DARCY: Automatic Architectural Inconsistency Resolution in Java

Negar Ghorbani, Tarandeep Singh, Joshua Garcia, *Member, IEEE*, and Sam Malek, *Member, IEEE*

***Abstract*—Many mainstream programming languages lack extensive support for architectural constructs, such as software components, which limits software developers in employing many benefits of architecture-based development. To address this issue, Java, one of the most popular and widely-used programming languages, has introduced the Java Platform Module System (JPMS) in its 9th and subsequent versions. JPMS provides the notion of architectural constructs, i.e., software components, as an encapsulation of modules that helps developers construct and maintain large applications efficiently—as well as improving the encapsulation, security, and maintainability of Java applications in general and the JDK itself. However, ensuring that module declarations reflect the actual usage of modules in an application remains a challenge that results in developers mistakenly introducing inconsistent module dependencies at both compile- and run-time. In this paper, we studied JPMS properties and architectural notions in-depth and defined a defect model consisting of eight inconsistent modular dependencies that may arise in Java applications. Based on this defect model, we also present DARCY, a framework that leverages the defect model and static analysis techniques to automatically detect and repair the specified inconsistent dependencies within Java applications at both compile- and run-time. The results of our experiments, conducted over 52 open-source Java 9+ applications, indicate that architectural inconsistencies are widespread and demonstrate DARCY's effectiveness for automated resolution of these inconsistencies.**

***Index Terms*—Software Architecture, Java Platform Module System, Architectural Inconsistency, Software Analysis, Software Repair.**

I. INTRODUCTION

EVERY software system has an architecture comprising the principal design decisions employed in the system's construction [1], which is not usually explicitly documented, e.g., in the form of UML models. Furthermore, the architecture of a system is often conceptualized in terms of high-level constructs, such as software components, connectors, and their interfaces. However, programming languages used to implement the software systems provide low-level constructs, such as classes, methods, and variables. Therefore, in many software systems, the architecture as intended or even documented, known as the *prescriptive* architecture, does not match the architecture reflected in the system's implementation, known as the *descriptive* architecture, and it is a non-trivial task to map one to the other. Hence, ensuring the conformance between the prescriptive and descriptive architecture has been a significant challenge in the software engineering literature [1].

Inconsistencies between prescriptive and descriptive architectures are of utmost concern in any software project since architecture is the primary determinant of a software system's key properties. One promising approach for abating the occurrence of architectural inconsistencies is to make it easier to bridge the gap between architectural abstractions and their implementation counterparts. To that end, the software engineering research community has previously advocated for *architecture-based development*, whereby a programming language (e.g., ArchJava [2]) or a framework (e.g., C2 [3]) provides the implementation constructs for realizing the architectural abstractions.

In spite of this prior work in the academic community, until fairly recently, Java—arguably the most popular programming languages over the past two decades—lacked extensive support for architecture-based development. This all changed with the introduction of Java Platform Module Systems (JPMS) in Java 9 and subsequent versions[1]. Modules are intended to make it easier for developers to construct large applications and improve the encapsulation, security, and maintainability of Java applications in general as well as the JDK itself [4].

Using Java's module system, the developer explicitly specifies the system's components (i.e., modules in Java) as well as the specific nature of their dependencies in a file called `module-info`. However, Java does not provide any mechanism to ensure the prescriptive architecture specified in the `module-info` file is in fact, consistent with the descriptive architecture of the implemented software, i.e., whether the declared dependencies in the `module-info` file are accurately reflecting the implemented dependencies among the system's components. Similarly, in case of any changes to the specified dependencies of modules at run-time, Java cannot ensure whether those changes are consistent with the applications' implemented dependencies.

Inconsistencies between the prescriptive and descriptive architectures in Java 9+ matter. The Java platform uses the `module-info` file to determine the level of access granted to each module and to determine which modules should be

Manuscript received 4 August 2023; revised 18 April 2024; accepted 22 April 2024. Date of publication 3 May 2024; date of current version 14 June 2024. This work was supported by the National Science Foundation award numbers 2211790 and 2106306. Recommended for acceptance by R. Kazman. *(Corresponding author: Negar Ghorbani.)*

Negar Ghorbani, Tarandeep Singh, Joshua Garcia, and Sam Malek are with the Department of Informatics, University of California Irvine, Irvine, CA 92697-3440, USA (e-mail: negargh@uci.edu; tarandes@uci.edu; joshug4@uci.edu; malek@uci.edu).

Digital Object Identifier 10.1109/TSE.2024.3396433

[1]We may refer to Java 9 and subsequent versions as Java 9+ in this paper.

packaged together for deployment. As a result, inconsistencies between prescriptive and descriptive architecture in Java have severe security and performance consequences. These inconsistencies also affect an engineer's ability to use the prescriptive architecture to understand a system's properties or to make maintenance decisions.

In this paper, we present DARCY, an automated framework that leverages a defect model consisting of 8 types of architectural inconsistencies we formally defined and static analyses to automatically resolve the identified inconsistencies within Java applications at both compile- and run-time. The results of our experiments, conducted over 52 open-source Java 9+ applications, indicate that architectural inconsistencies are widespread and demonstrate the benefits of DARCY in automated detection and repair of these inconsistencies. DARCY found 567 instances of inconsistencies in our dataset of Java applications. By automatically fixing these inconsistencies, DARCY was able to measurably improve various attributes of the subject applications' architectures at both compile- and run-time by reducing the attack surface of applications, improving their encapsulation, and producing deployable applications that consume less memory.

This paper presents several new non-trivial extensions to the preliminary version of our work published previously [5] as follows: (1) We have extended DARCY to detect and repair architectural inconsistencies not only statically at compile-time, but also dynamically at run-time. JPMS allows for the architecture of a software system to be changed dynamically at run-time. These dynamic changes may introduce inconsistencies that could not be detected using the techniques described in our prior publication. (2) Since at the time of our previous work [5] JPMS was a newly introduced concept, open-source repositories lacked an adequate number of appropriate Java 9 applications. Therefore, in this paper, we have expanded our evaluation dataset to include ten additional, larger and more mature Java 9+ applications to better evaluate DARCY's effectiveness. (3) We have designed and reported on new experiments to evaluate DARCY's ability in terms of resolving architectural inconsistencies at run-time and improving architectural metrics followed by the resolution.

To summarize, this paper makes the following contributions:

- Construction of a defect model representing formal definitions of 8 modular inconsistencies that may occur in Java 9+ applications at both compile- and run-time.
- An automated framework that leverages the defect model and static analysis techniques to automatically (1) detect the specified inconsistencies within Java applications, and (2) repair them at both compile- and run-time. DARCY is also publicly available [6].
- An extensive empirical evaluation on real-world Java 9+ applications demonstrating DARCY's effectiveness in resolving compile- and run-time architectural inconsistencies.

The remainder of this paper is organized as follows. Section II introduces the module system of Java and its design goals. Section III formally specifies a defect model consisting of the architectural inconsistencies in the context of Java 9+. Section IV provides the details of our approach and its implementation. Section V presents the experimental evaluation of the research. Section VI includes the threats to validity of our approach. The paper concludes with an outline of related research and future work.

II. JAVA PLATFORM MODULE SYSTEM

To aid the reader with understanding architectural specification in Java 9+, we introduce the new module system for Java 9+, called Java Platform Module System (JPMS). We overview JPMS's goals and the architectural risks that arise from its misuse. We then discuss the details of modules in Java 9+—including module declarations, module directives, and their behavior at run-time.

A. JPMS Goals and Potential Misuse

JPMS enables specification of a prescriptive architecture in terms of key architectural elements—specifically components in the form of Java modules, architectural interfaces, and resulting dependencies among components. JPMS aims to enable reliable configuration, stronger encapsulation, modularity of the Java Development Kit (JDK) and Java Runtime Environment (JRE) to solve the problems faced by engineers when developing and deploying Java applications [7].

Software designers and developers can achieve strong encapsulation in their Java 9+ systems by modularizing them and allowing explicit specification of interfaces and dependencies. Encapsulation in Java 9+ is achieved by allowing architects or developers to specify which of a Java module's public types are accessible or inaccessible to other modules [8]. A module must explicitly declare which of its public types are accessible to other modules. A module cannot access public types in another module unless those modules explicitly make their public types accessible. As a result, JPMS has added more refined accessibility control—allowing architects and developers to decrease accessibility to packages, reduce the points at which a Java application may be susceptible to security attacks, and design more elegant and logical architectures [9].

