Call graph analysis: Leveraging public test suites

Version of August 14, 2023

Jingyu Li

Call graph analysis: Leveraging public test suites

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jingyu Li
born in Taiyuan, China

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

©2023 Jingyu Li. All rights reserved.

Call graph analysis: Leveraging public test suites

Author: Jingyu Li
Student id: 5481376

Abstract

Call graphs are useful tools for representing method relationships within software projects and correlations between dependencies. Although static analysis is a prevalent method for call graph construction, it has its limitations such as struggling with handling dynamic features and lambda expressions. In this research, we introduced an approach that utilizes test suites from public Java Maven projects to construct dynamic call graphs and then merge them with static call graphs. Our objective is to explore the efficacy of the merged call graph in uncovering additional information. We employed OPAL and Soot to generate static call graphs and Java agents to trace edges during actual execution. Subsequently, a merging procedure, coupled with a filtering mechanism, was implemented to eliminate duplications. We conducted vulnerability detection analysis to assess the results and a version analysis to investigate the potential for extending our approach by merging multiple versions. Our results indicated that the merged call graph offers a modest increase in edges compared to solely static analysis. Additionally, we discovered that vulnerability identification followed a comparable pattern, supporting the consistency of our methodology. Additionally, we discover that combining several minor/patch versions of a project is a successful tactic for enhancing test coverage. Our research highlights the value of using test suites from open-source projects to build more in-depth call graphs.

Thesis Committee:

Responsible Professor: Prof. Dr. A. van Deursen, Software Engineering, EEMCS
Defense Chair: Dr. Z. Erkin, Cyber Security, EEMCS
University supervisor: Dr. S. Proksch, Software Engineering, EEMCS

Preface

This thesis represents the final milestone in my journey to obtaining my Master's degree. It has been a path of both challenges and rewards, full of enlightening experiences and valuable lessons.

I owe my deepest gratitude to my supervisor, Professor Sebastian Proksch, and Professor Arie van Deurson. Their expertise and unwavering support were invaluable. They not only provide academic guidance but also encouraged me to question, explore, and push my limits. I am thankful for their consistent mentorship and constructive feedback. I must also express my heartfelt thanks to my family and friends. Their unconditional love, unwavering belief in my abilities, and continuous encouragement have been my pillars of strength. Their support during difficult times and celebrations of my successes have been sources of constant motivation. It is their faith in me that made this journey less arduous and more rewarding.

As I approach graduation, I feel a sense of accomplishment and pride. This journey was filled with hard work, perseverance, and continuous learning. There were obstacles along the way, but each challenge became an opportunity for growth, and each failure, a lesson learned. I am excited to bring into practice what I have learned during all those years and to see what the future holds.

Jingyu Li
Delft, the Netherlands
August 14, 2023

Contents

Preface iii

Contents v

List of Figures vii

1 Introduction 1
- 1.1 Research Questions . . . 2
- 1.2 Contribution . . . 3

2 Related Work 5
- 2.1 Static analysis . . . 5
- 2.2 Dynamic analysis . . . 7

3 Methodology 9
- 3.1 Static analysis . . . 9
- 3.2 Dynamic analysis . . . 11
- 3.3 Call Graph Merging . . . 18
- 3.4 Vulnerability analysis . . . 20
- 3.5 Version analysis . . . 21

4 Results and evaluation 25
- 4.1 Data processing . . . 25
- 4.2 Call graph construction . . . 26
- 4.3 Vulnerability detection . . . 30
- 4.4 Version analysis . . . 31

5 Discussion and Future Work 35
- 5.1 Threats to Validity . . . 38

6 Summary 41

Bibliography **43**

A Glossary **47**

List of Figures

3.1 The process of mapping package to Maven coordinate 13
3.2 Dynamic call graph construction 16
3.3 Construction procedure of dependency X dynamic call graph 17
3.4 Call graph construction procedure 19
3.5 Example of vulnerability tracing 21
3.6 Public vulnerable methods propagation and filtering 21
3.7 Version analysis 23

4.1 Visualization example in Neo4j 27
4.2 Edge type distribution in combined call graphs 28
4.3 Contribution of analyses in combined call graphs 29
4.4 Example of vulnerability changes 32

Chapter 1

Introduction

In today's software development ecosystem, dependency management has become a critical aspect given the increasing interdependency of software packages. Developers nowadays lean heavily on external libraries and frameworks to enhance their productivity, allowing them to focus on their application's core functionality while leveraging readily available solutions for common, repetitive tasks.

To reuse existing code and the availability of a large number of open-source libraries, and to speed up the development process, the trend of increasing dependencies emerged. However, the growing complexity and escalating number of these dependencies lead to explosions in the size and complexity of the projects, particularly in understanding the behavior of software, identifying possible vulnerabilities, and ensuring its security.

Faced with these challenges, call graphs (CG) have emerged as a powerful tool. A call graph contains almost all the invocation paths between functions and classes in a project and provides valuable insight into various complex interdependencies. By mapping the control flow between various methods, CG provides a simplified conceptual map of the operation of a software system which provides insight into the structure. Thus, it became the cornerstone of many software analyses. Covering areas as diverse as impact analysis and vulnerability detection, these analyses enhance our understanding of software behavior and help proactively detect and resolve issues.

To construct call graphs, there are mainly two approaches: static and dynamic analysis. Static analysis scans the program's byte code to extract all possible paths, while dynamic analysis traces all the executed paths during runtime. Static analysis tends to be more comprehensive and constructs more complete call graphs. However, it also falls short on dynamic language features and its conservative strategies also produce more false positives and redundant edges in the call graph. Dynamic analysis is more accurate than static analysis because there are no false positives during runtime. But compared to static analysis, it relies entirely on the test suites. The dynamic analysis tracks every method executed within the test suites during runtime. If test suites have low coverage of the project, the dynamic analysis misses most of the edges of a project. In our study, we intend to combine static and dynamic analysis making them complementary.

In an effort to explore the full potential of call graphs in navigating software complexity, our study focuses on public Java Maven projects. These projects, with their Maven unit test

cases, serve as a potentially valuable source of run-time data for call graph construction and augmentation. Moreover, it is possible to replace the test suite generation. We seek to leverage these test suites in constructing and analyzing call graphs that account for both static and dynamic analysis. By exploring this approach, we aim to augment the understanding of software dependencies and explore the possibility of better vulnerability detection.

1.1 Research Questions

Among the programming languages, Java presents compelling use cases for the application of call graphs. Given the intricate dynamic nature of Java and its rich ecosystem of libraries and frameworks, creating a comprehensive CG that encapsulates both static and dynamic behavior is particularly challenging, but essential. In this study, we explore the following research questions:

- The Impact of Public Test Suites on Call Graphs: How do public test suites in open-source projects contribute to the construction of call graphs, specifically in terms of dynamic analysis? How much additional information can they provide, and under what conditions is this information most valuable?
- Vulnerability Detection: How does the inclusion of dynamic analysis, supplemented by public test suites, enhance the detection of vulnerabilities compared to static analysis alone? Does the combination of static and dynamic analysis offer significant advantages in vulnerability detection and if so, under what circumstances?
- Effect of Version Analysis: How does the merging of call graphs from different versions of software and the subsequent optimization of these graphs affect the comprehensiveness and accuracy of software analysis? What kind of additional information can be extracted through this process, and how does it contribute to a more efficient software analysis?

For the first research question, we download Maven projects from GitHub and analyze their dependencies and package information. Static and dynamic analyses were then performed to construct the call graph. The increase in information content resulting from the integration of these two analyses was carefully evaluated to quantify the contribution of the test suite to the overall call graph. To address the second research question, we identify vulnerabilities in the constructed call graphs. These vulnerabilities are collected from different public sources (e.g., national vulnerability databases) and propagated in the integrated call graph. We then check for the presence of additional vulnerabilities discover through the propagation process that are not initially found in the public sources. Regarding the third research question, we employ a version-based analysis. By selecting one version as a baseline, we merge the call graphs of different versions. This merging process includes the merging of the minor or patch versions, as well as the optimization of the merged graphs. Our goal is to explore the possibility of obtaining edges from other versions that were undetected in the base version, specifically from their test suites.

1.2 Contribution

Our study focuses on the interplay of static and dynamic analysis methods in constructing call graphs for software analysis. While both approaches have their distinct advantages, their combination provides a more comprehensive understanding of the system. However, given the resource intensity of dynamic analysis, we advocate its use primarily in projects with extensive dynamic behavior.

The following is a summary of the main findings:

1. Complementarity of Static and Dynamic Analysis: Static and dynamic analyses play complementary roles in assessing software systems, with static analysis providing the primary evaluation and dynamic analysis supplementing it. Dynamic analysis cannot fully replace static analysis due to its limitations in test coverage and efficiency. Despite drawing information primarily from official repositories test suites, dynamic analysis effectively utilizes test resources to provide extra information, aiding in the exploration of certain unprobed methods. The value of dynamic analysis is project-specific. It offers significant extra information in IO-intensive projects and those heavily utilizing Java's dynamic features, but its value is limited in projects not exploiting these features.

2. Vulnerability Detection: Our research shows that combining static and dynamic analysis can help identify more vulnerabilities.

3. Version Analysis: Merging different versions of the project leads to a marginally more comprehensive and refined call graph. However, we also discovered the same trend as in the dynamic efficacy that the effectiveness and applicability of this approach may vary widely among different projects due to the usage of Java dynamic features.

Following this introduction, the structure of the remainder of this paper is as follows: In Chapter 2, we provide an overview of existing related work, setting the foundation and context for our work. In Chapter 3, we outline our research methodology, detailing the approaches and techniques used to investigate our research questions. In Chapter 4, we present our research results, followed by a thorough analysis of the results. In Chapter 5, we discussed the analysis results and future Work and the implications of our findings which suggests directions for future research based on our results. In Chapter 6, we summarize our work.

Chapter 2

Related Work

Call graphs represent the structure of a software program. They are directed graphs with nodes being methods and edges representing the method calls. With these edges, analysis can be conducted to track the invocation paths within the component or the entire software. The graphical representation of a program is the basis of many software analysis tasks like vulnerability analysis, control flow analysis, and refactoring[19, 40]. The construction and analysis of call graphs have been a subject of considerable interest in software engineering ever since 1979[31]. Call graph construction is not a new concept in computer science. The goal is to build accurate and sound call graphs. There are 2 ways of constructing call graphs: static analysis and dynamic analysis. Static analysis is considered sound and conservative and dynamic analysis is considered accurate however incomprehension. Although state-of-the-art approaches are relatively comprehensive when it comes to call graph generation, they still cannot confidently say whether a branch in the software or its dependency is reached or not. Which makes the call graph generation problem undetermined[9].

In this chapter, we delve into the concept of call graphs, discussing both static and dynamic analyses employed for their construction. We then summarize the current state of call graph analysis in contemporary software engineering research and practice.

2.1 Static analysis

A large number of research has been conducted on call graph construction. Many algorithms have been developed and proven effective and efficient while making trade-offs between precision and time. Tip and Palsberg[37] carried out an analysis where they compared many of the widely used popular algorithms. Notably, Class Hierarchy Analysis (CHA)[15] is recognized as a fast yet lower-precision algorithm that utilizes class hierarchy data for resolving virtual methods. Additionally, the Rapid Type Analysis (RTA)[8] algorithm takes into account allocation sites. On the other hand, Variable Type Analysis (VTA)[34] handles the assignments between various variables by formulating subset constraints.

