DEPENDENCY ANALYSIS FOR SOFTWARE LICENSING AND SECURITY

Hannah Elizabeth Ahlstrom

A THESIS

Presented to the Faculty of Miami University in partial
fulfillment of the requirements
for the degree of

Master of Science

Department of Computer Science & Software Engineering

The Graduate School
Miami University
Oxford, Ohio

2025

Dr. Christopher Vendome, Advisor
Dr. Eric Rapos, Reader
Dr. Hakam Alomari, Reader

©

Hannah Elizabeth Ahlstrom

2025

ABSTRACT

As software systems grow in size and complexity, dependency management has become increasingly challenging. Dependencies, particularly transitive ones, can introduce security vulnerabilities and licensing incompatibilities even if only direct dependencies are explicitly invoked. Existing tools detect such issues but often overlook whether problematic dependencies are actually necessary for build or distribution. This thesis proposes a pruning approach that analyzes full dependency trees to remove uninvoked and test-scope dependencies, thereby enhancing security and legal compliance. Validation on synthetic and real-world Java projects demonstrated high effectiveness. Across 514 open-source projects, pruning reduced license incompatibilities by 86.6% (Maven) and 94.4% (Gradle), and decreased vulnerable dependencies by 56.9% (Maven) and 91.0% (Gradle). CVEs dropped by 60.7% for Maven and 79.2% for Gradle. Manual validation across 68 projects yielded precision rates of up to 87.5% (Maven) and 95.7% (Gradle). Total dependency counts were reduced by 61.5%–91.5%, simplifying systems and lowering maintenance burdens. By concentrating dependency management efforts on directly used components, this approach improves software security, compliance, and maintainability.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I am sincerely thankful to my thesis advisor, Dr. Vendome, for his guidance and mentorship throughout this research, and for continually encouraging me to learn and grow as a researcher. His support has been instrumental from my undergraduate studies through my graduate research. Additionally, I am very grateful to my committee members, Dr. Rapos and Dr. Alomari, for their time, valuable insights, and support throughout this process.

Chapter 1

Introduction

Software systems rely on dependencies to enable code reuse by leveraging existing software components or knowledge within their systems. Software reuse, such as the use of dependencies, enables the creation of such software systems with multiple software components. However, the growing use of dependencies has made it difficult for developers to manage or trace them manually. Not managing the dependencies of a software system can lead to licensing compliance issues and inherited security vulnerabilities that may exist within these external software components.

In response to this, there have been numerous dependency management tools designed to automate the oversight of multiple dependencies in a software system such as NuGet, Nanny, Bower, and Sprockets [1]. Although dependency management tools can provide benefits, their management of dependencies in a software system may be hindered by certain limitations. When dependency management tools are used on software systems with complex dependency trees, they may not be able to illustrate how the external software components used may impact the overall software system. Any dependencies that are not tracked or managed properly could create issues within the security or the licensing compatibility of the overall software system. Some dependency management tools also do not account for license compatibility between the multiple software libraries used within a software system which can lead to future licensing conflicts or legal implications. The motivations behind this thesis show a need for improved dependency management approaches in software development.

This thesis proposes an approach for analyzing and pruning software dependency trees to identify dependencies that are actively invoked and essential for building and redistributing the software system. Security vulnerability and license incompatibility analyses are integrated into this refined dependency structure, enabling developers to better manage and mitigate risks introduced by unnecessary or test-scope dependencies. By systematically pruning dependencies that are neither invoked nor required for redistribution, this approach helps focus dependency management efforts exclusively on components needed by the software system. This approach was also applied to a selection of open-source software projects to evaluate its effectiveness in improving dependency tracking and enhancing security and licensing compliance.

1.1 Motivation

The motivation for this thesis is to help address the growing complexities and risks introduced by modern software dependency management. While dependencies enable rapid development by leveraging software reuse, they also introduce significant management challenges that can cause issues in the security and legal compliance of software systems. As noted in *Surviving Software De-*

pendencies, developers often find themselves navigating extensive and complex dependency trees that are difficult to trace and maintain manually [2]. These dependencies can introduce vulnerabilities or license incompatibilities indirectly through dependencies brought in by other libraries. This risk is compounded by the challenge of ensuring that the software remains secure and compliant, given that developers must also manage evolving dependencies and updates that could affect the entire project. Many developers are unaware of the full list of transitive dependencies their projects rely on, leaving their systems vulnerable to disruptions from unexpected failures. A notable example occurred in 2016, when the removal of the tiny left-pad package broke builds for major JavaScript projects like Babel and React, exposing the fragility of dependency chains [2].

The limitations of current dependency management tools further emphasize the need for enhanced solutions. Although some tools automate aspects of dependency tracking, they often fail to distinguish which dependencies are essential for the use and redistribution of a software system. This thesis addresses such risks by proposing an analysis of software dependencies to ensure greater security and legal compliance through improved dependency tracking and management.

1.2 Contributions

This thesis aims to make the following contributions:

1. Propose an approach to analyze and prune uninvoked and test-scope dependencies while integrating security vulnerability and license compatibility assessments.
2. Evaluate the effectiveness of dependency pruning using synthetic data and assess its impact on security and licensing issues within the controlled dataset.
3. Apply the approach to open-source software to empirically evaluate the effectiveness of dependency pruning and assess its impact on security and licensing risks.

Chapter 2

Background & Related Work

This chapter presents background information on software reuse, dependency management, software licensing, and software security. Figure 2.1 shows the intersection of the research backgrounds that are discussed in this chapter and how they culminate into the topic of this thesis.

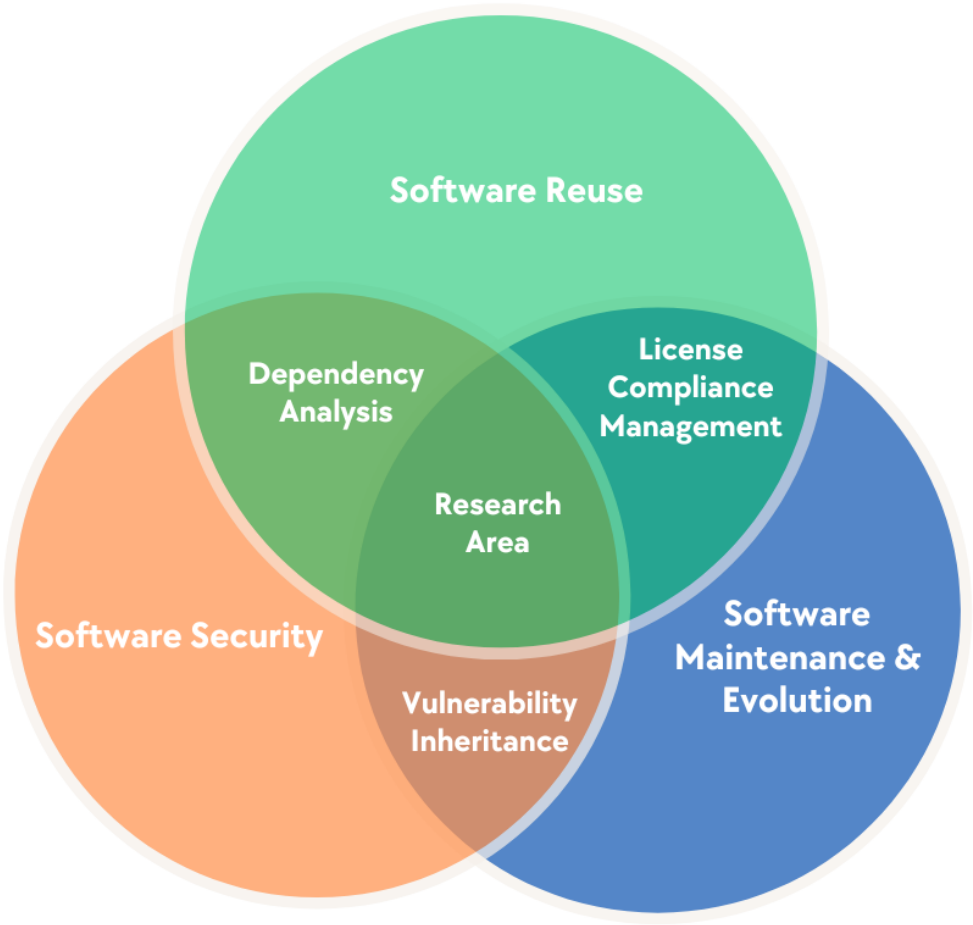

Figure 2.1: Research Backgrounds: Venn Diagram

2.1 Software Reuse

Software reuse is the practice of integrating existing software within a subject system and can occur at different granularities, from frameworks to code segments. By using existing components, software reuse facilitates improvements in multiple areas of software development [3].

- One benefit of software reuse is that it lessens development time. The reuse of existing software components reduces redundancy within the software development projects [4]. By utilizing an existing library or package, developers can avoid tedious tasks of repeating work already completed by other developers in areas such as testing, debugging, writing, and maintaining code [2]. This allows for the quicker production of new code by building upon ideas implemented in previous work.

- Utilizing already established software components leads to better quality software systems as these components already have established reliability [3]. When previously tested software components are reused in the development of a new software system, that software system is more likely to be error-free than if a developer wrote the entire software system from scratch. This is because publicly available software components are frequently tested and used, which makes them less likely to contain errors or bugs [5].

- By allowing developers to build upon established software components, software reuse also facilitates the collaboration of knowledge and expertise in the software development process by linking new ideas and software contributions across large groups of developers.

Commonly, software reuse involves linking to the existing code in the form of a *software dependency*. Software dependencies occur when an existing software component (in the form of a package, library, module, or framework) is reused in the development of a new software component [6].

Figure 2.2 illustrates the two types of dependencies relative to a given software system: direct and transitive dependencies. Direct dependencies occur when libraries or packages are directly called by the software system. Transitive dependencies occur when libraries or packages are indirectly called through other dependencies, meaning they are not directly invoked by the software system itself. The directed graph example in Figure 2.2 represents a dependency tree with its relationships and hierarchical levels of dependencies. In this example, the node representing the software system has three direct dependencies labeled A, B, and C. Examples of transitive dependencies (D, E, F, and C) are also presented, highlighting that dependency C can be classified as both direct and transitive. Most software systems contain dependency structures resembling the example provided in Figure 2.2, although typically on a much larger scale.

Dependency chains describe the multiple, complex dependencies that make up many modern-day software systems. These chains of dependencies are complex in that they would be infeasible and time-consuming for developers to manually trace [7]. Transitive dependencies mainly make up these dependency chains because they are the libraries and packages called by other dependencies. Transitive dependencies can be harder to manage manually as they are not explicitly declared when a developer is viewing the dependencies of their software system [6]. Even if a developer is utilizing one open-source library, they may not be aware of the dependency chain that that library contains.

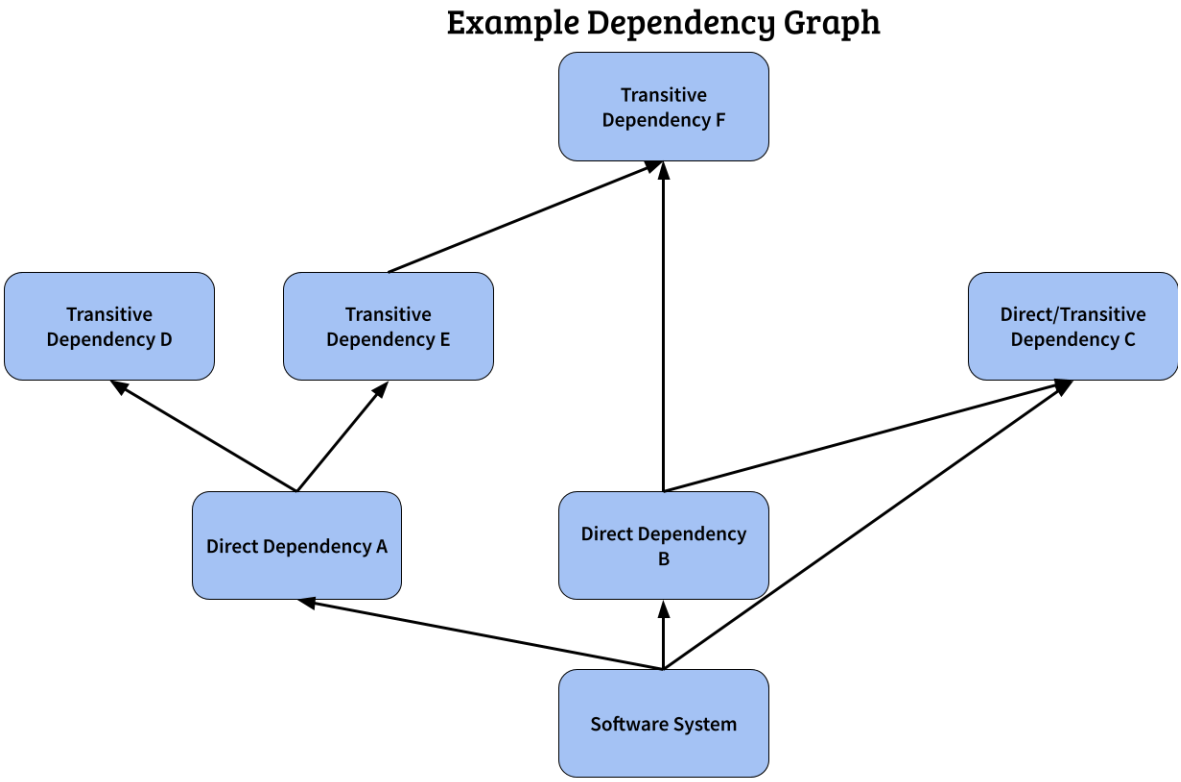

Figure 2.2: Example dependency graph

2.2 Dependency Management

This thesis focuses on the phases of deployment and maintenance in the Software Development Life Cycle (SDLC) that are both heavily impacted by the management of dependencies. These processes occur after a software system is fully developed and tested.

The maintenance process is essential to ensure that the deployed software system remains updated so that it can continue functioning effectively with changing demands from both its environment and its users. Given the rapidly evolving nature of software, the maintenance process is considered crucial to the continued functionality and effectiveness of a software system. This maintenance phase is also associated with the concept of changing requirements throughout development. Often used synonymously, both software maintenance and software evolution are defined as the adaptation of a software system after its deployment [8]. Examples of software maintenance in practice are observed through frequent updates of applications on smartphones. These updates do not necessarily indicate a deterioration of the software system but rather may represent adaptations to evolving environmental conditions or user requirements.

One part of the maintenance of a software system is the management of its dependencies. Complications during the software maintenance phase can stem from the management of structural dependencies between source files. These complications can be defects that are propagated between

files, the occurrence of multiple bugs or changes, and the significant cost of having to maintain dependencies within a software system [9]. There are many steps required to effectively manage the dependencies of a software system. One important step within dependency management is the ability to trace and identify the dependencies within the system. This can either be done manually through tracing source code files, potential build files, and other documentation or it can be done through the use of a software tool.

In order to use software dependencies, they must first be downloaded and installed. Dependency management tools or package managers are commonly used to easily install individual dependency packages. Some commonly used build automation tools that have dependency management functionalities include Maven, Gradle, and npm [2]. The goal of these build automation tools is to orchestrate the entire build process of forming executable programs from the source code files, libraries, and other data contained within a given software system [10]. These tools simplify the compilation and handling of software systems, leveraging different programming languages and also allowing for more portability in the use of packages [11].

2.3 Dependency Analysis

Understanding the dependencies of a software system requires extracting dependency information from multiple sources, primarily the build system and the source code itself. Build systems such as Maven and Gradle explicitly declare dependencies in configuration files, specifying which libraries a software system relies on. However, build declarations alone do not indicate whether dependencies are actively used in the code [12]. To address this, static analysis techniques such as Abstract Syntax Tree (AST) analysis can be applied to source code to examine dependency usage, including import statements and method invocations. This section details how dependencies are retrieved from both build systems and Java source code.

2.3.1 Dependencies in Maven Build Files

Maven is primarily used within Java projects [10]. Maven specifies build processes in a Project Object Model (POM) build file (*i.e.,* a pom.xml file within the directory of the software system). The POM file is analyzed when the build management tool runs to determine what commands need to be executed based on the build specifications of that software system.

The POM file contains project information and configuration specifications fundamental to building the project [13]. Within the POM file of a Maven software system, there are XML tags that map to different Maven build specifications and phases [14]. These XML tags follow a parent-child hierarchy and are used to specify configurations such as a project's dependencies. Within the specification of project dependencies in the POM file there are additional tags such as *groupId, artifactId, version, and scope* which can also be specified [13].

In a POM file, the required attributes when specifying a dependency are groupId, artifactId, and version. The groupId describes the group or organization to which the dependency belongs, while the artifactId is the specific project or library upon which the system depends [13]. The version specifies the release of the artifact so that the correct dependency can be identified. Optionally, the

scope can be declared, which can specific the relevant build tasks in which the dependency is used (*e.g.,* a test dependency for a framework like JUnit).

Listing 1 Example Declaration in pom.xml

```
<dependency>
    <groupId> org.mockito </groupId>
    <artifactId>mockito-core</artifactId>
    <version> 3.12.2</version>
    <scope> test </scope>
</dependency>
```

An example illustrating how a dependency would be documented within the POM file of a Maven software system is provided in listing 1. In this example, only one instance of a dependency is shown. If the complete XML file were presented, this dependency would be nested within the dependencies XML tag and within the root project XML tag. The Mockito core library, a mocking framework utilized for testing, is imported by this dependency [15]. The library being imported can be identified through the `groupId`, which indicates the source from which the library is retrieved, and the `artifactId`, which specifies the name of the library. Additionally, the version used within the software system is shown as release 3.12.2. The scope attribute is also included, indicating that this dependency is utilized exclusively for testing purposes.

2.3.2 Dependencies in Gradle Build Files

Gradle builds upon Java's core runtime and API. Gradle build scripts can be written using either Groovy or Kotlin Domain-Specific Languages (DSLs):

- Groovy: The build file is named build.gradle.
- Kotlin: The build file uses the .kts extension and is named build.gradle.kts

Although these scripts differ syntactically, they are functionally equivalent, allowing the declaration of plugins, dependencies, tasks, and repositories. When equivalently configured, both script types produce identical build outcomes.

Gradle manages dependencies by locating and resolving declared components required for specific build tasks. Once resolved, the necessary dependency files are stored in a local dependency cache, enabling reuse across subsequent builds and reducing the need for repeated network requests [16].

The examples in listing 2 illustrate how a runtime dependency on the library `spring-core` version `2.5` from the group `org.springframework` would be declared in each syntax. Despite minor syntactic differences between Groovy and Kotlin Gradle scripts, both declarations are semantically identical. Each clearly specifies the group (org.springframework), the artifact name (spring-core), and the version (2.5), indicating the specific component the software requires at runtime. Also, we can see that the dependency attributes listed in the build file example closely resembles some of

Listing 2 Example Declaration in build.gradle

```
// build.gradle (Groovy)
dependencies {
runtimeOnly group: 'org.springframework', name: 'spring-core', version: '2.5'
}

// build.gradle.kts (Kotlin)
dependencies {
runtimeOnly("org.springframework:spring-core:2.5")
}
```

the attributes previously observed in the Maven POM file. It is also important to note that both Maven and Gradle automatically include transitive dependencies, the dependencies of the declared dependencies listed in the build file [17, 18].

2.3.3 Dependencies in Java Source Code

In the context of our approach a dependency is seen as invoked if it has both an import statement and a method invocation in the code. To determine invocation we need to look for this within the source code of the software system.

In Java, import statements at the beginning of the source code file indicate the library or part of a library that is going to be used within that file. After importing the library or a specified class within that library the source code can then refer to the class(es) within that library [19].

Listing 3 Example Import in Java

```
    import org.apache.commons.lang3.StringUtils;
```

In the example shown in listing 3, only the `StringUtils` class from the `org.apache.commons.lang3` library is imported.

Following an import statement, a method from the imported class may be invoked. An example of direct method invocation is shown in listing 4. In this example we see a method call from the `StringUtils` class of commons-lang3, confirming that this dependency is actively used in the codebase.

Listing 4 Example Direct Method Invocation in Java

```
boolean result = StringUtils.isEmpty("hello");
```

2.4 License Compliance

One complication in the use of software dependencies is compliance with the software license that governs that dependency. Software licenses describe the legal regulations of how the code of that

dependency can be used and distributed. In some licenses that come with a software dependency, the ability of a developer to reuse and redistribute a software system with that dependency can be limited or completely blocked. Legal implications can occur when the restrictions and conditions of a software's license are violated by developers that use it [20].

This thesis deals with the use of open-source software dependencies which have multiple types of licenses governing the use and redistribution of that software component. Open source software is software that is distributed with source code and a license that permits free use, modification, and redistribution. To be considered open source, the license must meet specific criteria ensuring non-discrimination, technological neutrality, and the preservation of user rights across all use cases [21].

2.4.1 License Types

According to Arnoud Engelfriet's 2010 article *Choosing an Open Source License*, more than 50 open-source licenses are certified by the Open Source Initiative, each generally falling into one of two broad categories: permissive and copyleft [22]. Permissive licenses, such as MIT, BSD, and Apache, allow users to modify, distribute, and integrate the software freely, including in proprietary works, without imposing significant restrictions. These licenses are often referred to as "academic" or "free-for-all" licenses because they only require derivative works to credit the original authors. By contrast, copyleft licenses, such as the GNU General Public License (GPL) developed by Richard Stallman, enforce stricter conditions by requiring that derivative works adopt the same license, thereby ensuring that modified versions remain open-source. Copyleft licenses can further be divided into strong and weak categories: strong copyleft licenses (e.g., GPL) require that all software linked with GPL-licensed code also adopt the GPL, while weak copyleft licenses (e.g., LGPL) permit integration with proprietary software under certain conditions [22].

Engelfriet's article also introduces the concept of "keep-open" licenses, which mandate that any derivative works based on the original open-source component remain open-source. Examples include the Mozilla Public License and the LGPL. Additionally, "share-alike" licenses—such as the GPL and the Open Software License (OSL)—require only modified or extended versions of the open-source software to be made available under the same license. This categorization helps showcase the trade-offs between licensing models: permissive licenses maximize flexibility, while copyleft licenses ensure the ongoing availability of software as open-source [22].

The study conducted by Capiluppi et al. in the publication *Characteristics of Open Source Projects*, found that the GPL license was the most popular license in their sample as it was used in 77% of projects, followed by the LGPL license which was used in 6% of projects, and the BSD license which was used in 5% of projects [23].

2.4.2 License Compatibility

Licensing compatibility describes whether the combination or merging of two or more software components with different licenses is permitted or prohibited according to their respective licenses. License compatibility is only considered a concern when the software components being combined do not have the same license. When licenses differ, it must be ensured that they are compatible, meaning that code from varying licenses can be merged while still complying with all relevant

license conditions [24]. A licensing incompatibility occurs when one or more licenses are violated through the combination of software components. This compliance is important, as failure to adhere to licensing conditions when utilizing third-party software components may result in legal implications and can restrict or prohibit the redistribution of the software system containing those components.

License compatibility is further complicated by its directional nature, meaning that a license governs differently based on whether code flows into a project (upstream) or out of it (downstream). Permissive licenses like BSD allow downstream integration into nearly any project but restrict upstream compatibility to other permissive licenses. Copyleft licenses like GPL permit incorporating upstream code from permissive or GPL-licensed sources, but the resulting project must remain GPL-licensed, creating potential challenges when combining code from multiple sources with varying license obligations [25].

2.5 Software Security

Security vulnerabilities describe flaws within a software system that can cause it to behave in unintended or unexpected ways. A software component is considered vulnerable if it contains weaknesses that attackers can exploit, often to gain unauthorized access or obtain sensitive information [26].

Software security vulnerabilities are extremely important as the results of a successful attack on a vulnerability could be catastrophic [26]. An example of the impact of a successful attack on a vulnerability is a data breach such as an attacker gaining access to private and identifiable information of the users of that software system.

Another complication in the use of software dependencies is the possibility of these aforementioned security vulnerabilities. An example of the impact of security vulnerabilities in open-source software components is the Heartbleed bug. This bug occurred when a popular open-source software component, OpenSSL, contained a vulnerability that allowed thousands of malicious users to read memory contents. This allowed the malicious users to gain access to keys that protected a user's personal information. The exploited vulnerability was "detected two years later after it had affected the web servers that were powering 66% of the active websites at that time" [27]. In addition to the vulnerability of some open-source libraries, developers do not tend to update the dependencies of their software system, even when there is a known security vulnerability. They found that developers often avoid updating dependencies due to limited resources for handling breaking changes introduced in newer versions, as well as restrictions imposed by company policies on when and how dependencies can be updated [28].