Prior to Java 9, the Java platform was a monolith consisting of a massive number of packages, making it challenging to develop, maintain, and evolve. Software developers could not easily choose a subset of the JDK as a platform for their applications. This results in software bloat and more potential points of attack for malicious agents. With the introduction of JPMS in Java 9, the Java platform was modularized into 95 modules. Furthermore, many internal APIs are hidden from apps using the platform [8], potentially reducing problems involving software bloat and security.

Using JPMS, Java developers can create lightweight custom JREs consisting of only modules they need for their application or the devices they are targeting. As a result, the Java platform can more easily scale down to small devices, which is important for microservices or IoT devices [10]. For example, if a device does not support GUIs, developers could use JPMS to create a runtime environment that does not include the GUI modules, significantly reducing the runtime memory size [9].

JPMS also allows developers to modify Java applications at run-time. More specifically, developers can dynamically load new modules into a running Java application or unload them [11].

Although JPMS allows for specification of prescriptive architectures, the descriptive architecture of a Java application may be inconsistent with the prescriptive architecture. Such inconsistencies may arise due to architects or developers misunderstanding of a software systems' architectures (e.g., an architect mistakenly specifies a more accessible interface than he intended), or simply due to mistaken implementations (e.g., a developer neglects to use a module's interface, even though the architect intended such a use). This can result in (1) a poorly encapsulated architecture, making an application harder to understand and maintain; (2) bloated software; or (3) insecure software. In terms of security, for instance, one of the potential problems is the granting of unnecessary access to internal classes and packages, potentially resulting in security vulnerabilities. In terms of software bloat, inconsistent dependencies can compromise scalability and performance of Java software (e.g., requiring many unnecessary modules from the JDK).

B. Understanding JPMS Modules

In JPMS, a *module* is a uniquely named, reusable group of related packages, as well as resources (such as images and XML files) [4]. Each module has a descriptor file, `module-info.java`, which contains meta-data, including the declaration of a named module. A named module should specify (1) its dependencies on other modules, i.e., the classes and interfaces that the module needs or expects, and should specify (2) which of its own packages, classes, and interfaces are exposed to other modules.

A module can be a *normal module* or an *open module*. A normal module allows access from other modules at compile-time and run-time to only explicitly exported packages; an open module allows access from other modules (1) at compile-time to only explicitly exported packages and (2) at run time to all its packages [12].

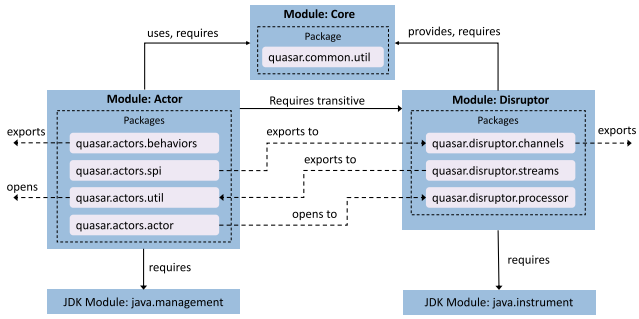
The module declaration file consists of a unique module name and a module body. Any module body can be empty or contain one or more module directives, which specifies a module's exposure to other modules or the modules it needs access to.

Fig. 1 shows an example of a Java application. This example is inspired by one of the subject applications studied in our work (see Section V), called Quasar. For the purpose of introducing JPMS module constructs and directives, we only present a subset of Quasar's architecture consisting of three modules, namely `core`, `actor`, and `disruptor`. The declarations of each module provided in its `module-info.java` file is described in Fig. 1(a). Fig. 1(b) is a diagram that depicts the relationship between the same modules based on dependencies in their declarations.

A module body can utilize combinations of the following five module directives [12], which specify module interfaces and their usage: the *requires* directive specifies the packages

```
1  module actor {
2      requires core;
3      requires transitive disruptor;
4      requires java.management;
5
6      exports quasar.actors.behaviors;
7      exports quasar.actors.spi to disruptor;
8
9      opens quasar.actors.util;
10     opens quasar.actors.actor to disruptor;
11
12     uses quasar.common.util.ProcessUtil;}
13
14 module disruptor {
15     requires core;
16     requires java.instrument;
17
18     exports quasar.disruptor.channels;
19     exports quasar.disruptor.streams to actor;
20
21     provides quasar.common.util.ProcessUtil with quasar
           .disruptor.processor.ImplProcess;}
22
23 module core {
24     exports quasar.common.util;}
```

(a) Module declarations and their directives provided in their *module-info.java* files.



(b) Specified dependencies between modules based on their directives

Fig. 1. Three example modules with their inter-dependencies.

that a module needs access to, the *exports* and *opens* directives make packages of a module available to other modules, the *provides* directive specifies the services a module provides, and the *uses* directive specifies the services a package consumes. These directives can be declared as described below:

- The `requires` directive with declaration `requires` m_2 of a module $m1$ specifies the name of a module m_2 that m_1 depends on. m_2 can be a user-defined module or a module within the JDK. For example, in Fig. 1, module `disruptor` requires module `java.instrument`. The requires declaration of a module m_1 may be followed by the `transitive` modifier, which ensures that any module m_3 that requires m_1 also implicitly requires module m_2. As an example, in Fig. 1, module `actor` requires module `disruptor` and any module that requires `actor` also implicitly requires `disruptor`.
- The `exports` directive with declaration `exports` p of a module m_1 specifies that m_1 exposes package p's public and protected types, and their nested public and protected types, to all other modules at both runtime and compile-time. For example, in Fig. 1, the module `disruptor` exports the package `quasar.disruptor.channels`. We can also export a package specifically to one or

more modules by using the `exports` p `to` $m_2, m_3, ..., m_n$ declaration. In this case, the public and protected types of the exported package are only accessible to the modules specified in the *to* clause. As an example, in Fig. 1, module `actor` exports `quasar.actors.spi` to the module `disruptor`.

- The `opens` directive with declaration `opens` p specifies that package p's nested public and protected types, and the public and protected members of those types, are accessible by other modules at runtime but not compile-time. This directive also grants reflective access to all types in p, including the private types, and all its members, from other modules. For example, in Fig. 1, module `actor` makes package `quasar.actors.util` available to other modules only at runtime, including through reflection.

  This directive may also be followed by the *to* modifier, resulting in the `opens` p `to` $m_2, m_3, ..., m_n$ declaration. In this case, the public and protected types of p are only accessible to the modules specified in the *to* clause. For instance, in Fig. 1, module `actor` makes package `quasar.actors.actor` available only at runtime, including through reflection, to the module `disruptor`. Unlike the other directives that can only be used in the body of a module's specification, `open` can be used in both the body of a module's specification and in its header (i.e., before the module's name). The latter usage is a shorthand way of denoting all packages in the module are `open`.
- The `provides with` directive with declaration `provides` c_1 `with` $c_2, c_3, ..., c_n$ of module m_1 specifies that a class c_1 is an abstract class or interface that is provided as a service by m_1. The *with* clause specifies one or more service provider classes for use with `java.util.ServiceLoader`. A service is a well-known set of interfaces and (usually abstract) classes. A service provider is a specific implementation of a service. The class `java.util.ServiceLoader<S>` is a simple service-provider loading facility. It loads a provider implementing the service type `S` [13]. For instance in Fig. 1, module `disruptor` provides the abstract class `quasar.common.util.ProcessUtil` as a service using the `quasar.disruptor.processor.ImplProcess` class as the service's implementation.
- The `uses` directive with declaration `uses` c_1 of a module m_1 specifies that m_1 uses a service object of an abstract class or interface, c_1, provided by another module. For this purpose, the module should discover providers of the specified service via `java.util.ServiceLoader`. As an example from Fig. 1, module `actor` uses the service object of class `quasar.common.util.ProcessUtil`, which is provided by module `disruptor`.

Note that, as depicted in Fig. 1, both `provides with` and `uses` directives need the module being declared to `require` the service module as well.

Depending on different use cases, Java modules can be dynamically loaded or unloaded at run-time [11], altering the system's prescriptive and descriptive architecture.