Owing to their unique prowess in call graph construction, Soot, WALA, and OPAL are distinguished frontrunners in the realm of static analysis tools.

Soot, a framework developed by McGill University, operates as a Java optimization

framework. It's equipped with a suite of transformation algorithms and analytical tools. Soot streamlines optimization and analysis by translating Java bytecode into various intermediate representations, Jimple being a notable example[39]. This tool further extends its utility by facilitating easy navigation and manipulation of built-in call graphs, backed by versatile algorithms such as Class Hierarchy Analysis (CHA), Rapid Type Analysis (RTA), and certain context-sensitive pointer analysis variants.

Originating from IBM, the T.J. Watson Libraries for Analysis (WALA)[5] provides a toolkit that empowers users to craft, scrutinize, and alter call graphs. WALA, with its innate capability to support a spectrum of programming languages including Java and JavaScript, generates call graphs underpinned by pointer analysis. A notable feature of WALA is its adaptable pointer analysis, encompassing both object and context sensitivity. Its modular and extensible architecture renders WALA a versatile instrument for diverse static analysis endeavors.

Lastly, the OPAL Project[16] presents a dedicated framework tailored for the static analysis of Java bytecode. The hallmark of OPAL is its unwavering emphasis on precision, dependability, and efficiency. OPAL's robust pointer alias analysis is both comprehensive and scalable, aiding in the formulation of sophisticated call graphs. Moreover, its declarative analysis specification is a standout feature, granting analyses heightened transparency and simplifying their configuration and extension.

2.1.1 limitations

However, the static analysis could not precisely depict the structure of the actual call graph. Due to its conservative nature, the static analysis approximates every possible edge which introduces many false positives. Ever Since JDK v1.3, dynamic features like Reflection API[3] are available. Now they are widely used to instantiate classes and transform executions during run-time. Considering that dynamic approaches rely on workloads to run the program under study, they are intrinsically flawed. These workloads won't cover every potential execution route for real-world apps. It turns out that most static evaluations are flawed because Java programs frequently use dynamic features. These dynamic properties are notoriously challenging to represent.

We chose Soot and OPAL for this study for a few compelling reasons. First off, their analyses of Java bytecode are both thorough and fast, which fits the focus of our investigation. Soot offers a variety of call graph construction algorithms, allowing us to choose the one that best suits our requirements. Our analysis is also made easier by its capacity to convert Java bytecode into condensed intermediate representations. On the other hand, OPAL's emphasis on accuracy and performance enables us to create call graphs that are incredibly precise. We have more control over the analysis process thanks to its declarative specification of analyses. By combining these tools, we can build a thorough static call graph that serves as a solid starting point for our further investigation of dynamic analysis and their combination.

2.2 Dynamic analysis

Dynamic analysis constructs a dynamic call graph by running the program and observing its runtime behaviors. Unlike static analysis, it is highly precise and doesn't require approximation. However, its speed is typically slower, as it hinges on the program's execution time, which is usually much longer than the time required for static analysis.

Dynamic call graphs present the actual execution paths and are generally more compact than static call graphs. Although they ideally represent subgraphs of static call graphs, they can offer valuable insights into dynamic features that enrich the static analysis. In Java, there are primarily two techniques for call graph generation: bytecode-level instrumentation and source code-level instrumentation[36]. Both methods involve inserting probes into the source code or bytecode to monitor executioncite[24]. Java agent is a well-known tool for performing bytecode-level operations. As a part of the Java Instrumentation API[4], it aids in the instrumentation of Java Virtual Machine (JVM)-based programs, featuring the 'premain' method and specific attributes outlined in its manifest. Upon the initialization of JVM, the 'premain' method oversees the startup process. It loads the method bodies and user-initiated property alterations into the JVM during this phase. Owing to its bytecode instrumentation ability, Java Agent facilitates several functionalities, including Aspect-Oriented Programming (AOP), profiling, mutation testing, and some forms of monitoring.

Unlike static call graphs, dynamic call graphs depict the interplay between dynamic behavior—representing specific program activities—and program structure, which outlines how these activities are implemented[41]. The value of dynamic analysis is heavily dependent on the quality and quantity of test cases. Well-structured and selected test cases can reveal more undetected branches and invocations in static analysis. Greater test coverage enables more invocation tracking, thus enhancing the comprehensiveness of the dynamic call graph. One popular approach to increase test coverage is the test case generation from static call graphs. Arcuri, Andrea, and Fraser et al. used bytecode instrumentation to automatically separate code from its environmental dependencies and extended the EvoSuite Java test generation tool to explicitly set the environment state in the sequences of calls[7]. They observed a significant increase in test coverage to over 80% or even 90%. Parasoft Jtest[6] generates Java unit test cases to supplement the manually written test suite. Various other studies have shown that dynamic analysis is very effective when the test coverage is high enough. The study also revealed that dynamic analysis uncovers a significant number of false negatives in static analysis, indicating that hybrid techniques can be very effective.

However, some test suite generation tools have their own limitations. Some test cases are randomly generated, which results in very poor overall coverage. Some of them even fail to execute the program[17]. In [21], the authors discovered the possibility of augmenting the existing test cases by generating guessed test cases to test the accuracy of detected invariants. Test generation is expensive, and the study demonstrated that it is not as effective in discovering dynamic program behavior as manually written tests. While state-of-the-art analysis with reflection support can significantly improve recall, its high cost renders it impractical for many applications. Multiple other researchers also revealed the same issue[14, 29]. Li and Jens et. al investigated the challenges static analyses face

when attempting to model dynamic language features soundly while maintaining acceptable precision[33]. The authors conducted a study on 31 real-world Java programs using an oracle of actual program behavior recorded from executions of built-in and synthesized test cases with high coverage. Unlike these studies, we propose a new approach to utilize the test suites in existing public projects to generate dynamic call graphs. We expect that this approach would be more efficient to cover more execution paths and generate a more complete call graph.

Chapter 3

Methodology

The design of call graph combination and version analysis is discussed in this chapter. The three sections in this chapter are listed below:

1. Call graph combination. Combine call graphs from dynamic analysis and static analysis using multiple different tools to increase edge coverage.

2. Vulnerability analysis. Find vulnerabilities in the call graph and compare the results with pure static or dynamic analysis.

3. Version analysis. Merge several versions of the project to acquire higher test coverage for dynamic analysis.

In the following 2 sections, we introduce the utilized methodology to answer the RQ1. We present the data collection and processing strategy for static and dynamic analysis to combine their call graphs. In addition, we identify projects with informative test suites and construct partial call graphs derived from the invocation paths. These partial call graphs are then merged to form a comprehensive call graph for the dependency. In Sections 3.4 and 3.5, we introduced the procedure of vulnerability detection and version analysis.

3.1 Static analysis

Static analysis plays a vital role in understanding the structure and behavior of software. The information from static analysis enables the identification of potential vulnerabilities, optimization opportunities, and architectural improvements. In many studies, static analysis has been proven to be the most informative approach regarding software analysis such as program comprehension, performance analysis, and security assessment.

In our study, the static analysis tools scan the JAR files obtained from Maven Central[28]. It is a public repository for Java libraries commonly accessed by Java developers to acquire multiple types of compiled dependencies. During the process, JAR files with various dependencies are carefully chosen and examined. We scanned all of the packages in the JAR files and identified the classes and methods within them. The extraction of call graphs and other pertinent data for static analysis was then conducted. We found that the JAR files larger

than 10 MB were too extensive for our analysis. They were excluded from the analysis due to time and resource limitations for a variety of reasons:

- Large JAR files require significantly more computational resources and time, which impacts the overall efficiency of the analysis.
- The complexity of the call graphs generated from larger JAR files may render them challenging to interpret and analyze.
- Excluding large JAR files helps maintain a manageable dataset size, enabling more in-depth analysis of the remaining projects within the available resources and time limit.

In our study, we also created a mapping between the jar file and its Maven coordinate to enable static and dynamic analysis and to create nodes in graph databases, which is introduced in the processing later in this chapter.

3.1.1 Static Analysis Tools

Soot and OPAL are powerful and versatile static analysis frameworks for Java byte code. Both provide a comprehensive range of analyses, including points-to analysis, control-flow analysis, and call graph construction. In particular, we are using fasten project from the Delft University of Technology[13]. It is a tool for generating call graphs using OPAL call graph generator version '3.0.0'. It calls the OPAL framework to generate call graphs and store them inside a customized data structure. OPAL loads the Java bytecode before starting the abstract interpretation process. The program's execution is simulated by the abstract interpretation, but instead of using real inputs, it makes use of abstract values that stand in for sets of potential inputs. This tool can also merge the resulting call graphs with their dependencies which helps to acquire more information on the dependency we are analyzing. In Soot, the process begins by loading the program's Java bytecode for analysis. The code is then simplified and immediately represented by Jimple by Soot, making it simpler for tools to understand and work with. In the call graph, each node denotes a method, and each edge denotes a method call. Multiple edges may be added from the call site to each potential target method if dynamic dispatch makes it impossible to pinpoint the actual target method of an invocation. Furthermore, Soot uses iterative refinement to increase the call graph's precision. The call graph is progressively updated and improved during this process depending on the findings of additional Points-to Analyses and other analyses. The program can then be subjected to various downstream analyses or modifications using the generated call graph. Control flow analysis, data flow analysis, call graph analysis, and many other types of studies are all supported by Soot out of the box. Although neither tool is capable of handling all dynamic features properly, OPAL's use of abstract interpretation enables it to handle them more accurately than Soot. OPAL offers a less complicated intermediate representation, which offers several distinct intermediate representations for Java bytecode.

However, it is important to acknowledge that both Soot and OPAL, like any other static analysis tools, have their limitations and weaknesses. Soot may struggle with handling incomplete library code, which can adversely affect the precision of call graph construction.

On the other hand, OPAL is criticized for its limited support for higher language-level features. These constraints inevitably impact the quality of the call graph constructed. To offset these limitations and harness the strengths of each tool, we integrated the results of Soot and OPAL for static call graph analysis. This integration allowed our study to offer a more robust and comprehensive understanding of the structure and behavior of dependencies. We were also able to cross-validate the call graph construction by using two different static analysis tools. If the call graphs generated by Soot and OPAL were similar, it gave us more confidence that our analysis was accurate and correct.

3.1.2 Strategy

In our research, we aim to perform a thorough and accurate static analysis of the selected JAR files by combining the results of Soot and OPAL. We discovered that the complexity of analyzing larger JAR files increases dramatically. The number of edges and dependencies within these files has increased to the point where the analysis cannot be completed within a reasonable time limit. As a result, we decided to skip JAR files larger than 10 MB from our analysis. Furthermore, larger JAR files frequently contain a higher percentage of irrelevant or infrequently used code. This may result in a lower actual edge-to-noise ratio in the analysis results, lowering the value of the resulting data. To ensure consistency in call graph generation and combination, we generated call graphs using the Class Hierarchy Analysis (CHA) available in both tools. The static analysis results serve as a foundation for subsequent sections that address the research questions posed in the introduction. This lays a strong foundation for understanding software behavior and identifying areas where dynamic analysis can supplement static analysis.