2.5.1 Common Vulnerabilities and Exposures (CVE)

Software vulnerability tracking through Common Vulnerabilities and Exposures (CVEs) is crucial in modern software development. These vulnerabilities, as explained by Wang et al., increase the risk of exploitation through flaws in software or design, often introduced via dependencies in complex systems [29]. While dependencies enable reuse and faster development, they also expand a

system's attack surface by incorporating external code, which may contain vulnerabilities. Developers frequently struggle to keep dependencies updated, which leaves software exposed to security risks [30].

A significant challenge in managing these risks is dealing with transitive dependencies, which are indirectly included by direct dependencies. As more dependencies accumulate, identifying and addressing vulnerabilities becomes more complicated. Nguyen and Tran highlight the effectiveness of using dependency graphs to predict vulnerable components, showing how the inclusion of dependencies can obscure security risks [31]. Developers also face the challenge of balancing security fixes with functional requirements, which can delay necessary updates. Research by Pashchenko et al. shows that these delays often stem from resource limitations and the potential for breaking changes [28]. Open-source projects have been found to experience an increase in vulnerabilities as their dependency count grows, reinforcing the need for proactive monitoring and dependency management [27].

While automated tools and public CVE databases help identify vulnerabilities, their effectiveness depends on developer intervention. Developers must carefully manage the interplay between security and functionality to ensure their systems remain both functional and secure [6]. Thus, improving dependency management practices is essential for mitigating risks, ensuring updates are implemented efficiently, and reducing the likelihood of unaddressed security vulnerabilities within software ecosystems [30].

This thesis proposes a software analysis approach that builds upon existing tools for identifying known CVEs, detecting project-level licenses, and evaluating license compatibility. By analyzing the full dependency tree, the approach systematically prunes dependencies that are either uninvoked or limited to test scope, focusing exclusively on components necessary for the software's build and distribution. This pruning process consequently can prune unnecessary dependencies containing security vulnerabilities (CVEs) or licensing incompatibilities, thus reducing overall risks to security and licensing compliance.

2.6 Related Work

The growing reliance on software dependencies has led to challenges in security and license compliance, making dependency management a critical issue in modern software development. Prior research has highlighted difficulties developers face in managing vulnerabilities and licensing risks, the limitations of existing dependency management tools, and the prevalence of security and compliance issues in dependency ecosystems. While various approaches have been proposed to extract and analyze dependencies, few address the necessity of dependencies within the software system itself. This thesis extends the field by integrating dependency pruning into security and license compatibility analysis, systematically removing uninvoked and test-scoped dependencies to reduce risks associated with unnecessary dependencies.

Prior research highlights the challenges that currently exist when managing dependencies, particularly in maintaining security and legal compliance. Pashchenko et al. conducted a qualitative study on how developer practices influence the retention of vulnerable and outdated dependencies [30]. Their study analyzed how developers select, update, and resolve vulnerabilities in de-

pendencies, showcasing the difficulty of keeping dependency trees secure and compliant. Decan et al. studied how vulnerabilities propagate through package ecosystems and found that security vulnerabilities persist for long periods due to poor dependency management [32]. Similarly, Wang et al. examined vulnerabilities introduced by dependencies between security requirements, highlighting the risks of insufficient dependency oversight [29]. Almeida et al. found that while developers were able to interpret isolated licenses with reasonable accuracy, they often struggled to understand how these licenses interact when used in combination [33]. The authors highlight a clear need for tool support to guide developers through license interactions and compatibility concerns. These studies emphasize the importance of dependency tracking but do not provide solutions for mitigating these risks beyond better visibility. Our approach builds upon these findings by determining which dependencies within a software system can be pruned, reducing to potential for inherited vulnerabilities and compliance issues.

Specifically some works have stressed the prevalence software license incompatibilities in the context of multi-licensing. Moraes et al. investigated the prevalence of multi-licensed JavaScript projects and found that 373 projects exhibited licensing incompatibilities [34]. Vendome et al. categorized different types of licensing bugs in free and open-source software, highlighting the challenges of maintaining compliance in multi-licensed projects [35]. These studies show that licensing conflicts are widespread, but they primarily focus on identifying incompatibilities rather than mitigating them. Our approach extends this work by demonstrating how dependency pruning can remove uninvoked or test-scoped dependencies that could be introducing licensing conflicts.

Several studies have examined methods for dependency information extraction and analysis. Zhang proposed four techniques for extracting dependencies, including analyzing C/C++ source code, parsing build scripts, identifying dynamically linked libraries, and retrieving package metadata from system specifications [20]. Like this work, our approach leverages static analysis of the project's source code and leverages the project's build system to retrieve dependency information. However, our work focuses on applying dependency information extraction and analysis to determine whether a dependency is needed for the functionality and distribution of a software system.

Multiple studies have combined dependency analysis methods with security analysis. Márquez et al. developed *Advisory*, a technique for automating dependency analysis for security vulnerabilities. Their approach constructs a dependency graph and annotates it with security risk information [36]. While *Advisory* provides insights into where vulnerabilities exist, it does not address whether the dependencies introducing these vulnerabilities are necessary. Kula et al. conducted an empirical study of over 4,600 GitHub projects and found that 81.5% of systems retain outdated dependencies even when security advisories are available [37]. Their findings highlight a significant gap in developer awareness and responsiveness to vulnerability disclosures, particularly when the perceived migration effort is high or project responsibilities are ambiguous. This underscores the need for approaches that not only detect security issues but also reduce the burden of unnecessary dependencies—such as our method, which eliminates uninvoked or test-scope libraries to streamline maintenance and minimize inherited vulnerabilities.

Multiple studies have combined dependency analysis methods with licensing analysis. Prior work by Van der Burg et al. introduced Concrete Build Dependency Graphs (CBDGs) to analyze

dependency inclusion and exclusion in software deliverables. This dependency analysis was also paired with the identification of license compliance inconsistencies in real software systems [38]. Vendome further advanced this area by developing an infrastructure that automatically evaluates license compatibility across the full dependency tree of a software system. The tool identifies specific libraries and the corresponding conflicting licenses when an incompatibility arises, allowing developers to proactively address compliance issues that would otherwise prevent legal distribution [39]. Our work builds on this foundation by incorporating security vulnerability analysis in addition to licensing, and introducing a pruning mechanism that eliminates unused and test-scope dependencies based on their actual usage in the source code to reduce the overall license and security risk in a software system.

Several prior works have explored techniques related to the mitigation of either security vulnerabilities or license incompatibilities, though they differ from our approach in key ways. Gohmann proposed a semi-automated method for assessing license compatibility and suggesting alternative methods from compliant libraries [40]. This approach utilizes AST analysis but is limited to direct dependencies, whereas our approach extends pruning to transitive dependencies across the entire dependency tree. This approach also has a main focus on licensing and the suggestion of replacement libraries to reduce license incompatibilities whereas our approach focuses on which dependencies can be removed in a dependency tree to enhance security and license compliance. Our work differs in that it actively removes unnecessary dependencies, reducing both security and licensing risks.

Recent work has also begun incorporating AI and LLM-based assistance for dependency analysis and software maintenance. Nitin et al. which proposed CARGO, a context- and flow-sensitive partition refinement technique that enhances the output of existing decomposition tools. CARGO uses a system dependency graph to capture both code and external resource interactions, addressing limitations of prior approaches that resulted in distributed monoliths or reliance on distributed transactions [41]. Similarly, Pandey et al. explored AI-driven methodologies for mitigating technical debt in legacy systems, employing machine learning, natural language processing, and graph-based techniques to automate code refactoring and dependency management. Their work demonstrates the effectiveness of AI in identifying architectural bottlenecks and optimizing component relationships to improve maintainability [42]. While our work differs in focus, it similarly emphasizes the importance of accurate dependency modeling to improve downstream decisions, in our case for license and security risk mitigation. While these works explore the intersection of AI and dependency management, they do not integrate AI techniques for dependency tree refinement or pruning. Future work may explore how AI can aid in identifying removable dependencies with minimal developer intervention.

While existing studies have analyzed dependency retrieval, security vulnerabilities, and licensing conflicts, they primarily focus on identifying issues rather than addressing them. Our research uniquely integrates dependency pruning into security and licensing evaluations, demonstrating how removing uninvoked and test-scoped dependencies directly mitigates these risks. By refining dependency trees, our approach ensures that only essential dependencies remain in the software system, contributing to more secure and compliant software development practices.

Chapter 3

Approach

This research presents an approach for large-scale dependency analysis, specifically targeting single-module, open-source Java projects that use Maven or Gradle as their build systems. The following sections outline the overall architecture of the approach, detailing how it resolves, analyzes, and prunes dependencies and also how it determines which of these dependencies introduce security vulnerabilities or license incompatibilities.

3.1 Overview of Approach

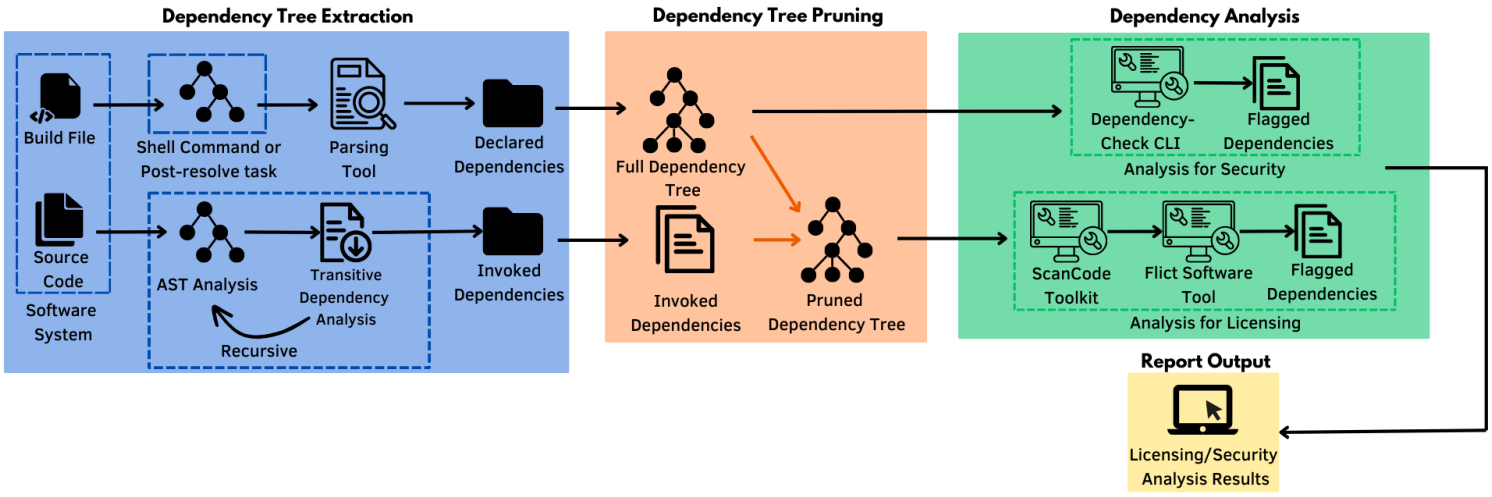

Figure 3.1: System Architecture

Figure 3.1 shows the overall system architecture of the approach. The approach extracts the

dependency information for a project and performs licensing or security analysis over the full set of dependencies.

The approach first extracts the comprehensive dependency tree from software systems by utilizing the native build command *mvn dependency:tree* for Maven projects or *gradle :dependencies* for Gradle projects. The dependency tree is then parsed into a standardized Node object model capturing all direct and transitive relationships. Subsequently, static analysis of the project's source code is conducted using Abstract Syntax Trees (ASTs) to identify explicitly invoked dependencies. This enables us to identify and prune dependencies that are declared but have no invocations within the source code.

For each node of the dependency tree this process is repeated, downloading the source code and performing static analysis to determine which of the declared dependencies at that level were imported and invoked. This information of which dependencies have invocations within their parent's source code is then used to perform dependency tree pruning. During dependency tree pruning we begin with the full dependency tree, removing nodes and their subtrees if they were found to have not been invoked.

The pruned dependency trees one of two analysis types: security or licensing compatibility analysis. During security analysis the OWASP Dependency-Check tool is employed to detect known vulnerabilities associated with each dependency [43]. During the licensing analysis, ScanCode-Toolkit is used to extract the project-level license of each dependency. To determine compatibility, the FLICT tool is used on each parent child relationship. Further detail on each part of this architecture is provided below.

3.2 Dependency Tree Extraction

The first step in the process is extracting the full dependency tree of all declared dependencies (direct and transitive) of a software system. To do this we leverage the build system, using the native commands `mvn dependency:tree` for Maven build systems and `gradle :dependencies` for Gradle. A prerequisite for running these native commands is that each software system must be single-module, meaning exactly one build file is present; otherwise, these commands cannot be executed successfully. To execute shell commands programmatically within the approach we use a process.

3.2.1 Maven Build System Dependency Tree Extraction

We use mvn dependency:tree to extract the complete set of declared dependencies in a Maven-based software system, including both direct and transitive dependencies. This command leverages Maven's native resolution process to provide an exhaustive and structured representation of all dependencies explicitly or transitively declared in the project's build configuration.

An example of executing this command on a Maven build system is illustrated by gathering the dependency tree from `com.google.guava:guava`, which can be found on Libraries.io [44]. After the Guava jar file has been downloaded, the files must be extracted to locate the directory containing the `pom.xml` file.

Before using this command, Maven must be installed. Once Maven installation is verified, the command `mvn dependency:tree` can be executed from the directory containing the `pom.xml` file. Following these steps for the Guava project, the dependency tree shown in listing 5 is generated.

Listing 5 Example Result from mvn dependency:tree Command

```
[INFO] com.google.guava:guava:bundle:31.1-jre
[INFO] +- com.google.guava:failureaccess:jar:1.0.1:compile
[INFO] +- com.google.guava:listenablefuture:jar:9999.0-empty-to-avoid-conflict-with-guava:compile
[INFO] +- com.google.code.findbugs:jsr305:jar:3.0.2:compile
[INFO] +- org.checkerframework:checker-qual:jar:3.12.0:compile
[INFO] +- com.google.errorprone:error_prone_annotations:jar:2.11.0:compile
[INFO] +- com.google.j2objc:j2objc-annotations:jar:1.3:compile
[INFO] \- jdk:srczip:jar:999:system (optional)
```

The next step involves parsing this information. For Maven build systems, this is achieved using the existing `maven-dependency-tree-parser` tool, which contains a set of parsers designed to process the output of the command `mvn dependency:tree`. This tool converts the command output into a `Node` object representation that can be easily traversed [45].

The `Node` object model provided by the `maven-dependency-tree-parser` tool was adapted to better suit our purposes and was used throughout the approach to store the dependency trees once parsed for both Maven and Gradle. The `Node` object model captures the hierarchical relationships within the dependency tree by following a parent-child structure. In this model, the software system is represented as the root node, with its direct dependencies stored as immediate child nodes. Transitive dependencies are structured as further descendants (e.g., grandchildren, great-grandchildren), forming a comprehensive hierarchical tree. Each node in the tree holds relevant attributes, including any flags raised during analysis that may impact the final evaluation. The full dependency tree, along with a pruned version focusing only on invoked dependencies, is leveraged in the upcoming analysis to identify and track security vulnerabilities or license incompatibilities.

3.2.2 Gradle Build System Dependency Tree Extraction

An example of executing this command on a Gradle build system is illustrated by obtaining the dependency tree from a small Gradle example project available on GitHub [46]. Once Gradle installation is confirmed,the command `gradle :dependencies` should be executed in the directory containing the `build.gradle` file. Following these steps for the small Gradle example project, the dependency trees shown in listing 6 are generated.

For Gradle build systems, a parser was developed to process the dependency tree into the modified `Node` object model, ensuring compatibility with the Maven dependency tree representation. Unlike Maven, Gradle lists dependencies under multiple configurations, meaning that a dependency declared as `implementation(org.apache.logging.log4j:log4j-core:2.20.0)` may also appear under additional scopes in the generated dependency tree. To standardize the representation across both build systems, the Gradle dependency tree is consolidated into a format similar to Maven, where each dependency is listed only once, and its `scope` attribute includes all configurations under which it appears. This consolidation ensures that the `Node` object model accurately

Listing 6 Example Result from gradle :dependencies Command (Reduced)

```
annotationProcessor - Annotation processors and their dependencies for source set 'main'.
No dependencies

compileClasspath - Compile classpath for source set 'main'.
+--- org.codehaus.groovy:groovy:3.+ -> 3.0.17
+--- org.codehaus.groovy:groovy-json:3.+ -> 3.0.17
|    \--- org.codehaus.groovy:groovy:3.0.17
+--- com.codevineyard:hello-world:{strictly 1.0.1} -> 1.0.1
\--- :simple-jar

implementation - Implementation only dependencies for source set 'main'. (n)
+--- org.codehaus.groovy:groovy:3.+ (n)
+--- org.codehaus.groovy:groovy-json:3.+ (n)
+--- com.codevineyard:hello-world:1.0.1 (n)
\--- simple-jar (n)

runtimeClasspath - Runtime classpath of source set 'main'.
+--- org.codehaus.groovy:groovy:3.+ -> 3.0.17
+--- org.codehaus.groovy:groovy-json:3.+ -> 3.0.17
|    \--- org.codehaus.groovy:groovy:3.0.17
+--- com.codevineyard:hello-world:{strictly 1.0.1} -> 1.0.1
\--- :simple-jar

...
```

reflects all applicable scopes for each dependency. A dependency in Gradle is only considered to be of test scope if all of its listed configurations belong exclusively to test-related configurations.

3.2.3 Pruning

To construct the pruned dependency tree, the source code must be analyzed to determine which declared dependencies are imported and directly invoked. This process involves examining the complete dependency tree and pruning dependencies that are used only for testing or are not explicitly referenced or invoked within the code. The result of this pruning process is a dependency tree that contains only the essential dependencies required for building and distributing the system. Eliminating test-scoped and unused dependencies and any of their associated security vulnerabilities or license incompatibilities improves the security and licensing compliance of the overall software system without affecting software functionality.

Since this approach focuses exclusively on Java-based software systems, static analysis is applied using Abstract Syntax Trees (ASTs) to identify direct dependencies referenced within the code. ASTs provide a hierarchical representation of the code, where nodes represent constructs such as method calls, class definitions, and import statements. Parsing the AST allows meaningful information to be extracted from these constructs. To facilitate this parsing, the *org.eclipse.jdt.core.dom* ASTParser class is utilized. In order to generate an AST, each source code file in the software system must first be read and processed into a string format. This string is then provided to the AST parser, which converts it into a tree representation of the code structure [47]. By traversing the generated ASTs, imported dependencies can be accurately identified, and direct method invocations associated with them can be located. This enables an understanding of which declared libraries are actively used within the codebase.

To identify instances of imported dependencies and their method invocations, the ASTVisitor class is used to visit the *ImportDeclaration* and *MethodInvocation* nodes [48]. The output from each file's AST analysis is continuously compiled as these nodes are visited, forming a list of dependency package names that represent the invoked direct dependencies of the analyzed software system. To determine the invoked transitive dependencies, this process is repeated, where all source code files for each direct dependency identified as imported and invoked in the initial AST analysis are analyzed. For instance, if we have the dependency chain A->B->C we would start at A's source code to analyze and determine if B was invoked. If B was determined to have been invoked, we would download and analyze B's source code to determine if C was invoked, and so on for all subtrees in the dependency tree. This iterative process continues until all direct and transitive dependencies of the original software system have been examined.

We see an overview of this process in Figure 3.2 which shows the high-level flow of operations performed on each dependency tree node from the source code download to the output of the AST analysis. To facilitate this recursive analysis, the source code of each invoked library must be downloaded and processed. To download the source code we try two approaches. The first approach involves retrieving the source code of a given library from the Maven Central Repository. For example, if we needed to download the source code of commons-lang3 at version 3.12.0, we would construct a URL to its repository location based on its group ID, artifact ID, and version, https://repo1.maven.org/maven2/org/apache/commons/commons-

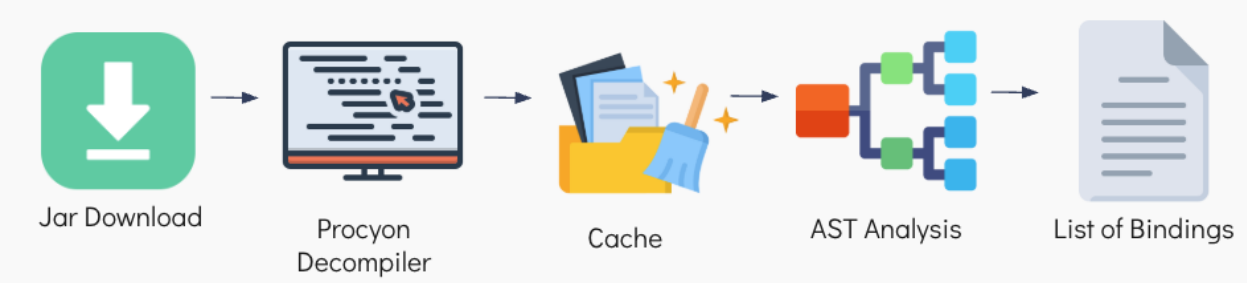

Figure 3.2: Workflow of source code download and AST analysis that is performed recursively

lang3/3.12.0/. The Jsoup library is then used to locate the corresponding JAR file of the dependency by matching its name with the provided artifact ID and version. For our commons-lang3 example, this would be https://repo1.maven.org/maven2/org/apache/commons/commons-lang3/3.12.0/commons-lang3-3.12.0.jar. If the JAR file is found and does not already exist in the target directory, the necessary directories are created, and the jar is downloaded and subsequently extracted by executing the shell command (`jar xf`). If extraction completes successfully, all `.class` files are traversed and decompiled into `.java` files to enable AST analysis. For this decompilation process, the Procyon decompiler, integrated as a Maven dependency, is used within the approach [49].

The second approach leverages the Libraries.io API to locate the repository associated with a given `artifactId`. For example, an API call such as https://libraries.io/api/search?q=jackson-core&api_key=<valid_api_key> is used to retrieve relevant repository metadata. Responses for the specified `artifactId` are processed, and matches between the `groupId` and `artifactId` of the dependency and the repository name and owner from the API response are identified. To improve matching accuracy, both the `groupId` and `artifactId` are normalized by removing dashes and periods. A match is assumed when the normalized `artifactId` contains the repository name and the normalized `groupId` matches the repository owner or vice versa. Once a match is identified, repository metadata is retrieved, and the GitHub API is used to locate repository tags corresponding to the dependency version (e.g., https://api.github.com/repos/FasterXML/jackson-core/tags). If a matching version is found, the zipball URL for that specific version is obtained (e.g., https://api.github.com /repos/FasterXML/jackson-core/zipball/refs/tags/jackson-core-2.15.3), and the source code is downloaded as a ZIP file, which is then extracted to access the dependency's source code.

If neither the Maven Central strategy nor the Libraries.io API yields valid results, a *sourceCodeDownloadFailureFlag* is assigned to the corresponding dependency node in the tree. This flag indicates that the pruning of the dependency's subtree may be less accurate due to the inaccessibility of its source code. To optimize performance, all JAR downloads and source code downloads are cached. If the same library version is required again, it can be retrieved directly from the cached files, avoiding the need for repeated downloading and decompilation.

Before AST generation can be performed for a dependency, all of its direct dependencies must be downloaded as JARs. To download these needed JARs we use the same two approaches identified earlier, maven repository and Libraries.io API. As the source code and required JARs

for each dependency are recursively downloaded, the code is analyzed by traversing its AST to identify imported libraries and invoked methods. The AST nodes `ImportDeclaration` and `MethodInvocation` are visited and tracked to extract relevant dependency information. Each Java file is processed to first collect all identified imports and then determine the methods invoked within that file. To confirm active dependency usage, import statements are matched with invoked methods, ensuring that only dependencies with both import and invocation references are retained. Unique dependencies identified through this analysis are compiled into a list and used to remove uninvoked or test-scoped dependencies from the declared dependencies of the project.

To perform pruning, invoked dependencies must be mapped to their declarations in the build file, as their formats may differ (e.g., `org.apache.commons.io.FileUtils` versus its Maven identifier `commons-io:commons-io`). Each dependency node in the full dependency tree is examined to determine whether it is invoked, using a combination of two strategies.

To avoid unneeded API calls, we maintain a cache file that maps prior successful searches given a package name to its corresponding groupId:artifactId coordinate to avoid redundant network requests. If the result is not found in the cache, it sends a query to the Maven Central search API, applying rate limiting and retry logic with exponential backoff in case of transient failures. If a valid response is returned, the result is extracted, stored in the cache for future use, and returned. All cache operations are protected by read-write locks to ensure thread safety, and any I/O or network-related errors are logged.