III. Defect Model: Inconsistent Module Dependencies

We found certain types of inconsistencies may arise between the specification of module dependencies in the `module.info` file (i.e., prescriptive architecture) and the actual dependencies among the implemented modules comprising the system (i.e., descriptive architecture). These inconsistencies occur in modules' specifications at both compile- and run-time. Insufficiently specified dependencies (e.g., a module that attempts to use a package it does not have a requires directive for) are already checked by the Java platform. However, *excess dependencies*, where a module either (1) exposes more of its internals than are used or (2) requires internals of other modules that it never uses, are not handled by the Java platform. These inconsistencies can affect various architectural attributes:

A1: Encapsulation and Maintenance—Requiring unneeded functionalities of other modules increases the complexity of the module unnecessarily, compromises its encapsulation, and decreases its maintainability.

A2: Software Bloat and Scalability—Requiring unneeded modules, especially from JDK, can result in bloated software, which compromises scalability of the application.

A3: Security—Excessively exposing the internals of a module can result in errors or security issues arising in the module.

To achieve a systematic and comprehensive coverage of all types of inconsistent module dependencies, we studied all potential inconsistencies resulting from developers' misuse of each type of module directive. In the remainder of this section, we focus on specifying eight types of inconsistent dependencies that may arise when using JPMS and the functions needed to specify those dependencies.

Table I includes 11 functions that directly model different variations of the five module directives in JPMS. To describe a class loading a service using `java.util.ServiceLoader` API, we define the *LoadsService* function. For actual code usage among packages, as opposed to those specified through module directives, we define the *Dep* function.

By leveraging the functions in Table I, we introduce eight types of excess inconsistent dependencies: requires, JDK requires, requires transitive, exports(to), provides with, uses, open, and opens(to) modifiers. For each inconsistent dependency type, there is a dependency explicitly defined in a `module-info` file that is not actually used in the source code of the module. Using these formal definitions, Section IV detects and repairs the following inconsistent dependencies.

Inconsistent Requires Dependency: This scenario describes an inconsistent $requires$ dependency in which (1) module m_1 explicitly declares that it requires another module m_2 and (2) no class of m_1 actually uses any class inside exported packages of m_2. As a result, this inconsistency mostly affects attribute A1. It can also affect attribute A2.

$$Req(m_1, m_2) \wedge (\nexists\ p_1 \in m_1,\ p_2 \in m_2 : Dep\ (p_1, p_2)) \quad (1)$$

Inconsistent JDK Requires Dependency: This scenario describes an inconsistent $requires$ dependency in which module m_1 explicitly declares that it requires a module inside the Java JDK, m_{jdk}. However, none of the classes inside m_1 uses

TABLE I
FUNCTIONS DESCRIBING DEPENDENCIES BASED ON MODULE DIRECTIVES OF JPMS

Function	Description
$Req(m_1, m_2)$	Module m_1 requires module m_2.
$ReqJDK(m_1, m_{jdk})$	Module m_1 requires the JDK module m_{jdk}.
$ReqTransitive(m_1, m_2)$	Module m_1 requires transitive module m_2.
$Exp(m, p)$	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 $\{m_2, m_3, \ldots\}$.
$Open(m)$	Module m is open.
$Opens(m, p)$	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 $\{m_2, m_3, \ldots\}$.
$Uses(m, s)$	Module m uses Service s.
$ProvidesWith(m, s, \{c_1, c_2, \ldots\})$	Module m provides service s with the set of classes $\{c_1, c_2, \ldots\}$.
$LoadsService(c, s)$	Class c loads Service s via the `java.util.ServiceLoader` API.
$Dep(p_1, p_2)$	Source code in package p_1 uses classes of package p_2.
$ReflDep(p_1, p_2)$	Source codes in package p_1 uses classes of package p_2 via reflection.

any class inside exported packages of m_{jdk}. Hence, it affects attribute A1, and more importantly A2. We distinguish this scenario from the previous one because an inconsistency involving JDK modules has a greater effect on portability than the previous more generic scenario.

$$Req(m_1, m_{jdk}) \wedge (\nexists p_1 \in m_1, p_2 \in m_{jdk} : Dep(p_1, p_2)) \quad (2)$$

Inconsistent Requires Transitive Dependency: An excess $transitive$ modifier in a $requires$ dependency consists of the following (1) a module m_1 explicitly declares in its `module-info` file that it transitively requires another module m_2—which means any module that requires m_1 also implicitly requires m_2; and (2) no class of a module that requires $m1$ actually uses any class in $m2$. This type of inconsistency mostly affects attribute A1, but also affects A2.

$$ReqTransitive(m_1, m_2) \wedge (\forall\, m : Req(m, m_1), \forall\, p \in m, \forall\, p_2 \in m_2 : \neg Dep(p, p_2)) \quad (3)$$

Inconsistent Exports/Exports to Dependency: An inconsistent *exports* dependency occurs when a module m_1 explicitly exports a package p_1 to all other modules, while no package in those other modules use p_1.

$$Exp(m_1, p_1) \wedge (\forall\, p \notin m_1 : \neg Dep(p, p_1)) \quad (4)$$

For an *exports to* directive, this inconsistency occurs when m_1 exports the package p_1 to a specific list of modules M, while no class outside m_1, or inside module list M, uses any class inside p_1.

$$ExpTo(m_1, p_1, M) \wedge (\forall\, p \in M : \neg Dep(p, p_1)) \quad (5)$$

These inconsistencies mostly affect attribute A3 by granting unnecessary access to classes and packages. They also affect attribute A1 due to complicating the architecture.

Inconsistent Provides With Dependency: An inconsistent *provides with* dependency has two key parts: (1) a module m explicitly declares that it provides a service s, which is an abstract class or interface that is extended or implemented by a set of classes $E = \{c_1, c_2, ..., c_k\}$ inside m; and (2) none of the classes inside other modules uses service s via the `java.util.ServiceLoader` API. Consequently, this inconsistency type—similar to inconsistent `requires` dependency—affects attribute A1 and A2 because the `provides with` dependency necessitates a `requires` directive as well. Additionally, this inconsistency type grants unnecessary access to a subset of the application's classes via the `ServiceLoader` API which affects attribute A3.

$$ProvidesWith(m, s, E) \wedge (\forall m' \neq m : \neg Uses(m', s)) \quad (6)$$

Inconsistent Uses Dependency: An inconsistent *uses* dependency occurs when (1) a module m explicitly declares in its `module-info.java` file that it uses a service s and (2) none of the classes inside m actually use the service s via the `java.util.ServiceLoader` API. This inconsistency type, similar to the previous type, will affect attribute A1 and A2, due to adding an additional `requires` directive.

$$Uses(m, s) \wedge (\forall\, c \in m : \neg\, LoadsService(c, s)) \quad (7)$$

Inconsistent Open Modifier: An excess *open* modifier occurs in the following scenario: (1) a module m declares that it opens all its packages to all other modules—recall from Section II-B that unlike the other directives, `open` can be used in the header of a module's specification to denote all its packages are open; and (2) there is at least one package p inside m that no class outside m reflectively accesses. As a result, any such package p is potentially open to misuse through reflection, e.g., external access to private members of a class that should not be allowed by any other class. This inconsistency type will affect attribute A3—and make the architecture inaccurate and more complicated, affecting attribute A1.

$$Open(m) \wedge (\exists p \in m : \forall p' \notin m : \neg ReflDep(p', p) \quad (8)$$

Inconsistent Opens/Opens To: An inconsistent *opens* dependency occurs when a module m declares that it opens a package p to all other modules via reflection, while none of the classes outside m reflectively accesses any classes of package p.

$$Opens\ (m, p) \wedge \forall\, p' \notin m : \neg ReflDep(p', p) \quad (9)$$

Similarly, for *opens to*, the *to* modifier specifies a list of modules M for which module m opens a package p to access via reflection, while no package of m reflectively accesses p.