3.2 Dynamic analysis

Although static analysis already constructs complex call graphs, they are still considered over-approximation to the actual call graph. This means there are still a lot of the reachable methods missing. The dynamic call graph is a record of the runtime execution of a program, they are built entirely different from the static call graphs[32]. Unlike static analysis's nature of scanning jar files to directly construct call graphs, the dynamic analysis uses test suites to trace actual call executions in practical executions, which provides more insight into the actual executions of the project. Static analysis may face challenges when analyzing Java programs due to:

- Dynamic proxy & Polymorphism: Precise method calls may be hard to infer during inheritance or interface-implementation relationships, as actual object types in Java are determined at runtime.
- Reflection: Reflection allows inspection of classes, interfaces, fields, and methods at runtime, complicating static analysis due to its unpredictability at compile-time.
- Dynamic Loading & Dynamic Proxy: Classes and methods dynamically loaded and invoked at runtime can obscure the foresight of static analysis.

- Lambda Expressions & Functional Interfaces: Introduced in Java 8, these constructs can alter the code control flow, presenting a challenge to static analysis.

To counter these challenges, we introduce dynamic analysis in our project.

In this section, we present the steps and frameworks taken to perform the dynamic analysis on the processed projects we mentioned earlier.

3.2.1 Leveraging Public Projects for Dynamic Analysis

One of our key research questions is the utilization of public projects to construct dynamic call graphs. We intend to extract essential call patterns and edges from the test suites of these projects. After that, we combine the partial call graphs to cover more essential paths in the dependency library. To ensure a diverse and representative sample of Maven projects, data was gathered from various open-source repositories hosted on GitHub. We collected them and classified them into two types:

official projects: Official projects are the projects developed and maintained by the dependency's authors. It typically contains the source code, documentation, and other necessary resources of the dependency. They represent the core functionality(internal calls) and features provided by the dependency. Official projects generally have more comprehensive test suites with sufficient coverage of the methods and classes.

Client projects: These projects rely on the project we are analyzing and integrate the dependency into their codebase to leverage its features. Client projects show how the library is used in real-world scenarios and can help identify common patterns.

We intend to compare the edges from both to give suggestions for future analysis. While collecting projects, we took multiple factors into consideration, such as project size and complexity. Some projects with multiple modules or costs For the test suite, we aim to get as higher test coverage as possible. The library projects offer more comprehensive internal test suites and the test suites from client projects are informative as well. In order to extract useful projects and obtain test suites, the data collection process was guided by the following criteria:

- Collect the compilation of all Java Maven projects on GitHub from the specified period (2009 to 2023).
- Exclude uninformative projects lacking test cases, only projects with more than 10 stars were considered.
- Filter Maven projects by scanning for the presence of a pom.xml file in their web pages, verifying their usage of Maven as the build tool.

We found that GitHub has a 1000-result limit for a single GitHub API search[2], while the number of projects we need for the analysis is well above 10000. We decided to search the dependencies month by month and accumulate all pertinent projects. We also tried to use the existing GitHub activity dataset on Google Cloud[12]. However, we encountered the challenge of extracting Maven projects from the dataset. As a result, we decided to rely on the GitHub API to search to collect all of the projects with 5 to 10 stars but eventually

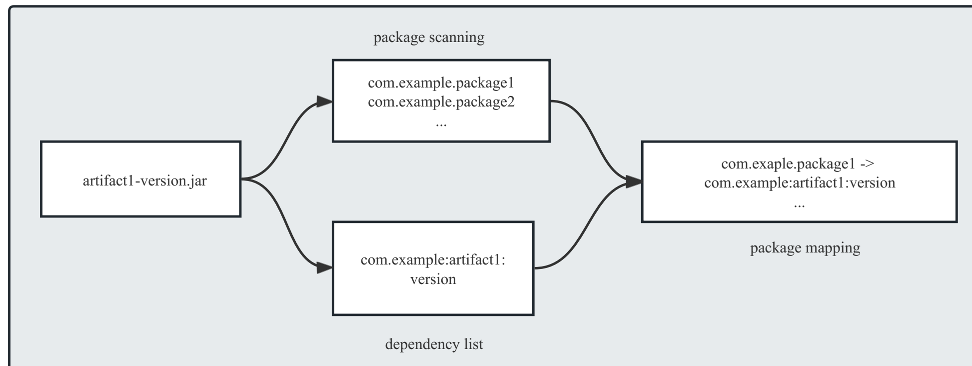


Figure 3.1: The process of mapping package to Maven coordinate

found a large number of them lack basic maven test cases and were not suitable for our study.

3.2.2 Data Processing

Although in the dynamic analysis method call information is collected at run-time at the JVM level, this data we collected does not provide direct insight into the usage patterns of dependencies, as it does not correlate with the specific dependencies employed in the project. Similarly, in the static analysis, scanning the methods in the source code does not necessarily correlate with the dependency either.

After cloning the collected projects, we extracted all dependencies into a dedicated library folder using the Maven command ”dependency:copy-dependencies”. The classes in the JAR files are extracted to build the package mapping for each dependency. These mappings detail the relationship between the JAR files and their respective packages. Since all dependency JAR files are retrieved from Maven Central, they adhere to the standard naming convention[20](artifactId-version.jar). Therefore, Maven coordinate information can be derived from the filenames, facilitating the construction of a comprehensive JAR-to-coordinate mapping. We found that although most dependencies follow the naming convention suggested by Maven, packages and jar files are not necessarily named the same as their coordinates (e.g., for the Gson project, the package name is com.google.gson but the coordinates are com.google.code.gson) which leads to problems with the correspondence of dependencies and packages.

Therefore, we combined the mappings (JAR-to-packages and JAR-to-coordinate) mentioned above to construct a package-coordinate mapping as shown in 3.2. All the mappings are stored by version in the Redis server. We first store the call graph of each project using their names as the key. Then the partial graphs of its dependencies are extracted and merged into the dependency call graph.

3.2.3 Dynamic analysis setup

In this subsection, we outline the steps taken to perform dynamic analysis on GitHub projects mentioned earlier.

Dependency resolution

As mentioned earlier in the data processing section, we have collected more than 28,000 projects and analyzed their dependency usage. To proceed with the dynamic analysis, we first collected projects that are using the version of the dependency needed. We use the Maven command ”dependency:copy-dependencies” to copy all the dependencies into a library folder so that we can process the dependencies of the projects and construct mapping as mentioned earlier.

Java Agent

In dynamic analysis, the Java agent is a bytecode-based instrumentation approach that is commonly used. It is inserted into the Java bytecode as a probe to actively detect all related methods[25]. The other method is the source code approach which does not depend on the JVM but requires a separate build. Research done by Horvath proved that with filtering mechanisms, the difference between them is negligible[23]. So in our approach, we utilized the capabilities of two popular byte code manipulation libraries, Byte Buddy[1] and Javassist, to develop a Java agent that intercepts method calls throughout the project execution processes. In live executions, there are multiple types of Java dynamic features to be intercepted, such as reflection calls, proxy methods, and dynamic invocation. Such features are overlooked in the static analysis and are captured and recorded in our dynamic analysis.

Byte Buddy is written on top of ASM, a mature and well-tested library for reading and writing compiled Java classes. Byte Buddy deliberately exposes the ASM API to its users in order to support complicated type manipulations, so that a Byte Buddy user is not limited to its higher-level functionality but can easily implement custom implementations when necessary. For ByteBuddy, the Java agent structure is outlined as follows:

As we are using Java agents, the premain method is an essential part of the Java instrumentation framework and provides a powerful mechanism for the dynamic analysis of Java programs. The extracted packages are configured in the element matcher to avoid unneeded call sites and stack overflow errors caused by extensive method calls in memory. The ByteBuddy agent is then installed and configured in the Agent Builder to transform the matching classes. The transformation encompasses applying the method tracing sub-class to all methods within the selected classes. The probes are inserted into the bytecode from here as an entry point. On method invocation, the probe detects method executions and returns the corresponding signature and information. They are then resolved into package name, class name, method name, parameters, return type, and access modifier. Subsequently, the analyzed method is pushed into a stack as the caller method. The later recorded method will capture the method on top of the stack so that edges between the ”caller” and ”callee” methods can be established during run time. These calls combined together are deemed to be the paths of the execution of calls between various components in the dependencies. Upon

exiting a method or an exception thrown inside a method, ByteBuddy captures the method signatures to inform the termination of the execution path. Removing the top method from the thread-local call stack helps the call stack remain consistent during method calls.

During the analysis process, we also designed a filter to facilitate the filtering process, ensuring the edges we captured during the analysis are valid. All test-related classes, classes with names containing ”test”, and all calls originating from test cases and test frameworks (e.g., JUnit, TestNG) are excluded from the call graphs, as they are not a part of the core functionalities of the dependency and are considered less valuable for call graph analysis. Calls from the native Java framework(e.g., rt.jar before Java 1.8 or jrt-fs.jar after) are also excluded since the majority of them are stable method calls, which would not enhance the information density needed in the study, as most of them hold little significance in terms of analyzing the actual call graph.

```
void enhanceMethod(Method method, String clazzName) {
packageName, className, methodName =
    extractClassAndMethodInfo(method, clazzName)
method.insertBefore("pushMethodCallInfoToStack(packageName,␣
    className,␣methodName)")
method.insertAfter("popMethodCallInfoFromStack()")
}
```

Listing 3.1: Javassist Method Call Tracing Pseudocode

Javassist is a byte code engineering library that allows us to modify and manipulate Java classes at run-time which shares the same structure as Byte Buddy. The standard reflection API of Java does not allow users to alter program behavior. In order to overcome this limitation, Javassist extended the reflection API[10], which enables structural reflection in Java as opposed to other extensions that only enable behavioral reflection. It also allows structural reflection before a class is loaded into the JVM and prevents a performance issue. They also included a custom compiler to improve the performance of reflective architecture in their later work[11].

Both tools offer comprehensive support for handling dynamic features in Java executions. Dynamic instances and delegate methods in dynamic invocations, and instrumentation of classes in reflection calls are captured using multiple APIs so that the corresponding edges are extracted correctly. In contrast with Byte Buddy, Javassit requires the byte code manipulated and method calls recorded using instrumentation and a transformer. As shown in Listing 3.1, Javassist requires the manual insertion of custom byte code snippets at the beginning and end of the methods using the insertBefore and insertAfter methods. Which calls the designated methods before and after the execution of the scanned methods.

Configuration

We made several other configurations to improve our dynamic analysis process in addition to the Java agent configuration we previously discussed. This was accomplished by modify-

Table 3.1: Dynamic analysis parameters configuration

Parameter	Value
Java agent	Packages specified in analysis
Multi-threaded test execution	Enabled
Parallel level	Method
TestFailureIgnore	True
ParallelTestTimeout	10 minutes
ReuseForks	True
SkipIntegrationTests	True

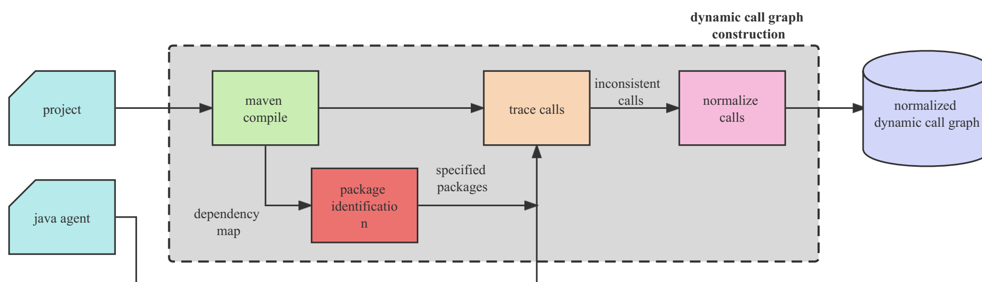


Figure 3.2: Dynamic call graph construction

ing the project's POM files in the Maven project(including all pom.xml files in the project's path) to ensure full coverage of the Java agent.