If a match is found, the dependency is retained in the pruned tree; otherwise, it is excluded. This analysis and pruning process begins at the root node and recursively continues down to each leaf node, resulting in a pruned version of the complete dependency tree. By removing unused dependencies, this approach ensures that the final tree accurately represents the dependencies required for building and distributing the software, ultimately reducing potential security and licensing risks.

3.3 Dependency Analysis

3.3.1 Analysis for Security

To determine whether a software component within the dependency tree of a software system contains a security vulnerability, the analysis in the approach is based on the Common Vulnerabilities & Exposures (CVE) database, which documents libraries with known security vulnerabilities. The identification of libraries with documented CVEs within a software project's dependencies is conducted by leveraging the OWASP Dependency-Check CLI tool. This is achieved by traversing the dependency tree and running the tool on each node to determine whether any of its direct dependencies have CVEs. This process is repeated for every node that functions as a parent within the tree until it has been determined whether any CVEs are present for every dependency in the software system's dependency tree. We can see the flow of operations performed for security analysis at a high level in Figure 3.3.

Similarly to the AST analysis process, caching is utilized to optimize efficiency when downloading required source code files or dependency JARs. Before security analysis is performed on a given library, a check is conducted to determine whether an exact version of that library already ex-

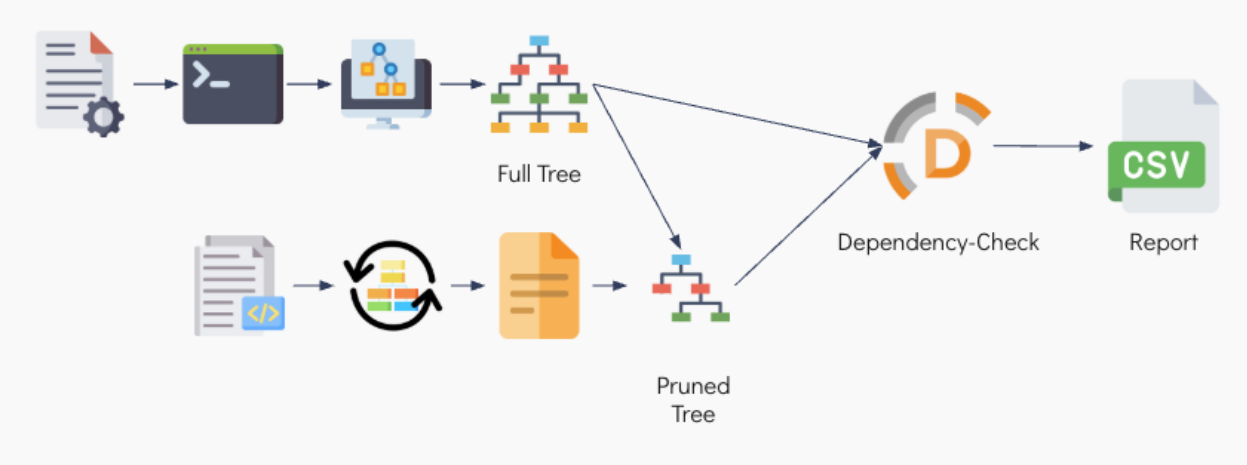

Figure 3.3: Workflow of security analysis applied to both full and pruned dependency trees

ists and whether a security analysis file has already been generated. If the security analysis has been previously conducted for that dependency at the specified version, the existing security analysis file is reused to avoid rerunning the Dependency-Check CLI tool on an already analyzed library.

If an existing security analysis file is not found within the source code, the process begins by identifying the build system used by the project, whether it is Maven or Gradle, by searching for corresponding build files such as `pom.xml` or `build.gradle`. Next, an attempt is made to compile the project using the relevant command (e.g., `mvn clean install` or `gradle build`). If the build fails or no build file is found, the issue is flagged with the `sourceCodeBuildFailureFlag`, and further analysis for that library is halted.

For the Dependency-Check CLI tool to function properly, a `lib` folder containing all direct dependencies of the libraries, stored as JARs, must be accessible. First, a check is performed to determine whether a `lib` folder already exists with all the required JARs for the library under analysis. If this folder is not present, it is created, and two attempts are made to download the required JARs (i.e., the direct dependencies or child nodes of the library currently being analyzed). If both attempts fail, a `setJarDownloadFailureFlag` is assigned to indicate the failure.

To obtain the JAR files, the same two approaches used for downloading JARs during AST analysis are followed. Initially, an attempt is made to download them from the Maven Central Repository. If this fails, the Libraries.io API is queried as an alternative source. The retrieval process begins by constructing a download URL based on the dependency's `groupId`, `artifactId`, and `version` to locate the JAR file on Maven Central. If the JAR file is not available in Maven Central, the Libraries.io API is queried to retrieve repository metadata as an alternative. The JAR files for the direct dependencies of the analyzed library are then stored in a designated directory, where they serve as input for the Dependency-Check CLI scan.

The Dependency-Check CLI command is configured to scan the directory containing these dependencies while using API keys to access the National Vulnerability Database (NVD). This process ensures that known vulnerabilities are detected by matching dependency metadata against the NVD entries, with a detailed JSON report generated as the output [43].

The approach determines whether a dependency has a CVE by processing the JSON security reports generated by the Dependency-Check CLI to identify known vulnerabilities within a software system. When analyzing a project, a check is first performed to determine whether a previously generated JSON report exists for a given dependency (as part of the caching strategy implemented throughout the approach). If the report is found, the content of the JSON file is read as a string and then parsed into structured Java objects.

The JSON data is deserialized using the Jackson library into objects representing individual dependencies and their associated vulnerabilities. Each dependency object contains detailed information, including any linked vulnerabilities, such as CVE identifiers, severity levels, descriptions, and references. If the report is unavailable or cannot be read, the absence of this data limits the ability of the approach to conduct security checks for the affected software system. By leveraging these reports, all relevant CVEs are identified and appropriately associated with their corresponding dependencies.

The analysis for security outputs two detailed security analysis reports. Each output file prints out all dependency tree nodes in the tree providing metadata such as its level, groupId, artifactId, version, scope, parent node, any flags, and associated vulnerabilities. The level column shows the user where the dependency sits in the dependency tree to allow for traceability where 0 represents the root node, 1 represents a direct dependency, 2 represents a transitive dependency that is the child of a direct dependency, and so on. To give more insight on the hierarchal structure of the dependency tree, each dependency's parents are also listed. The flag columns are used to inform the user of any failures at any point in the approach that have the potential to impact the results shown in the final analysis. Vulnerability details are also displayed including the CVE name, severity, description, references, and the affected software. If there are multiple CVEs associated with one dependency then that dependency will have multiple rows within the report. The user will be able to tell if a dependency has multiple CVEs because it will consecutively have the same information on each row with only the CVE data differing.

Table 3.1: Excerpt from output security analysis report showing selected fields. The full report includes additional metadata for each node, such as all analysis flags, CVE source, CVE description, CVE references, and affected software.

Level	Parent Node	Artifact ID	Group ID	Scope	Version	CVE Name	CVE Severity
0	–	tempus-unit-conversion	com.hashmapinc.tempus	–	1.0.4	–	–
1	tempus-unit-conversion	poi	org.apache.poi	compile	4.0.1	CVE-2019-12415	MEDIUM
1	tempus-unit-conversion	poi	org.apache.poi	compile	4.0.1	CVE-2022-26336	MEDIUM
2	poi	commons-codec	commons-codec	compile	1.11	–	–
2	poi	commons-collections4	org.apache.commons	compile	4.2	–	–
2	poi	commons-math3	org.apache.commons	compile	3.6.1	–	–

The reports generated follow two potential formats: full and pruned. We can see an excerpt of an example report that shows the format for security analysis in Table 3.1. In this table we see the dependency **poi** is listed twice with the exact same dependency information. This is because it has multiple CVEs and each CVE's information is listed on its own row. Full reports include every dependency node in the tree, along with detailed information about vulnerabilities, flags indicating build issues, and metadata about the relationship between dependencies. In contrast, pruned reports

exclude any test dependencies or dependencies with no usage in the codebase, focusing only on actively used dependencies needed for the build and redistribution of the software system.

3.3.2 Analysis for Licensing

The approach employs several strategies to determine the project-level license of a node. It begins by searching for licensing-related files within the source code directory, specifically targeting files such as LICENSE, license.txt, and license.md. ScanCode-Toolkit is then applied to these files to extract license information. If a license is found, it is recorded and associated with the node. We can see the flow of operations performed for licensing analysis at a high level in Figure 3.4.

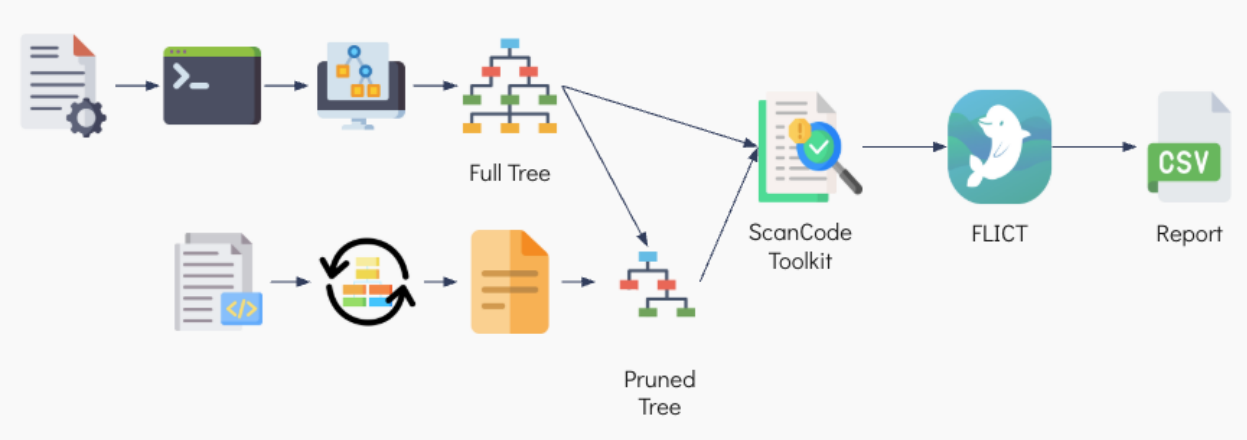

Figure 3.4: Workflow of licensing analysis applied to both full and pruned dependency trees

If the license is not found in the primary license files, the approach attempts to extract it from the project's build file if one of the following exists within that node's source code: build.gradle, pom.xml, build.xml. If a valid build file is found, ScanCode-Toolkit is applied to it in an attempt to extract licensing information. If this step is successful, the identified license is stored, and the process concludes. Otherwise, the approach proceeds to check the README.md file, applying the same scanning process to capture any license references embedded within it.

If no license is identified from the license file, build file, or README, the approach escalates to a full source code scan, applying ScanCode-Toolkit to the entire project directory. This step extracts any residual license information embedded within the source code. If ScanCode-Toolkit returns an "unknown-license-reference" the approach either discards it entirely or removes it from expressions containing multiple licenses. Additionally, "WITH" clauses, which indicate license exceptions (e.g., "GPL-2.0 WITH Classpath-exception-2.0"), are stripped since they are not recognized by the compatibility tool FLICT, ensuring the processed license expressions remain valid for further analysis.

Throughout this process, the detected license expressions are formatted and matched against the SPDX licenses expressions that Flict supports. To do this, the approach first executes the command "flict simplify <license>" which attempts to simplify the given license expression. The Flict output, returned as JSON, is parsed to extract the "simplified" field, which contains the normalized license.

If this result matches a known license from Flict or SPDX, the license is accepted. If a license fails to be normalized the licenseNormalizationFailureFlag is set since a license that is not in SDPX format will not be recognized by Flict when compatibility analysis is run.

To determine if there are any license incompatibilities between software systems in the dependency tree, the approach uses the Flict / FOSS License Compatibility Tool [50]. The approach traverses the dependency tree and uses the Flict tool to check compatibility between all parent-child nodes. Before utilizing the tool the approach checks that both the parent and child node both have at least one identified license. If both the parent and child each have at least one license, the approach runs the "flict display-compatibility" command.

The example shown in listing 7 demonstrates the output of the FLICT compatibility tool when checking the compatibility of BSD-3-Clause, MIT, and GPL-2.0-only licenses [50].

Listing 7 Example Output from 'flict display-compatibility'

```
flict display-compatibility BSD-3-Clause MIT GPL-2.0-only

{
  "compatibilities": [
    {
      "license": "MIT",
      "licenses": [
        { "license": "BSD-3-Clause", "compatible_right": "true", "compatible_left": "true" },
        { "license": "GPL-2.0-only", "compatible_right": "true", "compatible_left": "false" }
      ]
    },
    {
      "license": "BSD-3-Clause",
      "licenses": [
        { "license": "MIT", "compatible_right": "true", "compatible_left": "true" },
        { "license": "GPL-2.0-only", "compatible_right": "true", "compatible_left": "false" }
      ]
    },
    {
      "license": "GPL-2.0-only",
      "licenses": [
        { "license": "MIT", "compatible_right": "false", "compatible_left": "true" },
        { "license": "BSD-3-Clause", "compatible_right": "false", "compatible_left": "true" }
      ]
    }
  ]
}
```

In this command, license compatibility is checked by Flict while considering the directionality of the relationship. Within the scope of the approach, the compatibility between two software licenses is evaluated: the parent license (project license) and the child license (dependency license). For example, when a parent license `MPL-2.0` and a child license `Apache-2.0` are analyzed, the executed command is *flict display-compatibility MPL-2.0 Apache-2.0*. The command's output is captured and parsed into a JSON object. The compatibility details are stored in the 'compatibilities' array of the JSON output. Each entry in the array corresponds to a specific license (parent) and

contains an array of related licenses (children) with compatibility fields. The key fields are:

- compatible_right: Indicates whether the parent license can include and redistribute the child license.
- compatible_left: Indicates whether the child license can be redistributed under the terms of the parent license.

The approach first iterates through the compatibilities array to find an entry matching the parent license. It then iterates through the associated licenses to locate the child license. When both matches are found, the method evaluates the compatible_left field. If compatible_left is true, the licenses are deemed compatible, if it is false the licenses are incompatible. For example, this can be seen in the scenario shown in listing 8 where the parent license is MPL-2.0, and the child license is Apache-2.0. When the flict command is executed, the output is:

Listing 8 Example Output from 'flict display-compatibility' command

```
{
  "compatibilities": [
    {
      "license": "MPL-2.0",
      "licenses": [
        { "license": "Apache-2.0", "compatible_right": "false", "compatible_left": "true" }
      ]
    },
    {
      "license": "Apache-2.0",
      "licenses": [
        { "license": "MPL-2.0", "compatible_right": "true", "compatible_left": "false" }
      ]
    }
  ]
}
```

The approach processes this output by first identifying the entry where the "license" field matches the parent license MPL-2.0. Within this entry, it searches the "licenses" array for an entry where the "license" field matches the child license Apache-2.0. The corresponding compatibility details are "license": "Apache-2.0", "compatible_right": "false", "compatible_left": "true". From this entry, the approach extracts the *compatible_left* value, which is true. This indicates that the dependency (licensed under Apache-2.0) can be redistributed within a project licensed under MPL-2.0 without violating license terms. In the case that a violation exists(the compatible_left value is false), the child node's hasLicenseViolationWithParent flag is set to true, signaling a conflict.

The approach handles compatibility for several licensing scenarios, including cases where:

1. Both the parent and child have multiple licenses – the code checks all combinations of parent-child license pairs to find at least one compatible pair.
2. Only the parent or the child has multiple licenses – the code evaluates whether any license in the multi-license set is compatible with the other single license.

3. Both nodes have a single license – a direct comparison is performed using Flict to determine compatibility.

The analysis for licensing outputs two detailed licensing analysis reports. Each output file prints out all dependency tree nodes in the tree providing metadata such as its level, groupId, artifactId, vers ion, scope, parent node, any flags, its project level license(s), and whether or not it has a license violation with its parent.

Table 3.2: Excerpt from output licensing analysis report showing selected fields. The full report includes additional metadata such as analysis flags and license identification source

Level	Parent Node	Artifact ID	Group ID	Scope	Version	License(s)	License Violation w/ Parent
0	–	tempus-unit-conversion	com.hashmapinc.tempus	–	1.0.4	BSD-3-Clause	Unknown
1	tempus-unit-conversion	poi	org.apache.poi	compile	4.0.1	–	Unknown
2	poi	commons-codec	commons-codec	compile	1.11	Apache-2.0	Unknown
2	poi	commons-collections4	org.apache.commons	compile	4.2	Apache-2.0	No
2	poi	commons-math3	org.apache.commons	compile	3.6.1	Apache-2.0	No

The reports generated follow two potential formats similarly to the security analysis reports: full and pruned. We can see an excerpt of an example report that shows the format for licensing analysis in Table 3.2. Full reports include every dependency node in the tree, along with detailed information about all licenses identified and any incompatibilities between them, flags indicating build issues, and metadata about the relationship between dependencies. Pruned reports exclude any test dependencies or dependencies with no usages in the codebase, focusing only on actively used dependencies needed for the build and redistribution of the software system.

The reports are easy to read and analyze using any CSV reader, with the ability to filter and sort based on fields like CVE severity or license violations. This flexibility allows teams to prioritize the resolution of license incompatibilities introduced by dependencies that are essential to the building and redistribution of the software system.

Chapter 4

Experimental Design

This thesis developed an approach for improving security and licensing compliance by pruning uninvoked and test-scope dependencies from a software system's dependency tree, reducing unnecessary risk and complexity. This approach addresses the increasing use of dependencies within modern-day software systems that become infeasible to manually trace. Without the proper management of the dependencies of a software system, issues such as licensing compliance and the security of the overall software system can arise within the software deployment and maintenance phases of the Software Development Life Cycle.

While there exist multiple dependency management systems that can help the management of a software system's dependencies, these management systems have limitations such as overlooking dependencies that may not need to be included within the build and redistribution of a software system or not considering how the use of external software components impacts the license of the overall software system. These limitations are particularly stressed in software systems with complex dependency trees consisting of multiple software components. Due to this, there is a need to address the management of complex dependency trees, especially in the realms of software licensing and software security. This thesis proposes an approach that identifies dependencies in a software system's dependency tree that are not needed for its functionality and distribution which may also introduce security vulnerabilities and licensing incompatibilities.

The work presented in this thesis addresses the following two research questions:

1. **RQ1:** How effective is the proposed approach in accurately pruning uninvoked and test-scope dependencies from software dependency trees?

2. **RQ2:** How does dependency pruning impact the security and licensing compliance of a software system?

    - RQ2.A: What is the effect of pruning on security vulnerability analysis?
    - RQ2.B: What is the effect of pruning on licensing compliance analysis?

To evaluate these research questions, both synthetic and open-source datasets were used. For RQ1, a controlled synthetic dataset was constructed consisting of 40 test cases per build system (Maven and Gradle), with known dependency characteristics including invocation status, scope, and license or CVE presence. This enabled precise measurement of true positives, false positives, precision, recall, and F1-score. In addition, a sample of 38 Maven and 30 Gradle open-source projects was manually validated to assess the pruning accuracy in real-world settings. For RQ2, the same datasets were used to examine how pruning affected the presence of licensing incompatibilities and security vulnerabilities. In the synthetic dataset, incompatibilities and CVEs were

deliberately injected at specific locations in the dependency tree. In the open-source dataset, analysis reports were compared before and after pruning to measure reductions in identified issues. This approach enabled systematic evaluation of the pruning method in both controlled and realistic conditions.

4.1 Synthetic Dataset Construction

To construct the synthetic dataset, we manually constructed synthetic software systems that reflect the structure of real software systems. The purpose of these synthetic software systems is to mimic cases that may be encountered by the approach within the open-source dataset, allowing its precision and recall to be documented. This synthetic dataset provides full control over the version, scope, and depth of every dependency within the synthetic software system's dependency tree. By generating this synthetic software system from scratch, different licensing incompatibilities and security vulnerabilities can be injected within the dependency tree. The creation of these synthetic systems also enables the manipulation of hierarchical relationships between software components, such as transforming a direct dependency into a transitive dependency.

To evaluate the approach's ability to flag dependencies that introduce a license incompatibility, licenses are strategically placed within the dependency tree of the synthetic system to create cases of license incompatibility with the dependency's parent. This process involves introducing conflicting licenses and compliant licenses between software components in the synthetic system by modifying the licensing metadata for dependencies. By utilizing these different cases a dataset is built for evaluating the pruning and licensing analysis of the approach.

To evaluate the approach's ability to flag dependencies that introduce a security vulnerability, dependencies are systematically placed within the dependency tree of the synthetic system to create cases where none, one, or multiple security vulnerabilities occur. This process involves introducing dependencies with known CVEs and those with no known CVEs at different levels of the dependency tree. By utilizing these different cases, a dataset is built for evaluating the pruning and security analysis of the approach.

For both analysis types in the synthetic dataset, if a parent project declares and invokes the same dependencies as its child project, mvn dependency:tree only displays them under the parent. However, when the parent's declarations and invocations are removed, the child's usage appears in the output. This behavior is due to Maven's optimization, where transitive dependencies are omitted if they are already direct dependencies of a higher-level project. The gradle :dependencies command does not omit transitive dependencies in this way. While not a limitation of the approach itself, this should be noted when interpreting results. To ensure consistency in synthetic tests, parent projects did not declare or invoke the same dependencies as their children.

Table 4.1 summarizes the overall structure of the constructed synthetic dataset. In this context, **L1** refers to transitive dependencies at depth 1 (i.e., a child of a direct dependency), and **L2** refers to transitive dependencies at depth 2 (i.e., a child of a first level transitive dependency). These results reflect the outcome of carefully designing test cases to cover a wide range of invocation states, dependency levels, scope types, and licensing or security conditions.

Table 4.1: Synthetic Dataset Summary

Analysis Type	# of Test Cases	Dependency Levels	Invocation States	Scope Types
Licensing	28	Direct, Transitive L1, L2	Invoked, Not Invoked	Compile, Test
Security	14	Direct, Transitive L1, L2	Invoked, Not Invoked	Compile, Test

4.2 Open-source Dataset Construction

To construct the open-source dataset we leveraged Libraries.io Open Data and the Software Heritage Archive.

Libraries.io Open Data is a dataset containing over 2.7 million unique open-source packages, 33 million repositories, and 235 million inter-dependencies [51]. This data provides insight into open-source software that can be utilized in constructing the open-source data sample. To build this sample, the Open Data dataset is first downloaded and extracted. Once extracted into a CSV file, filtering is applied to identify Maven Java projects using the fields provided within the dataset, after which the file rows are randomly shuffled. Once filtering and shuffling are completed, a script is executed to clone each Maven project and inspect whether it is a valid candidate for inclusion in the open-source dataset. To qualify as a valid candidate, a project must contain exactly one `pom.xml` file, ensuring that it is a single-module project, and at least one `.java` file, confirming the presence of source code required for AST analysis and pruning. If a cloned repository is found to be an invalid candidate, it is removed from the open-source dataset.

To collect open-source data for Gradle-based Java software systems, the Software Heritage API is utilized to query the Software Heritage Archive. The Software Heritage Archive is an initiative that collects and provides access to source code and development histories across multiple repositories. Its purpose is to safeguard software artifacts by continuously crawling platforms such as GitHub and GitLab, ensuring long-term availability for research and analysis. In this study, repository data is sourced directly from the Software Heritage Archive through its API, ensuring access to reliable, archived versions of the dependencies analyzed [52]. To retrieve relevant Gradle projects, a Python script is used to interact with the API's origins endpoint, such as https://archive.softwareheritage.org/api/1/origins/?origin_count=10&page_token=-9223371018704180675.

From the API response, the listed origins are iterated through, and if the `url` attribute contains a valid GitHub link, the repository is cloned locally to determine whether it meets the inclusion criteria for the open-source dataset. These criteria require that the project be a single-module Gradle system (containing exactly one `build.gradle` file) and that it includes at least one `.java` file for source code analysis. Once a valid Gradle project is identified, it is logged in a CSV file that tracks projects suitable for inclusion in the dataset. A script is then applied to randomly sample projects from this compiled CSV and download their source code using the recorded repository links.

By incorporating two large and diverse sets of open-source software systems into the open-source dataset, the likelihood of including a sufficient and varied number of security vulnerabilities and licensing incompatibilities within the dependency trees of the selected software systems is increased.

Table 4.2 summarizes the results of this construction process. It shows the total number of valid

projects identified from each data source and how many were successfully analyzed for licensing and security.