$$OpensTo(m, p, M) \wedge \forall\, p' \in M : \neg ReflDep(p', p) \quad (10)$$

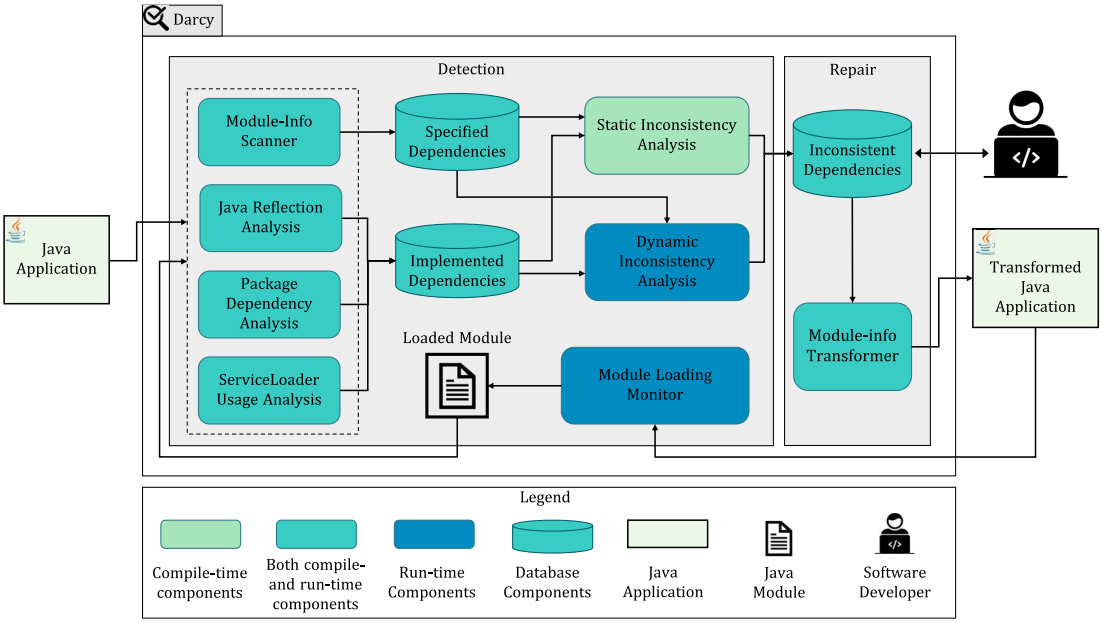


Fig. 2. A high-level overview of DARCY.

For these inconsistency types, private members of p are open to dangerous misuse through undesired access and reflection, affecting attribute A3, and can also affect attribute A1 due to unnecessarily complicating the architecture.

IV. DARCY

In the previous section, we introduced various types of inconsistent dependencies. This section describes how we leverage these definitions to design and implement DARCY. Fig. 2 depicts a high-level overview of DARCY comprised of two phases: *Detection* and *Repair*. Each DARCY's component works at either compile-time, run-time, or both, which is color-coded in Fig. 2. DARCY is implemented in Java and Python.

A. Detection

The detection phase takes a Java application as input and identifies any instance of the eight inconsistent dependencies described in Section III.

To identify the implemented dependencies of an input Java application, DARCY relies on static analysis, represented as *Package Dependency Analysis* in Fig. 2. In the implementation of DARCY, we leveraged Classycle [14] for *Package Dependency Analysis*. More precisely, the information about implemented dependencies in the source code of the input application is collected by running Classycle, which provides a complete report of all dependencies in the source code of a Java application at both the class and package levels. We only need the extracted dependencies among packages since the dependencies defined in modules are at the package level. *Package Dependency Analysis's* results are stored in *Implemented Dependencies*, which is a database component.

A Java application may contain multiple modules, each with a `module-info` file describing the module's dependencies. For extracting a prescriptive architecture, we developed *Module-Info Scanner* which examines all `module-info.java` files within the input Java application and extracts all specified dependencies which are defined at the package level. The collected information of specified dependencies is stored in another database component, *Specified Dependencies*.

Java Reflection Analysis leverages a custom static analysis [15], which we have implemented using the Soot framework [16], to identify the usage of reflection in the input application. The traces of any actual usage of reflection in the Java application is then stored in *Implemented Dependencies*.

Java Reflection Analysis extracts reflective invocations that occur in cases where non-constant strings, or inputs, are used as target methods of a reflective call. Reflective invocation of a method, for both constructor and non-constructor methods, occurs in three stages: (1) class procurement (i.e., a class with the method of interest is obtained), (2) method procurement (i.e., the method of interest to be invoked is identified), and (3) the method of interest is actually invoked. *Java Reflection Analysis* attempts to identify information at each stage.

A simple example, based on those found in real-world apps, of reflective method invocation, not involving constructors, is depicted in Fig. 3. In this example, a **`ClassLoader`** for **`MyClass`** is obtained (line 3), which is responsible for loading classes. The **`NetClass`** class is loaded using that **`ClassLoader`** (line 3). The **`getAddress`** method of **`NetClass`** (line 3)—which performs network operations—is retrieved and eventually invoked using reflection (line 3).

Our analysis identifies reflectively invoked methods using a backwards analysis. That analysis begins by identifying all reflective invocations (e.g., line 3 in Fig. 3). Next, the analysis follows the use-def chain of the invoked **`java.lang.reflect.Method`** instance (e.g., **`m`** on line 3)

```
1  ClassLoader cl = MyClass.getClassLoader();
2  try { Class c = cl.loadClass("NetClass");
3        ...
4        Method m = c.getMethod("getAddress",...);
5        ...
6        m.invoke(...); }
7  catch { ... }
```

Fig. 3. Reflective method invocation example.

```
1  ServiceLoader<MyService> loader;
2  loader = ServiceLoader.load(MyService.class);
3  for (MyService s : loader) {
4       if (s.getService != null){... } }
```

Fig. 4. Service loader example.

to identify all possible definitions of the **Method** instance (e.g., line 3). Our analysis considers various methods that return **Method** instances, i.e., using **getMethod** or **getDeclaredMethod** of **java.lang.Class**.

The analysis then records each identified method name. If the analysis cannot resolve the name, this information is also recorded. In this case, the analysis conservatively indicates that any method of the package opened for reflection can be accessed. Similarly, in situations where the class names cannot be determined, the analysis considers any classes of the opened package can be accessed.

For constant strings, the analysis attempts to identify the class name that is being invoked. Similar to the resolution of method names, the analysis follows the use-def chain of the **java.lang.Class** instance from which a **java.lang.Class** is retrieved (e.g., following the use-def chain of c on line 3). We model various means of obtaining a **java.lang.Class** instance. For example, the class may be loaded by name using a **ClassLoader's loadClass(...)** method (e.g., line 3), using **java.lang.Class's forName** method, or through a class constant (e.g., using **NetClass.class**). The analysis then records the class name it can find statically, or stores that it could not resolve that name. Note that our analysis considers any subclass of **ClassLoader**. Our reflection analysis involving constructors works in a similar manner by analyzing invocations of **java.lang.reflect.Constructor** and invocations of its **newInstance** method.

Similar to our analyses for reflectively invoked methods, We perform analyses for any **set*** methods of **java.lang.reflect.Field** (e.g., **setInt(...)**) or **get*Field*** methods of **java.lang.Class** (e.g., **getDeclaredField(String)**).

For extracting the implemented dependencies of type *uses* we implemented *ServiceLoader Usage Analysis* which leverages a custom static analysis using the Soot framework to identify usage of **java.util.ServiceLoader** in the input application. The traces of any actual usage of a service is then stored in *Implemented Dependencies*.

An application obtains a service loader for a given service by invoking the static **load** method of **ServiceLoader** API. A service loader can locate and instantiate providers of the given service using the **iterator** or **stream** method [17], through which an instance of each of the located service providers can be created. As an example, Fig. 4 depicts the code that obtains a **ServiceLoader** for **MyService** (line 4). The **ServiceLoader** loads providers of **MyService** (line 4) and can instantiate any of the located providers of this service using its iterator—created by the for loop in line 4. In this example, the service provider with the **getService** method is desired (line 4).

Our analysis identifies the usage of the **ServiceLoader** API using a backward analysis by following the use-def chain of **ServiceLoader** instances (e.g., s on line 4) to identify all possible definitions of a **ServiceLoader** (e.g, line 4 in Fig. 4). The results of the **ServiceLoader API** usage is then stored in *Implemented Dependencies*.

All the above-mentioned components are used at both compile-time, for the whole application, and run-time, for the dynamically loaded modules, to collect all specified and implemented dependencies of Java modules to further assist the detection of architectural inconsistencies.

1) Detection at Compile-Time: Static Inconsistency Analysis's main goal is to identify all types of inconsistency scenarios described in Section III at compile-time. For each directive in a `module-info.java` file, *Static Inconsistency Analysis* explores implemented and specified dependencies, stored in their respective database components, to identify any occurrence of an inconsistent dependency defined in Section III. If a matching instance is found, *Static Inconsistency Analysis* reports the identified architectural inconsistency, the modules affected, and the specific directive involved. The component then stores the identified inconsistencies in *Inconsistent Dependencies*, which are then used in the repair phase.

2) Detection at Run-Time: A Java application can dynamically load a module while running. Hence its architecture and module dependencies can change at run-time. Subsequently, the loaded module can contain architectural inconsistencies, which will be introduced to the Java application at run-time. The *Module Loading Monitor* and *Dynamic Inconsistency Analysis* components have been added to our previous work [5], extending DARCY to automatically detect architectural inconsistencies at run-time.

