UC Irvine
UC Irvine Electronic Theses and Dissertations

Title

Automatic Detection of Breaking Strong Encapsulation in Java Modules

Permalink

https://escholarship.org/uc/item/2tf1d64x

Author

Chen, Yongbo

Publication Date

2024

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

Automatic Detection of Breaking Strong Encapsulation in Java Modules

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Software Engineering

by

Yongbo Chen

Thesis Committee:
Assistant Professor Joshua Garcia, Chair
Professor Sam Malek
Associate Professor James A. Jones

2024

© 2024 Yongbo Chen

TABLE OF CONTENTS

Page

LIST OF FIGURES **iv**

LIST OF TABLES **v**

ACKNOWLEDGMENTS **vi**

ABSTRACT OF THE THESIS **vii**

1 Introduction **1**
1.1 Motivation 1
1.2 Organization of the Thesis 5

2 Java Platform Module System and BSE Problem **6**
2.1 Overview of Java Platform Module System 6
2.1.1 Module and Module Directives 6
2.1.2 Unnamed Module 11
2.2 Overview of Breaking Strong Encapsulation (BSE) problem 12

3 Design Principles for Detecting Abuse Instances **15**
3.1 Module Abuse Instance Construction 15
3.2 BEAD Detection Approach and Implementation 20
3.2.1 Step 1: JDK Info Scanning 21
3.2.2 Step 2: JDK Info Combining 22
3.2.3 Step 3: Invocation Analysis 24
3.2.4 Step 4: Abuse Instance Analysis 27

4 Case Study: Evaluation of BEAD **28**
4.1 RQ1: JDK Module Interfaces Design 29
4.2 RQ2: Result Analysis Of BEAD for
Detecting BSE Abuse Instances 30
4.3 RQ3: Detected JDK Abuse Instances and
Commonly-Abused Packages 32
4.3.1 Abuse Instances of Reflection 32
4.3.2 Abuse Instances of Compile-Time Invocation 34
4.4 RQ4: BEAD Against JDK Compiler's Abuse Detection 35

4.4.1 Javac's Detection BSE Detection at Compile Time 36
4.4.2 Detection of Reflection-Oriented BSE Abuse 39
4.5 RQ5: Efficiency of BEAD 42

5 Discussion **43**
5.1 Implications 43
5.2 Threats to Validity 45

6 Related Work **47**

7 Conclusion **49**

Bibliography **50**

LIST OF FIGURES

Page

1.1 Example of BSE problem propagation 3

2.1 Example of module declarations and their directives provided in `JUnit 5` `module-info.java` files 7
2.2 Specified dependencies between modules based on their directives 8

3.1 A high-level overview of BEAD 21
3.2 *JDKModule* structure example 23

4.1 Compile Error message for Un-exported Packages 37
4.2 Example Output of BEAD Abuse Detection Under Compile Time 37
4.3 JVM Abuse Warning Message Example 40
4.4 BEAD Reflection Abuse Report Example 40

LIST OF TABLES

Page

3.1 Functions describing exports and opens directive dependencies based on JDK implementation and the way that source code invoke members 17

4.1 Subject Applications 29
4.2 Identified Abuse Instances of Subject Applications 30
4.3 Abused Instance via. Reflection Invocation in Subject Applications 32
4.4 Top 5 Abused Packages Under Compile-Time Invocation 34
4.5 Abuse Information Provided by Javac at Compile-Time 36
4.6 Abuse Information Provided by BEAD at Compile-Time 38
4.7 Abuse Information Provided by JVM via. Reflection 39
4.8 Abuse Information Provided by BEAD via. Reflection 41
4.9 Result of Execution Time 42

ACKNOWLEDGMENTS

I would like to extend my sincere gratitude to Professor Joshua Garcia, Professor James A. Jones, Professor Sam Malek, and all the professors and friends who helped me during my master's years.

ABSTRACT OF THE THESIS

Automatic Detection of Breaking Strong Encapsulation in Java Modules

By

Yongbo Chen

Master of Science in Software Engineering

University of California, Irvine, 2024

Assistant Professor Joshua Garcia, Chair

Starting from JDK 9, the Java Platform Module System (JPMS) was introduced to facilitate resolving the monolithic ball-of-mud architecture of the original JDK by offering architectural modules in the Java language that provide various benefits for Java projects, especially in terms of protecting its internals from dangerous static or dynamic usage. However, a recent study has shown evidence that many existing projects bypass Java module boundaries to access internal APIs which, in turn, break the strong encapsulation (BSE) of Java modules. BSE leads to exceptions, errors, and other maintenance issues for Java systems. To address BSE, we propose a detection tool, BEAD, that leverages static analysis to identify BSE abuse instances within Java projects and evaluate BEAD on open-source Java projects. Our analysis revealed that JDK modules are designed with strong encapsulation principles, but BEAD detected numerous abuse instances, predominantly at compile-time, indicating prevalent unauthorized access to internal packages. Besides, our analysis found evidence of a correlation between the detected abuse instances and reported GitHub issues, validating the ecosystem impact of those detected abuse instances. Our findings demonstrate BEAD's effectiveness of detection capabilities in terms of BSE problem.

Chapter 1

Introduction

1.1 Motivation

As modern software systems grow in size and complexity, maintaining them only at the code level of abstraction (e.g., traditional functions, classes, and packages) has become a challenge for maintainers, especially as the lines of source code for a single system increasingly reach into the hundreds of millions. The traditional approach to dealing with such large and complex software systems is through the lens of the software architecture's constructs (e.g., components, connectors, and configurations) [42] [45] [46]. Unfortunately, ensuring consistency between architecture and code-level abstractions has been a core problem of *software architecture drift and erosion* [43] [47], which we collectively refer to as *architectural decay*.

A significant example of *architectural drift and erosion* is in the Java ecosystem, especially the Java Development Kit (JDK), which has described as a ball-of-mud architecture [40] [44]. To address the architectural rigidity associated with the monolithic structure, the JDK now includes the Java Platform Module System in the Java Language. This new system

brought a new architectural module called *Java Module*. Those Java modules have a higher abstraction level and serve as package containers, which allows developers to declare the visibility and related access permissions of a module's internal package to external APIs and other modules by setting specific directives, like *exports* and *opens*. Besides, JPMS can provide developers with an editable architecture, a file called *module-info.java*, used to store the contents of the modules and the categories used for inter-communication.

Despite the numerous benefits of JPMS, such as robust encapsulation for Java projects, it's crucial to address the architectural decay issues that have surfaced in real-world implementations. One such issue is the *over-* and *under-exposure* of Java modules.

Over-exposure occurs when a Java module grants access to its internal packages, even though none of the external classes utilize them. Conversely, *under-exposure* is when external usage of specific packages within the Java module is allowed, but the module's declaration doesn't open access to these packages to the outside world.

Previous studies have investigated the problem of *over-exposure* and *under-exposure* of Java modules to some extent. For example, Ghorbani et al.'s study [28] proposes an automated framework, Darcy, for detecting and automatically repairing defect models consisting of formally defined eight architectural inconsistencies, i.e., eight *over-exposure* scenarios in Java modules, through static analysis techniques.

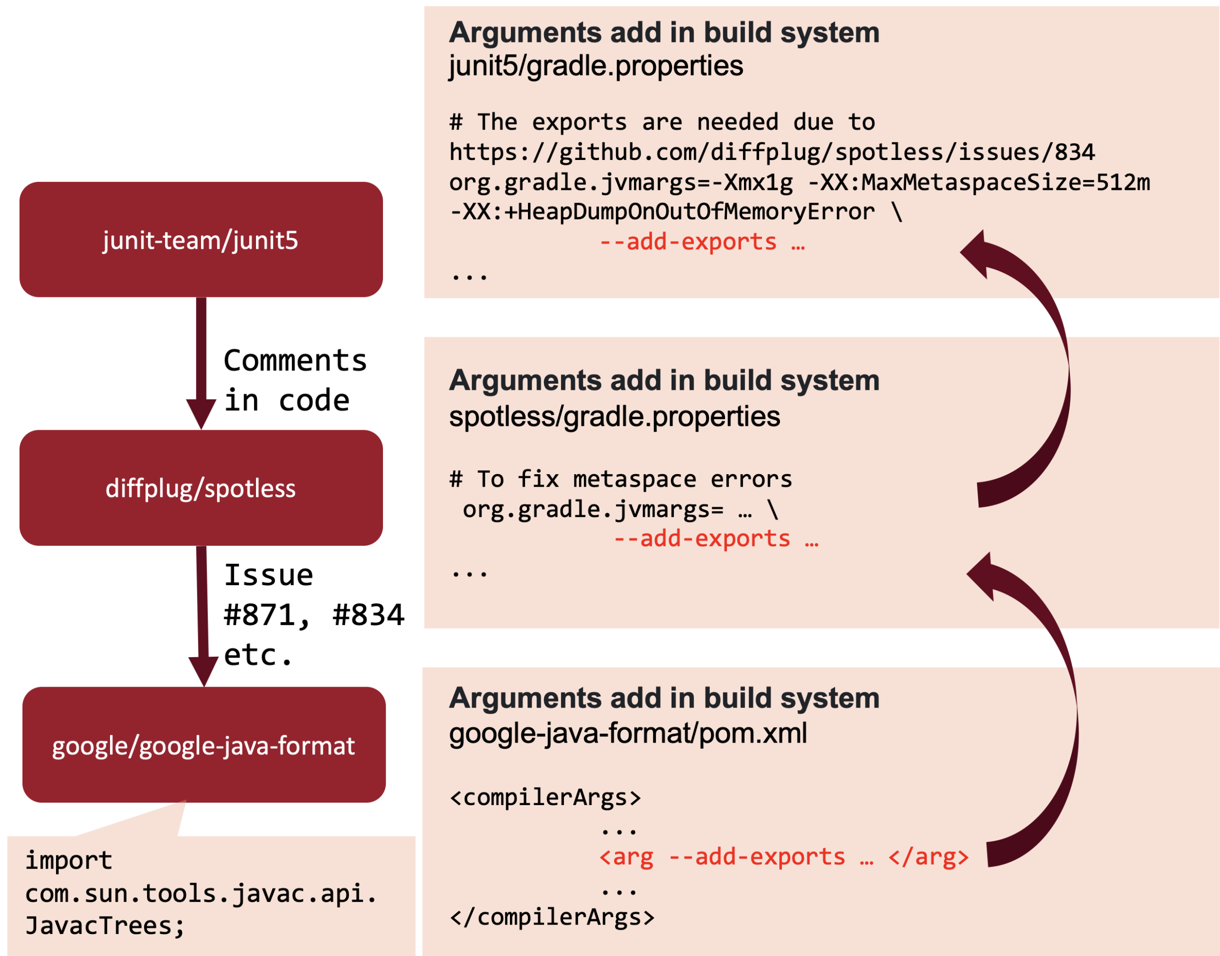

Figure 1.1: Example of BSE problem propagation

The *under-exposure* problem occurs in the opposite case, where an external class tries to access an internal package in a Java module that does not grant access permissions. Existing studies [33] have already found evidence that many projects still attempt to bypass the strong encapsulation provided by JPMS in order to access internal APIs for different reasons and lead to technical debts that affect the broader ecosystem. Such kind of abuse problem has been identified as Breaking Strong Encapsulation (BSE) problem [24] [25]. One BSE example is described in Figure 1.1, *JUnit 5* [11] tries to access internal APIs by adding Java Virtual Machine (JVM) arguments, i.e., the flag `--add-exports`, which allows code in the target module to access types in the named package of the source module. A deeper investigation