Table 4.2: Open-Source Dataset Summary

Build System	Source	Valid Projects	Analyzed Projects
Maven	Libraries.io	7,768	392 (Licensing), 502 (Security)
Gradle	Software Heritage	7,885	115 (Licensing), 116 (Security)

4.3 Synthetic Testing for Licensing Analysis

Test Case	License Type	Parent ← Child	Is Invoked	Is Compatible
DDEP_d,d,i,compat	Permissive ← Permissive	MIT ← Apache-2.0	Yes	Yes
DDEP_d,d,ni,compat	Permissive ← Permissive	MIT ← Apache-2.0	No	Yes
DDEP_d,d,i,incompat	Permissive ← Strong Copyleft	Apache-2.0 ← GPL-3.0	Yes	No
DDEP_d,d,ni,incompat	Permissive ← Strong Copyleft	Apache-2.0 ← GPL-3.0	No	No
DDEP_d,d,i,multi,compat	Permissive (AND) Permissive ← Permissive	MIT AND BSD-3 ← Apache-2.0	Yes	Yes
DDEP_d,d,ni,multi,compat	Permissive (AND) Permissive ← Permissive	MIT AND BSD-3 ← Apache-2.0	No	Yes
DDEP_d,d,i,multi,incompat	Permissive (AND) Permissive ← Strong Copyleft	Apache-2.0 AND MIT← GPL-3.0	Yes	No
DDEP_d,d,ni,multi,incompat	Permissive (AND) Permissive ← Strong Copyleft	Apache-2.0 AND MIT← GPL-3.0	No	No
DDEP_d,d,t,i,compat	Permissive ← Permissive	MIT ← Apache-2.0	Yes	Yes
DDEP_d,d,t,ni,compat	Permissive ← Permissive	MIT ← Apache-2.0	No	Yes
TDEP_t1,d,i,compat	Weak Copyleft ← Permissive	LGPL-2.1 ← Apache-2.0	Yes	Yes
TDEP_t1,d,ni,compat	Weak Copyleft ← Permissive	LGPL-2.1 ← Apache-2.0	No	Yes
TDEP_t1,d,i,incompat	Permissive ← Strong Copyleft	MIT ← AGPL-3.0	Yes	No
TDEP_t1,d,ni,incompat	Permissive ← Strong Copyleft	MIT ← AGPL-3.0	No	No
TDEP_t1,d,i,multi,compat	Permissive (OR) Permissive ← Permissive	Apache-2.0 OR BSD-2 ← MIT	Yes	Yes
TDEP_t1,d,ni,multi,compat	Permissive (OR) Permissive ← Permissive	Apache-2.0 OR BSD-2 ← MIT	No	Yes
TDEP_t1,d,i,multi,incompat	Weak Copyleft ← Permissive (AND) Strong Copyleft	LGPL-2.1 ← Apache-2.0 AND GPL-3.0	Yes	No
TDEP_t1,d,ni,multi,incompat	Weak Copyleft ← Permissive (AND) Strong Copyleft	LGPL-2.1 ← Apache-2.0 AND GPL-3.0	No	No
TDEP_t2,d,i,compat	Strong Copyleft ← Permissive	GPL-3.0 ← MIT	Yes	Yes
TDEP_t2,d,ni,compat	Strong Copyleft ← Permissive	GPL-3.0 ← MIT	No	Yes
TDEP_t2,d,i,incompat	Permissive ← Weak Copyleft	Apache-2.0 ← LGPL-2.1	Yes	No
TDEP_t2,d,ni,incompat	Permissive ← Weak Copyleft	Apache-2.0 ← LGPL-2.1	No	No
TDEP_t2,d,i,multi,compat	Weak Copyleft (AND) Permissive ← Permissive	LGPL-2.1 AND Apache-2.0 ← MIT	Yes	Yes
TDEP_t2,d,ni,multi,compat	Weak Copyleft (AND) Permissive ← Permissive	LGPL-2.1 AND Apache-2.0 ← MIT	No	Yes
TDEP_t2,d,i,multi,incompat	Permissive (AND) Permissive ← Weak Copyleft (AND) Strong Copyleft	Apache-2.0 AND BSD-3 ← LGPL-2.1 AND GPL-3.0	Yes	No
TDEP_t2,d,ni,multi,incompat	Permissive (AND) Permissive ← Weak Copyleft (AND) Strong Copyleft	Apache-2.0 AND BSD-3 ← LGPL-2.1 AND GPL-3.0	No	No

Table 4.3: Synthetic Test Cases for Licensing Pruning in Single-Module Systems. License types specify whether they are Permissive, Weak Copyleft, or Strong Copyleft.

Key for Test Case Identifiers

$DDEP$ = Direct Dependency
$TDEP\_t1$ = Transitive Dependency at Level 1
$TDEP\_t2$ = Transitive Dependency at Level 2
d = Declared
t = Test Scope
i = Invoked
ni = Not Invoked
$compat$ = Compatible License
$incompat$ = Incompatible License
$multi$ = Multi-License Scenario (AND/OR)

Table 4.3 illustrates the coverage of various license compatibility scenarios, focusing on evaluating the pruning of identified license incompatibilities within the dependency tree. Each scenario represents a relationship between a parent node (e.g., the root project or a dependency) and its child node (e.g., a direct or transitive dependency). The table explores these relationships across different levels of the dependency tree:

1. Root Project ← Direct Dependency: The root project forms a parent-child relationship with its direct dependencies.
2. Direct Dependency ← Transitive Dependency: A direct dependency may introduce transitive dependencies, creating a parent-child relationship between them.
3. Transitive Dependency ← Transitive Dependency: Transitive dependencies can themselves depend on other transitive dependencies, forming additional parent-child relationships.

To evaluate the approach's performance, a synthetic dependency tree was constructed to represent these scenarios. Each dependency was assigned a specific license (e.g., GPL-3.0, Apache-2.0, MIT, LGPL-3.0) to simulate real-world license configurations. For each case, the approach's ability to accurately retain invoked license incompatibilities and prune non-invoked license incompatibilities was tested. The table above demonstrates:

- Invoked Scenarios: Dependencies with a direct method invocation in the source code are retained, regardless of license incompatibilities, as they are essential to the project.
- Non-Invoked Scenarios: Dependencies that are not invoked are pruned, as they are not essential to the project.
- Test-Scope Scenarios: Dependencies that are of testing scope are pruned regardless of invocation as they are not essential to the project.

To simulate these scenarios, licenses were modified at different points in the synthetic dependency tree, reflecting real-world OSS license structures.

- For the root project, the LICENSE.txt file was directly edited.
- For direct and transitive dependencies, the approach's caching mechanism, which stores downloaded dependencies for AST analysis, was leveraged. When dependencies were cached, an injected license was simulated by modifying the existing license file or adding a new one, depending on the license compatibility scenario being tested.

To thoroughly evaluate the approach's pruning capability, which is designed to assist developers in identifying and removing unnecessary dependencies for software building and redistribution, multiple runs were conducted with varying configurations. In each run, a different node was designated as the primary target for retention or pruning, ensuring coverage of all dependency types, including direct, transitive, multi-license, and test scope dependencies.

Table 4.4: Synthetic Dependencies for Pruning

Group ID	Artifact ID	Version	Notes
com.fasterxml.jackson.core	jackson-databind	2.13.0	Similarly named with jackson-core
com.fasterxml.jackson.core	jackson-core	2.13.0	Similarly named with jackson-databind
org.apache.commons	commons-collections4	4.4	Similarly named with commons-lang3
org.apache.commons	commons-lang3	3.12.0	Similarly named with commons-collections4
org.slf4j	slf4j-api	1.7.30	Popular library for logging
org.apache.httpcomponents	httpclient	4.5.13	HTTP client library, Common for REST clients
org.springframework	spring-core	5.3.8	Popular library for application development
org.apache.httpcomponents	httpclient	4.5.13	Popular HTTP client library
com.google.guava	guava	31.0.1-jre	Popular utility library
org.eclipse.jetty	jetty-server	9.4.43.v20210629	Popular library for server applications

For each run, a different node was designated as the primary target for retention or pruning, as shown in Table 4.4, ensuring comprehensive coverage across direct dependencies, transitive dependencies at different levels, multi-license dependencies, and test scope dependencies. Each test was designed to focus on a single node for pruning or retention, with a distinct dependency declared and either invoked or not invoked in the source code. This approach enables a structured evaluation of false positives (pruning of invoked, non test-scope dependencies), false negatives (retention of uninvoked or test-scope dependencies), true positives (pruning of uninvoked dependencies or test-scope dependencies), and true negatives (retention of invoked, non-test scope dependencies).

To evaluate the approach's ability to handle false positives in pruning, particularly those caused by name similarity due to its reliance on mapping declared package names to bindings identified through AST analysis, test cases were structured to reflect real-world dependency configurations. Two sets of similarly named dependencies (jackson-databind vs. jackson-core and commons-collections4 vs. commons-lang3) were incorporated into the pool from which test cases were derived to assess the approach's ability to differentiate dependencies during pruning.

To ensure the test cases align with common dependency structures, libraries were selected based on their widespread use in Java projects. Dependencies with multi-license expressions were in-

cluded to examine the approach's handling of licensing complexity, while test scope dependencies were incorporated to confirm that they are consistently pruned regardless of invocation.

This approach establishes a structured evaluation framework that systematically assesses the approach's pruning mechanism under multiple conditions. The exact specifications of each cases setup can be found in Table A.1 and Table A.2. The results for both Maven and Gradle are presented and discussed in Chapter 5.

4.4 Synthetic Testing for Security Analysis

Dependency ID	Is Invoked	Has CVE(s)
$DDEP\_d, d, i, vuln$	Yes	Yes
$DDEP\_d, d, ni, vuln$	No	Yes
$DDEP\_d, d, i, noVuln$	Yes	No
$DDEP\_d, d, ni, noVuln$	No	No
$DDEP\_d, d, t, i, vuln$	Yes	Yes
$DDEP\_d, d, t, ni, vuln$	No	Yes
$TDEP1\_d, d, i, vuln$	Yes	Yes
$TDEP1\_d, d, ni, vuln$	No	Yes
$TDEP1\_d, d, i, noVuln$	Yes	No
$TDEP1\_d, d, ni, noVuln$	No	No
$TDEP2\_d, d, i, vuln$	Yes	Yes
$TDEP2\_d, d, ni, vuln$	No	Yes
$TDEP2\_d, d, ni, noVuln$	Yes	No
$TDEP2\_d, d, ni, noVuln$	No	No

Table 4.5: Synthetic Test Cases for Security Pruning in a Single-Module System. Has CVE(s) indicates if one or more CVEs of any severity are associated with the target dependency.

Key for Test Case Identifiers

$DDEP$ = Direct Dependency
$TDEP\_t1$ = Transitive Dependency at Level 1
$TDEP\_t2$ = Transitive Dependency at Level 2
d = Declared
t = Test Scope
i = Invoked
ni = Not Invoked
$vuln$ = Has at Least 1 CVE
$noVuln$ = Has No CVEs

Table 4.5 illustrates the coverage of various security vulnerability scenarios, focusing on evaluating the pruning of identified security vulnerabilities within the dependency tree. Each scenario

represents a different level of the dependency tree (i.e. direct or transitive), a scope (i.e. compile or test), and an invocation status (i.e. invoked or not invoked).

Table 4.6: Common Java Libraries With Known Direct CVEs

Group ID	Artifact ID	Version	Known Direct CVEs
org.apache.struts	struts2-core	6.3.0.1	CVE-2023-50164, CVE-2024-53677
org.apache.commons	commons-compress	1.25.0	CVE-2024-26308, CVE-2024-25710
org.apache.commons	commons-io	1.3.2	CVE-2021-29425
org.apache.commons	commons-collections4	4.0	CVE-2015-7501, CVE-2015-6420
org.apache.logging.log4j	log4j-core	2.17.0	CVE-2021-44832
org.springframework	spring-core	6.0.15	CVE-2024-22233
org.apache.httpcomponents	httpclient	4.5.12	CVE-2020-13956
com.google.guava	guava	31.1-jre	CVE-2023-2976, CVE-2020-8908
org.eclipse.jetty	jetty-server	12.0.8	CVE-2024-8184
org.dom4j	dom4j	2.1.1	CVE-2020-10683

Table 4.7: Common Java Libraries With No Known CVEs (Direct or Transitive)

Group ID	Artifact ID	Version
org.apache.commons	commons-lang3	3.12.0
org.apache.commons	commons-collections4	4.5.0-M3
org.slf4j	slf4j-api	2.0.17
org.mockito	mockito-core	5.15.2
junit	junit	4.13.2
org.apache.commons	commons-text	1.13.0
org.springframework	spring-beans	6.1.17
org.hibernate.orm	hibernate-core	6.6.9.Final
org.apache.logging.log4j	log4j-api	2.24.3

Note: The above CVE and non-CVE dependencies in Table 4.6 and Table 4.7 were last verified as of 02/2025, it is important to note if replicated that this data is subject to change over time.

In the security synthetic dataset, modifying detected CVEs was not feasible, unlike in the licensing cases where cached license files could be altered to simulate different license expressions. CVE data is inherently tied to the published version of a dependency and cannot be manipulated through caching or file modifications. To ensure accurate evaluation, focus nodes were selected from predefined pools of dependencies. Libraries confirmed to have CVEs were used in cases requiring vulnerable dependencies, while libraries verified to have no reported CVEs were used in cases requiring non-vulnerable dependencies. Validation for both categories was conducted using https://mvnrepository.com/, where each dependency's artifact ID, group ID, and version were cross-referenced against known vulnerability records. This ensured that the synthetic dataset accurately represented real-world security conditions while enabling a controlled evaluation of the pruning and security analysis logic of the approach.

The exact specifications of each cases setup can be found in Table A.3 and Table A.4. The results for both Maven and Gradle are presented and discussed in Chapter 5.

4.4.1 Synthetic Data Metrics

In order to answer the research questions on the synthetic datasets, we evaluate the precision, recall, and F1-score. Precision shows the proportion of true positives among the total predicted positives. This metric is calculated with the following formula where a higher precision value indicates lower amounts of false positives:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{4.1}$$

Recall shows the proportion of true positives among anything that should have been predicted as positive. This metric is calculated with the following formula:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{4.2}$$

The F1-score shows the harmonic mean of precision and recall. This metric can be calculated with the following formula:

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.3}$$

An F1-score close to 1 means the software system's analysis is highly accurate, successfully balancing high precision and high recall.

The synthetic datasets allow us to accurately calculate recall because we have a known ground truth that identifies which software components contain licensing incompatibilities or security vulnerabilities, their locations in the dependency tree, and whether they are invoked in the codebase. This enables us to assess how well the approach correctly prunes or retains dependencies, a level of certainty that is not feasible with our open-source project dataset.

To evaluate the precision and recall of the proposed approach, we utilize synthetic data in which security vulnerabilities and licensing incompatibilities are systematically injected, with explicit distinctions between invoked and non-invoked dependencies. By analyzing this structured dataset, we can quantitatively assess the approach's accuracy in identifying and handling these injected issues, specifically measuring its ability to correctly prune or retain dependencies based on their invocation status. By evaluating the recall rate of the proposed approach we can ensure that it is pruning all or most of the dependencies within the dependency tree of a given software system that are not invoked or are of testing scope.

4.4.2 Open-source Data Metrics

To evaluate the effectiveness of our approach on open-source projects, we measure its precision. As outlined in the previous section, precision quantifies the proportion of true positives among all predicted positives and is calculated using the formula:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{4.4}$$

To determine the precision score for the approach when applied to our open-source dataset, we first need to establish the expected correct outcomes of what should be pruned versus what should be retained. We then monitor how often the approach correctly prunes a dependency that is uninvoked or of testing scope (true positives) and how often it produces incorrect results, pruning a dependency that is invoked, (false positives) for each software system analyzed.

To calculate precision we need to randomly sample and manually analyze a statistically significant number of projects from each build system. We narrowed down our sample by only considering projects where both security and licensing analysis successfully ran.

A higher precision score indicates that the approach is more effective at correctly pruning dependencies that are uninvoked within the dependency tree of a software system, minimizing false positives. This ensures the approach provides more reliable and actionable insights for software security and licensing compliance.

From Libraries.io open data, we obtained 7,768 Java Maven projects that were determined to be valid candidate projects for our approach.

Maven Open-source Data Licensing Analysis

Within 392 successfully analyzed projects for licensing, 4,342 dependencies were analyzed in total, 1,840 of which were direct dependencies and 2,502 of which were transitive dependencies. The pruned dependency trees contained a total of 1,671 dependencies, 763 of which were direct dependencies and 908 of which were transitive dependencies.

We can see the effect of the pruning algorithm shown in Figure 4.1.

For licensing analysis, we see that around 0.94% of all analyzed dependencies had source code download failures (41 dependencies were identified to have source code download failures where the approach was unable to resolve and download the source code for use in AST analysis) this is important as our approach pruned the children of any dependencies where there was a source code failure which could lead to potential false positives.

We found that 1,833 of all dependencies analyzed failed to have their license identified, triggering a flag in the report to indicate its potential impact on the security analysis. This could result from ScanCode-Toolkit being unable to detect a license across all identification methods used or from the license not being distributed with the software system. For dependencies where we were not able to identify the project-level license, we were unable to determine if that dependency had an incompatibility with its parent. Consequently, the 247 license incompatibilities identified in the full Maven dependency trees should be considered a lower bound, as additional conflicts may exist among the dependencies where licenses were not resolved.

When the licensing analysis was applied, we found that 247 license incompatibilities were detected in the full dependency trees and 33 license incompatibilities were detected in the pruned dependency trees. This reduction represents a substantial improvement in license compatibility across the analyzed projects. The pruning mechanism effectively removed dependencies whose license terms conflicted with those of their parent modules, eliminating 214 of the 247 flagged incompatibilities.

Figure 4.3 shows the distribution of license incompatibilities identified per project that successfully ran Maven licensing analysis. The results in this plot suggest that pruning removed depen-

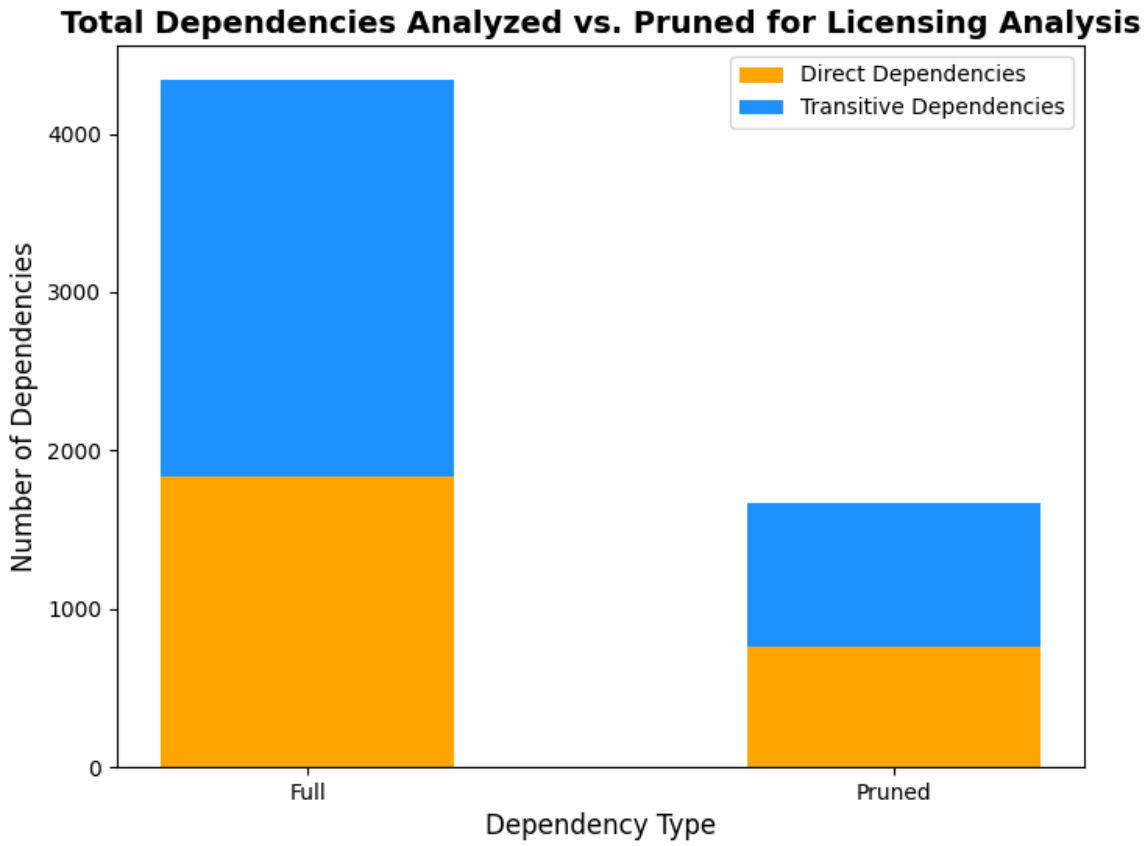

Figure 4.1: Comparison of Full vs. Pruned Dependency Trees

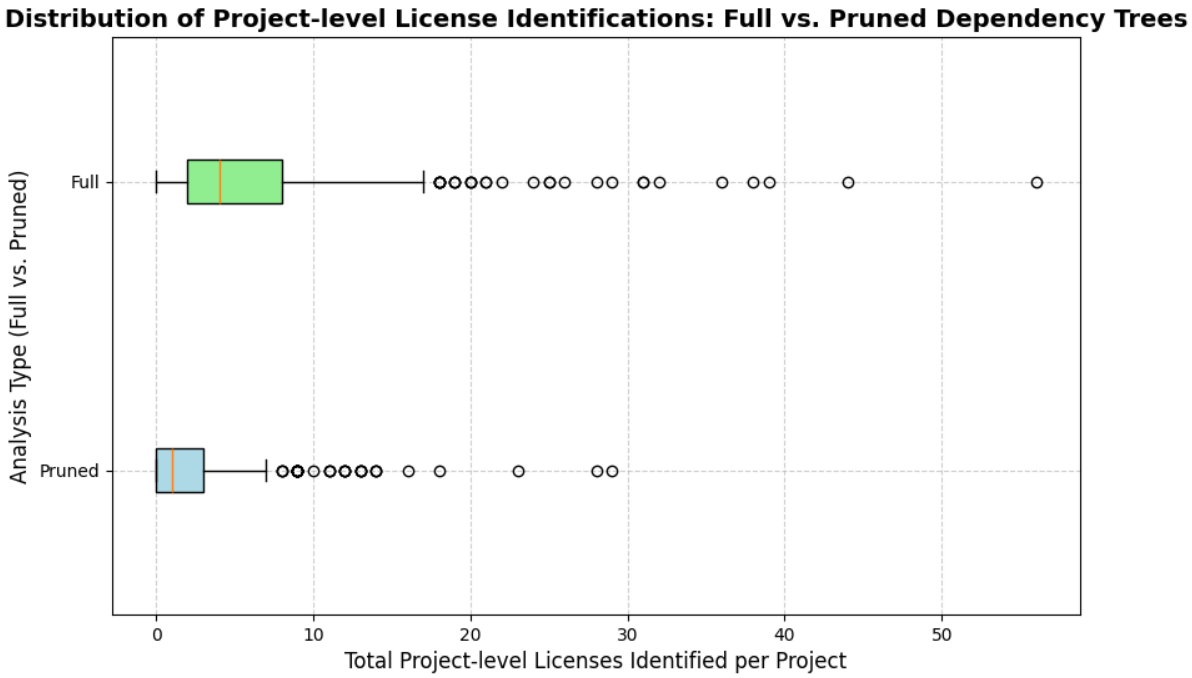

Figure 4.2: Comparison of Full vs. Pruned Dependency Trees: Identified Project-level License Distributions

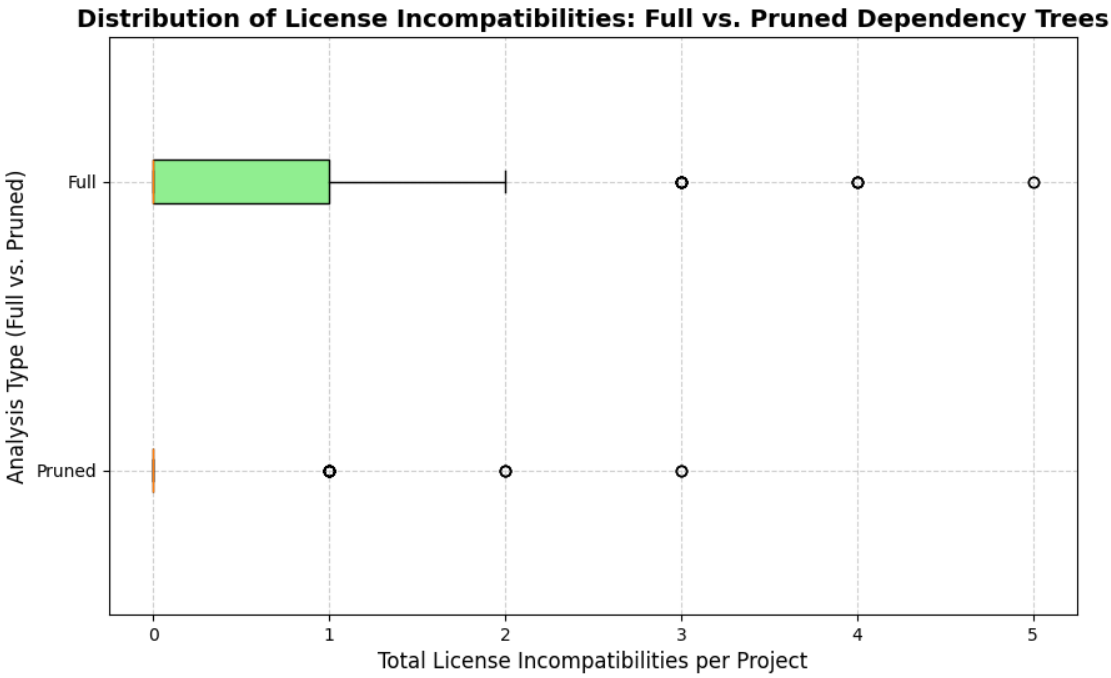

Figure 4.3: Comparison of Full vs. Pruned Dependency Trees: Identified License Incompatibility Distributions

dencies contributing to license conflicts.