Module Loading Monitor monitors the running Java application for any dynamic changes to its dependencies, i.e., dynamically loading a new module. More specifically, the *Module Loading Monitor* has been implemented as a wrapper module that resides in between the application modules and the JDK modules, allowing it to monitor the loading and unloading of any new application module at run-time. For this purpose, the *Module Loading Monitor* needs to interact with the JDK and its specific classes in `java.lang.module` and record all module load/unload requests. Additionally, the *Module Loading Monitor* component needs to analyze and load any dependent third-party libraries that the loading module introduces. For this purpose, we implemented an algorithm that searches the introduced libraries in the module path and collects all required information along with the loaded module. This step

is crucial as it prevents false negatives in extracting the implemented dependencies later by their corresponding components. Accordingly, in case a new module is dynamically loaded in the system, as illustrated in Section II, *Module Loading Monitor* notifies and passes the loaded module's information to the following components: *Module-info Scanner*, *Java Reflection Analysis*, *Package Dependency Analysis*, and *ServiceLoader Usage Analysis*. Unlike the detection phase at compile-time, the components mentioned above only analyze the loaded module and collect the additional implemented and specified dependencies into their corresponding database components

Afterward, *Dynamic Inconsistency Analysis* (1) analyzes the change in the architecture using the previously-stored dependencies at compile-time, and (2) identifies all types of inconsistency scenarios described in Section III that arise by the loaded module at run-time. *Dynamic Inconsistency Analysis* analyzes the added dependencies by the loaded module at run-time, whereas *Static Inconsistency Analysis* analyzes the whole application at compile-time. Although identifying the inconsistencies at run-time follows the same algorithm as the one at compile-time, unlike the *Static Inconsistency Analysis* component, the *Dynamic Inconsistency Analysis* component needs to build a dynamic architecture graph of the system and update it with any changes at run-time in an efficient way, i.e., only analyzing the changes as opposed to analyzing the whole application. For this purpose, we implemented the *Dynamic Inconsistency Analysis* as an additional module loaded alongside other application modules with an algorithm that stores and updates the dynamic dependency graph of the application. Finally, the inconsistencies detected by *Dynamic Inconsistency Analysis* are then added to *Inconsistent Dependencies* to be used later in the repair phase.

B. Repair

To repair inconsistent dependencies, the repair phase transforms modules with detected inconsistent dependencies both statically at compile-time and dynamically at run-time. *Module-Info Transformer* is responsible for transforming module specifications to match with their implemented dependencies.

To repair inconsistent dependencies in a module, *Module-Info Transformer* deletes or modifies the explicit dependencies defined in the `module-info` files. Inconsistencies found in the previous phase are all unnecessarily defined dependencies among an application's modules and packages. Therefore, *Module-Info Transformer* needs to omit those inconsistent dependencies specified in the `module-info` files.

The detection phase results include the type and details of identified inconsistencies. For instance, in the case of an inconsistent *exports* dependency, one result stored in *Inconsistent Dependencies* includes the module in which this dependency is specified, the type of the inconsistent dependency (*exports* in this case), and the package that is unnecessarily exported. The repair phase takes the results of the detection phase as input. For each module, the repair phase finds the related records of inconsistent dependencies defined in that module and modifies the affected lines in `module-info`.

For this purpose, we leveraged ANTLR [18] to transform the `module-info.java` files to repair the inconsistent dependencies. ANTLR is a parser generator for reading, processing, executing, or translating structured text. Hence, we generated a customized parser using Java 9+ grammar and modified it to check the records of inconsistent dependencies found in the detection phase of DARCY.

More precisely, we have implemented the generated parser so that if it finds any match between the tokens of `module-info` files and the inconsistent dependencies, it skips or modifies the specific token with respect to the type of the inconsistency. As a result, depending on the type of dependency, the corresponding line in the `module-info` file is omitted or modified.

Module-Info Transformer repairs each type of inconsistent dependency at both compile- and run-time. In most cases, *Module-Info Transformer* deletes the entire statement. However, for `requires transitive`, *Module-Info Transformer* only removes the token `transitive`.

In case of inconsistencies involving `open` module m (Equation 8 in Section III), the `open` modifier is removed from the header of the module declaration. However, there may be some packages in m that other modules reflectively access. For each of these packages, *Module-Info Transformer* adds an `opens to` statement that make private members of the package accessible to the modules that reflectively access the package. If other modules reflectively access no package in m, no statement will be added to the module's body.

In certain situations, the DARCY user may disagree with how it repairs the specified dependencies because DARCY is not aware of the architect's or developer's intentions. For example, this situation may occur if the user wants to develop a library and export some packages for further needs or even allow other modules to reflectively access the internals of some classes and packages. DARCY warns the developers and architects about potential threats caused by architectural inconsistencies in their Java application and allows them to override DARCY prior to application of repairs.

After the repair phase at compile-time, the transformed Java application can be executed, but it is continuously monitored by *Module Loading Monitor*. In case of dynamically loading a module at run-time, the detection and repair phases are repeated for the loaded module.

V. EVALUATION

To assess the effectiveness of DARCY, we study the following research questions:

- **RQ1:** How pervasive are inconsistent, architectural dependencies in practice?
- **RQ2:** How accurate is DARCY in resolving inconsistent, architectural dependencies statically at compile-time?
- **RQ3:** How accurate is DARCY in resolving inconsistent, architectural dependencies dynamically at run-time?
- **RQ4:** To what extent does DARCY reduce the attack surface of Java modules?
- **RQ5:** To what extent does DARCY enhance encapsulation of Java modules?

TABLE II
SUBJECT APPLICATIONS

No.	Application Name	Rel. Version	# Modules	# Pckgs	# Directives
1	Auto-sort	1.0.0	3	35	13
2	Ballerina-lang	2.0.0	65	4486	532
3	Blynk-Server	0.28.0	9	742	122
4	BunnyHop	1.0.0	2	22	28
5	Codersonbeer-app	1.0.0	4	5	9
6	Constantin	0.1	4	15	9
7	Eclipse Jetty	10.0.5	56	10437	413
8	Java-9-lab	1.0.0	5	6	15
9	Java-9-modularity	1.0.0	4	5	11
10	Java-Bookstore	1.0.0	6	11	17
11	Java-SPI	1.0.0	6	6	26
12	Java9-demo	1.0.0	4	12	10
13	Java9-junit	1.0.0	3	15	13
14	Java9-labs	1.0.0	4	6	10
15	java9-modules	1.0.0	2	2	5
16	Java9-TLB-modules	1.0.0	5	8	12
17	JavaUtils	2.0.0	6	39	36
18	Jigsaw-resources	1.0.0	2	2	5
19	Jigsaw-tst	1.0.0	4	5	11
20	Jwtgen	0.0.1	2	9	13
21	Logback	1.3.0	2	1250	65
22	Meetup	1.0.0	4	4	14
23	Music-UI	1.0.0	3	7	15
24	Number-to-text	1.0.0	3	11	11
25	Practical-Security	1.0.0	4	8	20
26	Quasar	0.8.0	4	783	42
27	QuestDB	6.0.4	2	3222	65
28	Rahmnathan-utils	1.0	3	16	14
29	RecolInline-Server	0.3	7	73	42
30	Rhizomatic-IO	0.3.4	7	111	58
31	Sense-nine	1.0	6	48	31
32	Sirius	0.0.1	7	392	37
33	Spring-mvn-java9	1.0.0	3	6	18
34	Springuni-java9	1.0.0	3	2	6
35	Tascalate-Javaflow	2.7.0	10	258	56
36	The-Message	1.0.0	3	32	16
37	TRPZ	1.0.0	4	20	19
38	Vstreamer	1.0.0	6	6	25

RQ6: To what extent does DARCY reduce the size of runtime memory?

RQ7: What is DARCY's runtime efficiency in terms of execution time?