of *JUnit 5* reveals that *JUnit 5* needs to break strong encapsulation because it relies on a plugin called *Spotless* [13], which employed similar JVM arguments in its configuration file. Eventually, an investigation of *Spotless*'s GitHub issue [9] [12] revealed that the source of the behavior that caused *Spotless* to break strong encapsulation was a tool called *Google Java format* [8], designed to reformat Java source code according to the Google Java Style guidelines [10]. This example shows in detail how a BSE issue can cause problems at the project level and affect other projects in its dependency chain, affecting the relevant Java ecosystem.

While Darcy focuses on detecting and repairing the *over-exposure* problem, none of the prior studies focus on detecting *under-exposure* problem, i.e., BSE problems in actual programs [33]. In order to bridge the gap, this thesis will first briefly introduce (1) the Java Platform Module System and (2) the BSE problem. Then, I will propose BEAD (Breaking Encapsulation Abuse Detector), a detection tool that combines static analysis techniques to identify potential abuse instances within Java projects that could lead to illegal access bugs.

More precisely, BEAD helps the program (1) obtain the specific JDK module information under the current running environment, (2) locate and acquire the reflected and static calls of the analyzed program, and (3) check whether the relevant calls of the analyzed program will abuse the original definition of JDK's module specific declarations. Finally, to test the BEAD, I have performed a case study by analyzing five open-source Java projects within the BEAD, including the abuse detection result, social impacts of abuse instances, and the running efficiency of the BEAD.

1.2 Organization of the Thesis

The rest of the thesis will follow: Chapter 2 introduces the Java Platform Module System and BSE problems. Chapter 3 presents the design principles of detecting abuse instances in the program. Chapter 4 describes the case studies to test BEAD and shows the result. Chapter 5 presents further discussions about the findings. Chapter 6 presents some related work. Chapter 7 concludes the thesis.

Chapter 2

Java Platform Module System and BSE Problem

2.1 Overview of Java Platform Module System

2.1.1 Module and Module Directives

In Java Platform Module System (JPMS), a module can be treated as a container with a unique name designed to store reusable groups of related packages and resources like images and XML files [2]. Each module requires a descriptor file named `module-info.java`, which stores the module's internal meta-data and declarations of a named module.

The structure of a module declaration file consists of a unique module name and a module body. The module body can be empty or contain one or more module directives that specify the module's contacts with other modules or modules that need to be accessed [29].

```
module org.junit.platform.launcher {
    requires transitive org.junit.platform.engine;
    uses org.junit.platform.engine.TestEngine;
    ...
}

module org.junit.jupiter.engine {
    requires org.junit.platform.commons;
    provides org.junit.platform.engine.TestEngine
      with org.junit.jupiter.engine.JupiterTestEngine;
    opens org.junit.jupiter.engine.extension to
      org.junit.platform.commons;
   ...
}

module org.junit.platform.commons {
    exports org.junit.platform.commons.logging to
      org.junit.platform.engine;
    ...
}

module org.junit.platform.engine {
    exports org.junit.platform.engine;
    ...
}
```

Figure 2.1: Example of module declarations and their directives provided in `JUnit 5` `module-info.java` files

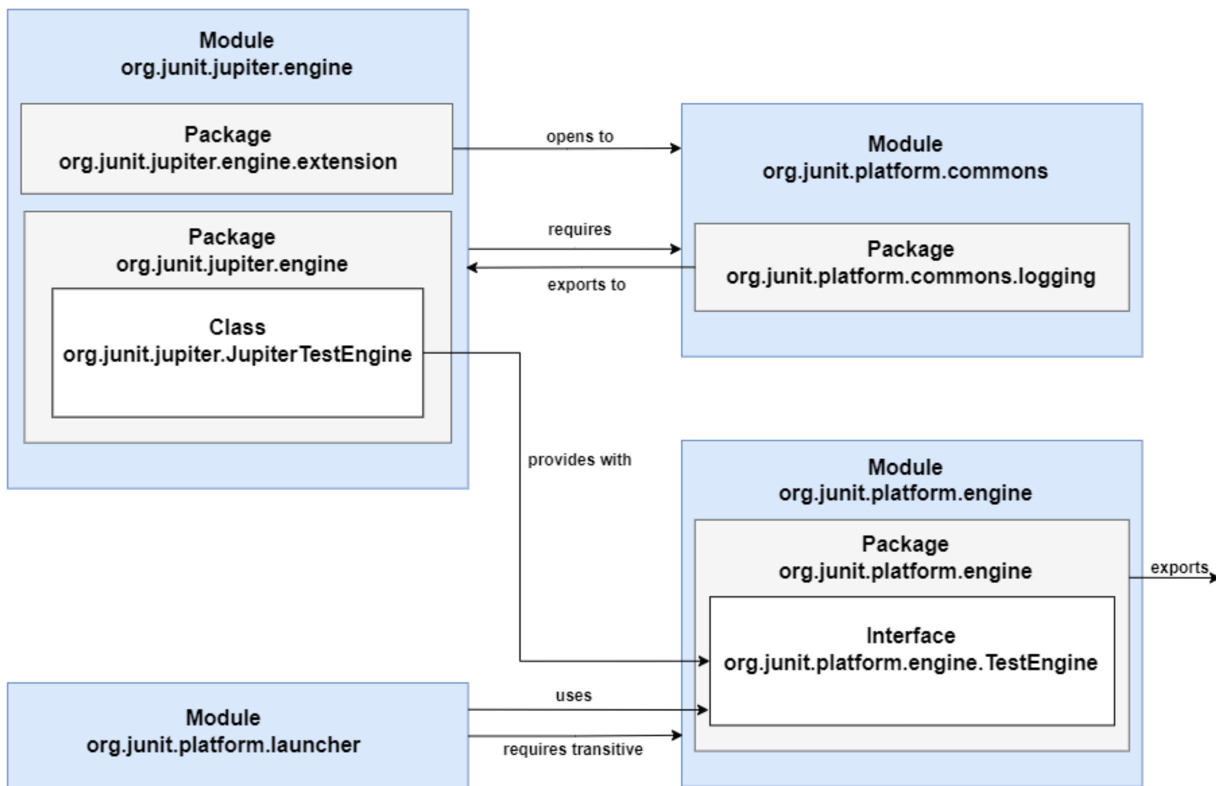

Figure 2.2: Specified dependencies between modules based on their directives

The above two figures, Figure 2.1 and 2.2, show an example of `JUnit5`, which adopted JPMS in its implementation [14]. The declarations of 4 modules and their relationship to each other is illustrated in Figure 2.1: `org.junit.jupiter .engine`, `org.junit.platform.commons`, `org.junit.platform.launcher`, and `org.junit .platform.engine`. Figure 2.2 describes the detailed relationship between the same modules based on dependencies in their declarations.

To build module dependencies and access relationships, JPMS provides the following five directives to specify a module's interface and its usage: the *requires* directive specifies the packages that the module needs to access, the *exports* and *opens* directives enable the module's packages to be available to other modules, the *provides* directive specifies the services

provided by the module, and the *uses* directive specifies the services consumed by the package. The specific declaration rules for these five directives are explained below [29]:

- The `requires` directive allows developers to declare the module dependency by requiring another module. When module A `requires` module B, it is said that module A has both a run-time and compile-time dependency on module B. For instance, module `org.junit.jupiter.engine` is depicted in Figure 2.1 which requires module `org.junit.platform.commons`. Another option JPMS provides for building module dependencies is the `requires transitive` directive. This directive specifies that any module that requires the declaring module implicitly gains a dependency on another specified module. In the example of Figure 2.1, module `org.junit.platform.launcher` has transitively required module `org.junit.platform.engine`. Therefore, when module C requires module `org.junit.platform.launcher`, module C can access module `org.junit.platform.engine` without `requires org.junit.platform.engine`.

- The `exports` and `exports...to` directives specify the module's packages whose `public` types (and their nested `public` and `protected` types) should be accessible to all other modules or a comma-separated particular module list at both compile and run time. As an example, in Figure 2.1, module `org.junit.platform.engine` exports its package access to all other modules, while module `org.junit.platform.commons` specifically exports the access of package `org.junit.platform.commons.logging` to module `org.junit.platform.engine`.

- The `opens` and `opens...to` directives define the module's packages whose `public` types (and their nested `public` and `protected` types) should be accessible to all other modules or a comma-separated particular module list at only the run time but not compile time. These directives also grant reflective access to all the types in the module, including the private types and all its members from other modules. In Figure 2.1, package `org.junit.jupiter.engine.extension` was opened by module

org.junit.jupiter.engine to the module org.junit.platform.commons.

- The provides with directive describes a module's provision of a service. Its declaration is written as for module a, it provides $c1$ with $c2$, $c3$, ..., cn. This declaration indicates that this directive will be used by module $m1$ as a service provider and specifies $c1$ as an abstract class or interface and one or more service provider classes to be used with java.util.ServiceLoader via the with clause. A service is a well-known set of interfaces and (usually abstract) classes, and a service provider is a specific implementation of a service. In Java, the class java.util.ServiceLoader provides a simple service-provider implementation that loads a provider to implement the service with type S [1]. As exemplified by module org.junit.jupiter.engine provides the class org.junit.jupiter.JupiterTextEngine as a service using the interface org.junit.platform.engine.TestEngine in module org.junit.platform.engine as the service's implementation.