Maven Open-source Data Security Analysis

Within these 502 successfully analyzed projects, 11,572 dependencies were analyzed in total, 3,083 of which were direct dependencies and 8,489 of which were transitive dependencies. The pruned dependency trees contained a total of 3,816 dependencies, 1,238 of which were direct dependencies and 2,578 of which were transitive dependencies.

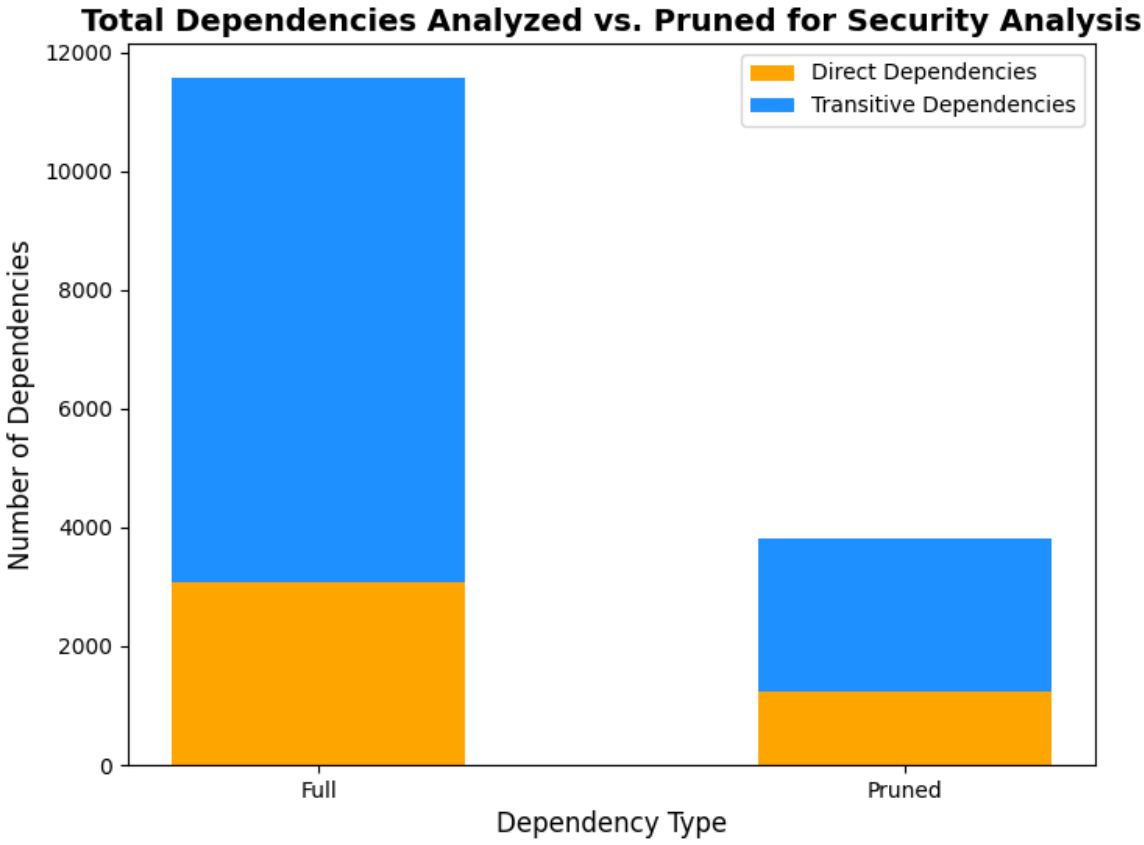

Figure 4.4: Comparison of Full vs. Pruned Dependency Trees

We can see the effect of the pruning algorithm shown in Figure 4.4.

When the security analysis was applied, we found 10,241 total CVEs detected. These CVEs correlated to 1,434 vulnerable dependencies, where a vulnerable dependency is said to be any dependency with at least one CVE. In the pruned dependency trees, we saw 4,026 CVEs retained which were correlated to 618 vulnerable dependencies. This means that after pruning, the number of vulnerable dependencies was reduced by approximately 56.92%, demonstrating the impact of pruning in lowering the presence of dependencies with known security vulnerabilities. This reduc-

tion is particularly significant given the original scale: among the 11,572 analyzed dependencies, 1,434 (12.4%) were associated with at least one CVE. By eliminating over half of these vulnerable dependencies, the pruning approach substantially reduces the potential attack surface of the software system. Notably, this reduction includes CVEs across all severity levels—critical, high, medium, and low.

For security analysis, we see that around 3.49% of all analyzed dependencies had source code download failures (404 dependencies were identified to have source code download failures where the approach was unable to resolve and download the source code for use in AST analysis).

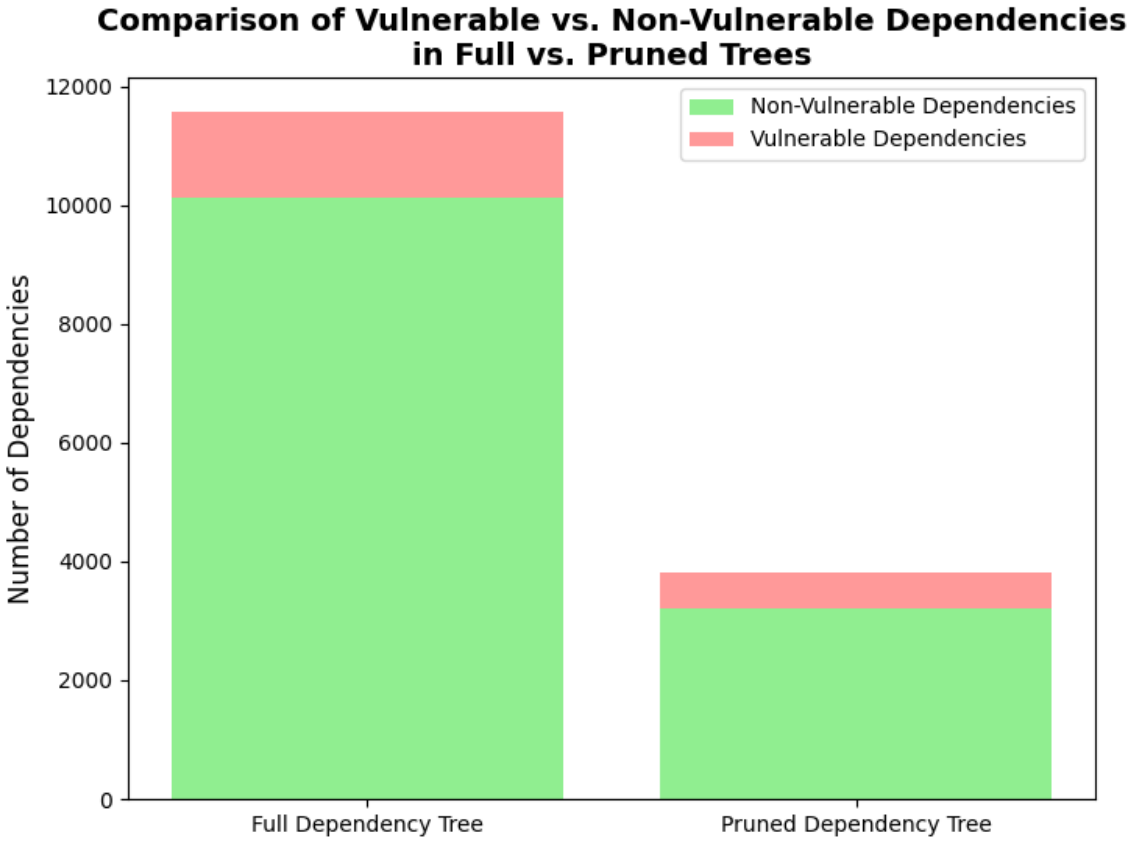

Figure 4.5: Comparison of Vulnerable Dependencies in Full vs. Pruned Dependency Trees

Of these CVEs, we found 1,451 were of critical severity, 3,270 were of high severity, 5,166 were of medium severity, and 354 were of low severity. In the pruned dependency trees we retained 722 of critical severity, 1,563 of high severity, 1,670 of medium severity, and 71 of low severity.

The distribution plot, Figure 4.15, further illustrates the effect of pruning on the number of vulnerable dependencies per project. The box plots for the full and pruned dependency trees show a clear reduction in the number of vulnerabilities retained after pruning. The full dependency tree exhibits a wider spread, with a higher median number of vulnerabilities. The pruned dependency

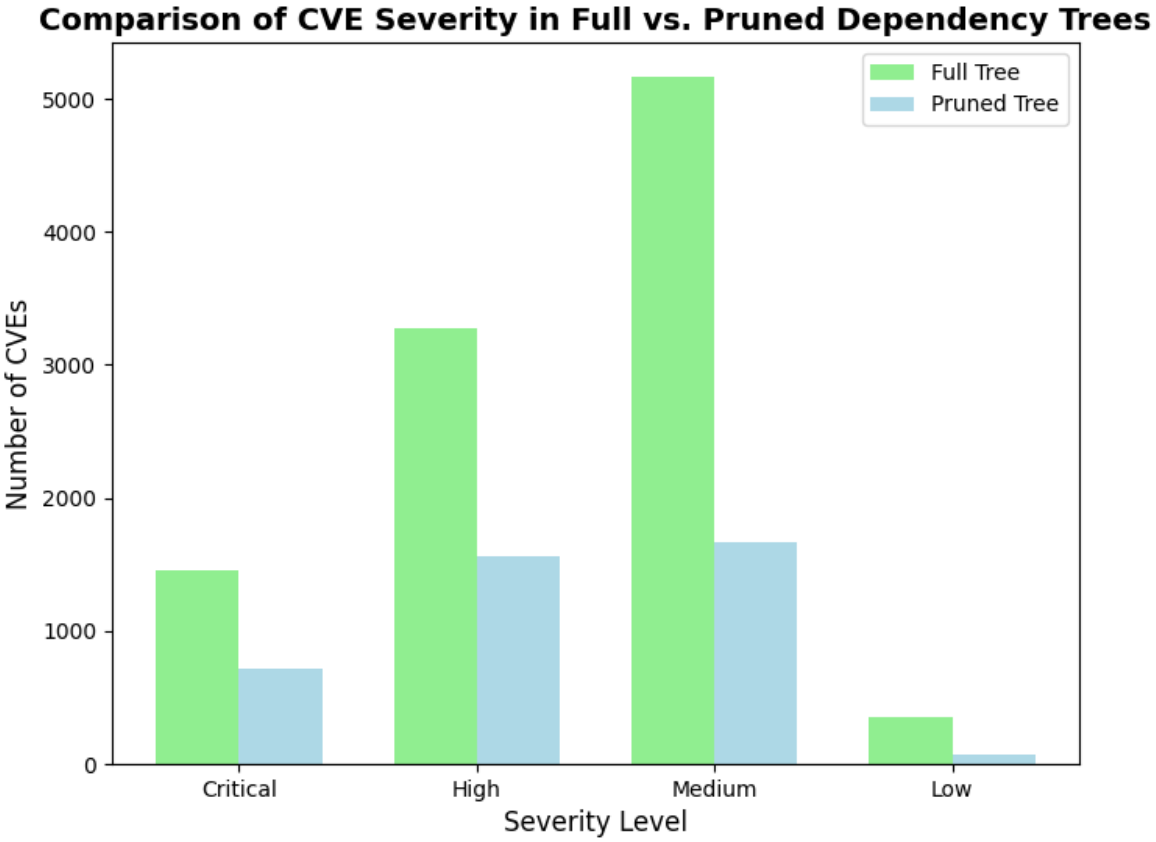

Figure 4.6: Comparison of CVE severities in Full vs. Pruned Dependency Trees

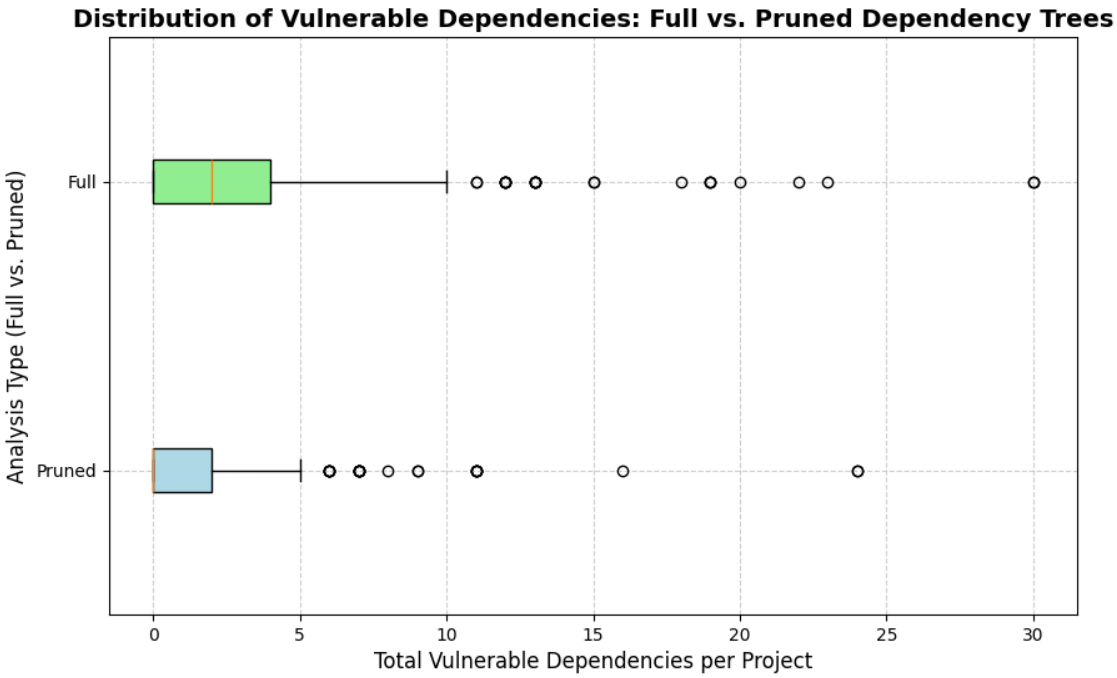

Figure 4.7: Distribution of Vulnerable Dependencies in Full vs. Pruned Dependency Trees

tree's median is visibly lower, and the overall spread of vulnerabilities is also reduced, suggesting that pruning eliminates many dependencies that introduce security risks.

From the Software Heritage Archive, we obtained 7,885 Gradle Java projects that were determined to be valid candidate projects for our approach.

Gradle Open-source Data Licensing Analysis

Within 115 successfully analyzed projects for licensing, 816 dependencies were analyzed in total, 337 of which were direct dependencies and 479 of which were transitive dependencies. The pruned dependency trees contained a total of 284 dependencies, 108 of which were direct dependencies and 176 of which were transitive dependencies.

We can see the effect of the pruning algorithm shown in 4.8.

For licensing analysis, we see that around 8.8% of all analyzed dependencies had source code download failures (72 dependencies were identified to have source code download failures where the approach was unable to resolve and download the source code for use in AST analysis). This is a percentage 7.8% higher than what we observed for Maven licensing analysis and is further discussed in Section 5.1.2.

When the licensing analysis was applied, we found that 18 license incompatibilities were detected in the full dependency trees and 1 license incompatibility was detected in the pruned dependency trees, representing a 94.4% reduction. This significant decrease indicates that most license

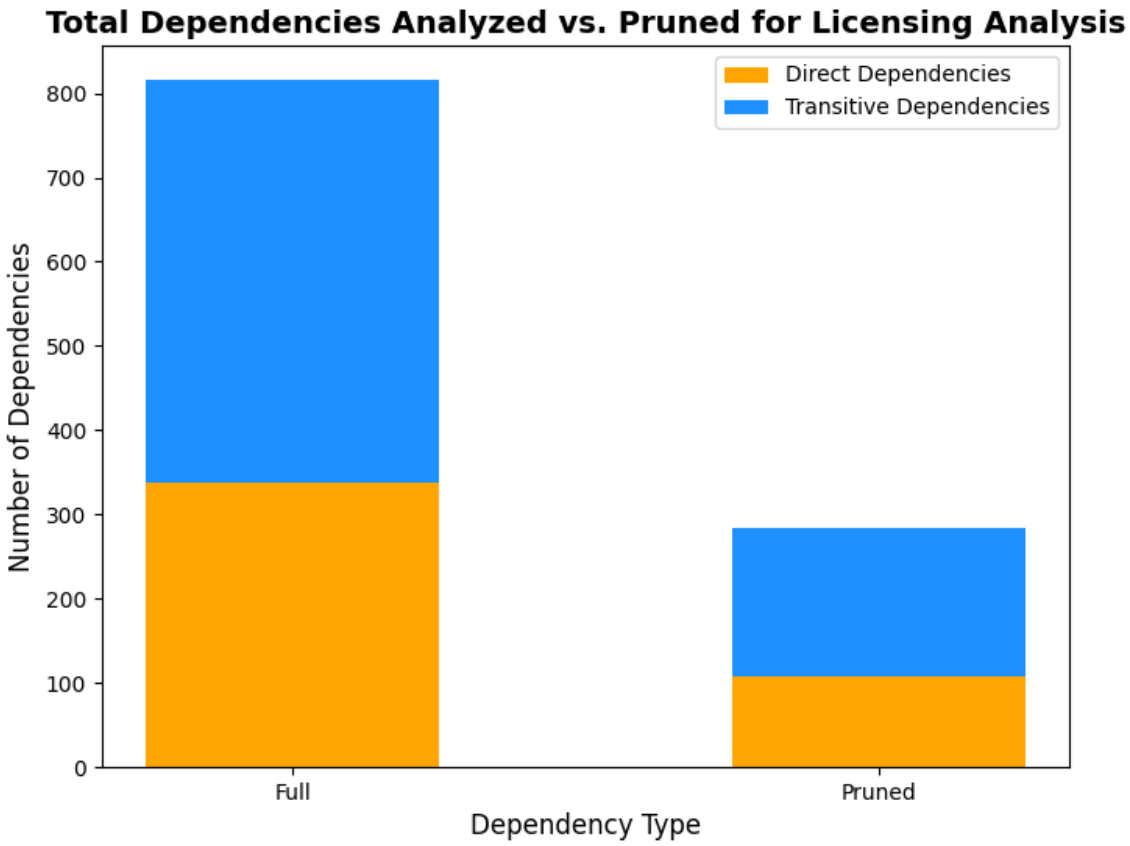

Figure 4.8: Comparison of Full vs. Pruned Dependency Trees

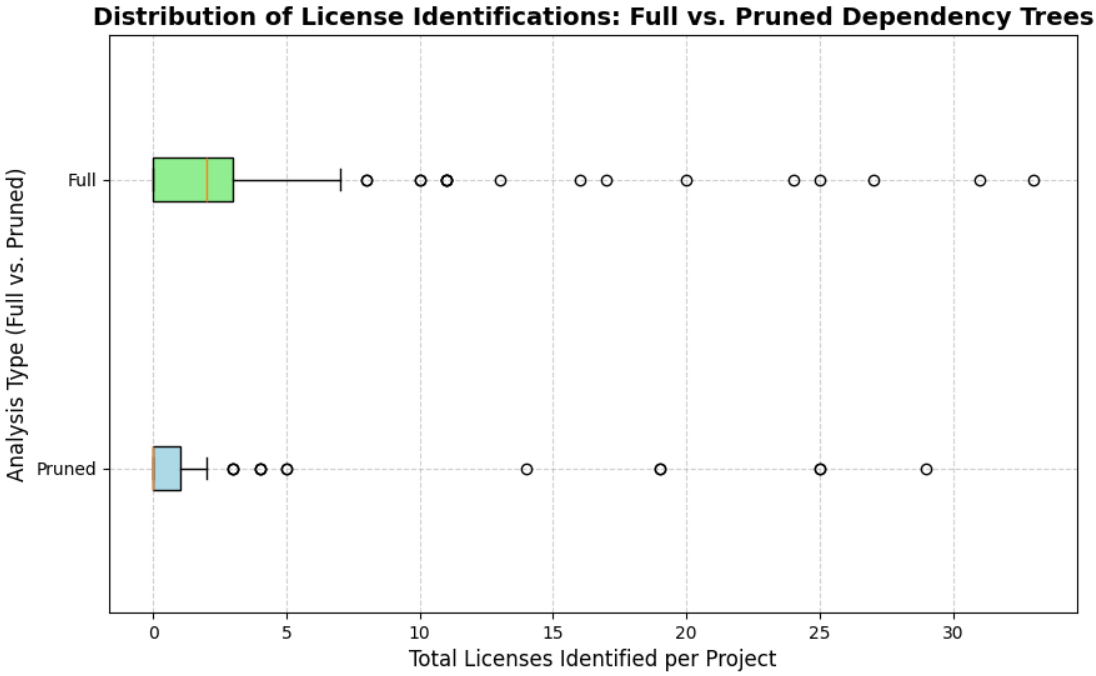

Figure 4.9: Comparison of Full vs. Pruned Dependency Trees: Identified Project-level License Distributions

conflicts in the analyzed Gradle projects were introduced by dependencies that were either uninvoked or used solely in a testing scope. By removing these non-essential dependencies, the pruning process effectively mitigated legal risk without compromising project functionality.

We also found that 46.3% of all dependencies analyzed failed to have their license identified, triggering a flag in the report to indicate its potential impact on the security analysis. In these cases, the approach is unable to determine the license compatibility between parent and child dependencies, as at least one of the required licenses for comparison is missing. This limitation prevents the detection of potential license conflicts and may result in underreporting of incompatibilities. Consequently, the 18 license incompatibilities identified in the full Gradle dependency trees should be considered a lower bound, as additional conflicts may exist among the dependencies where licenses were not resolved.