These options are shown in Table 3.1. The packages that needed to be scanned were specifically identified in order to avoid collecting unnecessary data that was irrelevant to our study. This selective scanning process is critical for reducing noise in our collected data and allowing us to focus on dependencies and method executions that are most relevant to the projects under consideration. Another configuration involved reusing forks, which allowed us to share the Java Virtual Machine (JVM) instance across multiple tests. This step is important because it improves performance while decreasing overhead. When dealing with large and complex projects, resource usage efficiency is critical. This configuration also enables a more streamlined and efficient analysis process, reducing disruptions and improving overall workflow.

We made additional changes to the POM settings to ensure optimal information collection within reasonable time constraints. We specifically set it up to allow multi-threaded test execution at the method level and the parallel test timeout to 10 minutes.

We also decided to skip the integration tests. They are generally more time-consuming and less informative than unit tests. We were able to save time and focus on the most relevant and informative data for our research by focusing on unit tests.

Furthermore, we discovered that not all of the projects could be fully compiled without errors. In these cases, we decided to remove all failed test cases and proceed with the remaining test cases. This choice was critical in allowing us to collect as much data from

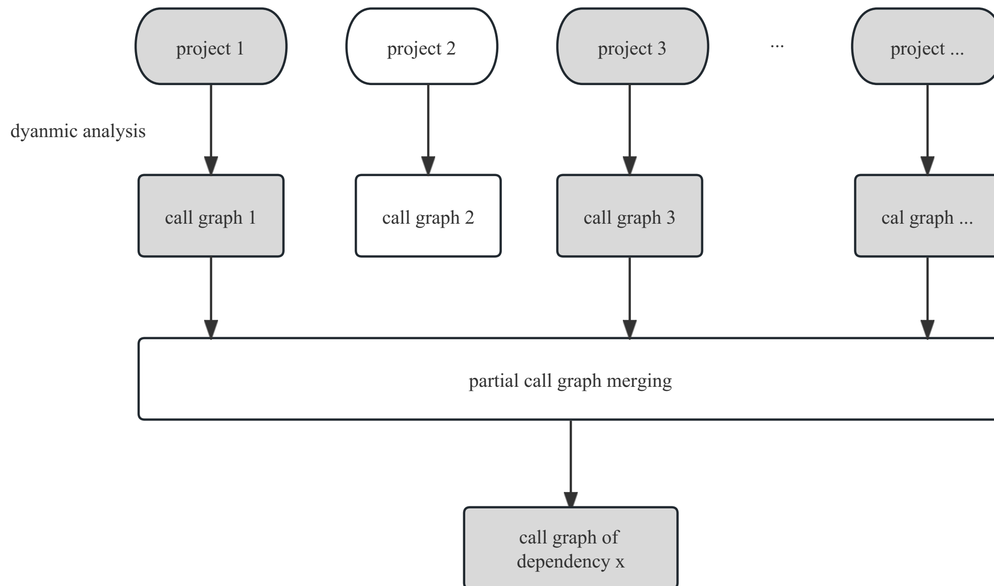


Figure 3.3: Construction procedure of dependency X dynamic call graph

the test suite as possible, regardless of individual test failures. This approach ensures that isolated test failures that may not be significant to our research goals do not have a negative impact on our analysis. Finally, the tag releases frequently indicate stable versions or significant project milestones. They mark a snapshot of the codebase of a version. So we conduct our dynamic analysis on the most recent tag.

3.2.4 Test Coverage Evaluation

The dynamic analysis depends heavily on test suites. A better and more comprehensive test suites result in more extensive coverage of the dependency methods which generates more edges in the call graph. To assess the comprehensiveness and representativeness of our dynamic analysis, we need to evaluate the test coverage of the dependency. In this subsection, we use JaCoCo to generate a test coverage report after the execution of the test suites. The report includes the coverage of classes, methods, lines, and branches for each package. We then calculate a combined test coverage percentage, which is a weighted average of line and branch coverage, as an overall metric of test coverage.

Using JaCoCo for Test Coverage Analysis

JaCoCo (Java Code Coverage) is a widely-used code coverage tool that provides a reliable method for measuring test coverage in Java projects. We compare the percentage of code executed during the test suite's execution between projects to find patterns and try to conclude its impact on . We integrate it into the projects' build process by configuring the JaCoCo Maven plugin[22]. This integration automatically generates code coverage reports after ex-

ecuting the test suites. We then extracted the dependency methods that were executed in the project from the reports to determine the dependency coverage for each dependency, focusing on the following coverage metrics:

- Line Coverage: The percentage of executed lines of code during the test suite's execution.
- Branch Coverage: The percentage of executed branches (e.g., if-else statements) during the test suite's execution.
- Method Coverage: The percentage of executed methods during the test suite's execution.

Combining Test Coverage and Analysis Results

To demonstrate the overall test coverage of our dynamic analysis, we combine the test coverage results of the individual projects. For projects with an official GitHub repository, we also include their test coverage results in our analysis(in which test cases are often more comprehensive). This aggregated coverage percentage provides a comprehensive view of how much our analysis covers the dependency's functionality.

The combined test coverage percentage obtained from the JaCoCo analysis is a strong indicator of the coverage's validity and comprehensiveness in our research. It highlights the effectiveness of our dynamic analysis in capturing method call relationships and dependencies between various components. With a high coverage percentage(more than 95% in this study), we gain assurance that our dynamic analysis results provide valuable insights into the dependency's behavior, and build complete dynamic call graphs, which ultimately contribute to more effective dependency management and vulnerability detection.

3.3 Call Graph Merging

Typical call graphs from dynamic and static analysis tend to be significantly huge with thousands of nodes and edges. It is not possible for manual de-duplication or comparison. Several algorithms designed to compare large call graphs have been proposed in recent years. However, due to the diverse approaches we utilize to generate the call graph, these methods are not directly applicable to our call graphs. The preprocessing of the call graphs generated by both the static and dynamic analysis tools is critical to our research. The goal is to establish consistency between the edges produced by the various types of analyses. As a result, we developed a merging procedure to pair nodes and harmonize the format of call graphs from all the analyzer tools.

Soot and OPAL each have their own way of representing Java method byte codes. We noticed that type descriptors were used to represent parameters and return types on occasion (for example, "[L" for arrays of objects and "[C" for arrays of char types). Similarly, Byte Buddy uses the same type descriptors during execution. However, Javassit sticks to the full name of each type which causes inconsistencies between the result of different tools. We parsed all types and arguments into their corresponding Java-type representations (like

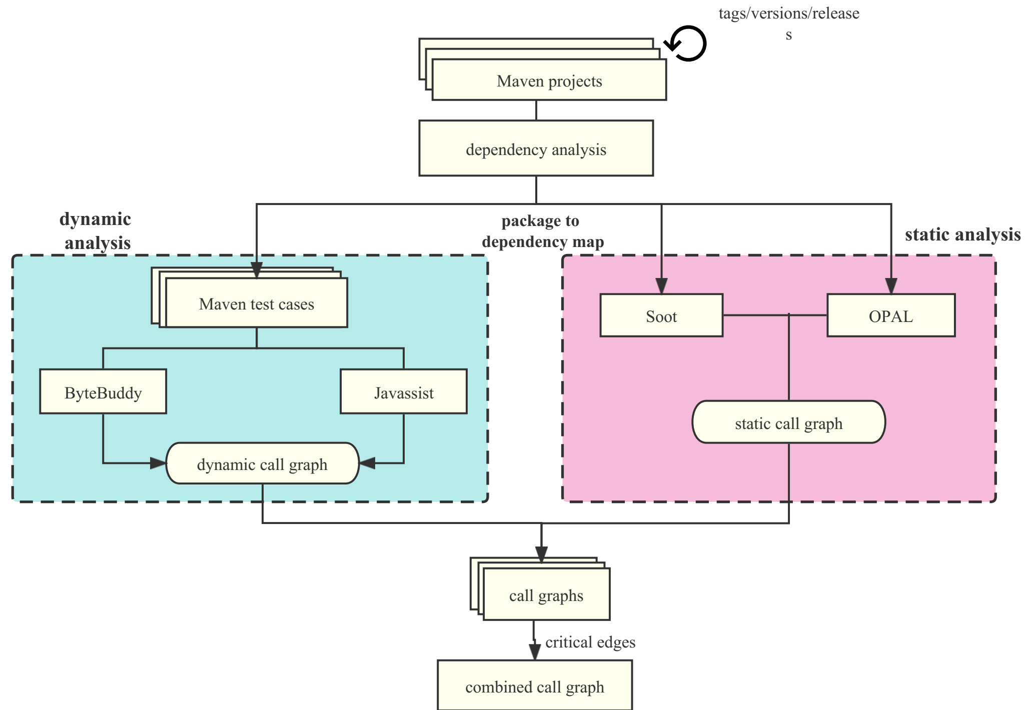


Figure 3.4: Call graph construction procedure

int, char[]) at each node to facilitate edge de-duplication in the call graph and to ensure uniformity across all return types and method arguments. Nonsensical edges produced by the static and dynamic analyses were also removed at this stage. The integration of dynamic and static analyses is essential for our methodology. This allows us to build a more complete and accurate picture of the behaviors of method invocations within the dependencies we're investigating. Distinct edges from each analysis are tagged 'dynamic' and 'static', while Edges that appear in both analyses are tagged as "both". The 'both' edges clearly mark the areas where dynamic and static analysis overlap.

To store the generated call graphs, non-relational databases (NoSQL) are a highly suitable choice compared to MySQL and other relational databases. Owing to their rapid solution deployment and scalable nature, NoSQL databases tend to be more efficient for this purpose [35]. MongoDB was chosen to be the graph database in this study for its document-oriented feature. It stores data as JSON-like documents with dynamic schemas (the format is called BSON). Traditional relational databases may struggle with complex, multi-level joins that are often required when working with graph data. In contrast, MongoDB can store related data in a single document structure. This can help to simplify the queries and improve overall performance [26]. For visualization and graph analysis, we transferred the data from MongoDB to the Neo4j database. Given that the storage formats for nodes and edges differ substantially between these two databases, which requires careful parsing and reorganizing.

Dynamic analysis, utilizing open-source projects, reveals connections between methods in the call graph generation process. Nevertheless, this method frequently duplicates nodes and edges. MongoDB's unique indexing feature was used to address these redundancies. A composite key using the combination of package name, class name, method name, parameters, return type, and access specifiers is used to define each method. Following this systematic and organized process, we perform the call graph merging operation, deduplicate existing edges from the examined projects, and identify calls present in both types of analysis. The dynamic analysis results give a thorough understanding of the call graph and highlight how effective dynamic analysis is compared to static analysis. However, in our study, project dependencies aren't taken into consideration in the current analysis, for the goal of our current research and methodology is to examine the viability of this approach.