To answer these research questions, we selected a set of Java applications from GitHub [19], a large and widely used open-source repository of software projects, all of which are implemented in Java 9+. To accomplish this task, we developed a GitHub crawling Python script that systematically explores GitHub repositories, actively identifying Java applications containing the module-info.java file. It leverages the GitHub search API to query repositories based on specific parameters, including the filename and desired file size range, then parses the HTML response using the BeautifulSoup [20] library to extract relevant information such as repository name, URL, and index time from the search results. To handle rate limits imposed by the GitHub API, the script implements a mechanism to pause execution for a specified duration before retrying the request, and it employs techniques to manage large result sets by splitting queries into smaller subsets when the total number of results exceeds a certain threshold. Subsequently, we filtered GitHub repositories with more than one module that could have been successfully built. By scouring tens of thousands of GitHub repositories, this method offers an automated solution for detecting Java applications containing module-info.java files. The result is a conclusive dataset comprising 52 Java 9+ applications, effectively minimizing potential selection biases.

DARCY detected 567 instances of architectural inconsistencies in 38 out of 52 applications. Table II includes these 38 subject applications, their number of modules, packages, and directives. At the time of our previous paper [5], JPMS was a new concept, and Java developers did not have enough time to incorporate JPMS into open-source Java applications. However, in this paper, we evaluated DARCY with 10 additional larger and more mature Java 9+ applications. Each of the additional applications includes 17 modules, 2175 packages, and 99K lines of code, on average, which is significantly larger than those of our previous dataset's applications, i.e., 4 modules, 13 packages, and 9K lines of code.

A. RQ1: Pervasiveness

Table III shows, for each application in our dataset, the total number of inconsistent dependencies DARCY found and separates them by their type. 73% of applications in our dataset (38 out of 52) have a total number of 567 inconsistent dependencies. Recall that even one existing inconsistent dependency could cause undesired behaviors or issues with encapsulation, security, or memory utilization (see Section III).

As depicted in Table III, most of the inconsistent dependencies are of types *exports* or *requires* because these two types of directives are used more frequently than others. The high frequency of inconsistent *exports* dependencies indicates that granting unnecessary access to internal packages is quite common in Java 9+ applications, which could cause security vulnerabilities. Furthermore, different types of inconsistent *requires* dependencies can impair encapsulation and maintainability of applications. Additionally, the *requires JDK* inconsistency increases the risk of loading unnecessary JDK modules and compromising portability.

Table III indicates that a few applications have inconsistent dependencies of type *provides with*, and only two applications have an inconsistent *uses* dependency. In fact, these directives are rare compared to other directives. For *provides with* and *uses*, Java platform already checks most of the conditions that may lead to inconsistent dependencies at compile-time. Therefore, the possibility of encountering an inconsistent *provides with* and *uses* dependencies decreases. Nevertheless, DARCY covers the inconsistent dependencies corresponding to these two directives because they are risky and may appear more frequently in future usage of JPMS.

Finally, we reported the identified architectural inconsistencies and their suggested repairs to the developers of 21 subject applications (out of 38) for which we were able to obtain any contact information. As of now, we have received 7 responses, out of which 5 have confirmed Darcy's detected inconsistencies and agreed with Darcy's repairs. These applications are: Sense-nine, Java-9-lab, Meetup, Java9-labs, and Practical-Security. One of the developers (of Java-9-lab application) acknowledged

TABLE III
IDENTIFIED INCONSISTENCIES AND ROBUSTNESS AT COMPILE-TIME

Application Name	# Total Incons.	Inconsistencies Types							% Correct Incons.	Compiled (After Repair)	Test Passing Rate (%)	
		R	R.J.	R.T	E	P	U	O			Before	After
Auto-sort	1	-	-	-	1	-	-	-	100	✓	97	97
Ballerina-lang	115	21	4	-	88	1	-	1	100	✓	100	100
Blynk-Server	26	-	-	-	26	-	-	-	100	✓	100	100
BunnyHop	17	-	1	2	11	-	-	3	100	✓	-	-
Codersonbeer-app	4	1	-	-	2	-	-	1	100	✓	-	-
Constantin	5	-	-	1	4	-	-	-	100	✓	100	100
Eclipse Jetty	147	3	1	90	49	4	-	-	100	✓	100	100
Java-9-lab	1	-	-	-	1	-	-	-	100	✓	-	-
Java-9-modularity	1	-	-	1	-	-	-	-	100	✓	-	-
Java-Bookstore	3	-	-	2	-	1	-	-	100	✓	-	-
Java-SPI	5	-	-	1	4	-	-	-	100	✓	-	-
Java9-demo	1	-	-	-	1	-	-	-	100	✓	-	-
Java9-junit	1	-	-	-	1	-	-	-	100	✓	-	-
Java9-labs	4	-	-	-	4	-	-	-	100	✓	-	-
java9-modules	1	-	-	-	1	-	-	-	100	✓	-	-
Java9-TLB-modules	1	-	-	-	1	-	-	-	100	✓	-	-
JavaUtils	29		14	9	6	-	-	-	100	✓	-	-
Jigsaw-resources	1	-	-	-	1	-	-	-	100	✓	-	-
Jigsaw-tst	1	-	-	-	1	-	-	-	100	✓	-	-
Jwtgen	2	1	-	-	1	-	-	-	100	✓	-	-
Logback	30	-	-	3	26	1	-	-	100	✓	100	100
Meetup	6	-	-	-	3	3	-	-	100	✓	-	-
Music-UI	4	-	-	1	1	-	-	2	100	✓	-	-
Number-to-text	4	-	-	-	4	-	-	-	100	✓	100	100
Practical-Security	4	-	1	-	3	-	-	-	100	✓	-	-
Quasar	22	-	1	4	17	-	-	-	100	✓	100	100
QuestDB	34	-	-	1	33	-	-	-	100	✓	100	100
Rahmnathan-utils	7	-	1	-	6	-	-	-	100	✓	-	-
RecolInline-Server	20	4	6	2	6	-	-	2	100	✓	-	-
Rhizomatic-IO	3	-	-	-	2	-	1	-	100	✓	100	100
Sense-nine	10	2	2		3	-	-	3	100	✓	-	-
Sirius	9	2	-	4	3	-	-	-	100	✓	-	-
Spring-mvn-java9	8	2	-	-	6	-	-	-	100	✓	-	-
Springuni-java9	3	2	-	-	1	-	-	-	100	✓	-	-
Tascalate-Javaflow	17	-	-	12	5	-	-	-	100	✓	-	-
The-Message	9	-	-	-	9	-	-	-	100	✓	-	-
TRPZ	5	-	-	-	3	-	-	2	100	✓	-	-
Vstreamer	6	1	-	-	2	-	3	-	100	✓	-	-

(R: Requires, R.J: Requires JDK, R.T: Requires Transitive, E: Exports, P: Provides With, U: Uses, O: Opens)

that Darcy's suggested repairs would improve the maintainability of their application. In the two responses with respect to QuestDB and Eclipse Jetty applications, the developers disagreed with Darcy's identified inconsistencies and stated that the dependencies were intentional. More specifically, a developer of QuestDB explained that they have intentionally exported some packages for other extensions which extend key classes of QuestDB with additional functionalities. One of the developers stated that in their opinion, one of the most common reasons for architectural inconsistencies in Java is the lack of understanding in the definitions of dependencies.

Additionally, we have examined the subject applications' latest versions, which were released while this paper was under review, and found that 35 identified inconsistencies in 9 applications have been removed in their latest versions, further confirming Darcy's results. The removed inconsistency types and their corresponding applications are as follows:

- Three *requires* and twelve *exports* in Ballerina-lang
- Two *exports* and one *opens* in BunnyHop
- Four *exports* in Logback
- Three *provides* in Meetup
- Two *exports* in QuestDB
- Three *exports* in RecolInline-Server
- Two *requires JDK* in Sense-nine
- Two *requires* in Tascalate-Javaflow
- One *exports* in TRPZ

B. RQ2: Inconsistency Resolution at Compile-Time

To answer RQ2 for DARCY's detection capability at compile-time, we ran the detection phase for each Java application in our evaluation dataset before running them and assessed whether DARCY can accurately detect inconsistent dependencies. To that end, we manually checked the inconsistent dependencies found by DARCY to ensure their correctness. More precisely, we compared the corresponding record in both *Implemented Dependencies* and *Specified Dependencies* to verify the correctness of the inconsistencies discovered by the detection phase. As shown in Table III, all inconsistent dependencies found by DARCY at compile-time are correct.