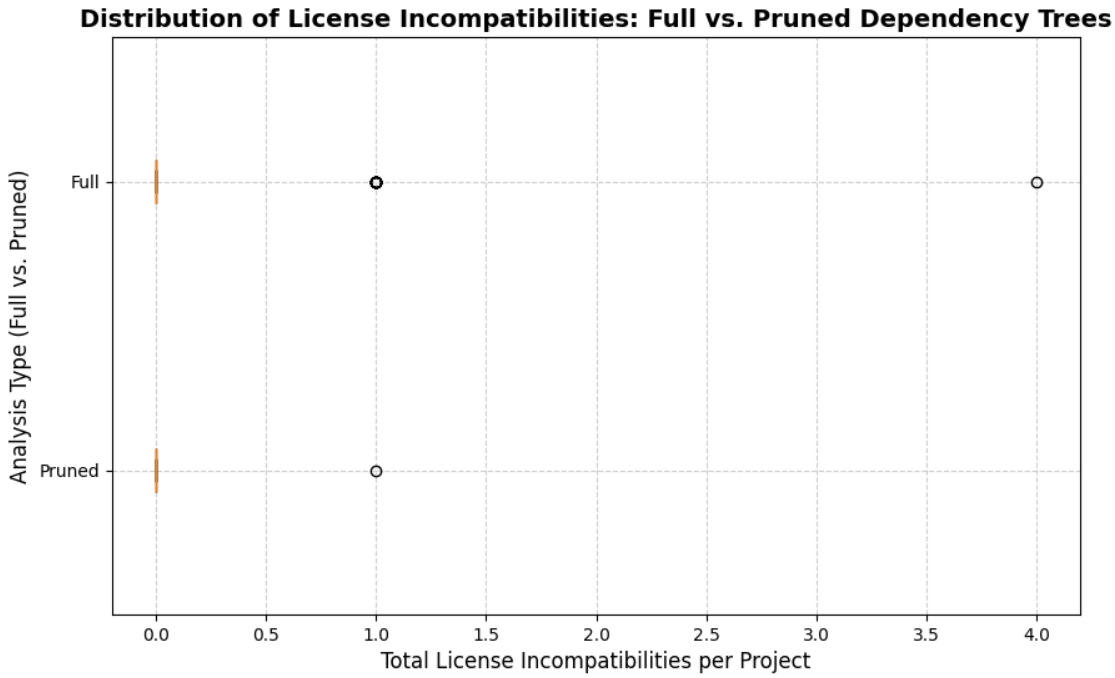

Figure 4.10: Comparison of Full vs. Pruned Dependency Trees: Identified License Incompatibility Distributions

Figure 4.10 illustrates the distribution of license incompatibilities detected in full and pruned dependency trees. The results indicate that license incompatibilities were identified only in the full dependency trees, with up to four per project, while pruned trees exhibited a max of one.

Gradle Open-source Data Security Analysis

Within these 116 successfully analyzed projects, 1,629 dependencies were analyzed in total, 277 of which were direct dependencies and 1,352 of which were transitive dependencies. The pruned

dependency trees contained a total of 119 dependencies, 42 of which were direct dependencies and 77 of which were transitive dependencies.

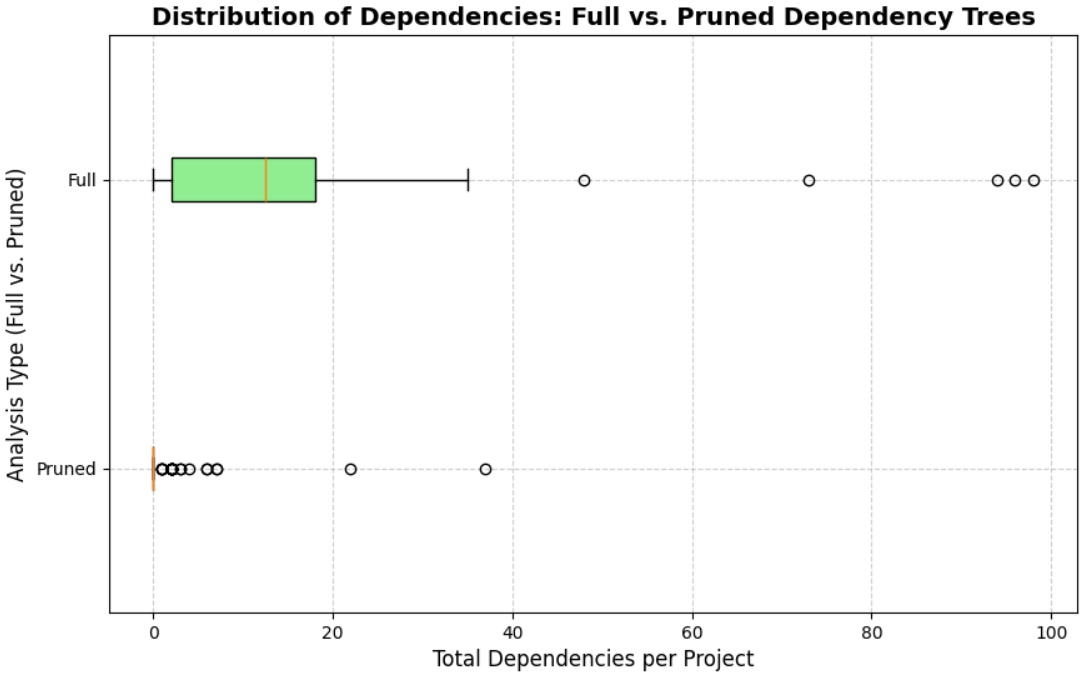

Figure 4.11: Gradle Dependency Distribution per Project Full vs Pruned

Upon further analysis of the distribution of total dependencies per project, we manually validated a subset of outliers reporting close to 100 dependencies. These counts were confirmed to be accurate based on the output from the 'gradle :dependencies' command. These outliers of dependencies per project appear in the Gradle security analysis but are less common in the licensing analysis. This is due to the higher computational cost of licensing analysis, which includes both license identification and compatibility checks. A one-hour time limit was applied during open-source data analysis. Runtimes exceeding this threshold were terminated to prevent indefinite execution.

These outliers can be seen in the distribution of dependencies per project shown in Figure 4.11. Such outliers inflate certain metrics and may not represent typical dependency tree structures. These projects were retained in the following statistics.

We can see the effect of the pruning algorithm shown in Figure 4.12.

When the security analysis was applied, we found 616 CVEs detected. These CVEs correlated to 200 vulnerable dependencies, where a vulnerable dependency is said to be any dependency with at least one CVE. In the pruned dependency trees, we saw 128 CVEs retained which were correlated to 18 vulnerable dependencies. This means that after pruning, the number of vulnerable dependencies was reduced by approximately 91%, demonstrating the impact of pruning in lowering the presence of dependencies with known security vulnerabilities. This highlights the impact of the proposed pruning approach in minimizing exposure to known security vulnerabilities. Notably, this

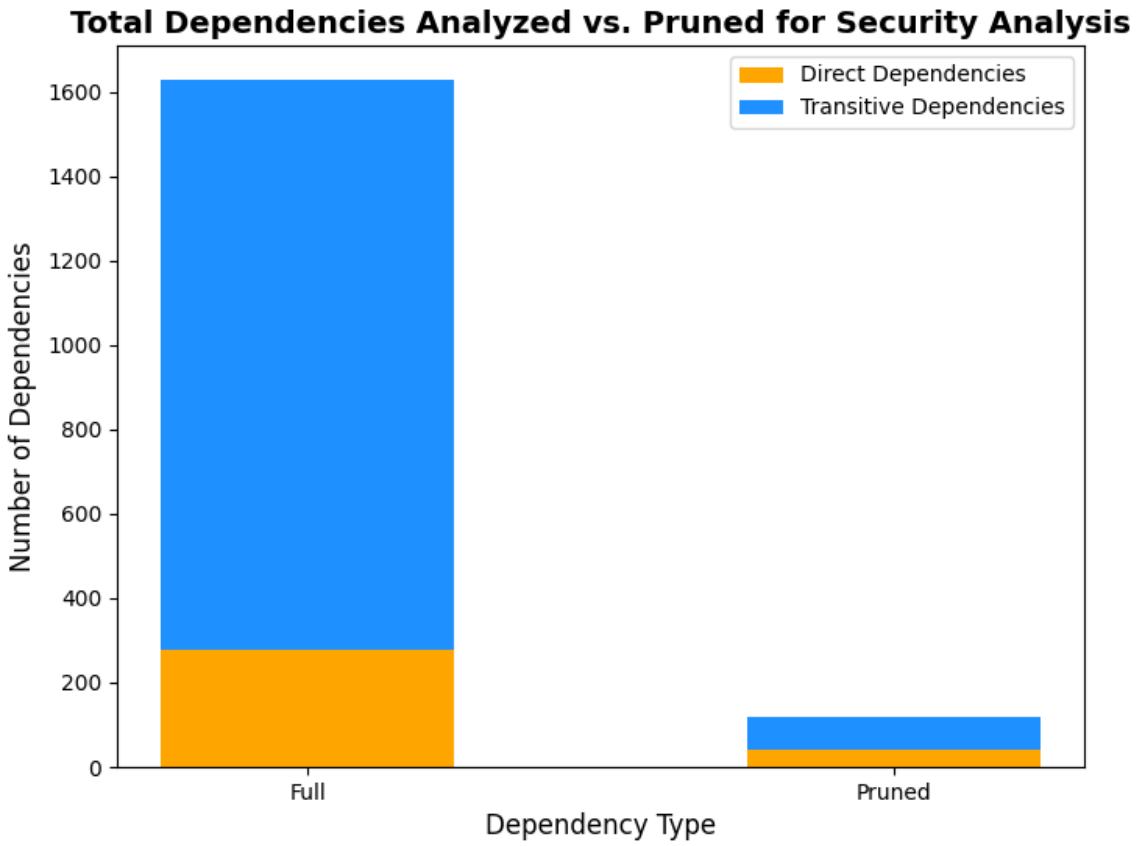

Figure 4.12: Comparison of Full vs. Pruned Dependency Trees

reduction was observed across dependencies with critical, high, medium, and low severity CVEs, indicating that pruning was effective at eliminating both severe and moderate risks.

For security analysis, we see that around 14.5% of all analyzed dependencies had source code download failures (237 dependencies were identified to have source code download failures where the approach was unable to resolve and download the source code for use in AST analysis).

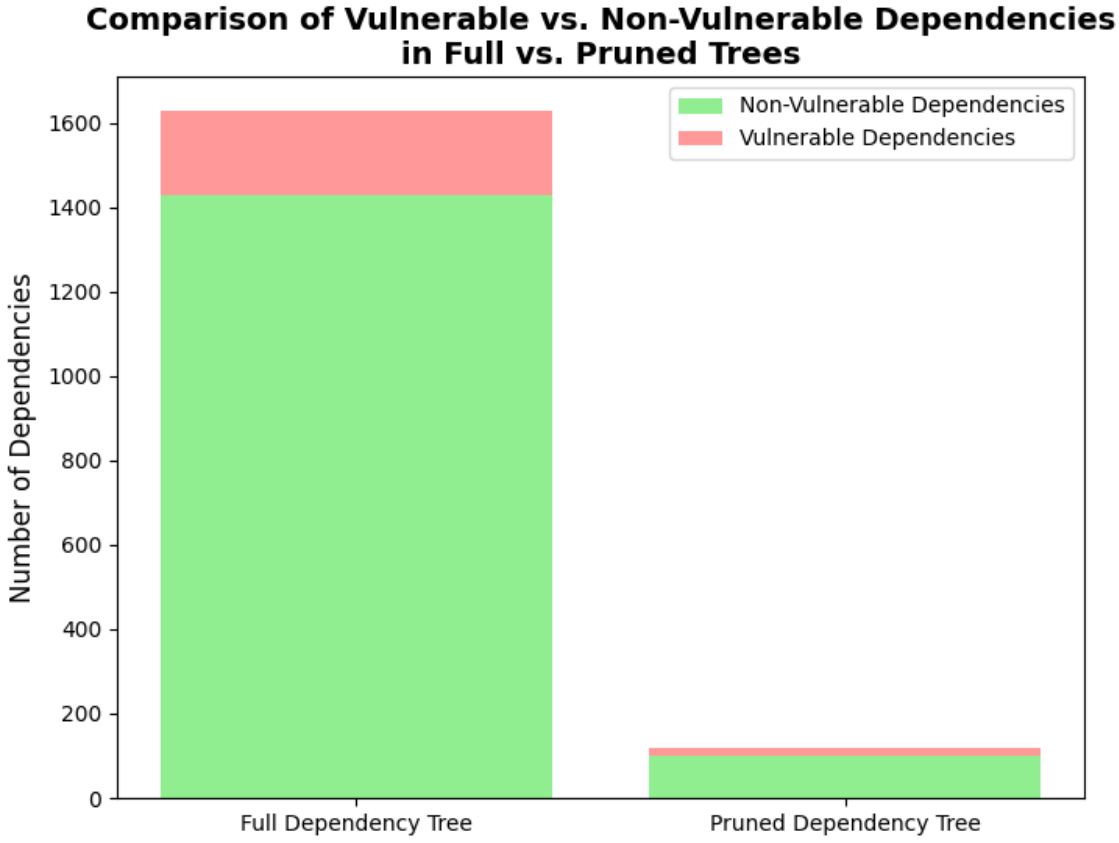

Figure 4.13: Comparison of Vulnerable Dependencies in Full vs. Pruned Dependency Trees

Of these CVEs we found 39 were of critical severity, 216 were of high severity, 320 were of medium severity, and 41 were of low severity. In the pruned dependency trees we retained 11 of critical severity, 29 of high severity, 87 of medium severity, and 1 of low severity.

The distribution plot, Figure 4.15, further illustrates the effect of pruning on the number of vulnerable dependencies per project. The box plots comparing the full and pruned dependency trees illustrate a substantial reduction in the number of vulnerable dependencies per project after pruning. The full dependency tree shows a higher median and a wider distribution, with several outliers exceeding 20 and even 30 vulnerable dependencies per project. In contrast, the pruned dependency tree exhibits a significantly lower median and a much narrower spread, with most projects having fewer than 5 vulnerable dependencies. The reduction in both central tendency and

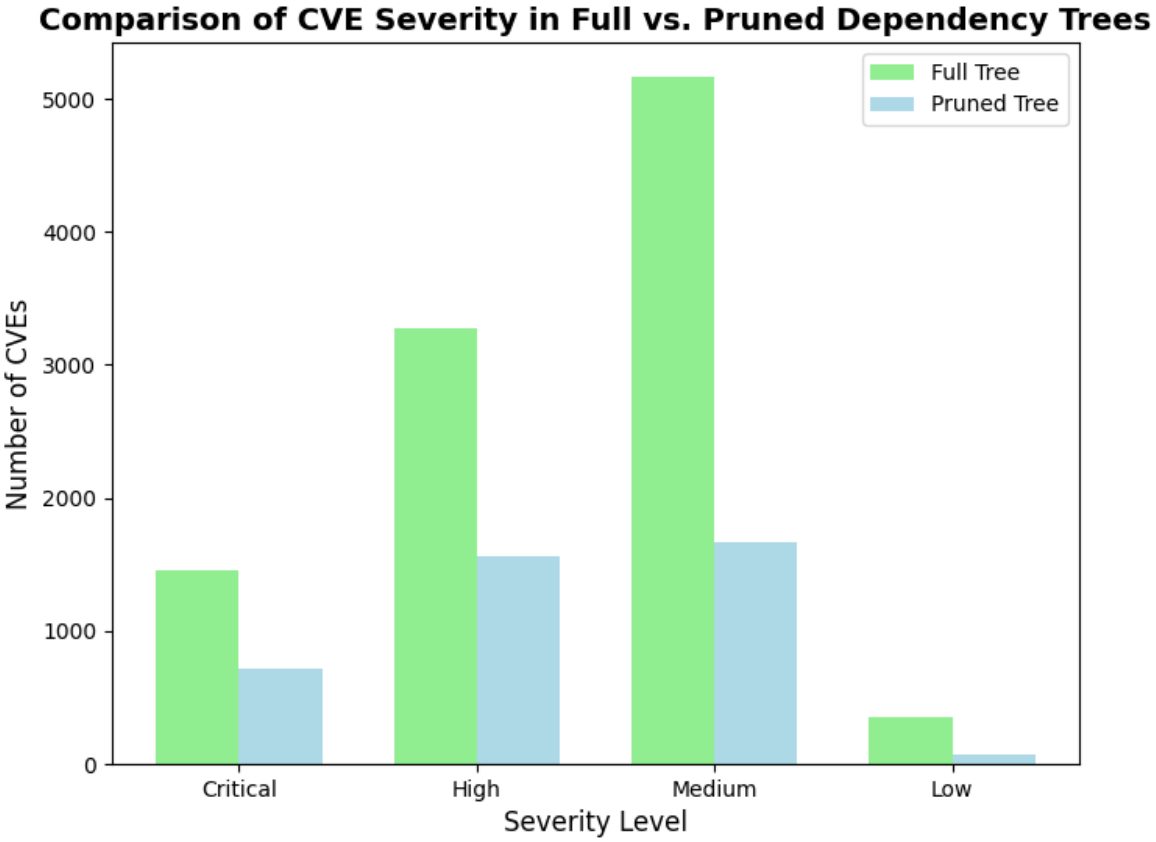

Figure 4.14: Comparison of CVE severities in Full vs. Pruned Dependency Trees

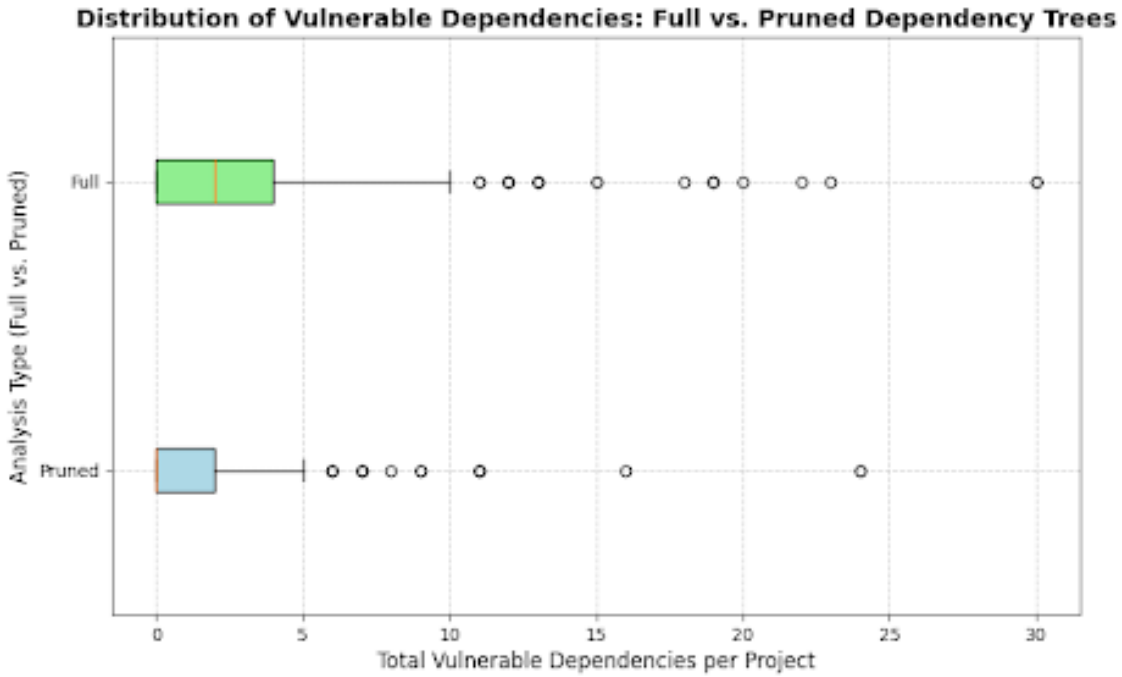

Figure 4.15: Distribution of Vulnerable Dependencies in Full vs. Pruned Dependency Trees

variability indicates that pruning effectively eliminates non-essential dependencies that contribute to known security vulnerabilities.

Table 4.8: Summary of Dependency Pruning Impact for Maven and Gradle

Metric	Maven Licensing	Gradle Licensing	Maven Security	Gradle Security
Total Projects Analyzed	392	115	502	116
Total Dependencies (Full Trees)	4,342	816	11,572	1,629
Total Dependencies (Pruned Trees)	1,671	284	3,816	119
% Reduction in Dependencies	61.5%	65.2%	67.0%	91.5%
License Incompatibilities (Full Trees)	247	18	–	–
License Incompatibilities (Pruned Trees)	33	1	–	–
% Reduction in License Incompatibilities	86.6%	94.4%	–	–
Vulnerable Dependencies (Full Trees)	–	–	1,434	200
Vulnerable Dependencies (Pruned Trees)	–	–	618	18
% Reduction in Vulnerable Dependencies	–	–	56.9%	91.0%
Total CVEs (Full Trees)	–	–	10,241	616
Total CVEs (Pruned Trees)	–	–	4,026	128
% Reduction in CVEs	–	–	60.7%	79.2%

Table 4.8 summarizes the impact of the proposed pruning approach across Maven and Gradle systems. The results demonstrate that the approach substantially reduces the total number of dependencies, security vulnerabilities, and license incompatibilities. For Maven licensing analysis, 86.6% of license incompatibilities were removed after pruning. In Gradle projects, 94.4% of license

incompatibilities were removed after pruning. For security analysis, Maven projects saw a 56.9% reduction in vulnerable dependencies and a 60.7% reduction in total CVEs. Gradle projects showed even greater improvements, with a 91.0% reduction in vulnerable dependencies and a 79.2% reduction in total CVEs. These results confirm the effectiveness of the pruning strategy in improving the security and licensing compliance of software systems.

Chapter 5

Validation

In this chapter, we address our research questions described in Chapter 4. First, we present the results of RQ1 for our approach in order to determine how well and how accurate the approach can prune uninvoked or test-scope dependencies. For RQ2, we determine the impact this pruning mechanism has on the licensing compatibility and security of the overall software system using both the synthetic dataset and the open-source dataset. For the open-source dataset, we extracted a representative sample of 30 analyzed Java projects where both security and licensing analysis ran successfully and manually validated them. Finally, we discuss the implications of the results.

5.1 RQ1: How effective is the proposed approach in accurately pruning uninvoked and test-scope dependencies from software dependency trees?

To evaluate the effectiveness of the proposed approach in accurately pruning uninvoked and test-scope dependencies from software dependency trees, we conducted a series of controlled experiments on synthetic data and performed a manual analysis on a sample of real-world software projects. This dual evaluation strategy allows for both comprehensive validation under known conditions and assessment of performance in realistic settings.

5.1.1 Synthetic Dataset

Table 5.1: Synthetic Data Pruning Metrics

Build System	Total TP	Total TN	Total FP	Total FN	Precision	Recall	F1
Maven	22	18	0	0	1.0	1.0	1.0
Gradle	22	18	0	0	1.0	1.0	1.0

Table 5.1 shows the number of true positives (TP), true negatives (TN), false positives (FP), false negatives (FN), precision, recall, and F1-score for the two build systems, Maven and Gradle.

For the synthetic projects using Maven, the proposed pruning process resulted in a precision of 1.0, a recall of 1.0, and an F1 score of 1.0. These results were obtained from a dataset of 40 total test cases, each designed to assess different aspects of dependency pruning. The cases varied in dependency scope (e.g., compile, test), and position within the dependency tree (direct or

transitive). Each test case assessed whether the designated focus node was correctly retained or pruned, depending on its invocation status. A total of 22 test cases correctly pruned an uninvoked or test scope dependency(true positives), while 18 test cases correctly retained invoked dependencies (true negatives). No cases resulted in incorrect pruning (false positives) or incorrect retention (false negatives). Although these results indicate that the pruning mechanism performed optimally on the synthetic dataset, this does not necessarily imply perfection across all possible licensing scenarios. Rather, the test cases were designed to reflect the range of conditions expected in open-source data, ensuring that the scope of this approach is evaluated against the primary cases it is intended to handle.

Similarly to maven, the synthetic evaluation of the Gradle licensing pruning mechanism yielded a precision of 1.0, an accuracy of 1.0, and an F1 score of 1.0, based on a dataset of 40 test cases. This similarity in pruning performance to Maven can be attributed to the fact that once the dependency trees outputted by each build system are parsed into the Node object model, both Gradle and Maven projects have the same pruning logic applied to their dependency trees.

It is important to note that the test cases were specifically designed to evaluate scenarios within the intended scope of the approach: single-module systems where a dependency is considered invoked if it has a direct method invocation. These controlled conditions allowed for precise labeling of each dependency and enabled clear evaluation of pruning correctness based on known ground truth. The results, while ideal, are not meant to suggest that real-world pruning will always achieve perfect accuracy. Instead, they validate that the pruning logic performs as expected when applied under ideal, interpretable conditions with complete and reliable dependency metadata.

Moreover, all dependencies used in these synthetic test cases were known to be available on the Maven repository and included complete source code, eliminating the risk of source code retrieval failure or incomplete invocation analysis. This setup isolates the pruning logic itself from external limitations such as missing artifacts or dynamic behavior, which are evaluated separately through the open-source dataset in Section 5.1.2. These synthetic results serve as a baseline verification that the proposed approach behaves correctly under expected inputs and controlled structural variations.

5.1.2 Open-source Dataset

To answer this research question, 38 software projects utilizing Maven and 30 software projects utilizing Gradle were manually validated to evaluate the pruning logic. These projects were randomly selected from a pool of 378 Maven and 123 Gradle open-source projects in which both security and licensing dependency analyses had run successfully. Manual validation was conducted once per project, as pruning results remain consistent regardless of whether the analysis type is security or licensing. This sampling ensured consistent, comparable, and meaningful results for evaluating the correctness of the pruning approach.