- The uses directive indicates a module's consumption of a service. When module A is declared using a directive of the form uses $c1$, this means module A uses a service object of an abstract class or interface, $c1$, which is provided by another module. For this purpose, module A needs to discover the specified service provider beforehand via class java.util.ServiceLoader, which is used to load service provider instances. For example, module org.junit.platform.launcher uses the service object of interface org.junit.platform.engine.TestEngine.

With these directives described above, JPMS can utilize Java modules to perform dynamic load and unload operations at runtime, thereby changing the normative and descriptive architecture of the system according to different use cases [36].

2.1.2 Unnamed Module

An additional concept in JPMS is the *Unnamed module*, which is a container for all "non-module classes," including (1) classes without module descriptors at compile time and (2) any class loaded from the classpath at compile and run time [15] [36].

Theoretically, the strong encapsulation system employed by JPMS was initially designed to treat each module as an independent unit of functionality. However, for backward compatibility reasons, creating modules in JPMS is not mandatory, and as a result, classes that are not part of any defined module are automatically placed into an unnamed module. An *Unnamed module* is a default, catch-all module for types not included in any named module, whereas a *Named Module* refers to any module explicitly defined with a name in the module declaration. More specifically, unnamed modules encapsulate all classes on the classpath, apply specific rules to them, and integrate them into the modular system, making unnamed modules less encapsulated than named modules [15] [24] [25] [27] [36].

On the other hand, unnamed modules can effectively run without a module descriptor in practice. Despite not having a formally defined name (which also makes it impossible for an unnamed module to be directly referenced in a module declaration), an unnamed module has the same three key attributes as a named module, i.e., it implicitly reads and exports all other modules contained in the module graph and opens up all of its packages for reflection, which is another reason for unnamed modules to be less encapsulated. For the service availability mechanism, unnamed modules can utilize `META-INF/services` to provide services to the outside via the `ServiceLoader` class [15] [25].

2.2 Overview of Breaking Strong Encapsulation (BSE) problem

In Java 9+, the strong encapsulation rules provided by JPMS make the behavior of the JVM strictly consistent with these rules, thus effectively preventing illegal accesses from occurring at compile time. However, given the fact that some Java programs written before Java 9+ may call internal APIs encapsulated by JPMS in their code implementations, this also makes it possible to cause *under-exposure* problems when attempting to migrate to version 9+, where the program tries to access a package within the JDK that is not declared open or export to the outside. As a result, the runtime environment initially adopted a more lenient stance in favor of backward compatibility [17] [18] [19], i.e., the default runtime setting is `--illegal-access=permit`. this mechanism is intended to enhance the compatibility of applications initially developed for Java 8 and earlier and to facilitate a smoother migration of these applications to versions 9 and above [16] [27] [41].

However, this mechanism of allowing illegal access by default at runtime has been phased out. In JDK 16, the default illegal access setting was changed from `permit` to `deny`, and in JDK 17, it was deprecated [20] [21]. Previously, JDK 16 and earlier versions would raise relevant warnings in the console log for illegal access operations. With JDK 17, these illegal access operations are recognized directly as exceptions and errors that are thrown without the need for specialized configuration. This shift emphasizes the JDK's commitment to enforcing a long-established principle: 'Packages beginning with 'java' or 'javax' are designated as public APIs, while all other packages are internal private packages by default,' which is an important step in enforcing module integrity and encapsulation in Java [22].

There are several strategies to address the underexposure problem and the program's dependence on internal APIs during porting. The most direct and durable solution is to replace the internal APIs with the supported APIs. If this is not feasible, the next best option

is to declare these dependencies during the runtime configuration of the module system. Developers can achieve this by using specific JVM options to export packages:

- Using command line: --add-exports <module>/<package>=<readingmodule> to export <package> from <module> to <readingmodule>

Meanwhile, there are two options for developers to open specific packages:

- Using command line: --add-opens <module>/<package>=<readingmodule> to open <package> from <module> to <readingmodule>
- Using command line: --illegal-access=(permit | warn | debug) to enable access to packages present in JDK 8 but encapsulated in JDK 9, facilitating static and deep reflective access, which give the permissions to inspect and modify a class's internal elements, such as private and protected fields, methods, and constructors that are normally shielded by strict encapsulation rules. The latter can be used through the platform's reflection API calls to allow unauthorized code into the JDK's internals to run as in previous JDK versions. Note that this option is only applicable from Java 9 through Java 16.

In short, Java has adopted a policy of loosening strong encapsulation to help programs transition from Java 8 to Java 9's modular system. Nevertheless, on the one hand, this policy behavior itself, especially the three options provided above, can break the strong encapsulation that JPMS was initially designed for. On the other hand, the enforcement of strong encapsulation has gradually strengthened over time with JDK version upgrades, especially considering that the privilege of --illegal-access was denied by default starting with JDK 16 and completely removed in JDK 17. As a result, if such under-exposure is not addressed on time, previous BSE warnings will turn into factual errors or exceptions

during future JDK version porting. At the same time, this trend highlights the increasing stringency of the JDK's work on strong encapsulation, emphasizing the need to address such issues promptly to avoid significant bugs.

Chapter 3

Design Principles for Detecting Abuse Instances

3.1 Module Abuse Instance Construction

Our earlier research on the GitHub issues reported on BSE [33] revealed that certain types of abuse instances might occur between the module dependencies implemented in some Java projects and the module dependencies described in the JDK's original `module-info` file. Those instances arising from BSE problems occur when either (1) an external class attempts to access a package in a module that has not opened or exported that package or (2) a module that has opened or exported a package is accessed incorrectly by an external class, or the class tries to access non-public members within the package. These abuses can affect various architectural attributes:

A1: Encapsulation and Maintenance – Failure to open access to required functionality in other modules may result in needing to modify the original module files to open these packages, thus increasing the complexity and maintainability of these modules. In addition,

without modifying the original module files, developers may gain access to the required functionality by breaking strong encapsulation.

A2: Adaptability – Lack of access to required functionality may force developers to use workarounds, such as exposure extra packages to outside and rewriting custom functionalities, which increases the system's complexity and reduces the system's adaptability for any future functional adjustments.

A3: Security – Inadequate exposure of modules may lead developers to obtain unexposed packages through illegal access, which may cause associated illegal access errors and security issues.

A4: Compatibility – Modules whose required functionality fails to open access may cause other modules that depend on the module to fail to compile or run, causing dependency failures. Module version changes can also cause access changes, which may lead to software depending on this module having a JDK version compatibility problem.

Considering that BSE problems can be caused by unauthorized external access to packages inside the module or *under-exposure* problems caused by internal module declarations, the detection of BSE problems will focus on checking for abuse instances under the `opens` and `exports` directives. To cover those module abuse instances caused by the BSE problem as comprehensively as possible, I examine the misuse that developers may make using two module directives, `exports` and `opens`, which can lead to BSE problems. In the rest of this section, I will focus on the eight abuse scenarios that can arise in BSE problems and the functions needed to specify them.

Constructing the abuse model requires retrieving the module dependencies and associated opens and exports directive declarations that the JDK implements in practice. Therefore, I scanned all the module-info.java files included in the JDK 17 source code, which is detailed in 3.2.1, and found four variations on the `exports` and `opens` directives in JDK implemented

module declarations, which have been listed in Table 3.1. In addition, it should be noted that each package in JDK 17 declares only one of the directives listed in Table 3.1, except for three packages: the `javax.swing.plaf.basic` package from the `java.desktop` module, which declares both the `exports` and `opens...to` directives, and the package `sun.reflect` and `sun.misc` from module `jdk.unsupported` declared both the `exports` and `opens` directives. Considering the need to cover all potential types of abuse instances comprehensively, we need to account for situations with a single directive and the particular module declaration scenarios provided by the three packages mentioned above in the following phase of abuse model construction.

Table 3.1: Functions describing exports and opens directive dependencies based on JDK implementation and the way that source code invoke members

Function	**Description**
$Exp(M, P)$	Module M exports package P.
$ExpTo(M_1, P_1, \{M_2, M_3, \ldots\})$	Module M_1 exports package P_1 to the set of modules $\{M_2, M_3, \ldots\}$.
$Opens(M, P)$	Module M opens package P.
$OpensTo(M_1, P, \{M_2, M_3, \ldots\})$	Module M_1 opens package p to the set of modules $\{M_2, M_3, \ldots\}$.
$CompInv(P_1, m, P2)$	Source code in package P_1 invoke member m in package P_2's class via compile-time invocation.
$ReflInv(P_1, m, P_2)$	Source code in package P_1 invoke member m in package P_2's class via reflection.

To describe how an external call to an open or exported package in a module can be made, I also defined two functions in Table 3.1 that show the way to invoke an exported package, i.e., the reflection invoke and the compile-time invoke, as explained in section 2.1.1.

By leveraging the functions in Table 3.1, I introduce the following eight types of BSE abuse instances under six implemented situations: exports, exports...to, opens, opens...to, exports & opens...to, and exports & opens. Based on the given situations, Section 3.2 detects the

following abuse instances:

Opens: The abuse of `opens` directive happens when a module M only declares that it opens a package P_i to all other modules, while outside package P_o invoke any type of members m in package P_i via compile-time invocation. Therefore, this abuse type leads to compile-time errors, and one way to fix the error requires developers to modify their programs' module dependency, which is not guaranteed to work under all versions of the JDK, thus affecting attribute A4; besides, more importantly, modifying the module file will affect A1 as well as it may increase the maintenance difficulty.

$$Opens(M, P_i) \land CompInv(P_o, m, P_i) \tag{3.1}$$

Exports: An abuse instance of `exports` directive happens when a module M only declares that it exports a package P_i to all other modules, while outside package P_o invokes non-public type member m in package P_i via compile-time invocation or reflection. This type of abuse involves illegally accessing non-public members, so it mostly affects A3 but also affects A1, as illegal access to non-public members breaks the encapsulation.

$$\begin{aligned} &Exports(M, P_i) \land \exists m \in NonPublicMembers(P_i) : \\ &\quad (CompInv(P_o, m, P_i) \lor ReflInv(P_o, m, P_i)) \end{aligned} \tag{3.2}$$

Opens To: An abuse of `opens...to` directive happens when a module M_1 only declares that it opens a package P_1 to module M_2, while (1) package P_2 from module M_2 invoke any type member m in package P_1 via compile-time invocation.

$$OpensTo(M_1, P_1, M_2) \land \exists P_2 \in M_2 : CompInv(P_2, m, P_1) \tag{3.3}$$

Or (2) package P_3 from module M_3 invoke any type member m in package P_1 via compile-time invocation or reflection.

$$\begin{aligned} &OpensTo(M_1, P_1, M_2) \land \exists P_3 \in M_3 : \\ &\quad (CompInv(P_3, m, P_1) \lor ReflInv(P_3, m, P_3)) \end{aligned} \tag{3.4}$$

Situation (1) mainly leads to compile errors and extra fixes, thus affecting A1 and A4; for situation (2), this type of abuse will affect A3 since it involves illegal access and will also affect A2 since it requires exposing the current package to extra target modules.