3.4 Vulnerability analysis

Once the call graphs are constructed, it's important to test them in practice. We selected vulnerability detection as our use case. Our aim is to identify more vulnerabilities and compare them with the vulnerabilities detected only using static call graphs in this study. Most vulnerabilities are concealed in some methods or classes which lead to potential malicious attacks causing security breaches or unauthorized access, even controlling the entire system. Hence, identifying vulnerabilities in a project is essential for its robustness and safety. The spread of security flaws is a common problem in ecosystems. If a library is compromised, all upstream dependencies could be potentially compromised as well[27]. Direct dependencies can update to a safe version right away, but indirect dependencies must wait until the entire chain in between has been updated. In our study, we employed the vulnerability API from FASTEN[18]. With this tool, developers can precisely analyze whether their applications are invoking vulnerable code and can determine whether dependency updates are required.

We aim to address these vulnerabilities with call graphs to extend the known issues to a more comprehensive understanding of the affected modules. We aim to identify vulnerable methods using propagation methods based on known vulnerabilities sourced from the National Vulnerabilities Database (NVD) and other public resources, employing the FASTEN project. Having identified the vulnerable methods, our next step involves highlighting these potential security risks on the call graph. Upon obtaining the vulnerable methods from FASTEN API, we trace all the methods that are calling them. We then propagate these vulnerabilities using a breadth-first search (BFS) algorithm. marking all the methods potentially affected by these weaknesses. Further, by combining this process with edge types—which indicate the analysis source of each edge—we enhance the reliability and precision of our vulnerability detection. The API we used from the FASTEN project accepts the version and artifact ID of the dependencies as input and returns the existing vulnerability ID. This allows us to trace back to the corresponding vulnerable methods. This is crucial as it provides a clear direction for subsequent fixes and updates, allowing developers to pinpoint problematic areas swiftly and effectively. The entire process of vulnerability detection is illustrated in Figure 3.6, providing a step-by-step guide to understanding our approach.

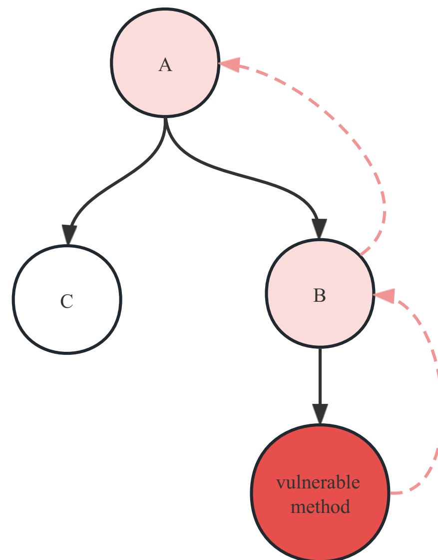


Figure 3.5: Example of vulnerability tracing

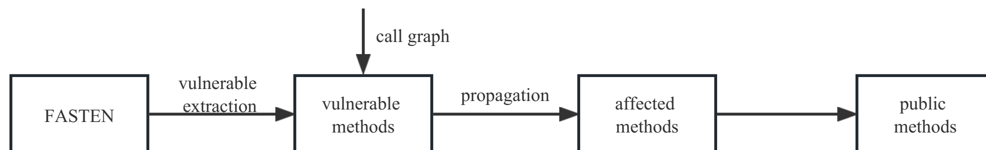


Figure 3.6: Public vulnerable methods propagation and filtering

To enhance the precision of our analysis, we look for vulnerabilities only accessible by dynamic edges, which are the edges that only appeared in dynamic analysis. Despite being resource-intensive, dynamic analysis dive deeper into the structure during run-time. This focus on dynamic edge analysis allows us to identify potential vulnerabilities that might be missed in a static analysis context. It also helps us understand the dynamic behavior of the software system under study better. We calculate the proportion of the extra vulnerabilities provided by the dynamic analysis so that we may be able to have more insights into the efficiency of the dynamic analysis and may further guide future methodologies on call graph construction and analysis.

3.5 Version analysis

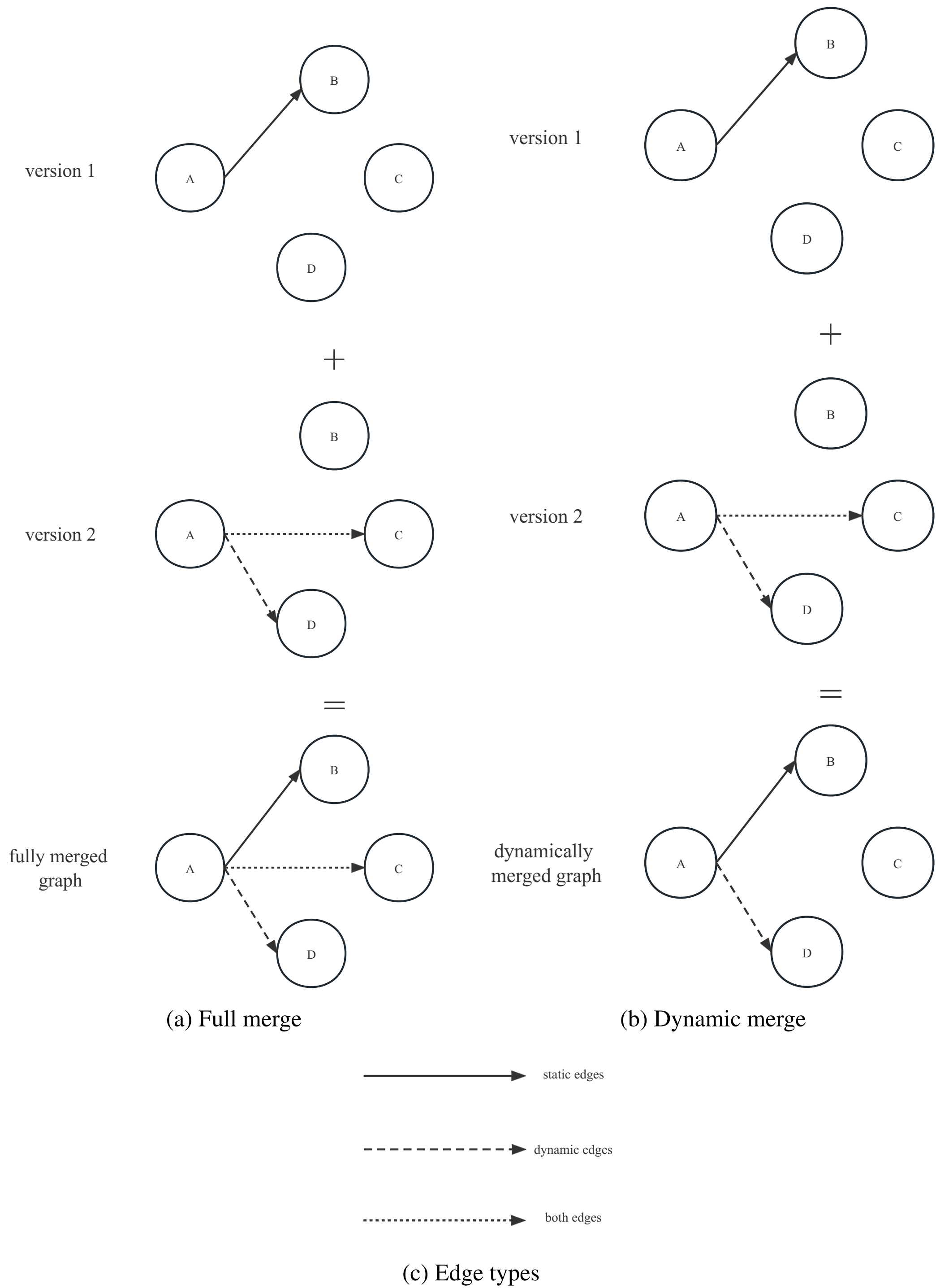
For large projects, the complexity of their codebases makes the high test coverage almost impossible to achieve. Compared to smaller projects, their test coverage is significantly

lower, leaving potential edges and paths unexplored. Although it is impossible to cover all the edges in the test suite of a single version of the projects, it is highly likely that other versions might have more test cases covering the same functions in this version. According to semantic versioning suggestions, which are followed by almost every single large project, the changes between minor versions and patch versions are small enough to make them compatible with each other. So, we proposed an approach to merge call graphs from different versions(minor and patch versions) of the project in a compatible manner. To ensure consistency, we designed appropriate filtering mechanisms, allowing us to reveal previously unseen edges within the version being analyzed.

As shown in Figure 3.7, we selected one version of the project as the baseline and then select another version of the call graph to merge with the baseline. In order to explore the most relevant updates and minimize the noise from major architectural changes, we based our selection process on the principles of Semantic Versioning[30]. It is a system that involves versioning a software product to communicate what kind of changes were made in the new release. It follows the pattern of MAJOR.MINOR.PATCH. Between major versions, there are incompatible changes that greatly change the methods and classes which makes the call graph incompatible. In this study, we focused only on the 'minor' and 'patch' versions, as these updates often include crucial bug fixes and functional improvements while maintaining backward compatibility. This focus allows us to maximize the chance of finding relevant test suites, eliminating potential confusion or distortion that may arise from major version changes.

The merged call graph is able to be built on the nodes of the existing project, complementing the missing edges due to incomplete coverage of test suites. We evaluated the effectiveness of this approach by examining the amount of new information brought by the merged graph. When merging the call graph, we only look for compatible nodes and edges. These nodes must have existed in the base version to ensure that the edges are complementary instead of added. One important presumption is that we assume the static analysis is accurate and complete, meaning that the edges detected by static analysis are never removed or excluded in the call graphs. We first merge 2 versions only with compatible edges, as shown in Figure 3.7a, the edges from dynamic and static analysis in version 2 are all merged into the call graph. In the merged call graph, 'Both' edges are the edges that appeared in both dynamic and static analysis. After directly merging these two graphs, we proceed to optimize the fully merged graph.

Our study operates under the assumption that the static analysis method should theoretically be capable of scanning all nodes (for this assumption, we also use the consistency guarantee means). We decided to primarily concentrate on the dynamic edges added to the baseline version by the new version. This decision meant that if we found static or 'both' edges that weren't present in the baseline version but appeared in version 2, we'd consider these as redundant edges that don't truly belong to the baseline version. This strategy ensured that we focused on genuine enhancements to the call graph and prevented the inflation of results with unrelated or insignificant additions introduced by the new version. The result of this process is what we've termed 'dynamic merge' graphs, as illustrated in Figure 3.7b. This graph more accurately reflects the unique information contributed by dynamic analysis, giving us a clearer view of its potential benefits.



(a) Full merge

(b) Dynamic merge

(c) Edge types

Figure 3.7: Version analysis

The constructed call graph from dynamic and static analysis with a merging step is still considered an oversimplification of the actual project structure. This simplification from the perfect call graph could cause problems in the study, information may be missing due to insufficient test coverage. For example, nodes are not found or tests do not cover some edges. So we seek to mitigate the influence of the changing factors as small as possible. To achieve this, we maintained consistency of dynamic analysis and static analysis, we are better equipped to accurately augment information from new versions to baselines. For these merged call graphs, while ensuring accuracy, we calculate their coverage of the edges in the baseline version and measure the amount of additional information provided by the merged graph. This process helps us to identify any potential enhancements to the completeness and detail of the call graphs.