To determine the number of Maven projects required for manual analysis, we calculated the sample size using the standard formula for estimating sample size within a finite population:

$$n = \frac{NZ^2p(1-p)}{Z^2p(1-p) + e^2(N-1)} \tag{5.1}$$

where:

- $N = 378$ is the total number of Maven projects where both licensing and security analysis were successfully conducted.
- $Z = 1.96$ is the critical value corresponding to a 95% confidence level.
- $p = 0.5$ is the assumed proportion of projects exhibiting relevant characteristics
- $e = 0.15$ is the margin of error (15%).

This resulted in a target of 38 projects for manual review.

To determine the number of Gradle projects required for manual analysis, we calculated the sample size using the same formula where:

- $N = 123$ is the total number of Gradle projects where both licensing and security analysis were successfully conducted.
- $Z = 1.96$ is the critical value corresponding to a 95% confidence level.
- $p = 0.5$ is the assumed proportion of projects exhibiting relevant characteristics
- $e = 0.15$ is the margin of error (15%).

This resulted in a target of 30 projects for manual review.

In our open-source data manual analysis we calculated the following:

1. TPs: True Positives (Pruned uninvoked or test scope dependencies)
2. FPs: False Positives (Pruned invoked, non-test scope dependencies)
3. PFPs: Potential False Positives (Pruned due to source code download failure, could have been invoked or uninvoked within the source code)
4. Precision(PFP=FP): Calculation of precision considering PFPs as FPs
5. Precision(PFP=TP): Calculation of precision considering PFPs as TPs

Table 5.2: Open-source Dataset Pruning Metrics

BuildSystem	Total TP	Total FP	Total PFP	Precision(PFP=FP)	Precision(PFP=TP)
Maven	90	13	1	86.5%	87.5%
Gradle	44	3	22	63.8%	95.7%

As seen in table 5.2, our manual analysis of 38 Maven open-source data projects found that among the dependencies pruned 90 were True Positives (TPs), 13 were False Positives (FPs), and 1 was a Potential False Positive (PFP). True positives were either test-scoped dependencies or dependencies not required for successful execution of *mvn clean compile* on the project. We used this

command as a validation method, as removing an essential dependency would cause a compilation failure. Notably, 'mvn clean compile' was executed within the project's source code where the dependency was directly declared. In cases where transitive dependencies were pruned, we retrieved the downloaded and decompiled cached source code and manually inspected the code, running the compilation command within that context.

In the 13 FPs identified, direct invocation was not detected. Instead, these cases involved imports and/or invocations that were not captured during AST analysis, including invocation through reflection, annotation-based static declarations, and fully qualified class references. Additionally, the one PFP observed resulted from a failed source code download, preventing us from determining whether it was correctly pruned. To account for this uncertainty, we compute precision using two approaches: one where the PFP is treated as a false positive (FP) and another where it is considered a true positive (TP), providing both an upper and lower bound for precision.

As shown in Table 5.2, our manual analysis of Gradle open-source data projects found that pruned true positives were either test-scoped or not required for the successful execution of *./gradlew clean compileJava*. The false positives followed similar patterns to those observed in the Maven analysis.

However, Gradle had a significantly higher number of PFPs, primarily due to dependencies linked to repositories specified in the Gradle build file that were external to Maven repository, leading to source code retrieval failures in our approach. When the source code of a dependency cannot be downloaded and decompiled, the approach cannot determine whether the dependency is invoked, and pruning outcomes involving that dependency are classified as PFPs. These cases cannot be definitively labeled as true or false positives without access to the source code. The precision range presented reflects this ambiguity. Future improvements may include expanding repository resolution to support additional Gradle-specific configurations, but these enhancements are out of scope for this work. With this, PFPs shown in Table 5.2 represent a conservative estimate and are explicitly flagged to preserve the validity of the overall evaluation.

The precision values in Table 5.2 provide insight into the accuracy of the pruning approach when applied to open-source data projects. For Maven, precision remains high regardless of how PFPs are classified, with a minimal impact on the overall result. This suggests effective pruning within our manually analyzed random sample with minimal misclassification. In contrast, Gradle exhibits a wider precision range, from 63.8% when PFPs are considered false positives to 95.7% when they are treated as true positives. This discrepancy is due to a higher number of PFPs in Gradle, primarily caused by source code retrieval failures linked to dependencies specified repositories not supported by our approach. Overall, these findings suggest that the pruning approach is effective, but improvements in source code retrieval and dependency resolution could further enhance precision, particularly for Gradle projects.

5.2 RQ2: How does dependency pruning impact the security and licensing compliance of a software system?

To address this research question, we analyze the effect of pruning uninvoked and test-scope dependencies on security vulnerability and license compliance assessments. Specifically, we examine whether removing these dependencies eliminates identified vulnerable or incompatible dependencies within the dependency tree.

5.2.1 RQ2.A: What is the effect of pruning on security vulnerability analysis?

We evaluate our approach's effect on security by checking how often our approach was able to successfully prune a dependency in its dependency tree that introduced a security vulnerability. We define a vulnerable dependency as one that is associated with one or more CVE(s).

The results in Table 5.1 for the cases shown in Table A.3 and Table A.4 show that 8 cases specifically tested the approach's performance when the focus node introduced a security vulnerability. Within these 8 cases, 4 were conducted with the focus node as a direct dependency and 4 were conducted with the focus node as a transitive dependency. We ran each as invoked and uninvoked with the focus node introducing various numbers of CVEs for each case. In all of the cases in our synthetic data for both Maven and Gradle all uninvoked vulnerable dependencies were successfully pruned, and all invoked vulnerable dependencies were successfully retained.

To answer this question in the context of our open-source data analysis we reviewed all of the manually classified TPs found as listed in table 5.2 to identify how many were vulnerable dependencies. In this analysis, 26 out of 90 Maven TPs were found to have been associated with one or more CVEs. We also found that 20 out of 44 Gradle TPs were found to be vulnerable dependencies. These results support that the approach's pruning effectively retains vulnerable dependencies.

Figure 5.1 presents the distribution of manually validated true positive Maven dependencies that were correctly pruned and were found to introduce at least one known vulnerability (CVE). The majority of projects had either zero or one such vulnerable dependency removed, with a median of 0 and an interquartile range between 0 and 1. A small number of projects had higher counts, as indicated by outliers at 3 and 4. The distribution of true positive vulnerable dependencies in Gradle projects (Figure 5.2) is similar to Maven (Figure 5.1), with a slightly lower maximum and fewer outliers, indicating pruning was similarly effective in removing unused vulnerable dependencies across both build systems. These distributions demonstrate that pruning uninvoked dependencies can eliminate components with known security vulnerabilities, reducing the overall security risk in the overall software system.

5.2.2 RQ2.B: What is the effect of pruning on licensing compliance analysis?

We evaluate our approach's effect on license compliance by checking how often our approach was able to successfully prune a dependency in its dependency tree that introduced a license incompatibility. We define a dependency with a license incompatibility as one whose license directly

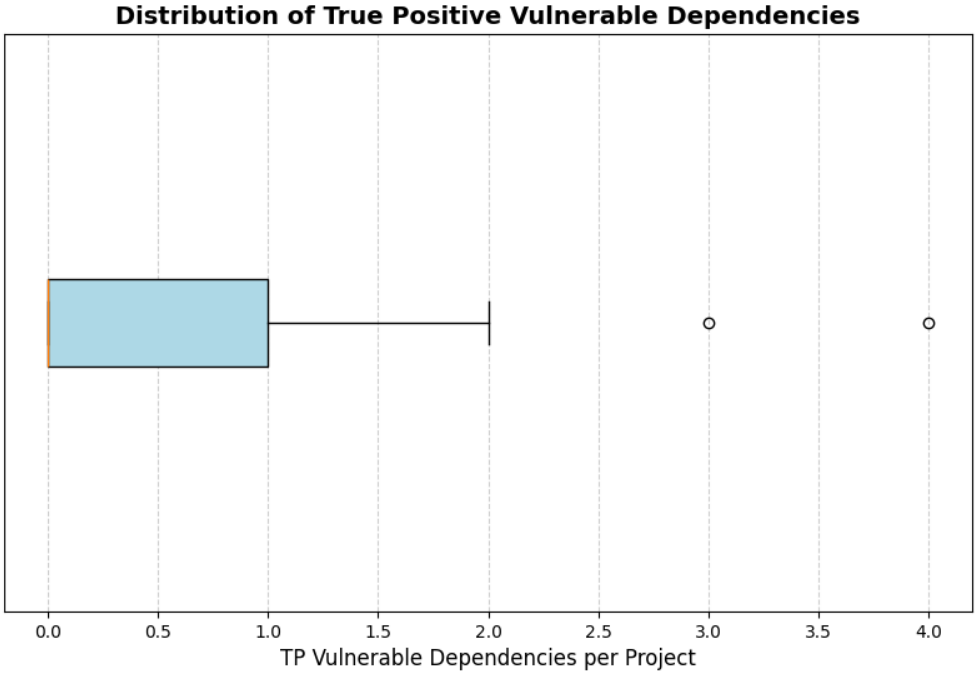

Figure 5.1: Maven Distribution of TPs that introduced at least one CVE

conflicts with its parent's license.

The results in Table 5.1 for the cases shown in Table A.1 and Table A.2 show that 12 cases specifically tested the approach's performance when the focus node introduced a license incompatibility. Within these 12 cases, 4 were conducted with the focus node as a direct dependency and 8 were conducted with the focus node as a transitive dependency. We conducted these with single and multi-licenses and ran each as invoked and uninvoked. In all of the cases in our synthetic data for both Maven and Gradle all uninvoked license incompatibilities were successfully pruned, all invoked license incompatibilities were successfully retained.

To answer this question in the context of our open-source data analysis we reviewed all of the manually classified TPs found as listed in table 5.2 to identify how many had license incompatibilities flagged. In this analysis, 19 out of 90 Maven TPs and 2 out of 44 Gradle TPs were found to have license incompatibilities with their parent. While the effect was less pronounced in Gradle, these results indicate a reduction in the total number of license incompatibilities present in the overall software system.

Figure 5.3 shows the distribution of manually validated true positive Maven dependencies that were correctly pruned and were found to introduce a license incompatibility with their parent. As with the vulnerability analysis, the majority of projects had zero or one such incompatible dependency removed. The median value is 0, and the interquartile range spans from 0 to 1. A single outlier at value 3 indicates one project where pruning removed multiple incompatible dependencies. In contrast to Maven, where multiple license-incompatible dependencies were pruned in some

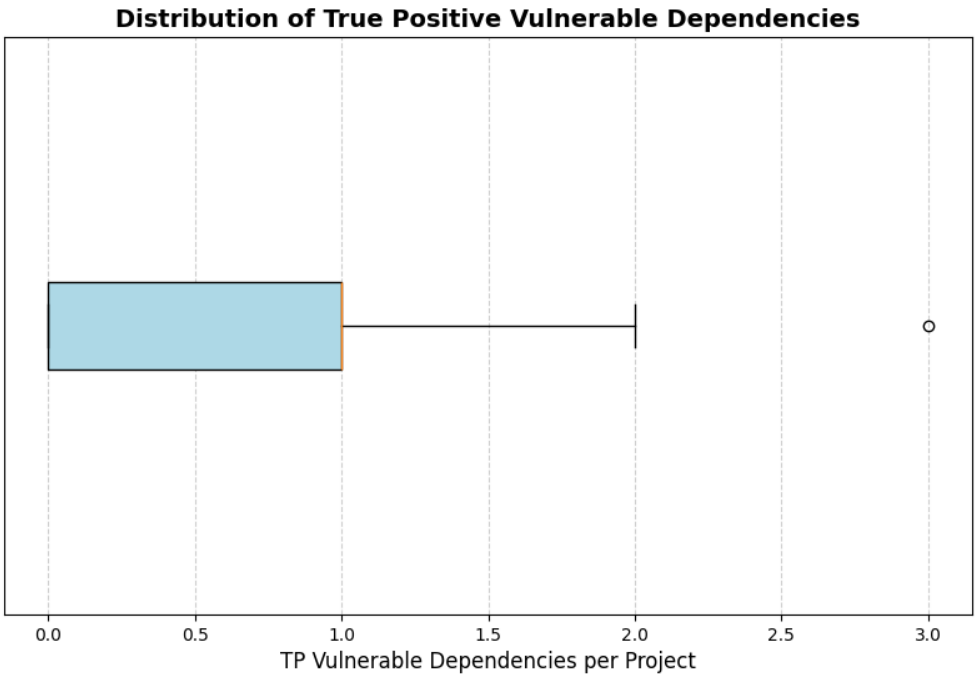

Figure 5.2: Gradle Distribution of TPs that introduced at least one CVE

projects (Figure 5.3), the Gradle distribution (Figure 5.4) shows that nearly all projects had zero true positives that introduced incompatibilities, with two projects having a single pruned incompatible dependency. This may be attributed to the 46.3% of analyzed Gradle dependencies for which license identification was unsuccessful, as previously discussed. These results shown in Figure 5.3 and Figure 5.4 support that pruning effectively eliminates license conflicts caused by unused dependencies, improving the system's overall license compliance.

5.3 Discussion

Using the data from our experimental design and validation, we evaluate the effectiveness of dependency pruning in improving software security and licensing compliance. This approach removes uninvoked and test-scope dependencies from a software system's dependency tree to reduce unnecessary risk and complexity. The results indicate that pruning significantly impacts both security vulnerability analysis and licensing compliance analysis, though limitations in project-level license identification and dependency source code retrieval require further examination.

Table 5.1 presents precision, recall, and F1-score for dependency pruning on synthetic test cases for both Maven and Gradle. The approach achieved precision, recall, and F1-scores of 1.0 for both build systems, indicating that all uninvoked or test-scope dependencies were correctly pruned, while all invoked dependencies were correctly retained. These results demonstrate the

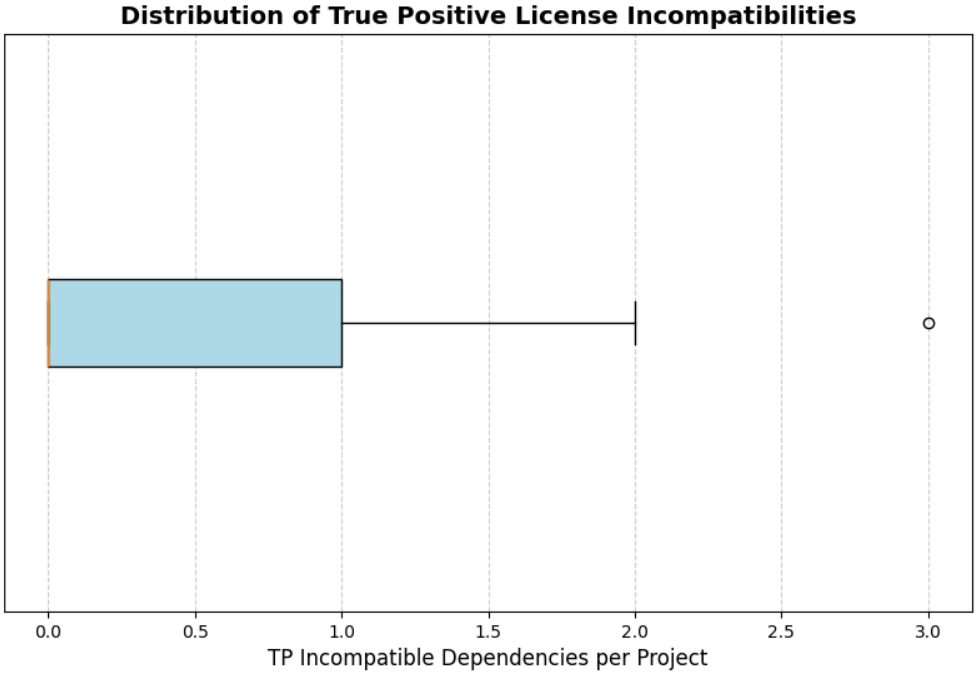

Figure 5.3: Maven Distribution of TPs that introduced license incompatibilities

approach's effectiveness in controlled environments where dependencies are imported and directly invoked when needed, licenses are present when needed in the expected format, and all dependency source code can be resolved through maven repository. However, real-world software projects introduced additional complexities such as indirect invocations, dynamic dependencies, and source code resolution failures that we identified in our open-source dataset through manual analysis.

A manual validation of 38 Maven and 30 Gradle open-source projects was conducted to assess pruning accuracy in real-world settings. The results, presented in Table 5.2, show a high precision rate for Maven projects, ranging from 86.2% to 87.7% depending on how potential false positives (PFPs) were classified. Gradle projects exhibited a broader precision range (63.8% to 95.7%) due to a higher number of PFPs caused by source code retrieval failures. These findings indicate that the pruning mechanism remains effective in real-world projects. However, future improvements could focus on dependency source code retrieval and dependency invocation analysis. This is especially applicable for Gradle projects, where dependencies sourced from repositories other than maven repository posed challenges in retrieving the dependency source code.

The security analysis results indicate that pruning substantially reduces the number of vulnerable dependencies. In Maven projects, 7,053 CVEs were detected in full dependency trees, affecting 925 dependencies, while the pruned trees retained 2,891 CVEs across 398 dependencies, representing a 57% reduction. In Gradle projects, 616 CVEs were detected in full dependency trees, affecting 200 dependencies, while the pruned trees retained 128 CVEs across 18 dependencies, representing a 91% reduction.

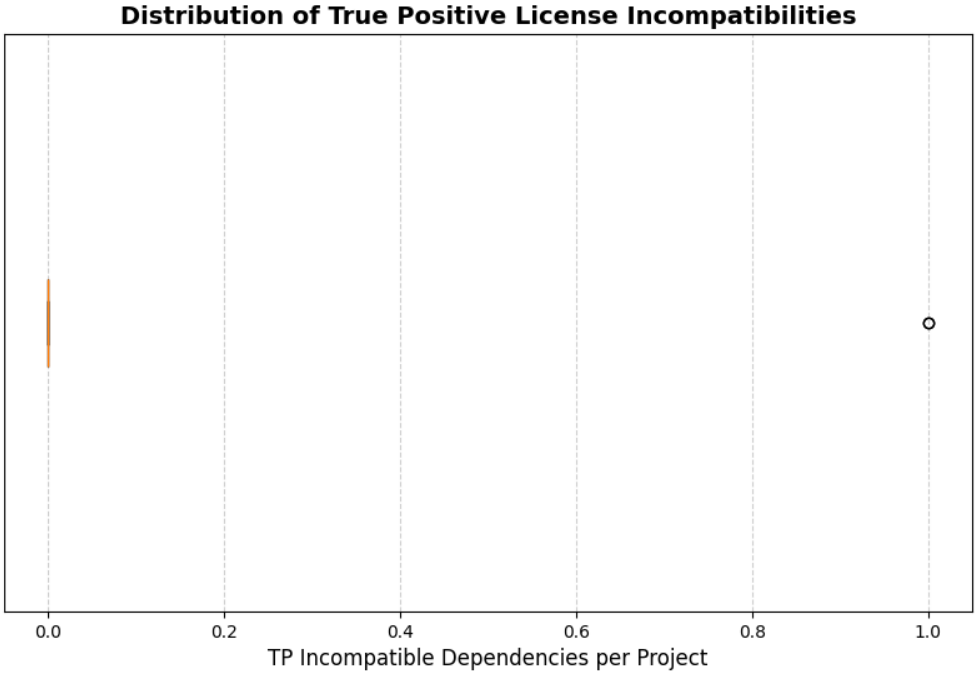

Figure 5.4: Gradle Distribution of TPs that introduced license incompatibilities

In Maven projects, 158 license incompatibilities were detected in full dependency trees, compared to only 23 in pruned trees. In Gradle projects, 18 license incompatibilities were detected in full trees, and only one was found in pruned trees. This shows a 86.6% reduction of license incompatibilities from full to pruned in Maven projects and a 94.4% reduction of license incompatibilities in Gradle projects when our approach was applied to our open-source dataset. We also observed that 41.5% of analyzed Maven dependencies and 46.3% of analyzed Gradle dependencies lacked a detected license. However, the tools chosen are among the state of the art. The high number of dependencies without licenses may be due to several factors including missing licensing, proprietary licensing, or misclassifications.

The results highlight the potential benefits of dependency pruning in improving software security and licensing compliance by removing uninvoked dependencies. However, limitations in license identification and dependency resolution must be considered. Overall, the findings suggest that dependency pruning is effective in reducing unnecessary licensing and security risks, but further validation of the results observed on our open-source dataset is needed.

Chapter 6

Conclusion

In this thesis, we present an approach that refines software dependency trees by systematically identifying and pruning dependencies that are either uninvoked or used solely in testing scopes. The approach first resolves the full dependency tree using the build system, then prunes dependencies based on invocation results obtained through AST analysis. For licensing analysis, it integrates ScanCode-Toolkit and Flict to assess license incompatibilities in both the full and pruned dependency trees. For security analysis, it utilizes the Dependency-Check CLI tool to determine which dependencies in both dependency trees are associated with known CVEs.

To validate the approach, we applied it to 40 synthetic test cases per build system, ensuring the approach could accurately prune dependencies at different levels of the dependency tree. These synthetic test cases were also used to evaluate the impact of pruning on security and license compatibility by comparing vulnerabilities and license incompatibilities in the full and pruned dependency trees. Additionally, we conducted pruning and security analysis on a total of 363 Maven and Gradle open-source projects, alongside pruning and licensing analysis on 514 Maven and Gradle open-source projects. Of these successful runs, we manually reviewed 30 projects per build system to verify the correctness of pruning decisions and assess their impact on security and licensing analysis.

Our findings demonstrate that this approach enhances visibility into the dependency trees of a software system and effectively removes uninvoked or test-scoped dependencies that are unnecessary for functionality and distribution. In some cases, pruning also eliminated dependencies introducing license incompatibilities or security vulnerabilities.

For RQ1, the synthetic test results confirmed that the pruning logic operates with perfect precision and recall within the defined scope, correctly retaining or removing dependencies based on invocation status. In the manually reviewed open-source projects, precision remained high—up to 87.5% for Maven and 95.7% for Gradle—depending on how failed source code retrieval cases were treated, further validating the approach's accuracy in real-world settings.

For RQ2, pruning reduced the number of license incompatibilities and security vulnerabilities across both datasets. In Maven projects, pruning led to an 86.6% reduction in license incompatibilities and a 56.9% reduction in vulnerable dependencies. In Gradle projects, all license incompatibilities were eliminated after pruning, and 91.0% of vulnerable dependencies were removed. These reductions indicate that pruning improves both licensing compliance and security posture by eliminating non-essential dependencies that contribute to risk.

The evaluation of both synthetic and real-world projects highlights the impact of pruning in reducing security and licensing risks. These results emphasize the benefits of structured dependency management, reducing risks associated with excessive or unmanaged dependencies while ensuring

that only essential components remain in the software system.

6.1 Limitations

There are several limitations to the current implementation of the approach that impact its applicability and effectiveness in analyzing dependencies. First, the approach is restricted to Java projects that use Maven, Gradle, or Ant/Ivy build systems, limiting its ability to analyze projects written in other languages, such as JavaScript, which rely on different dependency managers like npm. Expanding support for additional languages would require integrating their dependency management systems and adapting language-specific analysis techniques, similar to how Abstract Syntax Tree (AST) parsing is applied for Java.

Another limitation is that the approach only supports single-module projects, reducing its effectiveness for larger, more complex software systems. Single-module projects contain all dependencies and source code within a single build configuration file, while multi-module projects consist of multiple interdependent submodules, each with its own configuration. These interdependencies complicate dependency resolution and require additional logic to aggregate results across modules. Enhancing the approach to support multi-module projects would improve its applicability to large-scale software systems.

While ScanCode-Toolkit is effective at identifying standard open-source licenses, it may struggle with custom, proprietary, or modified licenses, potentially leading to misclassifications [53]. Additionally, although ScanCode-Toolkit detects license exceptions, our approach does not currently process them (e.g., license expressions containing "WITH" clauses). Since these exceptions modify the terms of a base license, excluding them results in a more conservative classification. However, this may not accurately reflect the actual licensing conditions of a dependency, leading to an incomplete representation of its compatibility status. Addressing these limitations in future work would improve the precision and reliability of license assessments.

6.2 Threats to Validity

There are several threats to validity that could affect the accuracy, reliability, and generalizability of the study results. A key issue in resolving full dependency trees arises from how Maven's dependency:tree handles dependency resolution. Specifically, when a parent project declares and invokes the same dependencies as its child project, dependency:tree lists them only under the parent, potentially omitting them from the child's output. This behavior may result in an incomplete or misleading representation of the software system's dependency structure. To ensure accuracy in such cases, an additional method for resolving full dependency trees in Maven projects would be beneficial.