Exports To: An abuse of `exports...to` directive happens when a module M_1 only declares that it exports a package P_1 to module M_2, while (1) package P_2 from module M_2 invoke non-public type member m in package P_1 via compile-time invocation or reflection.

$$\begin{aligned} &ExportsTo(M_1, P_1, M_2) \land \exists P_2 \in M_2 \land \exists m \in NonPublicMembers\ (P_1) : \\ &\quad (CompInv(P_2, m, P_1) \lor ReflInv(P_2, m, P_1)) \end{aligned} \tag{3.5}$$

Or (2) package P_3 from module M_3 invoke any type member m in package P_1 via compile-time invocation or reflection.

$$\begin{aligned} &ExportsTo(M_1, P_1, M_2) \land \exists P_3 \in M_3 : \\ &\quad (CompInv(P_3, m, P_1) \lor ReflInv(P_3, m, P_1))) \end{aligned} \tag{3.6}$$

Situation (1) affects A3 since it involves illegal access and affects A1 in terms of increasing maintenance difficulty. For situation (2), it is similar to the abuse situation (2) of `opens...to` directive, which affects A2 and A3.

Exports & Opens: An abuse of `exports & opens` directive happens when a module M declares that it exports and opens a package P_i to all other modules, while outside package P_o invoke non-public type member m in package P_i via compile-time invocation. Consequently, this abuse affects—similar to abuse of `exports` type—affects A1 and A3 because it leads to encapsulation breaking and illegal access to internal members.

$$Exports(M, P_i) \wedge Opens\ (M, P_i) \wedge \exists m \in NonPublicMembers(P_i) : \\ CompInv(P_o, m, P_i) \tag{3.7}$$

Exports & Opens...to: An abuse of `exports & opens...to` directive happens when a module M_1 declares that it exports a package P_1 to all other modules, and opens P_1 to module M_2, while package P_3 from module M_3 (1) reflectively invoke any type member m_1 in package P_1, or (2) invoke non-public type member m_2 in package P_1 via compile-time. This type of abuse will affect A3 as it may involve accessing non-public members and making the module open packages to extra target modules, affecting A2 as well.

$$Exports(MP_1) \wedge OpensTo(M_1, P_1, M_2) \wedge \exists P_3 \in M_3 : \\ (ReflInv(P_3, m_1, P_1) \vee \exists m_2 \in NonPublicMembers(P_1) : CompInv(P_3, m, P_1)) \tag{3.8}$$

3.2 BEAD Detection Approach and Implementation

In the previous section, I introduced various types of BSE abuse situations. This section describes how I leverage these definitions to design and implement BEAD. Figure 3.1 depicts

a high-level overview of BEAD, and BEAD is fully implemented in Java.

Specifically, BEAD consists of four steps, including JDK info scanning, JDK info combining, invocation analysis, and abuse instance static analysis. The rest of this section describes each step in depth.

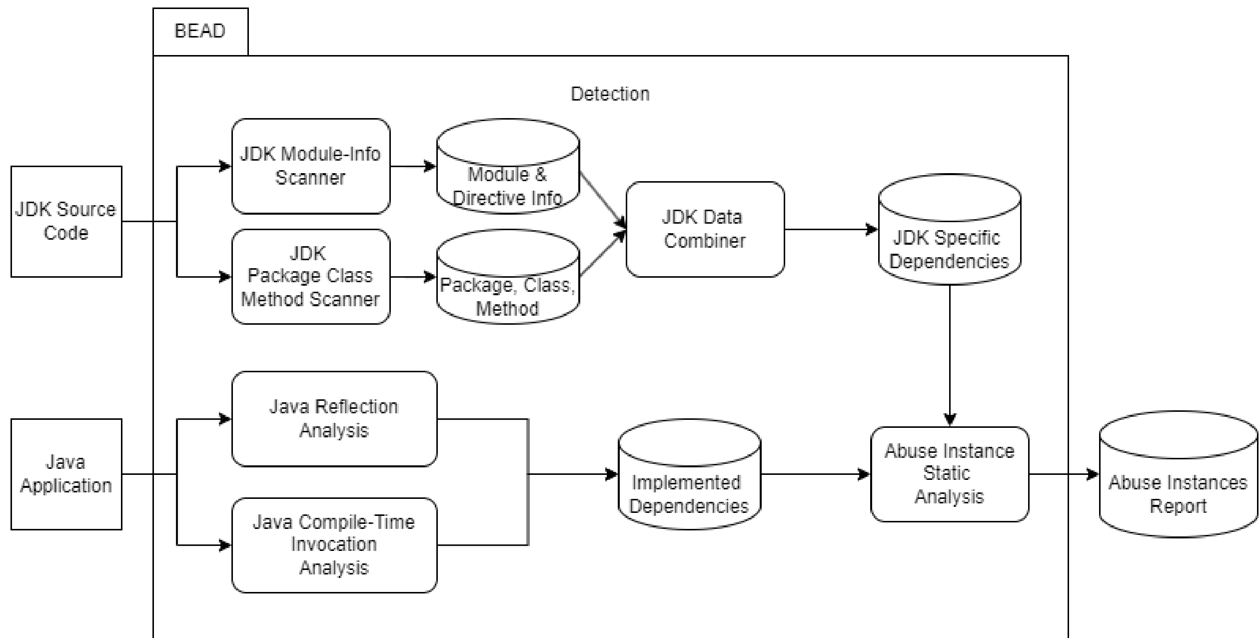

Figure 3.1: A high-level overview of BEAD

3.2.1 Step 1: JDK Info Scanning

The first step is to scan the following aspects of the selected version of JDK: all module declarations in all `module-info.class` files—and all packages in the JDK source code, classes in each package, and members in each class and their corresponding types (e.g., public, private, protected). BEAD's scanning of module declarations involves scanning all `module-info.class` files under the JDK source code with `javap`, the JDK's decompiler tool. Once the scan is complete, an additional Java script is used to extract all declarations of the exports and opens directives from the scan.

BEAD uses a parser tool called JavaParser [23] to scan for packages, classes, methods, and corresponding types. It scans the JDK source code, extracts the information of all packages, classes, methods, and corresponding types, and provides a list of information output.

3.2.2 Step 2: JDK Info Combining

The information obtained in the above steps is needed to obtain the full declaration of the `exports` and `opens` directives for a JDK module, including information about the packages, classes, and methods it contains. Therefore, BEAD has set up a data structure called *JDKModule* to record the inheritance information from the module to the method level.

Figure 3.2 shows the specific hierarchical information of *JDKModule*. The top-level stores the extracted modules from the JDK. The next level stores information at the package level, which includes (1) **Package:** the package name under the module that declares `exports(to)` or `opens(to)`; (2) **AccessRules**: specific directives for the package in the `module-info.java` file; (3) **AllowedModules**: target packages specified after the `to` in directives `exports...to` or `opens...to`. The next level of the package hierarchy stores information about each class under this package, and each class contains the signatures of each extracted method and its corresponding type.

The extraction and combination of the *JDKModule* data is done inside BEAD and output as a text file with the same structured format as depicted in Figure 3.2.