Chapter 4

Results and evaluation

In this chapter, we present and evaluate the result acquired using the methodologies mentioned in Chapter 3.

4.1 Data processing

By following the data processing approach in Chapter 3, we collected Maven projects from GitHub and properly formatted and organized them as follows:

- We obtained and processed a total of 28,000 projects, covering a wide range of Maven projects and contributing to the comprehensiveness of the call graph construction.
- A total of 241,536 packages were detected during the analysis of project dependencies, mapping between all of the packages and coordinates are constructed for further investigations of the dependencies.
- A small proportion of projects, accounting for less than 10% (approximately 2,000 projects), were removed due to their extensive size and the associated challenges in compiling and analyzing them.
- As a result of the data processing, 93,000 Maven coordinates were acquired, providing valuable information to facilitate the efficient retrieval and analysis of project dependencies.

While cataloging the Maven coordinates in our study, we found that a considerable number of dependencies did not hold additional value for our research. Dependencies such as logging frameworks (e.g., SLF4J, Log4j), while broadly used across most projects, possess a relatively stable character. Therefore, they do not offer substantial insights into the invocation paths and the correlated vulnerabilities. Furthermore, some dependencies are oversimplified in their functionalities which offer very limited APIs that contribute to dynamic call graph relationships and possible vulnerability analysis in this research. Finally, it is essential to note that our analysis does not include standard or utility libraries that are

inherent to the Java ecosystem. Although these libraries are an essential part of all Java applications, they are even more stable and it is not necessary to explore the internal features of Java language in this study.

In most projects, even though we did not analyze these packages individually, they still produced a significant number of edges. Especially when we are analyzing other projects, the interactions between other projects and these utility projects are not negligible. Their presence in the call graph indicates their ubiquitous use in various Java applications, which further emphasizes their importance in the overall structure and functionality of Java programs.

In order to ensure a more focused and efficient analysis, we performed a filtering procedure to exclude projects without proper test suites. Our dataset was consequently downsized to about 28,000 projects. This updated dataset highlights the size and complexity of the data we're working with while also better serving our study goals. In these projects, we tried to collect the projects with proper Maven test suites. We found a lot of the projects with dozens of stars but still lack sufficient test coverage. In addition, despite our best efforts to collect projects with adequate Maven test suites, many of them even failed at the compilation stage. For these projects, we still analyzed them and all the execution paths before the error were correctly traced and extracted. This limitation may affect the completeness of our dynamic analysis and potentially impact the accuracy of the resulting merged call graphs. However, these challenges do not diminish the idea of using test suites that are from public projects. It emphasizes the need for further research and improved practices in test suite selection and extraction.

4.2 Call graph construction

RQ1: What insights can be gained into dependency usage patterns and software behavior by combining static and dynamic analysis techniques on existing GitHub projects, focusing on Maven test suite execution for dynamic analysis?

In our study, we employed Neo4j, a popular graph database, as a tool to visualize our call graph. Neo4j successfully gives us a reliable way to show the intricate connections between representation techniques and query methods. Figure 4.1 provides an illustration of this visualization, with a cluster from the commons-io project that centers on the method "copyLarge". In this visual representation, the red nodes are the methods from the commons-io project, while the other yellow nodes are from external packages. The orange edges designate static edges as well as "both" edges (edges detected in both static and dynamic analyses), while blue edges are exclusively identified using dynamic analysis. The names of the methods are reflected in the node labels. Each node has all the properties we mentioned in Chapter 3, we only used the method name as a label of demonstration. The cluster under observation exhibits a high level of coverage, which is a property typically seen in object-oriented programming. The complicated linkages seen in the call graph can be intuitively understood and further interpreted with the help of this visualization technique. It's impor-

Figure 4.1: Visualization example in Neo4j

tant to underline that although the volume of dynamic edges in this particular cluster seems high, it doesn't reflect the overall composition of the entire call graph. Dynamic edges make up a considerably lower portion of the call graph overall when seen in a larger context. This distinction serves as the foundation for the function of dynamic analysis within our methodology, which enhances static analysis rather than substituting it. Therefore, despite the fact that dynamic analysis can catch real-time behavior and relations that static analysis misses, the latter remains a crucial cornerstone of call graph building due to its complete and thorough nature.

Figure 4.2 highlights the relative contribution of static and dynamic analysis in call graph construction. Static and dynamic edges represent the edges exclusively found in static and dynamic analysis, while 'both' edges are found in both of the analyses. Among the analyzed projects, we noticed that static analysis contributed over 95% edges in the combined call graph. While the remaining 4% is contributed by the dynamic analysis.

Figure 4.3 represents the distribution of edges detected by both static and dynamic analysis. The area denoted by 46% corresponds to the edges that were detected solely through static analysis. The 4% region represents the edges that were exclusively detected through dynamic analysis. The overlapping section, representing 49% of the total, indicates the edges that were detected by both static and dynamic analysis. This shows that while static

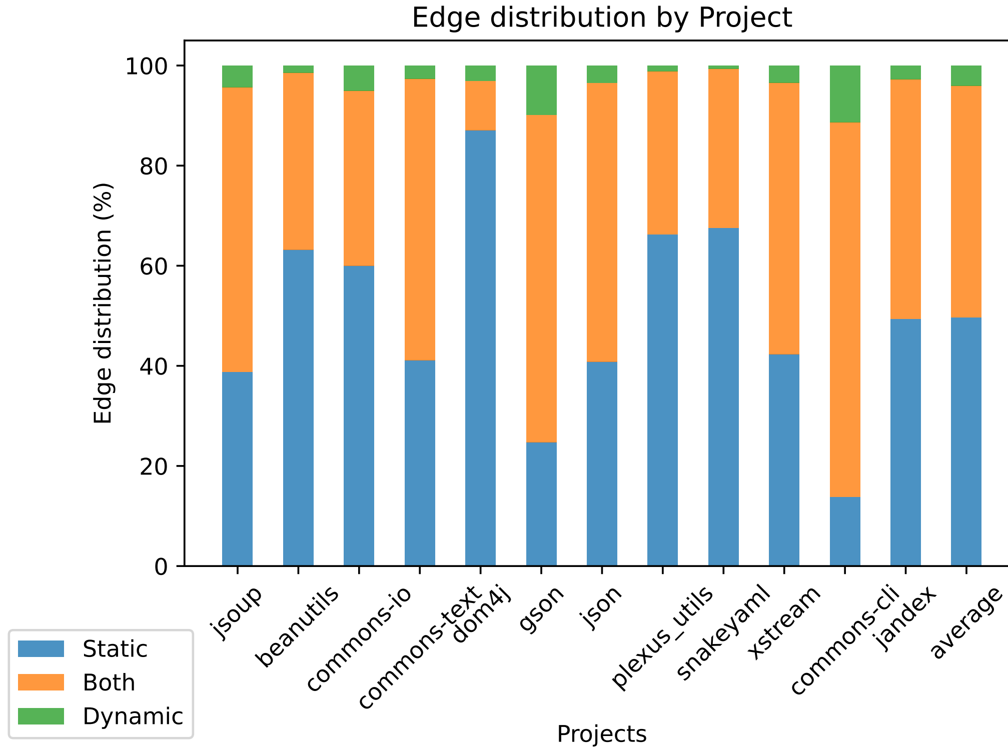


Figure 4.2: Edge type distribution in combined call graphs

analysis is capable of identifying a significant portion of the edges, dynamic analysis can detect additional unique edges that static analysis misses, even though the quantity might not be as substantial. Furthermore, nearly half of the edges are common to both analysis methods, highlighting the complementary nature of static and dynamic analysis.

Static analysis is the core of call graph construction and dynamic analysis act as a supplement. We also discovered that test coverage affects the effectiveness of dynamic analysis significantly. Projects with higher test coverage tend to have a higher percentage of dynamic contribution. In Table 4.1, we presented the test coverage of several projects. Combined with Figure 4.2, we can clearly discern the impact of test coverage on dynamic analysis. Projects with higher test coverage tend to contribute more dynamic edges in the call graph, suggesting a direct correlation between the comprehensiveness of test coverage and the effectiveness of dynamic analysis.

We also looked into the Maven projects in the dataset we collected. A notable observation from our analysis is the origin of the dynamic edges. A significant majority, precisely 80%, of these dynamic edges are derived from the official repositories of the dependencies. In contrast, client projects using these dependencies contribute only 20% of useful information. This distribution highlights the importance of considering official repositories in our analysis for a comprehensive understanding of the dependencies and their behaviors.

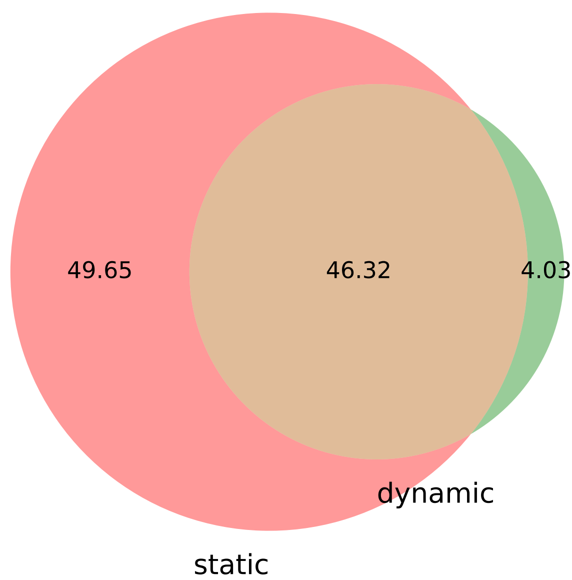


Figure 4.3: Contribution of analyses in combined call graphs

Table 4.1: Test coverage of multiple projects

Project Name	**Test Coverage**
plexus_utils	43%
xstream	75%
gson	83%
jsoup	84%
commons-cli	96%
commons-text	98%

4.3 Vulnerability detection

RQ2: How does the inclusion of dynamic analysis, supplemented by public test suites, enhance the detection of vulnerabilities compared to static analysis alone? Does the combination of static and dynamic analysis offer significant advantages in vulnerability detection and if so, under what circumstances?

In our study, we implemented a propagation method that plays a key role in our vulnerability detection process. This method was utilized within the constructed call graphs. The implementation of this method resulted in an increase in detected vulnerabilities. On average, we were able to identify an additional 5% of vulnerabilities across the analyzed projects. This increase, though might seem modest, in fact, constitutes a significant enhancement in the overall vulnerability detection rate.

Through dynamic analysis, we were able to unearth several vulnerabilities. As demonstrated in Listing 4.1, we identified several observations regarding polymorphism during dynamic analysis. Specifically, we found that the 'copy' function initiates a call to the 'read' method. This sequence reflects a clear instance of polymorphism. The 'copy' method directly invokes a vulnerable method, consequently impacting all methods found along the execution path, which was not detected in static analysis.

```
public static void copy(final Reader input, final Writer
    output, final int bufferSize) throws IOException {
      final char[] buffer = new char[bufferSize];
      int n = 0;
      while (0 <= (n = input.read(buffer))) {
          output.write(buffer, 0, n);
      }
      output.flush();
  }
```

Listing 4.1: Example of polymorphism detected by dynamic analysis

We also found some instances where certain lambda expressions remained undetected during static analysis, thereby leading to undetected vulnerabilities. For instance, as illustrated in Listing 4.2, the 'sort' function calls the 'compare' function, which encounters a vulnerability during the static initialization of the class. The presence of such scenarios reinforces the value of dynamic analysis in complementing static methods for a more comprehensive vulnerability detection process.