The approach also relies on Maven Central and Libraries.io as primary sources for retrieving source code and JAR files. If both retrieval methods fail, the approach flags the source as unavailable. This affects multiple aspects of the analysis: missing source code prevents AST-based dependency pruning, and missing JAR files hinder security analysis with the Dependency-Check

CLI tool. In the case where we are unable to download the source code of a dependency, its subtree (e.g. its children and their children, etc.) is omitted from the pruned dependency tree. This omission could lead potentially to a false positive if dependencies in that subtree were invoked in this codebase and should have been retained. This can impact the reliability of the pruned dependency tree in cases where there is a SourceCodeDownloadFailureFlag. Although this may impact the results in these cases, we note in 5.1.2 that this was found to occur when the dependency is sourced from a repository that is not Maven Central or when a dependency has been manually added where even the build tool fails to resolve it. Enhancing the approach with alternative retrieval methods could mitigate these issues and reduce reliance on these two sources.

Another threat to validity arises from the limitations of static analysis in determining dependency usage. Static analysis is unable to detect dependencies that are dynamically loaded at runtime and does not account for dependencies that are conditionally invoked based on configuration settings. If the source code is incomplete or cannot be parsed, AST-based analysis may fail, leading to incorrect pruning decisions. These limitations can result in false positives or false negatives in dependency pruning. Future improvements could incorporate dynamic analysis techniques to improve accuracy and address the gaps inherent in static analysis.

Another threat is the potential dataset bias introduced by Gradle's open-source dataset sourced from Software Heritage. This dataset contains small, incomplete, or educational repositories that do not accurately represent the complexity and structure of large-scale software systems. These projects may lack comprehensive dependency trees or accurate build files, potentially skewing results by overrepresenting unrealistic dependency structures. Expanding the evaluation dataset to include well-documented and widely used software systems would enhance the reliability and generalizability of the findings.

6.3 Future Work

Several improvements can be made to enhance the approach's effectiveness and expand its applicability. One key area for future development is broadening language support beyond Java. Many open-source projects are written in languages such as Python, JavaScript, and C++, each with distinct dependency management tools like Setuptools, npm, and CMake. Expanding the approach to support multiple programming languages would enable it to analyze a wider range of software ecosystems and improve its generalizability across different development environments.

Support for additional build systems is another avenue for improvement. The current implementation is designed for Maven, Gradle, and Ant with Ivy, which limits its applicability to projects using other widely adopted build systems. Extending compatibility to include tools such as Bazel, CMake, or Buck would enable the approach to process a more diverse set of software projects. Additionally, handling multi-module projects and those that use more than one build system would enhance its ability to analyze large-scale and complex software architectures.

The dependency pruning approach relies on static analysis of Abstract Syntax Trees (AST) to determine whether dependencies are invoked. While this method effectively captures common usage patterns, it does not account for dynamically loaded dependencies, such as those introduced through reflection, service loaders, or runtime configuration files. Since static analysis alone can-

not detect such cases, future work could explore integrating runtime analysis techniques, such as bytecode instrumentation or dynamic tracing, to improve accuracy in identifying actively used dependencies.

Gradle dependency resolution presents another challenge. The approach currently consolidates dependencies across multiple configurations into a single scope, classifying a dependency as test-scoped only if all of its configurations are test-related. However, dependencies that appear in both test and non-test configurations may serve different roles depending on the context. Future refinements could incorporate deeper usage analysis to distinguish between dependencies that are strictly test-related and those that are also required for production, thereby improving scope attribution for Gradle projects.

Enhancing licensing analysis is another critical area for future research. The approach currently relies on ScanCode Toolkit for license detection, which, while effective for standard open-source licenses, may struggle with custom, modified, or multi-license scenarios. Integrating additional license detection tools, such as FOSSology or SPDX-based analyzers, could improve the accuracy of project-level license identification. Moreover, heuristic-based techniques and machine-learning models trained on diverse license texts could enhance the approach's ability to process complex license expressions. Cross-referencing metadata from SPDX, open-source repositories, and natural language processing (NLP) techniques could further refine license detection, reducing false positives and negatives in real-world software projects.

Finally, Gradle's evaluation dataset could be expanded to include a broader set of widely used and well-documented software projects that accurately represent real-world dependency structures. The current reliance on the Software Heritage dataset introduces risks, as it contains toy projects and incomplete repositories that may not fully capture the complexity of modern software systems. While many entries were minimal or toy examples, we also observed projects with over 300 declared dependencies, which disproportionately inflated certain metrics and may not reflect typical project configurations. Incorporating a curated selection of widely used open-source projects would provide a more representative benchmark for assessing the approach's effectiveness in security and licensing compliance analysis.

Appendix A

Appendix

Test Case	Declared Dependencies	Invoked Dependencies
DDEP_d,d,i,compat	commons-lang3, guava, slf4j-api, log4j-core, jackson-databind, httpclient, spring-core	commons-lang3, guava, log4j-core, jackson-databind
DDEP_d,d,ni,compat	commons-collections4, guava, slf4j-api, log4j-core, jackson-core, httpclient, spring-core	guava, slf4j-api, log4j-core
DDEP_d,d,i,incompat	slf4j-api, log4j-core, jackson-core, jackson-databind, httpclient, spring-core, guava	slf4j-api, log4j-core, jackson-databind
DDEP_d,d,ni,incompat	jackson-databind, slf4j-api, guava, httpclient, commons-lang3, commons-collections4, spring-core	guava, slf4j-api, httpclient
DDEP_d,d,i,multi,compat	commons-lang3, guava, slf4j-api, log4j-core, jackson-core, httpclient, spring-core	slf4j-api, log4j-core, commons-lang3
DDEP_d,d,ni,multi,compat	commons-collections4, guava, slf4j-api, log4j-core, jackson-core, httpclient, spring-core	slf4j-api, log4j-core, guava
DDEP_d,d,i,multi,incompat	spring-core, slf4j-api, log4j-core, jackson-core, guava, httpclient, commons-lang3, commons-collections4	log4j-core, guava, spring-core
DDEP_d,d,ni,multi,incompat	slf4j-api, guava, httpclient, commons-lang3, commons-collections4, spring-core, jetty-server	guava, slf4j-api, httpclient
DDEP_d,d,t,i,compat	junit, mockito-core, jackson-core, commons-lang3, slf4j-api, log4j-core, jackson-databind, spring-core, guava	mockito-core, jackson-core, spring-core, guava
DDEP_d,d,t,ni,compat	junit, mockito-core, jackson-databind, commons-lang3, slf4j-api, log4j-core, spring-core, guava, commons-collections4	slf4j-api, log4j-core, guava
TDEP_t1,d,i,compat	spring-core, slf4j-api, jackson-databind, commons-lang3, guava, log4j-core	spring-core, slf4j-api, guava, log4j-core
TDEP_t1,d,ni,compat	guava, commons-lang3, log4j-core, jackson-core, slf4j-api, spring-core	commons-lang3, slf4j-api, log4j-core
TDEP_t1,d,i,incompat	commons-collections4, guava, slf4j-api, log4j-core, jackson-core, spring-core	slf4j-api, log4j-core, commons-collections4
TDEP_t1,d,ni,incompat	commons-lang3, guava, slf4j-api, log4j-core, jackson-core, spring-core	slf4j-api, log4j-core, guava
TDEP_t1,d,i,multi,compat	commons-lang3, guava, slf4j-api, log4j-core, jackson-core, spring-core	slf4j-api, log4j-core, commons-lang3
TDEP_t1,d,ni,multi,compat	commons-collections4, guava, slf4j-api, log4j-core, jackson-core, spring-core	slf4j-api, log4j-core, guava
TDEP_t1,d,i,multi,incompat	slf4j-api, guava, commons-lang3, commons-collections4, spring-core, jetty-server, javax.servlet-api	javax.servlet-api, guava, slf4j-api, jetty-server
TDEP_t1,d,ni,multi,incompat	slf4j-api, guava, commons-lang3, commons-collections4, spring-core, jetty-server, javax.servlet-api	guava, javax.servlet-api, slf4j-api, spring-core
TDEP_t2,d,i,compat	log4j-core, jackson-databind, commons-lang3, guava, junit, mockito-core, spring-core, slf4j-api, jackson-core	jackson-databind, commons-lang3, guava, spring-core, slf4j-api
TDEP_t2,d,ni,compat	jackson-core, jackson-databind, slf4j-api, guava, commons-lang3, log4j-core, junit, mockito-core, spring-core	guava, slf4j-api, log4j-core, spring-core
TDEP_t2,d,i,multi,compat	slf4j-api, javax.servlet-api, guava, commons-lang3, commons-collections4, spring-core, jetty-server	guava, javax.servlet-api, slf4j-api, jetty-server
TDEP_t2,d,ni,multi,compat	slf4j-api, guava, commons-lang3, commons-collections4, spring-core, javax.servlet-api, jetty-server	guava, javax.servlet-api, slf4j-api, commons-collections4
TDEP_t2,d,i,multi,incompat	commons-lang3, slf4j-api, guava, log4j-core, spring-core, jackson-core, commons-collections4, jetty-server	guava, log4j-core, commons-collections4
TDEP_t2,d,ni,multi,incompat	commons-lang3, slf4j-api, guava, log4j-core, spring-core, jackson-core, commons-collections4, jetty-server	slf4j-api, log4j-core, jetty-server
TDEP_t2,d,i,multi,incompat	commons-lang3, slf4j-api, guava, log4j-core, spring-core, jackson-core, commons-collections4, jetty-server	slf4j-api, guava, log4j-core, spring-core
TDEP_t2,d,ni,multi,incompat	commons-lang3, slf4j-api, guava, log4j-core, spring-core, jackson-core, commons-collections4, jetty-server	slf4j-api, log4j-core, spring-core

Table A.1: Dependencies Used in Synthetic Licensing Test Cases found in Table 4.3.

Note: Transitive dependencies are considered only within their level, meaning that pruning is determined by the pool of dependencies available at that level rather than the entire project's dependencies.

Test Case	Focus Node Group ID	Focus Node Artifact ID	Focus Node Version
DDEP_d,d,i,compat	org.apache.commons	commons-lang3	3.12.0
DDEP_d,d,ni,compat	org.apache.commons	commons-collections4	4.4
DDEP_d,d,i,incompat	org.apache.logging.log4j	log4j-core	2.14.1
DDEP_d,d,ni,incompat	com.fasterxml.jackson.core	jackson-databind	2.13.0
DDEP_d,d,i,multi,compat	org.apache.commons	commons-lang3	3.12.0
DDEP_d,d,ni,multi,compat	org.apache.commons	commons-collections4	4.4
DDEP_d,d,i,multi,incompat	org.springframework	spring-core	5.3.8
DDEP_d,d,ni,multi,incompat	org.eclipse.jetty	jetty-server	9.4.43.v20210629
DDEP_d,d,t,i,compat	org.mockito	mockito-core	4.0.0
DDEP_d,d,t,ni,compat	junit	junit	4.13.2
TDEP_t1,d,i,compat	org.springframework	spring-core	5.3.8
TDEP_t1,d,ni,compat	com.google.guava	guava	31.0.1-jre
TDEP_t1,d,i,incompat	org.apache.commons	commons-collections4	4.4
TDEP_t1,d,ni,incompat	org.apache.commons	commons-lang3	3.12.0
TDEP_t1,d,i,multi,compat	org.apache.commons	commons-lang3	3.12.0
TDEP_t1,d,ni,multi,compat	org.apache.commons	commons-collections4	4.4
TDEP_t1,d,i,multi,incompat	org.eclipse.jetty	jetty-server	9.4.43.v20210629
TDEP_t1,d,ni,multi,incompat	org.eclipse.jetty	jetty-server	9.4.43.v20210629
TDEP_t2,d,i,compat	com.fasterxml.jackson.core	jackson-databind	2.9.0
TDEP_t2,d,ni,compat	com.fasterxml.jackson.core	jackson-databind	2.9.0
TDEP_t2,d,i,multi,compat	org.eclipse.jetty	jetty-server	9.4.43.v20210629
TDEP_t2,d,ni,multi,compat	org.eclipse.jetty	jetty-server	9.4.43.v20210629
TDEP_t2,d,i,multi,incompat	org.apache.commons	commons-collections4	4.4
TDEP_t2,d,ni,multi,incompat	org.apache.commons	commons-collections4	4.4
TDEP_t2,d,i,multi,incompat	com.google.guava	guava	31.0.1-jre
TDEP_t2,d,ni,multi,incompat	com.google.guava	guava	31.0.1-jre

Table A.2: Focus Node Used in Synthetic Licensing Test Cases found in Table 4.3.

Test Case	Declared Dependencies	Invoked Dependencies
DDEP_d,d,i,vuln	guava, slf4j-api, log4j-core, commons-lang3, httpclient	httpclient, guava, slf4j-api, log4j-core
DDEP_d,d,ni,vuln	commons-collections4, slf4j-api, guava, log4j-core, spring-core	slf4j-api, guava, log4j-core
DDEP_d,d,i,noVuln	commons-lang3, slf4j-api, guava, log4j-api, spring-core	commons-lang3, slf4j-api, guava, log4j-api
DDEP_d,d,ni,noVuln	commons-text, guava, slf4j-api, spring-core, hibernate-core	slf4j-api, guava, spring-core
DDEP_d,d,t,i,vuln	jetty-server, slf4j-api, guava, junit, mockito-core	jetty-server, junit, mockito-core
DDEP_d,d,t,ni,vuln	dom4j, junit, mockito-core, slf4j-api, guava	junit, mockito-core
TDEP1_d,d,i,vuln	spring-core, slf4j-api, guava, log4j-core, commons-collections	slf4j-api, guava, log4j-core, commons-collections4
TDEP1_d,d,ni,vuln	spring-core, slf4j-api, guava, commons-compress, log4j-core	slf4j-api, guava, log4j-core
TDEP1_d,d,i,noVuln	guava, slf4j-api, commons-lang3, spring-core, log4j-api	slf4j-api, commons-lang3, guava, log4j-api
TDEP1_d,d,ni,noVuln	guava, slf4j-api, spring-core, commons-collections4, log4j-api	slf4j-api, guava, spring-core
TDEP2_d,d,i,vuln	struts2-core, slf4j-api, guava, log4j-core, commons-io	struts2-core, slf4j-api, guava, log4j-core
TDEP2_d,d,ni,vuln	log4j-core, slf4j-api, guava, commons-io, jetty-server	slf4j-api, guava, jetty-server
TDEP2_d,d,i,noVuln	slf4j-api, guava, log4j-api, spring-core, commons-text	slf4j-api, guava, log4j-api, commons-text
TDEP2_d,d,ni,noVuln	slf4j-api, guava, log4j-api, spring-core, hibernate-core	slf4j-api, guava, spring-core

Table A.3: Dependencies Used in Synthetic Security Test Cases found in Table 4.5.

Test Case	Focus Node Group ID	Focus Node Artifact ID	Focus Node Version
DDEP_d,d,i,vuln	org.apache.httpcomponents	httpclient	4.5.12
DDEP_d,d,ni,vuln	org.apache.commons	commons-collections4	4.0
DDEP_d,d,i,noVuln	org.apache.commons	commons-lang3	3.12.0
DDEP_d,d,ni,noVuln	org.apache.commons	commons-text	1.13.0
DDEP_d,d,t,i,vuln	org.eclipse.jetty	jetty-server	12.0.8
DDEP_d,d,t,ni,vuln	org.dom4j	dom4j	2.1.1
TDEP1_d,d,i,vuln	org.apache.commons	commons-collections4	4.0
TDEP1_d,d,ni,vuln	org.apache.commons	commons-compress	1.25.0
TDEP1_d,d,i,noVuln	org.apache.commons	commons-lang3	3.12.0
TDEP1_d,d,ni,noVuln	org.apache.commons	commons-collections4	4.5.0-M3
TDEP2_d,d,i,vuln	org.apache.struts	struts2-core	6.3.0.1
TDEP2_d,d,ni,vuln	org.apache.commons	commons-io	1.3.2
TDEP2_d,d,i,noVuln	org.apache.commons	commons-text	1.13.0
TDEP2_d,d,ni,noVuln	org.hibernate.orm	hibernate-core	6.6.9.Final

Table A.4: Focus Nodes Used in Synthetic Security Test Cases found in Table 4.5.

References

[1] Gavin. Codegeekz: 15 best dependency management tools for developers: https://-codegeekz.com/15-best-dependency-management-tools-for-developers/, Sep 2016.

[2] Russ Cox. Surviving software dependencies. *Communications of the ACM*, Vol. 62 No. 9:36–43, September 2019.

[3] R.W. Selby. Enabling reuse-based software development of large-scale systems. *IEEE Transactions on Software Engineering*, 31(6):495–510, 2005.

[4] Pedro Esteves Pinto. Promoting software reuse in a corporate setting: https://www.cs.cmu.edu/afs/cs/usr/ppinto/www/reuse.html, 1998.

[5] José L. Barros-Justo, Fernando Pinciroli, Santiago Matalonga, and Nelson Martínez-Araujo. What software reuse benefits have been transferred to the industry? a systematic mapping study. *Information and Software Technology*, 103:1–21, 2018.

[6] Liran Tal. Software dependencies: How to manage dependencies at scale: https://s-nyk.io/series/open-source-security/software-dependencies/, Jun 2022.

[7] Joel Ossher, Sushil Bajracharya, and Cristina Lopes. Automated dependency resolution for open source software. pages 130–140, 2010.

[8] Tom Mens. *Introduction and Roadmap: History and Challenges of Software Evolution*, pages 1–11. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[9] Di Cui. Early detection of flawed structural dependencies during software evolution. *IEEE Access*, 9:28856–28871, 2021.

[10] Xin Xia, Xiaozhen Zhou, David Lo, and Xiaoqiong Zhao. An empirical study of bugs in software build systems. In *2013 13th International Conference on Quality Software*, pages 200–203, 2013.

[11] Bram Adams. Co-evolution of source code and the build system. In *2009 IEEE International Conference on Software Maintenance*, pages 461–464, 2009.

[12] Fabian Oraze Christian Macho and Martin Pinzger. Dvalidator: An approach for validating dependencies in build configurations. *J. Syst. Softw.*, 209(C), March 2024.

[13] Maven apache guides: Introduction to the pom. https://-maven.apache.org/guides/introduction/introduction-to-the-pom.html, Feb 2009.

[14] Casimir De´sarmeaux, Andrea Pecatikov, and Shane McIntosh. The dispersion of build maintenance activity across maven lifecycle phases. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 492–495, 2016.

[15] Mockito framework site. https://site.mockito.org/.

[16] Gradle documentation: Dependency management in gradle. https://docs.gradle.org/current/userguide.

[17] Gradle documentation: Declaring dependencies: https://docs.gradle.org/current/userguide.

[18] Eric Redmond. Pom reference – maven. https://maven.apache.org/pom.htmlmaven_coordinates, Dec 2019.

[19] Oracle java documentation: Using package members. https://docs.oracle.com/javase/tutorial/java/package/usepkgs.html.

[20] Hanyu Zhang. *A comprehensive approach for software dependency resolution*. Library and Archives Canada = Bibliothèque et Archives Canada, 2012.

[21] Open source initiative: Open source software definition. https://opensource.org/osd.

[22] Arnoud Engelfriet. Choosing an open source license. *IEEE Software*, 27(1):48–49, 2010.

[23] Patricia Lago Andrea Capiluppi and Maurizio Morisio. Characteristics of open source projects. In *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.*, pages 317– 327, 04 2003.

[24] Richard Stallman. License compatibility and relicensing - gnu project - free software foundation. https://www.gnu.org/licenses/license-compatibility.en.html.

[25] Andrew Morin, Jennifer Urban, and Piotr Sliz. A quick guide to software licensing for the scientist-programmer.

[26] Carnegie mellon university software engineering institute: Security vulnerabilities. https://www.sei.cmu.edu/our-work/security-vulnerabilities/.

[27] Antonios Gkortzis, Daniel Feitosa, and Diomidis Spinellis. Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities. *Journal of Systems and Software*, 172:110653, 2021.

[28] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. A qualitative study of dependency management and its security implications. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1513–1531, New York, NY, USA, 2020. Association for Computing Machinery.

[29] Wentao Wang, Faryn Dumont, Nan Niu, and Glen Horton. Detecting software security vulnerabilities via requirements dependency analysis. *IEEE Transactions on Software Engineering*, 48(5):1665–1675, 2022.

[30] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. Preliminary findings on foss dependencies and security: A qualitative study on developers' attitudes and experience. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, ICSE '20, page 284–285, New York, NY, USA, 2020. Association for Computing Machinery.

[31] Viet Hung Nguyen and Le Minh Sang Tran. Predicting vulnerable software components with dependency graphs. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, MetriSec '10, New York, NY, USA, 2010. Association for Computing Machinery.

[32] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018.

[33] Daniel A. Almeida, Gail C. Murphy, Greg Wilson, and Mike Hoye. Do software developers understand open source licenses? In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 1–11, 2017.

[34] João Pedro Moraes, Ivanilton Polato, Igor Wiese, Filipe Saraiva, and Gustavo Pinto. From one to hundreds: Multi-licensing in the javascript ecosystem. *Empirical Software Engineering*, 26(3), 2021.

[35] Christopher Vendome, Daniel M. German, Massimiliano Di Penta, Gabriele Bavota, Mario Linares-Vásquez, and Denys Poshyvanyk. To distribute or not to distribute? *Proceedings of the 40th International Conference on Software Engineering*, 2018.

[36] Germán Márquez, José A. Galindo, Ángel Jesús Varela-Vaca, María Teresa López, and David Benavides. Advisory: Vulnerability analysis in software development project dependencies. *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume B*, 2022.

[37] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Softw. Engg.*, 23(1):384–417, February 2018.

[38] Sander van der Burg, Eelco Dolstra, Shane McIntosh, Julius Davies, Daniel M. German, and Armijn Hemel. Tracing software build processes to uncover license compliance inconsistencies. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 731–742, New York, NY, USA, 2014. Association for Computing Machinery.

[39] Christopher Vendome. Assisting developers with license compliance. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 811–814, 2016.

[40] Colin M. Gohmann. Finding library substitutions for open-source license violations. Master's thesis, Miami University, 2023. Available through OhioLINK Electronic Theses and Dissertations Center.

[41] Vikram Nitin, Shubhi Asthana, Baishakhi Ray, and Rahul Krishna. Cargo: Ai-guided dependency analysis for migrating monolithic applications to microservices architecture. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

[42] Brij Kishore Pandey, Ajay Tanikonda, Sudhakar Reddy Peddinti, and Subba Rao Katragadda. Ai-driven methodologies for mitigating technical debt in legacy systems. *Journal of Science & Technology (JST)*, 2(2), April 2021. Available at SSRN: `https://ssrn.com/abstract=5101827` or `http://dx.doi.org/10.2139/ssrn.5101827`.

[43] Owasp dependency-check documentation. https://jeremylong.github.io/dependencycheck/index.html.

[44] Libraries.io com.google.guava:guava documentation: https://libraries.io/maven/com.google.guava:guava.

[45] Alexandre Adutra. Adutra/maven-dependency-tree-parser: https://github.com/adutra/maven-dependency-tree-parser.

[46] Andy J Duncan. Andyjduncan/gradle-example: A small example gradle project: https://github.com/andyjduncan/gradle-example.

[47] org.eclipse.jdt.core.dom astparser documentation: https://www.ibm.com/docs, Aug 2014.

[48] org.eclipse.jdt.core.dom astvisitor documentation: https://www.ibm.com/docs, Aug 2014.

[49] Stefan Steiger. Ststeiger/procyon: Procyon java decompiler.

[50] Vinland-Technology. Flict/examples.md at main · vinland-technology/flict.

[51] Libraries.io open data: https://libraries.io/data.

[52] The software heritage archive: https://archive.softwareheritage.org/.

[53] Scancode toolkit reference documentation. https://scancode-toolkit.readthedocs.io/en/stable/reference/index.html.

ProQuest Number: 32063510

INFORMATION TO ALL USERS
The quality and completeness of this reproduction is dependent on the quality
and completeness of the copy made available to ProQuest.

Distributed by
ProQuest LLC a part of Clarivate (2025).
Copyright of the Dissertation is held by the Author unless otherwise noted.

This work is protected against unauthorized copying under Title 17,
United States Code and other applicable copyright laws.

This work may be used in accordance with the terms of the Creative Commons license
or other rights statement, as indicated in the copyright statement or in the metadata
associated with this work. Unless otherwise specified in the copyright statement
or the metadata, all rights are reserved by the copyright holder.

ProQuest LLC
789 East Eisenhower Parkway
Ann Arbor, MI 48108 USA