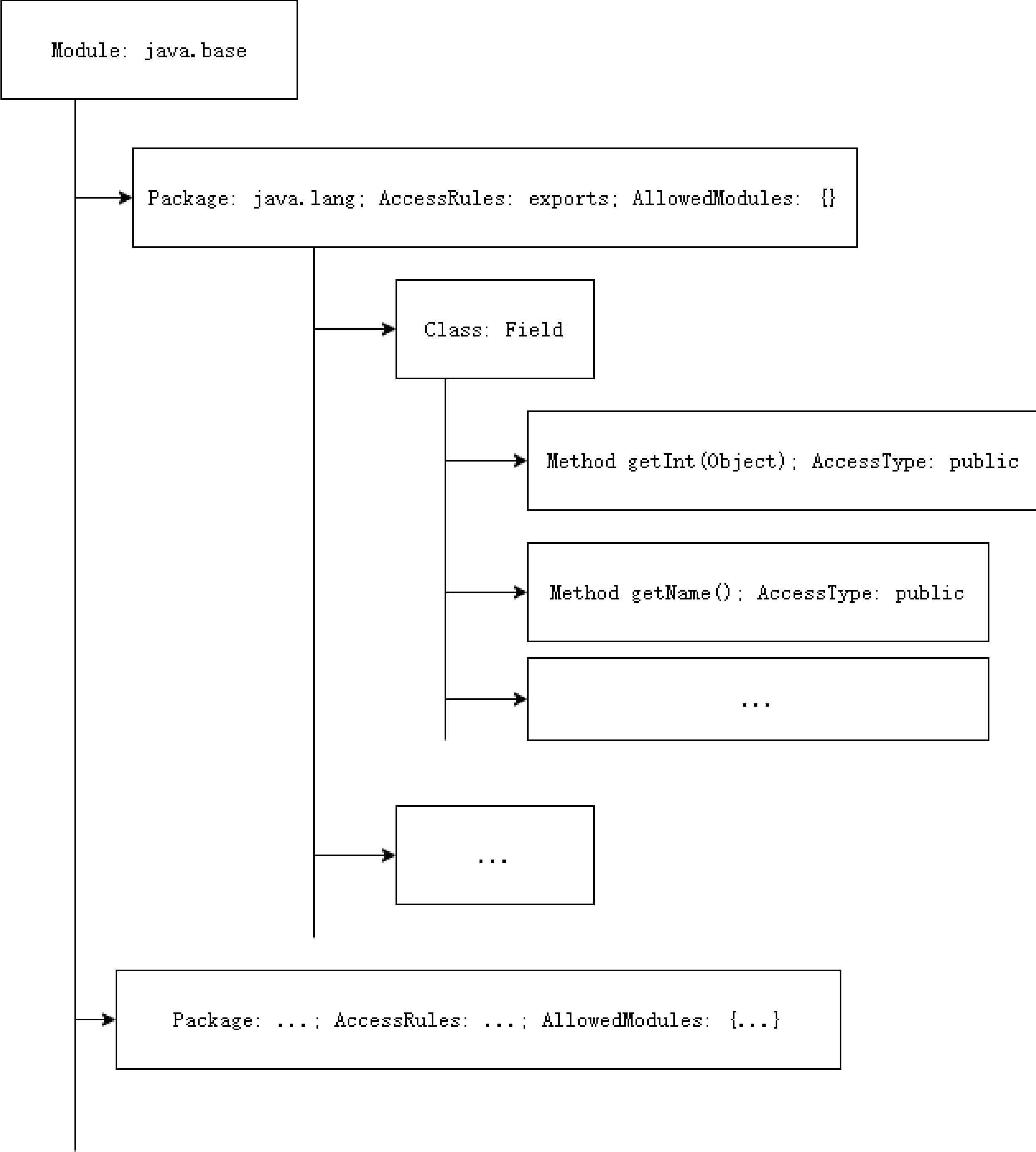

Figure 3.2: *JDKModule* structure example

3.2.3 Step 3: Invocation Analysis

Consider the definition in section 2.1.1; detecting BSE abuse instances requires analyzing both reflection and compile-time invoke to identify any instance of the eight abuse types described in section 3.1. This step takes a jar file as input and relies on static analysis to implement call analysis. More specifically, BEAD's implementation uses the Soot framework [48] to analyze the input Java application for both reflection and compile-time invocation.

The implementation of *Java Reflection Analysis* is inspired by Ghorbani et al.'s previous work on automatically detecting and repairing Java architectural abuses [28], to extract reflection invocations that occur when non-constant strings or inputs are used as the target method of a reflection call. The routine of reflection invocation analysis is presented in Algorithm 1.

Within each file path iteration, BEAD loads the input classes and analyzes each method in the loaded classes. For concrete methods, it retrieves the method body and analyzes each statement to identify if it is a reflection invocation. Once the reflection invocation is identified, BEAD will process the invoked method to obtain the identified method name, then update the method and its count into the stored map. The invocation map will be used to analyze abuse instances in section 3.2.4.

Algorithm 1: Reflection Invocation Analysis

input : A list of class file paths

output: Counts for Full Method Names

```
1  let s denote the set of class file paths and let C_full denote the map to store counts of full
     method names;
2  while s is not empty do
3      load and analyze class from s;
4      for each method in the class do
5          if method is concrete then
6              retrieve and analyze the method body;
7              for each statement in the method body do
8                  if statement involves a reflection invocation then
9                      if reflection call is identified then
10                         identify details of the reflection call;;
11                         update C_full based on the method and class name availability;
12                     end
13                 end
14             end
15         end
16     end
17     remove class file from s;
18 end
19 output stored methods in C_full;
```

The analysis of compile-time invocation for input Java programs is also implemented on the Soot framework. Using the call graph among methods generated by Soot, BEAD reads the nodes of these calls and extracts all the compile-time invocations towards JDK's internal classes. Algorithm 2 describes the specific algorithm routines.

Algorithm 2: Compile-Time Invocation Analysis

input : A list of class file paths

output: A map of compile-time method invocations to JDK classes

```
1  let s denote the set of class file paths and let C denote the map to store the compile-time
     call edge;
2  foreach class path in the input list do
3  |   load necessary classes from the class path;
4  end
5  determine entry points for analysis;
6  run analysis packs to generate the call graph;
7  foreach edge in the call graph do
8  |   if source and target methods exist then
9  |   |   if target method is static and declared in a JDK class then
10 |   |   |   if source method is not declared in a JDK class then
11 |   |   |   |   record the compile-time call edge from the source method to the target
   |   |   |   |     method;
12 |   |   |   end
13 |   |   end
14 |   end
15 end
16 output stored compile-time call edges in C;
```

Within each file path iteration, BEAD loads the input classes and determines the entry points to construct the call graph. For each edge in the generated call graph, BEAD first examines both the source and target methods that exist, then checks whether it is a compile-time invocation to a method from JDK by checking if the target method is static. After finishing the target method check, BEAD performs an extra check to filter out JDK internal compile-time invocation to those target methods. Finally, BEAD outputs the stored

compile-time call edge map for abuse instance analysis in section 3.2.4.

3.2.4 Step 4: Abuse Instance Analysis

By incorporating the combined *JDKModule* data obtained from step 3.2.2 and the analyzed reflection and compile-time invocation results generated from step 3.2.3 for the input Java program, BEAD detects BSE abuse instances described in section 3.1 and reports any potential BSE abuse instances to provide references for the further repair.

Chapter 4

Case Study: Evaluation of BEAD

To assess the effectiveness of BEAD, I study the following research questions.

- **RQ1:** To what extent does JDK declare `exports` and `opens` directives?
- **RQ2:** How frequently do abuse instances occur under subject applications?
- **RQ3:** What are the JDK APIs detected by BEAD as abused?
- **RQ4:** How does BEAD compare to the JDK's mechanisms for detecting abuse instances?
- **RQ5:** What's the efficiency of BEAD?

To answer those research questions, I conducted a case study and used JDK 17 as the module declaration reference for detecting BSE-related abuse instances. It should be noted that I chose JDK 17 as the module description reference because JDK 17 removes one of the launch options that can be used to access internal APIs in previous JDK versions, i.e. `--illegal-access`, which means that source code implemented before JDK 17 cannot access

internal APIs via the `--illegal-access` option when running with JDK 17, as explained in Section 2.2.

The case study is performed with five Java open-source projects on GitHub [3], a large and widely used open-source repository of software projects. The repository information of those five projects is listed in Table 4.1.

Table 4.1: Subject Applications

No.	Application Name	Rel. Version	# Stars	# Forks
1	arthas-core	3.6.7	34.7k	7.3k
2	error-prone-check-api	2.5.1	6.7k	724
3	google-java-format	1.22.0	5.4k	843
4	cglib	3.3.0	4.7k	885
5	darklaf-core	2.6.1	416	39

4.1 RQ1: JDK Module Interfaces Design

To better understand and detect potential abuse due to the BSE problem, Step 3.2.1 scanned all the declarations related to *opens(to)* and *exports(to)* in the JDK 17 module. The scanning revealed a total of 372 packages in JDK 17 that were found to be declared with the *opens(to)* and *exports(to)* directives. The details are distributed as follows:

- 228 packages declared with *exports* directive
- 142 packages declared with *exports...to* directive
- 2 packages declared with *opens* directive
- 3 packages declared with *opens to* directive

Besides, as mentioned in Section 3.1, there are three packages whose declarations involve more than two directives. Those packages are (1) package `javax.swing.plaf.basic` from module `java.desktop` declared both *exports* and *opens to* directives; (2) package `sun.misc` and `sun.reflect` from module `jdk.unsupported` declared both *exports* and *opens* directives at the same time. These findings indicate that JDK rarely opens its internal packages to the external or specified modules when it actually implements APIs and related module dependencies, which is consistent with the JPMS principle of maintaining greater encapsulation and reducing related security vulnerabilities by limiting reflection.

4.2 RQ2: Result Analysis Of BEAD for Detecting BSE Abuse Instances

Table 4.2: Identified Abuse Instances of Subject Applications

Application Name	# Total Abuse Inst.	# Reflection Abuse Inst.	# Compile-Time Abuse Inst.	Abused Type Of Expose Packages					
				O	O.T	E	E.T	E & O.T	E & O
arthas-core	4	1	3	-	-	1	3	-	-
error-prone-check-api	119	2	117	-	-	-	119	-	-
google-java-format	44	0	44	-	-	-	44	-	-
cglib	1	1	0	-	-	1	-	-	-
darklaf-core	19	1	18	-	-	1	18	-	-

(O: opens, O.T: Opens To, E: Exports, E.T: Exports To)

Table 4.2 shows, for each subject application, the total number of abuse instances BEAD found and separates them by their type. As depicted in Table 4.2, most of the abuse instances occurred during the compile-time invocations, which indicates that unauthorized access to internal packages through compile-time invocations is more common in Java 9+ applications, thus resulting in architecture rigidity and potential security vulnerabilities.

On the other hand, Table 4.2 indicates that most of the abuse instances are of types *exports* and *exports...to*. Among those detected abuse instances, the high frequency of type *exports...to* demonstrates that most instances of abuse occur when unspecified authorized modules access the JDK's internal APIs, which were designed to be used only for internal self-invocations inside the JDK. Furthermore, Table 4.2 also shows that multiple instances are abused with the *exports* directive, which means the existence of external packages accessing non-public members declared by the JDK's internal APIs. Both types of abuse increase the risk of encountering illegal access-related exceptions and errors at the runtime, thus increasing the maintenance difficulty of these Java applications.

Additionally, Table 4.2 indicates that no instances of abuse involving the directives *opens*, *opens to*, *exports & opens to*, and *exports & opens* have been detected. As detailed in Section 4.1, the occurrence of these four directives in JDK packages is significantly lower than that of the other two, which reduces the possibility of detecting abuse instances for these four directives and even unable to detect those four directives. Despite their low prevalence, BEAD covers all six directive situations because they have potential risks to appear during the future usage of JPMS.

Overall, BEAD has detected a total of 187 abuse instances under selected subject applications. Among them, 182 instances are detected during the compile-time, and 5 instances are detected under reflections. To evaluate the result, I performed an extra manual examination for source codes from each subject application and confirmed that all 187 abuse instances do exist from the code in each subject application, which shows that the BEAD is capable of correctly detecting abuse instances.