In our research, we have also examined the evolution of vulnerabilities across different versions of a software package. Figure 4.4 shows an example of this analysis using the xstream package. The bigger nodes represent vulnerable methods discovered in the older version, while the smaller nodes represent the corresponding methods in the newer version.

```
public File[] sort(final File... files) {
    if (files != null) {
        Arrays.sort(files, this);
    }
    return files;
}

@Override
public int compare(final File file1, final File file2) {
    final String suffix1 = FilenameUtils.getExtension(
        file1.getName());
    final String suffix2 = FilenameUtils.getExtension(
        file2.getName());
    return caseSensitivity.checkCompareTo(suffix1,
        suffix2);
}
```

Listing 4.2: Example of Lambda expression that leads to vulnerability

Our analysis showed that the vulnerabilities found in the older release had been successfully addressed and fixed in the newer release. Interestingly, we also discovered the emergence of new vulnerabilities in the updated version, in addition to some persistent vulnerabilities that remained unresolved. We believe the call graph constructed using our approach serves as a practical tool for the developers to have a better understanding of the evolution of vulnerabilities.

4.4 Version analysis

RQ3: How does the merging of call graphs from different versions of software and the subsequent optimization of these graphs affect the comprehensiveness and accuracy of the call graph?

In our version analysis, we sought to combine the base version with another minor or batch version sharing the same major version. The goal was to construct a compatible merged call graph. In order to build a compatible merge call graph, we only included nodes present in the base version so that the merged call graph guarantees its compatibility with the base version. As a result, we discerned that the unified call graph contributes an additional 1-2% of edges as shown in Table 4.2.

Initially, we operated under the presumption that the static analysis is precise. But this approach lead us to identify certain edges that, in theory, should not exist in the merged call

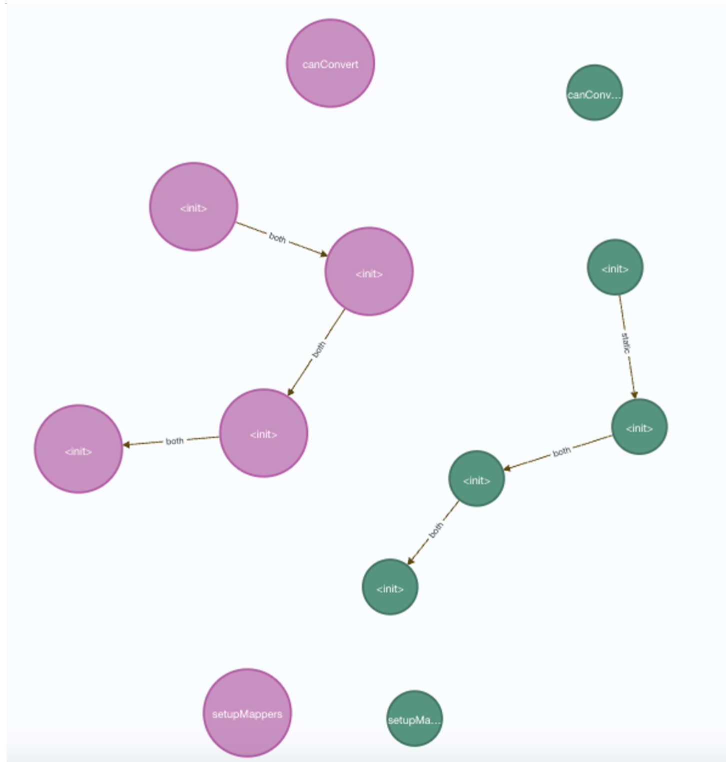


Figure 4.4: Example of vulnerability changes

graphs. Predominantly, these edges corresponded to new implementations or modifications present in the newer version but absent in the base version. In the context of our analysis, these edges were considered extraneous and misleading.

To address this issue, we refined our approach to exclusively incorporate edges deemed as compatible enhancements to our base version. We effectively pruned redundant edges identified earlier. This refined process yielded a minor, but significant, increase in nodes, in the range of 1-2%. Although the exact increase varied among projects. This refined approach allows us to provide more accurate and version-specific insights, focusing on the actual evolution of the software and enhancing the overall reliability of our study.

Table 4.2: Percentage of extra information found in version analysis

Project name	**compatible merge**	**dynamic merge**
commons-compress	1.88%	1.15%
commons-text	1.54%	0.72%
commons-io	2.80%	1.49%
xstream	2.31%	0.89%
plexus-utils	1.93%	0.60%

In Table 4.2, we noticed an interesting pattern that mirrors the one found in our first

research question. Particularly, for projects that make extensive use of dynamic features or information inaccessible during the static analysis, the addition of dynamic information clearly helps to complement and extend the completeness of the static analysis.

Chapter 5

Discussion and Future Work

In this chapter, we reflect upon the key findings of our research, discuss their implications, and propose directions for future research. We address the strengths and limitations of our method and also identify several challenging questions that emerged during this study, which we believe warrant further exploration in future work.

In this work, we demonstrate that leveraging the test suites from public GitHub projects for call graph construction and analysis is a powerful approach to understanding the intricate relationships between methods or functions. This methodology not only allows us to explore real-world projects but also to identify trends and patterns that may not be apparent in isolated or controlled environments.

Static call graph pruning We found that the combination of static and dynamic analysis resulted in a marginal increase in the number of edges in the call graph compared to using static analysis alone. This increase fully depends on the amount of information the dynamic analysis provides. While not significant, it has the potential to provide important insights missed by relying solely on static analysis, reflecting the complementary role that dynamic analysis can play in generating more comprehensive call graphs. We observed that dynamic analysis contributes to less than 5% of the new edges in the constructed call graph, suggesting that static analysis constitutes the core of the evaluation. It is worth noting that in cases where the test coverage is high (up to 95%), dynamic analysis can find more than 80% of the edges. However, in most cases, dynamic analysis accounts for less than 50% of the edges in the complete call graph. Despite the lack of dependence on perfect test coverage in our study, the observed correlation between edge distribution and test coverage suggests that higher test coverage leads to a more comprehensive dynamic analysis.

Although it is safe to say static analysis is the core of call graph construction, the conservative nature still makes it prone to redundancy, and dynamic analysis may be able to help with it. For instance, let's consider projects with high test coverage, such as commons-text, which has an impressive 99% test coverage. In such cases, it is theoretically expected that dynamic analysis would identify nearly all edges in the call graph. However, our findings reveal that dynamic analysis only identified about 70% of the total edges. This shortfall can be attributed to the over-approximation tendency of static analysis, which often introduces false-positive edges that are never invoked during actual execution. These undiscovered

static edges may be viewed as redundant in the context of great test coverage, hiding the true dependencies inside the software system. These findings lead us to the conclusion that a trimming technique could significantly aid static analysis in future studies. The overall analysis process would be made more effective and efficient by essentially getting rid of these extraneous edges. This would result in a call graph representation that is more precise [38].

Test coverage It is important to note that dynamic analysis is highly dependent on test suites, which are often difficult to collect and resource-intensive to run. Therefore, given the test coverage and efficiency issues, dynamic analysis is not a substitute for static analysis in most projects. Interestingly, the value of dynamic analysis varies from project to project. For projects that are IO-intensive or make extensive use of polymorphic and Java dynamic features, dynamic analysis provides a great deal of additional information. In some cases even more than 10% of extra edges. We conclude that the combined use of dynamic and static analysis techniques proves to be especially beneficial when examining these projects. This could potentially aid developers and project maintainers in making informed decisions, such as choosing suitable APIs or predicting possible software maintenance issues.

We went in-depth on the datasets and related test suites used in our research in Chapter 4. We discovered that more than 40% of the projects we gathered had dataset problems. These included test suites not existing at all, compilation errors, and runtime errors. These problems not only pose difficulties for our research but also highlight a more significant issue within the open-source community. Studies like ours, which rely on these resources for dynamic analysis, may experience significant differences in results depending on the standard and accessibility of test suites. Despite these difficulties, our approach and results offer insightful information about the potential of combining static and dynamic analyses using open-source test suites, laying the groundwork for further study. The future study should highlight the significance of maintaining thorough, high-quality test suites and illustrates the potential difficulties researchers may encounter when utilizing open-source projects for related research. Our analysis of the test suites revealed that more than 80% dynamic edges are detected in the test suite of the project's official repository due to their higher test coverage. At the same time, the contribution from client projects is disproportionate to the resource and time needed to gather them from public sources. Therefore, we conclude that if dynamic analysis is to be implemented, extracting information from official repositories will be sufficient and more efficient in terms of time consumption. However, a substantial number of dependencies that are either not open source or lack comprehensive test suites present challenges to this approach. These dependencies, although crucial for many projects, remain under-explored due to their limited accessibility or the absence of test suites. Consequently, this results in a less comprehensive call graph and may potentially omit significant relationships or vulnerabilities. A plausible direction for future work would be to diversify the sources of test suites and invest in innovative techniques to optimize test coverage, we could further enhance the robustness and comprehensiveness of our call graph.

call graph storage While our research establishes the validity of combining static and dynamic analysis, our investigation primarily pertains to the call graphs of individual projects and the discoverable entry and exit edges linking other projects. These edges reflect how other projects invoke methods within the project under study and how this project makes calls to external projects. We refrained from unifying all related graphs since it would necessitate altering the graph storage methodology and potentially require specialized storage for inbound and outbound edges. To achieve this integration, a plausible strategy involves the following steps: first, examine the dependencies of the project in question and verify if the dependent libraries have been analyzed previously. If these libraries are yet to be analyzed, they should undergo a comparable analysis and call graph construction process. The resulting information would then be saved to the graph database for future reference. If these libraries have been analyzed before, their call graphs would be retrieved directly from the database and integrated into the call graph of the current project. In the project's call graph, the connections between the project and its dependencies are independently extracted into a different database. A partial graph directly connected to the project can be gathered using these nodes from the dependencies. Propagation methods such as Breadth-First Search (BFS) or Depth-First Search (DFS) can be employed to extract relevant nodes and edges from the dependency call graphs, which can then be merged into the resulting call graph. The integrated call graph provides a more holistic view of the project and its dependencies. In essence, this proposed methodology facilitates the reuse of project analysis results, thereby enhancing project analysis efficiency and enabling more comprehensive and detailed analysis in future research endeavors.