To evaluate DARCY's ability to correctly resolve inconsistencies at compile-time, we ran the repair phase of DARCY for each

TABLE IV
IDENTIFIED INCONSISTENCIES AND ROBUSTNESS AT RUN-TIME

Application Name	The Loaded Module			# Runtime Incons.	Inconsistencies Types							% Correct Incons.	# Runtime Disruptions
	Name	# Packages	# Directives		R	R.J.	R.T	E	P	U	O		
Auto-sort	Unification Service	1	3	1	-	-	-	1	-	-	-	100%	0
Ballerina-lang	Shell	12	19	9	-	-	-	9	-	-	-	100%	0
Blynk-Server	Core	66	62	26	-	-	-	26	-	-	-	100%	0
BunnyHop	BunnyHop	20	23	17	-	1	2	11	3	-	-	100%	0
Codersonbeer-app	Data Model	1	3	2	-	-	-	1	1	-	-	100%	0
Constantin	Database	2	3	2	-	-	1	1	-	-	-	100%	0
Eclipse Jetty	Unixsocket Server	1	7	5	-	-	4	1	-	-	-	100%	0
Java-9-lab	Greeter	1	4	1	-	-	-	1	-	-	-	100%	0
Java-9-modularity	Analysis Service	1	3	1	-	-	1	-	-	-	-	100%	0
Java-Bookstore	Logging	1	2	1	-	-	-	-	-	1	-	100%	0
Java-SPI	API	1	3	2	-	-	1	1	-	-	-	100%	0
Java9-demo	Bank Impl	1	4	1	-	-	-	1	-	-	-	100%	0
Java9-junit	NFO	2	5	1	-	-	-	1	-	-	-	100%	0
Java9-labs	CMS	3	4	3	-	-	-	3	-	-	-	100%	0
java9-modules	Main	1	3	1	-	-	-	1	-	-	-	100%	0
Java9-TLB-modules	Poetry	1	4	1	-	-	-	1	-	-	-	100%	0
JavaUtils	JavaFX	2	7	6	-	-	4	2	-	-	-	100%	0
Jigsaw-resources	App	1	3	1	-	-	-	1	-	-	-	100%	0
Jigsaw-tst	Astro	1	2	1	-	-	-	1	-	-	-	100%	0
Jwtgen	Generator	6	11	1	-	-	-	1	-	-	-	100%	0
Logback	Core	38	33	4	-	-	3	1	-	-	-	100%	0
Meetup	Belarusian Lang	1	4	2	-	-	-	1	-	1	-	100%	0
Music-UI	UI	1	6	2	-	-	-	1	1	-	-	100%	0
Number-to-text	API	3	4	3	-	-	-	3	-	-	-	100%	0
Practical-Security	Banking Server	5	8	2	-	1	-	1	-	-	-	100%	0
Quasar	Core	21	33	16	-	-	-	16	-	-	-	100%	0
QuestDB	IO	67	57	34	-	-	1	33	-	-	-	100%	0
Rahmnathan-utils	Video Converter	4	6	3	-	-	-	3	-	-	-	100%	0
RecolInline-Server	SQL	3	13	7	3	-	1	3	-	-	-	100%	0
Rhizomatic-IO	IO Inject	4	9	2	-	-	-	1	-	-	1	100%	0
Sense-nine	Client	4	10	5	-	1	-	1	3	-	-	100%	0
Sirius	Frontend	1	3	2	-	-	1	1	-	-	-	100%	0
Spring-mvn-java9	Spring MVC	1	7	3	2	-	-	1	-	-	-	100%	0
Springuni-java9	Chat Model	1	2	1	-	-	-	1	-	-	-	100%	0
Tascalate-Javaflow	Agent Proxy	8	13	5	-	-	4	1	-	-	-	100%	0
The-Message	Base	10	11	1	-	-	-	1	-	-	-	100%	0
TRPZ	GUI	1	9	2	-	-	-	1	1	-	-	100%	0
Vstreamer	Player	1	4	3	1	-	-	1	-	-	1	100%	0

(R: Requires, R.J: Requires JDK, R.T: Requires Transitive, E: Exports, P: Provides With, U: Uses, O: Opens)

Java application in our evaluation dataset before running them and assessed whether DARCY repairs the detected inconsistencies without introducing any unexpected behavior. To assess the correctness of a repair, we (1) check if each application compiles successfully after running the repair phase, and (2) if the application contains a test suite, determine if the application obtains the same *test passing rate*, i.e., the ratio of the number of passing test cases to the total number of test cases, both before and after repairs. We also ran the detection phase after the repair actions. The result showed zero inconsistencies within the transformed Java applications.

The results for compilation after the repair phase are shown in Table III, indicating that all transformed Java applications compiled successfully. This confirms that the inconsistent dependencies have been robustly resolved in a way that does not prevent compilation of the applications. Additionally, eleven applications in our study contain a test suite. The passing rate for each of these test suites remains the same both before and after DARCY repairs, demonstrating that DARCY does not negatively affect the expected behavior of repaired applications.

C. RQ3: Inconsistency Resolution at Run-Time

Java applications can dynamically load new modules while running. Hence, their architecture changes and the new modules can introduce further architectural inconsistencies at run-time. To answer RQ3, we simulated the above-mentioned run-time behavior and assessed DARCY's capabilities of dynamically detecting and repairing architectural inconsistencies that may arise at run-time.

To simulate this scenario, we randomly selected a module among those containing one or more architectural inconsistencies in each dataset application and ran the application without that single module. While DARCY is monitoring the application during its run-time, we added the previously selected module as a new module loaded into the system. We then evaluated whether DARCY can accurately detect inconsistent dependencies of the recently added module at run-time. For this purpose, we manually checked whether the inconsistent dependencies found by DARCY match the actual inconsistencies existing in the added module. Table IV describes the results of this experiment, including the number of packages and directives in the

added module, as well as the detected run-time inconsistencies separated by their types. As shown in Table IV, all inconsistent dependencies found by DARCY at run-time are correct.

To evaluate DARCY's ability to correctly repair this type of run-time inconsistencies, we ran the repair phase of DARCY for each of the loaded modules in our experiment. We manually examined each repair, and all were correct. We further evaluated whether the transformed module can run sucessfully without any unexpected run-time behavior caused by DARCY's repairs. As shown in Table IV, DARCY was able to repair the detected run-time inconsistencies without introducing any disruptions to the applications' run-time, i.e., run-time exceptions, as it removes the excessively specified dependencies in module description files. Similar to RQ2, we also ran the detection phase after the repair actions, and the results showed zero inconsistencies within the transformed Java applications.

D. RQ4: Security

To assess DARCY's ability to enhance security, we consider the *attack surface* of Java 9+ applications. A system's attack surface is the collection of points at which the system's resources are externally visible or accessible to users or external agents. Manadhata et al. introduced an attack-surface metric to systematically measure the security of a system [21], [22], [23]. Every externally accessible system resource can potentially be part of an attack and, hence, contributes to a system's attack surface. This contribution reflects the likelihood of each resource being used in security attacks. Intuitively, the more actions available to a user or the more resources accessible through these actions, the more exposed an application is to security attacks [21], [22], [23].

For a Java 9+ application, the main resource under consideration is Java modules and their packages. As a result, we define an application's attack surface as the number of packages accessible from outside its modules. To measure the attack surface of Java 9+ applications, we count the number of packages exposed by *exports (to)* and *open(s to)* directives. These directives make the internals of packages accessible to other modules.

As shown in Table V, 36 out of 38 applications had an average attack-surface reduction of about 56% at their compile-time. DARCY was able to totally eliminate the attack surface in 5 applications.[2] Although eliminating the module-based attack surface does not result in perfect security, DARCY can maximize the protection of the asset (i.e., Java packages) through a module's interfaces by eliminating all unnecessary exports and opens directives of the module—other attack vectors (e.g., IPC over network sockets) still remain but are out of scope for DARCY.

On the other hand, to evaluate DARCY's ability to enhance the applications' security at run-time, we measured the attack surface reduction ratio followed by repairing the inconsistencies of the dynamically loaded modules at run-time. To simulate such a scenario for an individual module in an application, for each Java application in our dataset, we selected a module and ran DARCY on the rest of the application's modules. Then, we ran the repaired application without that specific module. While running, we dynamically loaded the chosen module and measured the attack surface ratio reduction in the whole application as the result of DARCY's repairing the loaded module at run-time. This ratio indicates to what extent DARCY is able to further reduce the attack surface of a transformed application by resolving inconsistent dependencies of a dynamically loaded module. We repeated this experiment for all modules in each application and reported the average attack surface reduction ratio per module. As shown in Table V, DARCY was able to reduce the attack surface of the loaded modules in 36 out of 38 applications by an average of about 17% at run-time. DARCY entirely eliminated the attack surface of the loaded module in 28 applications. Note that the average attack surface reduction ratios are usually less than those of compile-time as the run-time measurement includes the effects of repairing only one module on the whole application attack surface, whereas at compile-time we measured the effects of repairing all modules.