4.3 RQ3: Detected JDK Abuse Instances and Commonly-Abused Packages

4.3.1 Abuse Instances of Reflection

Table 4.3: Abused Instance via. Reflection Invocation in Subject Applications

Application Name	Abused API
arthas-core	java.lang.ClassLoader.defineClass (String name, byte[] b, int off, int len)
error-prone-check-api	com.sun.tools.javac.comp.Resolve.findIdent com.sun.tools.javac.util.Log.error
google-java-format	N/A
cglib	java.lang.ClassLoader.defineClass (String name, byte[] b, int off, int len, ProtectionDomain protectionDomain)
darklaf-core	javax.swing.JRootPane.setUseTrueDoubleBuffering (boolean useTrueDoubleBuffering)

Table 4.3 shows the detected abuse instances under the reflection invocation analysis by Step 3.2.3. Based on these five different abuse instances of reflection, I investigated these APIs and the intended functionality sought by developers.

- Method `java.lang.ClassLoader.defineClass(String name,byte[] b,int off, int len)` and `java.lang.ClassLoader.defineClass(String name,byte[] b, int off,int len,ProtectionDomain protectionDomain)`: Those two APIs are used in `ClassLoader` to convert an array of bytes into an instance of class `Class`, while the latter provides an option `ProtectionDomain`, a class that encapsulates the characteristics of a domain for granting a set of permissions to a class based on a policy at execution time.

- Method `com.sun.tools.javac.comp.Resolve.findIdent`: This is an internal API from the class `Resolve`, a helper class for name resolution. This method finds an unqualified identifier that matches a specific type (variable, type, or package) in the current compilation environment. The access level of this method is set to package-private, i.e., it can only be accessed by other classes in the same package.

- Method `com.sun.tools.javac.util.Log.error`: An internal API from the class `Log` inherited from class `com.sun.tools.javac.util.AbstractLog`, which is used to report a localized error message.

- Method `javax.swing.JRootPane.setUseTrueDoubleBuffering(boolean useTrueDoubleBuffering)` method: An internal API stored in class `JRootPane`, a container to hold all the components of the graphical user interface (GUI), including the title bar, menu bar, and content areas. The purpose of this method is to configure if the current GUI uses true double buffering to minimize or eliminate flickering when GUI elements are repainted and to enhance the visual appearance of the user interface. The access level of this method is set to package-private as well.

The above abuse instances illustrate that the internal APIs involved by those detected abuses may concern not only high-risk tasks, such as creating an anonymous class without standard verification (`java.lang.ClassLoader.defineClass`) but also low-level operations, such as accessing the compiler (`com.sun.tools.javac.comp.Resolve.findIdent` and `com.sun.tools.javac.util.Log.error`). To a broader impact, those detected abuse instances indicate that those detected abuse instances may also bring potential security vulnerability, compromise program stability, and increase related maintenance costs beyond breaking the strong encapsulation provided by JPMS.

On the other hand, it is worth noting that some instances have already been reported in the corresponding subject application's GitHub repository issues. For example, the

abuse of `java.lang.ClassLoader.defineClass` by `cglib` has been reported to cause an `InaccessibleObjectException` when running the program [6], which has significantly impacted the normal usage for `cglib` users. Unfortunately, this registered GitHub issue still remains the status as open, indicating that the `cglib` developers have not yet found a suitable fix solution, and other `cglib` users might also be unable to effectively run `cglib` due to this abuse.

4.3.2 Abuse Instances of Compile-Time Invocation

Regarding the detected abuse instances under the compile-time invocation, I analyzed all 184 instances of compile-time invocation-related abuse and investigated their contained modules and the intended functionality. Table 4.4 listed the five most frequently abused packages and their corresponding counts.

Table 4.4: Top 5 Abused Packages Under Compile-Time Invocation

Package Name	# Abuse Inst.	Proportion(%)
com.sun.tools.javac.util	46	25.27%
com.sun.tools.javac.code	37	20.33%
com.sun.tools.javac.parser	31	17.03%
sun.swing	18	9.89%
com.sun.tools.javac.comp	12	6.59%

As depicted in Table 4.4, four packages are all sub-packages under the same package, `com.sun.tools.javac`, and belong to the module `jdk.compiler`. Among them, `com.sun.tools.javac.util` provides general tools and data structures, `com.sun.tools.javac.code` manages the type system and symbol table, `com.sun.tools.javac.parser` is responsible for parsing source code into abstract syntax trees, and `com.sun.tools.javac.comp` per-

forms semantic analysis and other critical steps during the compilation process. These four packages constitute the main functionalities of the Java compiler `javac` and are declared with *exports...to* under the corresponding `module-info.java` file. In addition, Table 4.4 illustrates a concern about the wide abuse of package `com.sun.tools.javac` in existing Java programs. This suggests that developers need to be aware of the potential risk that the `com.sun.tools.javac` package could break the strong encapsulation provided by JPMS but also emphasizes the urgency of re-modifying the accessibility of the `Javac`'s module or finding an alternative API with similar functionality. Otherwise, external applications that rely on the functionality of `Javac` may have to force abusing those packages in their implementations, which leads to a break in the module's strong encapsulation and potentially causes more severe maintenance issues, such as exceptions or errors.

4.4 RQ4: BEAD Against JDK Compiler's Abuse Detection

To better assess the effectiveness of BEAD for detecting abusive instances, we need compare it with the current level of abuse detection available from the JDK compiler, javac. For the assessment, I've set up nine types of information to represent the information contained in an abuse instance as follows:

- **Source Module**: The module information of the abuse source.
- **Source Package**: The package information of the abuse source.
- **Source Class**: The class information of the abuse source.
- **Source Statement**: The source of the code statement that specifically causes the abuse.

- **Abuse Reason**: The brief explanatory information on the causes of abuse.
- **Target Module**: The specific JDK module being abused.
- **Target Package**: The specific package being abused.
- **Target Class**: The specific class being abused.
- **Target Function**: The specific function (or method) being abused.

4.4.1 Javac's Detection BSE Detection at Compile Time

Since javac can only handle compile-time tasks, when an external package tries to access a package that has not been exported, it raises a compile error and is caught by javac. Table 4.5 shows the details of the information that javac can detect for different abuse types of expose packages.

Table 4.5: Abuse Information Provided by Javac at Compile-Time

Abused Type Of Expose Packages	Abused Information								
	S.M	S.P	S.C	S.T	A.R	T.M	T.P	T.C	T.F
Opens	✗	✗	✗	✗	✗	✗	✗	✗	✗
Opens...To	✗	✗	✗	✗	✓	✓	✓	✓	✗
Exports	✗	✗	✗	✗	✓	✓	✓	✓	✗
Exports...To	✗	✗	✗	✗	✓	✓	✓	✓	✗
Exports & Opens...To	✗	✗	✗	✗	✓	✓	✓	✓	✗
Exports & Opens	✗	✗	✗	✗	✗	✗	✗	✗	✗

(S.M: source module, S.P: source package, S.C: source class, S.T: source statement, A.R: abuse reason, T.M: target module, T.P: target package, T.C: target class, T.F: target function)

As described in Table 4.5, although javac can catch compilation errors caused by abusing packages, the information provided by javac is quite limited. Figure 4.1 displays an example of abuse error messages returned by javac. It can be seen that javac only provides information about the abused package, involved target module, and very limited explanations for the abuse reason.

```
import com.sun.java.swing.plaf.windows.WindowsLookAndFeel;
                                  ^
  (package com.sun.java.swing.plaf.windows is declared in
    ↪ module java.desktop, which does not export it)
```

Figure 4.1: Compile Error message for Un-exported Packages

Besides, it should be noted that since the only two packages declared as `opens` in JDK 17 are also declared with the `exports` directive, javac is unable to detect `opens` type abuse because they do not cause compile errors at compile time.

```
Detected abuse under module jdk.compiler
Involved Source Method: <com.google.errorprone.ErrorProneAnalyzer:
   ↪ void finished(com.sun.source.util.TaskEvent)>; Involved Target
   ↪  Method: instance(com.sun.tools.javac.util.Context) in target
   ↪ class: com.sun.tools.javac.main.JavaCompiler from package com.
   ↪ sun.tools.javac.main
Abuse Reason: The project tries to invoke target method instance(com
   ↪ .sun.tools.javac.util.Context) at compile time, but com.sun.
   ↪ tools.javac.main.JavaCompiler only exports to [jdk.javadoc,
   ↪ jdk.jshell]
```

Figure 4.2: Example Output of BEAD Abuse Detection Under Compile Time

Figure 4.2 shows an example of an abuse report that has been detected by BEAD during compile time. Compared to javac, BEAD offers more comprehensive and detailed detection of abuse occurring at compile-time invocation, and Table 4.6 illustrates the specific types of abuse information that BEAD can detect and report compile-time invocation.

Table 4.6: Abuse Information Provided by BEAD at Compile-Time

Abused Type Of Expose Packages	Abused Information								
	S.M	S.P	S.C	S.T	A.R	T.M	T.P	T.C	T.F
Opens	✗	✓	✓	✓	✓	✓	✓	✓	✓
Opens...To	✗	✓	✓	✓	✓	✓	✓	✓	✓
Exports	✗	✓	✓	✓	✓	✓	✓	✓	✓
Exports...To	✗	✓	✓	✓	✓	✓	✓	✓	✓
Exports & Opens...To	✗	✓	✓	✓	✓	✓	✓	✓	✓
Exports & Opens	✗	✓	✓	✓	✓	✓	✓	✓	✓

(S.M: source module, S.P: source package, S.C: source class, S.T: source statement, A.R: abuse reason, T.M: target module, T.P: target package, T.C: target class, T.F: target function)

From Table 4.5 and 4.6, it is evident that, on the one hand, BEAD provides additional abuse detection for scenarios involving the "opens" and "exports & opens" directives beyond the abuse types that javac can handle. On the other hand, compared to the limited information on abuse instances provided by javac, BEAD can offer more detailed reports on abuse instances occurring at compile-time, which includes the source package, class, specific code statement causing the abuse, and the target function being abused. Such detailed information provided by BEAD can more effectively help developers pinpoint the source and target of the abuse, thus avoiding unnecessary time spent on debugging to locate which code caused the abuse.

4.4.2 Detection of Reflection-Oriented BSE Abuse

Regarding reflection invocation, since the JDK compiler primarily handles compile-time tasks, the access check of reflections occurs at runtime. The detection of abuse and associated access checks during runtime are handled by the `Java Virtual Machine (JVM)` [7], instead of javac.

By investigating the illegal access warning messages and specific error or exception reports provided by the JVM, I found that the JVM has already effectively captured potential abuse using reflection. Table 4.7 presents the types of abuse the JVM can detect, along with the details and categories of information it can return.