Vulnerability detection For further validation of the combined call graph using our approach, we chose vulnerability detection as a practical use case. In our vulnerability detection procedure, we found that the undetected method-level vulnerabilities in the static analysis came mainly from the methods using Java dynamic features, polymorphic and lambda expressions. These invocation paths are not discovered by the static analysis due to their conservative nature. It also could be our static analysis tools have their own configuration rules and limitations. We found that the increased percentage of detected vulnerabilities correlates well with the additional information that was discussed earlier in the paper. In other words, the additional vulnerabilities discovered are often associated with the extra edges retrieved through our analysis approach. This finding indirectly accentuates the importance of leveraging supplementary information from dynamic analysis in improving the comprehensiveness and accuracy of vulnerability detection. It emphasizes the value of integrating static and dynamic analysis and exploiting available test suites for an enriched and more accurate call graph. However, our vulnerability detection approach has its limitations. The FASTEN vulnerability detection failed to map a lot of the method-level vulnerabilities because the NVD description was too general. For instance, a vulnerability related to handling input could affect multiple methods while FASTEN failed to identify any vulnerable methods. This is very common in SQL injection vulnerabilities. In future work, if there are more accurate and comprehensive method-level vulnerability mapping tools, developers may be able to find more vulnerabilities using this approach.

Semantic versioning For the third research question, despite the low test coverage, different versions in large projects still provide a potential opportunity to discover more valuable information. In this study, we used the concept of semantic versioning. Combining minor and patch versions was logical as changes between them are very small. The test suites are similar so we are likely to find possible unseen edges in the base version. However, due to their similarity, we did not further extract much useful information from these versions. As we observed a similar trend in which the IO-intensive projects or projects use dynamic features as in research question one. We could further validate the conclusion we derived from the combination of static and dynamic analysis that dynamic analysis provides a small amount of extra information and the benefits from this analysis is proportionate to the information gain. This highlights the indispensable role of dynamic analysis, even if it only complements static analysis, in providing a more comprehensive picture of the software.

Future research could explore the opportunities of extracting useful edges by combining test suites from different major versions. This could be an interesting direction, considering that major versions typically involve substantial changes, and thus could potentially provide more informative edges. However, it is not clear whether we could further improve the test coverage using information from other major versions. The changes between major versions are not negligible. As we are searching for the same methods in the base version, the methods in other major versions could have been removed, their name could have been changed to an unrecognizable state. It is critical to find a solution to maintain rigorous edge validation to ensure the relevancy and accuracy of the additional edges introduced into the call graph.

5.1 Threats to Validity

Despite being thorough and using a solid methodology, our research could still be invalidated. These threats can be divided into several categories, such as tool-related restrictions, dataset representativeness, and approach-related presumptions. In our research, the static and dynamic analysis tools used have inherent limitations. Even though static analysis is thorough, the production of false positives occasionally causes an overestimation of the call graph. On the other hand, the standard and breadth of the test cases used during execution are dynamic analysis's natural limitations. The resulting dynamic call graph may not be a complete representation if these test cases are not sufficiently exhaustive or do not explore all potential code execution paths, which could affect the accuracy of our findings.

The validity may also be threatened by the dataset's representativeness. For our static analysis, due to time and resource limits, we only chose JAR files that are smaller than 10 megabytes as our analysis sample. The generalizability of our findings may be constrained if these projects do not accurately reflect the wide range of applications and complexity levels present in actual software systems. As a result, it is unclear to what extent all Java projects, particularly those with higher degrees of complexity or those created outside the Maven ecosystem, can be generalized to include our findings.

Thirdly, the thoroughness of the test suites used has a significant impact on how well our dynamic analysis performs. We anticipate finding the majority of the edges using dynamic

analysis with high test coverage. However, incomplete test suites in many projects increase the risk of overlooking certain dependencies and certain code paths. In our research, the balance and comparative analysis of static and dynamic analyses may be compromised as a result of underestimating the contribution of dynamic analysis.

Finally, we chose to exclude some logging and utility libraries from the analysis because they are stable and extensively used in the Java ecosystem. This could also be viewed as a limitation. This choice might omit potential security flaws in these libraries and their implications for the entire software system, such as the famous Log4J vulnerability found in 2021 which affected countless number of projects depending on it.

We think it's critical to consider these potential risks when interpreting our findings. Each of these threats represents a potential area for methodological improvement and might be the subject of future research.

Chapter 6

Summary

The representation of method relationships within software projects and the relationships between dependencies can be done with the help of call graphs. Although static analysis is frequently used to build call graphs, it has drawbacks, such as a poor ability to handle dynamic features and lambda expressions. In this study, we introduced an approach for constructing dynamic call graphs using test suites from open-source Java Maven projects and merging them with static call graphs. We investigated the effectiveness of the merged call graph in revealing additional edges. To generate static call graphs, we used OPAL and Soot. We used Java agents to track the invocation edges for dynamic analysis. After merging call graphs from both analyses, a merging process along with a filtering mechanism was conducted to get rid of duplicates. We performed a vulnerability detection analysis to evaluate the results, and a version analysis to look into the possibility of expanding our approach by merging multiple versions. According to our findings, the merged call graph offers a slight edge increase over static analysis alone. The majority of the edge information came from the test suites in the official repository of the projects. Additionally, we found a similar pattern in vulnerability identification, which supports the consistency of our methodology. Combining several minor/patch versions of a project is an effective strategy for increasing test coverage. In a word, our study emphasizes the importance of using test suites from open-source projects to construct more detailed call graphs.

Bibliography

[1] runtime code generation for the java virtual machine. URL https://bytebuddy.net/#/.

[2] Github docs. URL https://ghdocs-prod.azurewebsites.net/en/rest/overview/resources-in-the-rest-api?apiVersion=2022-11-28.

[3] Java reflection api. URL https://docs.oracle.com/javase/tutorial/reflect/.

[4] Java instrumentation api. URL https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html.

[5] Wala. URL https://wala.sourceforge.net/wiki/index.php/Main_Page.

[6] parasoft, Jul 2001. URL https://www.parasoft.com/.

[7] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. Automated unit test generation for classes with environment dependencies. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 79–90, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330138. doi: 10.1145/2642937.2642986. URL https://doi.org/10.1145/2642937.2642986.

[8] David F Bacon and Peter F Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 324–341, 1996.

[9] Paolo Boldi and Georgios Gousios. Fine-grained network analysis for modern software ecosystems. *ACM Transactions on Internet Technology (TOIT)*, 21(1):1–14, 2020.

[10] Shigeru Chiba. Load-time structural reflection in java. In *European Conference on Object-Oriented Programming*, pages 313–336. Springer, 2000.

[11] Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *International Conference on Generative Programming and Component Engineering*, pages 364–376. Springer, 2003.

[12] Google Cloud. Github repositories public dataset on bigquery. https://console.cloud.google.com/bigquery?p=bigquery-public-data&d=github_repos&page=dataset, 2021. [Online; accessed May 6, 2023].

[13] FASTEN community. javacg-opal, 2019. URL https://github.com/fasten-project/fasten/tree/develop/analyzer/javacg-opal. javacg-opal.

[14] Christoph Csallner and Yannis Smaragdakis. Jcrasher: an automatic robustness tester for java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[15] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95—Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995 9*, pages 77–101. Springer, 1995.

[16] Michael Eichberg and Ben Hermann. A software product line for static analyses: The opal framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, SOAP '14, page 1–6, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450329194. doi: 10.1145/2614628.2614630. URL https://doi.org/10.1145/2614628.2614630.

[17] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001. doi: 10.1109/32.908957.

[18] fasten project. Api endpoints for maven projects, Nov 2021. URL https://github.com/fasten-project/fasten/wiki/API-endpoints-for-Maven-projects.

[19] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 576–587, 2014.

[20] The Apache Software Foundation. Naming conventions, 2002. URL https://maven.apache.org/guides/mini/guide-naming-conventions.html. instrumentation API.

[21] Neelam Gupta. Generating test data for dynamically discovering likely program invariants. In *Proceedings of ICSE 2003 Workshop on Dynamic Analysis*, pages 21–24, 2003.

[22] Marc R. Hoffmann. Jacoco, 2006. URL https://www.jacoco.org/jacoco/. jacoco.

[23] Ferenc Horváth, Tamás Gergely, Árpád Beszédes, Dávid Tengeri, Gergő Balogh, and Tibor Gyimóthy. Code coverage differences of java bytecode and source code instrumentation tools. *Software Quality Journal*, 27:79–123, 2019.

[24] Torsten Kempf, Kingshuk Karuri, and Lei Gao. Software instrumentation. *Wiley encyclopedia of computer science and engineering*, pages 1–11, 2007.

[25] Torsten Kempf, Kingshuk Karuri, and Lei Gao. *Software Instrumentation*. 09 2008. ISBN 9780470050118. doi: 10.1002/9780470050118.ecse386.

[26] MongoDB. Mongodb. `https://www.mongodb.com/`, 2023. MongoDB can store related data in a single document structure, which can simplify queries and improve performance.

[27] Antoine Mottier. Overview - fasten. URL `https://www.fasten-project.eu/view/Main/Overview`.

[28] Fernando Rodriguez Olivera. maven, 2006. URL `https://mvnrepository.com/repos/central`. maven repository: central.

[29] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816, 2007.

[30] Tom Preston-Werner. Semantic versioning 2.0.0. URL `https://semver.org/`.

[31] B.G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, 1979. doi: 10.1109/TSE.1979.234183.

[32] B.G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, 1979. doi: 10.1109/TSE.1979.234183.

[33] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. On the recall of static call graph construction in practice. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1049–1060, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371216. doi: 10.1145/3377811.3380441. URL `https://doi.org/10.1145/3377811.3380441`.

[34] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. *ACM SIGPLAN Notices*, 35(10):264–280, 2000.

[35] Clarence Tauro, Aravindh S, and Shreeharsha A.b. Article: Comparative study of the new generation, agile, scalable, high performance nosql databases. *International Journal of Computer Applications*, 48:1–4, 06 2012. doi: 10.5120/7461-0336.

[36] Dávid Tengeri, Ferenc Horváth, Árpád Beszédes, Tamás Gergely, and Tibor Gyimóthy. Negative effects of bytecode instrumentation on java source code coverage. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 225–235. IEEE, 2016.

[37] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. *SIGPLAN Not.*, 35(10):281–293, oct 2000. ISSN 0362-1340. doi: 10.1145/354222.353190. URL `https://doi.org/10.1145/354222.353190`.

[38] Akshay Utture, Shuyang Liu, Christian Gram Kalhauge, and Jens Palsberg. Striking a balance: Pruning false-positives from static call graphs. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2043–2055, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. doi: 10.1145/3510003.3510166. URL `https://doi.org/10.1145/3510003.3510166`.

[39] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *Compiler Construction: 9th International Conference, CC 2000 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2000 Berlin, Germany, March 25–April 2, 2000 Proceedings 9*, pages 18–34. Springer, 2000.

[40] Tim A Wagner, Vance Maverick, Susan L Graham, and Michael A Harrison. Accurate static estimators for program optimization. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 85–96, 1994.

[41] Tao Xie and David Notkin. An empirical study of java dynamic call graph extractors. *University of Washington CSE Technical Report*, pages 02–12, 2002.

Appendix A

Glossary

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

CG: Call graph.

JDK: Java Development Kit.

JVM: Java virtual machine.

API: Application programming interface.

Combined call graph: Xall graph constructed by combining static and dynamic analysis.

Merged call graph: Call graph constructed by merging different versions of the same project.

Neo4j Bloom: Neo4j Bloom is a free visualization tool for the Neo4j graph database.

OPAL: OPAL is a static analysis platform for Java bytecode analysis such as call graphs.

POM file: A Project Object Model or POM file is where developers declare the usage of dependencies with Maven.

Maven: Maven is a software package management tool.

dynamic feature: The features that allow the program to change at runtime.