Table 4.7: Abuse Information Provided by JVM via. Reflection

Abused Type Of Expose Packages	Abused Information								
	S.M	S.P	S.C	S.T	A.R	T.M	T.P	T.C	T.F
Opens	✗	✓	✓	✓	✓	✓	✓	✓	✓
Opens...To	✓	✓	✓	✓	✓	✓	✓	✓	✓
Exports	✗	✓	✓	✓	✓	✓	✓	✓	✓
Exports...To	✓	✓	✓	✓	✓	✓	✓	✓	✓
Exports & Opens...To	✓	✓	✓	✓	✓	✓	✓	✓	✓
Exports & Opens	✗	✓	✓	✓	✓	✓	✓	✓	✓

(S.M: source module, S.P: source package, S.C: source class, S.T: source statement, A.R: abuse reason, T.M: target module, T.P: target package, T.C: target class, T.F: target function)

As depicted in Table 4.7, the JVM not only has the capability to detect each type of abuse but also provides comprehensive enough information to help developers identify the specific cause of the abuse. Additionally, the JVM returns different messages on vary scenarios. For instance, in JDK 17, the JVM returns error or exception (e.g., `IllegalAccessException`)

messages and corresponding stack traces to report abuse instances of accessing non-public members within an `unopened` package.

Moreover, the JVM returns the illegal access-related warnings, instead of error or exception reports, for abuse detection of Java programs running between JDK 9 and 16, especially those programs that adopt the `--illegal-access=permit` option. Figure 4.3 shows an example of an illegal reflective access warning message when running `cglib` [5]. From Figure 4.3, the JVM returns those warnings, including details about the source package, source class, target package, target class, and target function, during runtime. In contrast, BEAD provides equivalent effective details of abuse information without running the program as shown in Figure 4.4, which addresses potential abuse instances at an early stage.

```
WARNING: Illegal reflective access by net.sf.cglib.core.ReflectUtils
    ↪ \$1 (file:/xx/xx/cglib-3.3.0.jar) to method java.lang.
    ↪ ClassLoader.defineClass()
```

Figure 4.3: JVM Abuse Warning Message Example

```
Detected abuse under module java.base
Source method: <net.sf.cglib.core.ReflectUtils$1: java.lang.Object
    ↪ run()> from class: net.sf.cglib.core.ReflectUtils$1
Involved Method: java.lang.ClassLoader.defineClass in target class:
    ↪ ClassLoader from package java.lang
Abuse Reason: The project tries to reflectively invoke this method,
    ↪ but java.lang.ClassLoader.defineClass is protected
```

Figure 4.4: BEAD Reflection Abuse Report Example

Table 4.8 illustrates the detection of abuse types and the extraction of detailed information by BEAD in terms of reflection. Unlike the JVM, BEAD cannot extract the information

about the module to which the source package belongs, as Soot cannot analyze the package's modules. Nevertheless, BEAD still provides detailed abuse information and specifics, enabling developers to effectively identify the source and cause of the abuse.

Table 4.8: Abuse Information Provided by BEAD via. Reflection

Abused Type Of Expose Packages	Abused Information								
	S.M	S.P	S.C	S.T	A.R	T.M	T.P	T.C	T.F
Opens	✗	✓	✓	✓	✓	✓	✓	✓	✓
Opens...To	✗	✓	✓	✓	✓	✓	✓	✓	✓
Exports	✗	✓	✓	✓	✓	✓	✓	✓	✓
Exports...To	✗	✓	✓	✓	✓	✓	✓	✓	✓
Exports & Opens...To	✗	✓	✓	✓	✓	✓	✓	✓	✓
Exports & Opens	✗	✓	✓	✓	✓	✓	✓	✓	✓

(S.M: source module, S.P: source package, S.C: source class, S.T: source statement, A.R: abuse reason, T.M: target module, T.P: target package, T.C: target class, T.F: target function)

Overall, compared to the original detection mechanisms of the JVM, BEAD provides a detailed and comprehensive report on the abuse of JDK packages by Java programs. Another advantage of BEAD is its ability to detect abuse instances without running the program. Therefore, even though the JVM's detection of abuse related to reflection invocation is quite comprehensive, BEAD, through its adoption of static analysis techniques, can identify potential abuses of JDK packages at an early stage. This early detection can significantly reduce the maintenance burden associated with frequent debugging in later stages.

Additionally, BEAD focuses specifically on analyzing the reasons for the abuse, aiming to help Java developers more accurately locate the causes of the abuse, thus, in turn, helping mitigate the risks associated with the abusing JDK internal APIs as much as possible.

4.5 RQ5: Efficiency of BEAD

As described in Section 3.2, BEAD mainly builds on Soot [48]. As a result, to assess BEAD's efficiency, I answer the RQ5 in terms of Soot's execution time, as well as BEAD's execution time of inconsistency checking. I ran all the evaluation experiments under the Windows operation system (3.60 GHz AMD Ryzen 7 3700 8-Core, 32 GB, Windows 11 22H2).

Table 4.9: Result of Execution Time

Component	Avg. Execution Time (ms)
Reflection Invocation Analysis	6126
Compile-Time Invocation Analysis	7746
Reflection Abuse Analysis	7
Compile-Time Abuse Analysis	72
Total	13951

Table 4.9 shows the average execution time of BEAD for five subject applications. For invocation analysis, the BEAD takes about 6 seconds to analyze reflection invocation and about 7 seconds to analyze compile-time invocation. For the abuse detection phase, it takes about 0.07 seconds and 0.7 seconds to check the abuse instances via. reflection and compile-time invocation. It should be noted that the reason why the invocation analysis process takes the majority of the execution time is that Soot's analysis needs to check all the paths and load the relevant classes that depend strongly on the size of the analyzed programs. Still, on average, BEAD takes about 13 seconds to execute the invocation analysis and abuse detection process, which is reasonably efficient for the detection work.

Chapter 5

Discussion

5.1 Implications

In this study, I formally define the eight types of BSE abuse instances and propose BEAD to detect such abuse instances. In this section, I want to emphasize the importance of BEAD and highlight the need to design more reasonable module interfaces and develop tools for future repair work.

For practitioners: On the one hand, the introduction of BEAD can help program developers detect potential abuse of JDK packages in their source code before runtime, thus preventing them from adopting workarounds that would break the strong encapsulation of Java modules and leave technical debt that may increase maintenance complexity. BEAD can also help the client program developers check whether their dependencies have potential abuse to avoid the current client program abuse because those dependencies are abusing internal APIs, which may lead to severe security vulnerabilities. In addition, section 4.3 lists the commonly abused APIs in the BSE abuse instances detected by BEAD, which can be used for reference by Java platform maintainers as to which APIs are more likely to be

abused, and in turn, help them to adjust those APIs architecturally.

On the other hand, the BSE abuse instances detected by BEAD, as listed in Section 4.2, illustrate the extent to which the BSE problem exists and proliferates in JDK 9+ open-source programs. In addition, Section 4.3 lists the commonly abused APIs in the BSE abuse instances detected by BEAD, which emphasizes the urgency of adjusting the module dependencies of these APIs to prevent more possible severe errors and exceptions occurring in the future. Section 4.4 reveals the current limitations of current JDK in terms of abuse instance detection mechanisms, especially for the detection involving reflection, thus further emphasizing the significance of BEAD's work in helping developers to identify potential abuses in the source code of their programs more accurately, so that developers can fix their code with respect to the specific abuse cause.

For researchers: Our previous study of the BSE problem showed evidence that most BSE problems (including illegal access warnings, errors, and exceptions) occur during the runtime [33]. The introduction of BEAD can effectively help researchers achieve early detection of program abuse and serve as a reference for future studies that may automate repairs of BSE. Considering the importance of robust encapsulation brought by JPMS for the Java development community, automated BSE repair can significantly aid in the adoption of JPMS and facilitate the overall migration process. Moreover, automated repair of BSE can reduce the risk of functionality disruption of many downstream projects due to overly strict encapsulation.

In addition, determining effective strategies for modularizing Java projects is also a significant challenge for the maintenance work of JPMS. The reports of misuse instances provided by BEAD will, to some extent, assist researchers in determining the best arrangement of classes within Java modules [31] and designing proper module interfaces, which helps to maintain a fitting balance between allowing an appropriate degree of internal access and maintaining reliability, security, and maintainability.

5.2 Threats to Validity

Internal Threats. The main internal threat to validity stems from the potential inaccuracies of static analysis tools used during the invocation step, specifically the risk of false positives and negatives. Consequently, these inaccuracies could lead BEAD to incorrectly report abuse instances or overlook some important instances in the analysis and detection phase. This may result in certain service errors or unauthorized accesses going undetected. Since BEAD relies on Soot's result for the inconsistency analysis, it inherits all the limitations of Soot, which in turn impacts its ability to detect abuse instances accurately, based on the precision of Soot. However, Soot is a widely used [34] [35] and actively maintained framework for Java programs' static analysis work [4]. Therefore, to mitigate the risk of false positives and negatives, I further manually review the registered GitHub issues of subject applications and consult the database gathered from previous BSE study [33] to verify each identified abuse instance. This strategy reduces the possibility that inaccuracies in the underlying static analysis framework negatively affect the reliability of BEAD.

External Threats. One threat to external validity is that the current invocation analysis limits the target method at the JDK level. As a result, BEAD is currently unable to detect abuse instances not directed at the JDK itself (e.g., abuse instances of modules in a Java application or non-JDK platform). Nevertheless, the concepts and strategies for detection in this work can be extended relatively easily to Java modules outside of the JDK. More specifically, the target modules of a BSE type in our work can be replaced with non-JDK modules with minimal or no change to the defined abuse types or the approach design.

Another external threat is the JDK version used during the evaluation phase. In this study, I only used JDK 17 as the reference for the design of JDK module interfaces. Considering the fact that the JDK designers may modify JPMS designs and their corresponding implementations in different versions, this may cause differences in the results of detecting abuse

instances. Such a future study is out of the scope of this work, but future work can study abuse instances detection under different JDK 9+ versions and compare them in future work to mitigate the risk of detection result validity due to JDK version differences.

BEAD's evaluation is limited to the selection and quantity of Java subject applications used in the evaluation dataset. To minimize this threat, I randomly selected open-source Java applications that have already reported BSE problems based on our previous study [33]. Additionally, to address the quantity threat, future research is recommended to expand the scope of BEAD's evaluation by including more subject applications.

Chapter 6

Related Work

JPMS has offered a tool to transform from object-oriented design into component-based design. However, the transformation process is challenging. To simplify this process, Hammad et al. proposed the OO2CB tool [31] to introduce a simplified approach by promoting least privilege modularity. By introducing the implementations for automatic detection and corrections, Darcy's [28] work aimed to fix the redundant dependency inconsistent problem, as well as to improve the integrity of JPMS systems. To the sensitive information inside the module, Dann et al. implemented the ModGuard [26], which can pre-secure the sensitive information through dependency analysis to minimize unauthorized access to the data. Lastly, for understanding and managing architectural changes, Mondal et al. [39] proposed the technique of semantic partitioning changes to further improve the maintenance and review of JPMS projects.

On the other hand, studies on the Health of Java Software ecosystems are steadily progressing, which involves the understanding of Java libraries, dependency management, and the complexities of Java-based ecosystems. These studies have now spread to several domains: (1) the dependency conflict problem [50] [51] [52], (2) language features [54], (3) library

migration [30] [32] [49], and (4) the compilation process [38]. They explore the security risks [37] [49] [53] and reasons behind developer actions [32] [54] in these domains to introduce automated tools [50] [51] [53], frameworks [38], empirical solutions [52], and practical insights [30] [37] [49] that can better improve the Java development community and related ecosystem challenges.

Chapter 7

Conclusion

In this thesis, I formally define 8 types of abuse instances related to the BSE problem and introduce BEAD, an automatic detection tool leveraging static analysis to detect encapsulation abuse instances due to the BSE problem. In particular, BEAD scans the specific version of the JDK and extracts the information of module declaration details, inside packages, classes, and methods from the input JDK. BEAD also takes the jar file of the Java program as input, analyzes both the reflection and compile-time invocations to extract potential abuse information, including source & target packages, classes, functions, and specific abuse actions. Within retrieved information, BEAD checks inconsistencies between JDK-implemented module details and invocation details from subject applications and reports detected abuse instances to developers from subject applications. The result of the case study indicates a pervasive existence of abuse instances related to the BSE problem among open-source Java 9+ applications. Possible future directions of this study include (1) enhancing the detection capabilities of BEAD to cover non-JDK modules and applications, (2) adding more testing results under different JDK 9+ versions except JDK 17, (3) expanding the scope of the detection database by adding more open-source Java 9+ applications.

Bibliography

[1] Oracle Corporation. API specification for the Java Platform, Standard Edition: Class ServiceLoader. https://docs.oracle.com/javase/7/docs/api/java/util/ServiceLoader.html.

[2] Project Jigsaw. http://openjdk.java.net/projects/jigsaw/, 2017.

[3] GitHub. https://github.com, 2018.

[4] Soot GitHub Issue. https://github.com/Sable/soot/issues, 2018.

[5] jdk11 version warning. https://github.com/cglib/cglib/issues/184, 2020.

[6] Java 16 and 17 compatibility. https://github.com/cglib/cglib/issues/191, 2021.

[7] Java Virtual Machine Specification, Java SE 21 Edition. https://docs.oracle.com/javase/specs/jvms/se21/html/jvms-5.html#jvms-5.4.4, 2023.

[8] google-java-format. https://github.com/google/google-java-format, 2024[Accessed 04-22-2024].

[9] google-java-format (and removeunusedimports) broken on jdk 16+ (has workaround) #834. https://github.com/diffplug/spotless/issues/834, 2024[Accessed 04-22-2024].

[10] javaguide. https://google.github.io/styleguide/javaguide.html, 2024[Accessed 04-22-2024].

[11] junit5/gradle.properties. https://github.com/junit-team/junit5/blob/46d0f80db0d6fc5faced28e9827683a09e7f8fb9/gradle.properties#L13, 2024[Accessed 04-22-2024].

[12] removeunusedimports fails on java 17 #871. https://github.com/diffplug/spotless/issues/871, 2024[Accessed 04-22-2024].

[13] Spotless: Keep your code spotless. https://github.com/diffplug/spotless, 2024[Accessed 04-22-2024].

[14] Junit5. https://github.com/junit-team/junit5, 2024[Accessed 04-24-2024].

[15] Code on the class path - the unnamed module. https://dev.java/learn/modules/unnamed-module/, 2024[Accessed 04-28-2024].

[16] Java 9 migration guide: The seven most common challenges. https://nipafx.dev/java-9-migration-guide/, 2024[Accessed 04-28-2024].

[17] Java platform, standard edition tools reference. https://docs.oracle.com/javase/9/tools/java.htm#JSWOR624, 2024[Accessed 04-28-2024].

[18] Java platform, standard edition tools reference. https://docs.oracle.com/javase/10/tools/java.htm#JSWOR624, 2024[Accessed 04-28-2024].

[19] Java platform, standard edition tools reference. https://docs.oracle.com/en/java/javase/11/tools/java.html#GUID-3B1CE181-CD30-4178-9602-230B800D4FAE, 2024[Accessed 04-28-2024].

[20] Java platform, standard edition tools reference. https://docs.oracle.com/en/java/javase/16/docs/specs/man/java.html, 2024[Accessed 04-28-2024].

[21] Java platform, standard edition tools reference. https://docs.oracle.com/en/java/javase/17/docs/specs/man/java.html, 2024[Accessed 04-28-2024].

[22] A peek into java 17: Encapsulating the java runtime internals. https://blogs.oracle.com/javamagazine/post/a-peek-into-java-17-continuing-the-drive-to-encapsulate\-the-java-runtime-internals, 2024[Accessed 04-28-2024].

[23] Javaparser. https://github.com/javaparser/javaparser, 2024[Accessed 05-06-2024].

[24] A. Buckley and M. Reinhold. JEP 396: Strongly Encapsulate JDK Internals by Default — openjdk.org. https://openjdk.org/jeps/396, (Accessed on 04-28-2024), 2023.

[25] A. Buckley and M. Reinhold. JEP 403: Strongly Encapsulate JDK Internals — openjdk.org. https://openjdk.org/jeps/403 (Accessed on 04-28-2024), 2023.

[26] A. Dann, B. Hermann, and E. Bodden. Modguard: Identifying integrity & confidentiality violations in java modules. *IEEE Transactions on Software Engineering*, 47(8):1656–1667, 2019.

[27] P. Deitel. Understanding Java 9 Modules. https://www.oracle.com/corporate/features/understanding-java-9-modules.html, (Accessed on 04-28-2024), 2017.

[28] N. Ghorbani, T. Singh, J. Garcia, and S. Malek. Darcy: Automatic architectural inconsistency resolution in java. *IEEE Transactions on Software Engineering*, 2024.

[29] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, and D. Smith. The Java® Language Specification — docs.oracle.com. https://docs.oracle.com/javase/specs/jls/se9/html/index.html, (Accessed on 04-24-2023), 2017.

[30] H. Gu, H. He, and M. Zhou. Self-admitted library migrations in java, javascript, and python packaging ecosystems: A comparative study. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 627–638. IEEE, 2023.

[31] M. M. Hammad, I. Abueisa, and S. Malek. Tool-assisted componentization of java applications. In *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, pages 36–46. IEEE, 2022.

[32] H. He, R. He, H. Gu, and M. Zhou. A large-scale empirical study on java library migrations: prevalence, trends, and rationales. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 478–490, 2021.

[33] Y. He, Y. Chen, J. Ayala, Y. Zhou, Q. Wang, and J. Garcia. Breaking strong encapsulation: A comprehensive study of java module abuse. 2024.

[34] L. Hendren. Uses of the Soot Framework. `http://www.sable.mcgill.ca/~hendren/sootusers/`, 2018.

[35] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.

[36] S. Mak and P. Bakker. *Java 9 Modularity: Patterns and Practices for Developing Maintainable Applications.* " O'Reilly Media, Inc.", 2017.

[37] F. Massacci and I. Pashchenko. Technical leverage in a software ecosystem: Development opportunities and security risks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1386–1397. IEEE, 2021.

[38] M. R. H. Misu, R. Achar, and C. V. Lopes. Sourcererjbf: A java build framework for large-scale compilation. *ACM Transactions on Software Engineering and Methodology*, 2023.

[39] A. K. Mondal, C. K. Roy, K. A. Schneider, B. Roy, and S. S. Nath. Semantic slicing of architectural change commits: Towards semantic design review. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6, 2021.

[40] N. Parlog. Project Jigsaw is Really Coming in Java 9 — infoq.com. `https://www.infoq.com/articles/Project-Jigsaw-Coming-in-Java-9/`, (Accessed on 04-22-2024), 2015.

[41] N. Parlog. *The Java Module System.* Simon and Schuster, 2019.

[42] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT SEN*, 1992.

[43] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[44] R. Pressler and A. Buckley. JEP draft: Integrity and Strong Encapsulation — openjdk.org. `https://openjdk.org/jeps/8305968`, (Accessed on 04-22-2024), 2023.

[45] M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline*, volume 1. Prentice Hall Englewood Cliffs, 1996.

[46] R. Taylor, N. Medvidovic, and D. E.M. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, 2009.

[47] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.

[48] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.

[49] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. Liu. An empirical study of usages, updates and risks of third-party libraries in java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–45. IEEE, 2020.

[50] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S.-C. Cheung. Do the dependency conflicts in my project matter? In *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 319–330, 2018.

[51] Y. Wang, R. Wu, C. Wang, M. Wen, Y. Liu, S.-C. Cheung, H. Yu, C. Xu, and Z. Zhu. Will dependency conflicts affect my program's semantics? *IEEE Transactions on Software Engineering*, 48(7):2295–2316, 2021.

[52] Y. Wang, C. Xing, J. Sun, S. Zhang, S. Xuanyuan, and L. Zhang. Solving the dependency conflict of java components: A comparative empirical analysis. In *2020 IEEE 6th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing,(HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*, pages 109–114. IEEE, 2020.

[53] L. Zhao, S. Chen, Z. Xu, C. Liu, L. Zhang, J. Wu, J. Sun, and Y. Liu. Software composition analysis for vulnerability detection: An empirical study on java projects. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 960–972, 2023.

[54] M. Zheng, J. Yang, M. Wen, H. Zhu, Y. Liu, and H. Jin. Why do developers remove lambda expressions in java? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 67–78. IEEE, 2021.