TABLE V
RESULT FOR ATTACK-SURFACE REDUCTION AT COMPILE AND RUN-TIME

Application Name	App's Attack Surface Reduction (%)	
	Compile-time	Run-time (Avg. Per Module)
Auto-sort	50.00%	7.50%
Ballerina-lang	45.64%	3.09%
Blynk-Server	44.83%	4.02%
BunnyHop	87.50%	7.19%
Codersonbeer-app	75.00%	41.67%
Constantin	100.00%	100.00%
Eclipse Jetty	41.88%	5.14%
Java-9-lab	33.33%	3.13%
Java-9-modularity	-	-
Java-Bookstore	-	-
Java-SPI	50.00%	33.33%
Java9-demo	50.00%	11.43%
Java9-junit	25.00%	16.67%
Java9-labs	80.00%	31.25%
java9-modules	50.00%	3.37%
Java9-TLB-modules	20.00%	2.92%
JavaUtils	85.71%	19.28%
Jigsaw-resources	50.00%	5.00%
Jigsaw-tst	33.33%	6.00%
Jwtgen	100.00%	3.57%
Logback	52.00%	4.82%
Meetup	75.00%	9.38%
Music-UI	60.00%	1.90%
Number-to-text	25.00%	22.22%
Practical-Security	25.00%	18.75%
Quasar	77.27%	24.31%
QuestDB	63.46%	9.75%
Rahmnathan-utils	100.00%	42.86%
RecolInline-Server	47.06%	1.01%
Rhizomatic-IO	15.38%	0.61%
Sense-nine	83.33%	50.00%
Sirius	21.43%	0.70%
Spring-mvn-java9	100.00%	7.14%
Springuni-java9	100.00%	33.33%
Tascalate-Javaflow	45.45%	1.98%
The-Message	10.00%	15.79%
TRPZ	40.00%	50.00%
Vstreamer	66.67%	10.71%
Avg. Attack Surface Reduction	**56.37%**	**16.94%**

[2]These applications are essentially software utilities or libraries including different modules that provide functionalities for different situations, but do not have any dependency on one another.

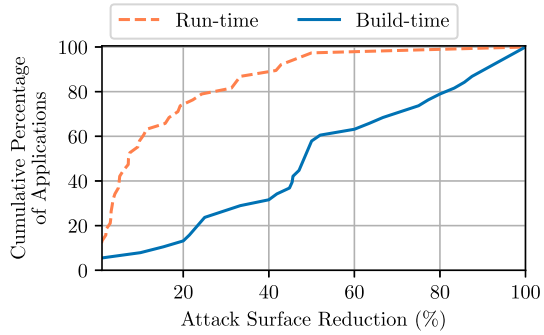


Fig. 5. Cumulative distribution function (CDF) of the attack surface reduction at compile- and run-time.

Fig. 5 shows a cumulative distribution function (CDF) of the ratio of attack surface reduction by DARCY among the applications in our dataset for both compile- and run-time. It indicates that in 60% of the applications, DARCY was able to reduce the applications' attack surface up to about 50% at compile-time. Furthermore, in 80% of the applications, the repairs by DARCY reduced the applications' attack surface up to 80%, which means that for the remaining 20% of the applications, it reduced the applications' attack surface by even more than 80% at their compile-time. At run-time, in about 90% of the applications, DARCY was able to reduce the attack surface up to 40% on average per loaded module. The relatively large reduction of the attack surface in applications achieved by DARCY indicates that it can significantly curtail security risks in Java 9+ applications.

E. RQ5: Encapsulation

To evaluate the ability of DARCY to enhance the encapsulation of Java 9+ applications, we leveraged two metrics selected from an extensive investigation by Bouwers et al. [24] about the quantification of encapsulation for implemented software architectures. We selected metrics that involve architectural dependencies and are appropriate for the context of modules in JPMS and Java 9+ applications.

The first metric we selected is the Ratio of Coupling (RoC) [25], which measures coupling among an application's modules. For Java 9+ modules, RoC is the ratio of the *number of existing dependencies among modules* to the *number of all possible dependencies among modules*. Ideally, the value of RoC would be low, meaning that only a small part of all possible dependencies among modules is actually utilized—making it less likely that faults, failures, or errors introduced by changes or additions to modules will propagate across modules.

The second metric we selected is a variant of Cumulative Component Dependency (CCD) [26] which is the sum of all outgoing dependencies for a component. For Java 9+ modules, outgoing dependencies are *requires* and *uses* dependencies of each module. The specific variant we used is Normalized CCD (NCD), which is the ratio of CCD for each module to the total number of modules. Ideally, the value of CCD, or NCD, is low, indicating lower coupling and better encapsulation.

TABLE VI
RESULTS FOR ENCAPSULATION IMPROVEMENT

Application Name	App's RoC Change (%)		App's NCD Change (%)	
	Compile-time	Run-time	Compile-time	Run-time
Auto-sort	7.69%	2.00%	-	-
Ballerina-lang	21.43%	1.41%	7.46%	0.53%
Blynk-Server	21.31%	2.74%	-	-
BunnyHop	60.71%	5.21%	25.00%	2.78%
Codersonbeer-app	30.77%	12.05%	12.50%	3.13%
Constantin	55.56%	23.33%	20.00%	5.00%
Eclipse Jetty	35.59%	4.46%	34.18%	4.26%
Java-9-lab	6.67%	1.25%	-	-
Java-9-modularity	9.09%	2.56%	12.50%	3.02%
Java-Bookstore	17.65%	5.07%	16.67%	4.42%
Java-SPI	15.38%	5.58%	5.56%	1.39%
Java9-demo	10.00%	4.40%	-	-
Java9-junit	7.69%	5.13%	-	-
Java9-labs	40.00%	11.90%	-	-
java9-modules	20.00%	2.45%	-	-
Java9-TLB-modules	8.33%	1.64%	-	-
JavaUtils	80.56%	20.86%	79.31%	20.37%
Jigsaw-resources	20.00%	3.13%	-	-
Jigsaw-tst	9.09%	2.31%	-	-
Jwtgen	15.38%	3.01%	8.33%	1.50%
Logback	46.15%	5.48%	21.43%	2.75%
Meetup	42.86%	6.82%	-	-
Music-UI	26.67%	2.68%	10.00%	1.27%
Number-to-text	9.09%	7.60%	-	-
Practical-Security	10.00%	5.00%	7.69%	1.92%
Quasar	52.38%	16.16%	25.00%	6.73%
QuestDB	52.31%	8.55%	8.33%	2.14%
Rahmnathan-utils	50.00%	16.98%	12.50%	3.17%
RecolInline-Server	47.62%	1.07%	48.00%	0.77%
Rhizomatic-IO	5.17%	0.27%	2.44%	0.13%
Sense-nine	29.03%	6.93%	16.00%	2.96%
Sirius	24.32%	1.03%	26.09%	0.90%
Spring-mvn-java9	44.44%	5.29%	16.67%	1.64%
Springuni-java9	50.00%	25.00%	40.00%	16.67%
Tascalate-Javaflow	30.36%	1.66%	26.67%	1.41%
The-Message	6.25%	12.00%	-	-
TRPZ	10.53%	7.54%	-	-
Vstreamer	16.00%	3.66%	20.00%	2.26%

RoC	Total # of Affected Systems	38
	Compile-time Reduction Avg.	**27.53%**
	Run-time Reduction Avg.	**6.69%**
NCD	Total # of Affected Systems	24
	Compile-time Reduction Avg.	**20.93%**
	Run-time Reduction Avg.	**3.80%**

Table VI presents the amount of RoC and NCD change in 38 Java 9+ applications with inconsistent dependencies at both compile- and run-time. Across all 38 applications, the amount of RoC is reduced by an average of about 28%, and up to about 81% at compile-time. The amount of NCD is also reduced in 24 applications by an average of about 21%, and up to 79% at compile-time.

To measure the encapsulation improvement at run-time, we ran the same experiment as RQ4. More specifically, we selected a module in each application and ran the transformed application without the chosen module and later at run-time we dynamically loaded the selected module. We then measured the amount of RoC and NCD change in the whole application followed by only repairing the loaded module. We repeated the same experiment for all modules and reported the average RoC and NCD change per module repair for each application. As shown in Table VI, DARCY was able to reduce the amount

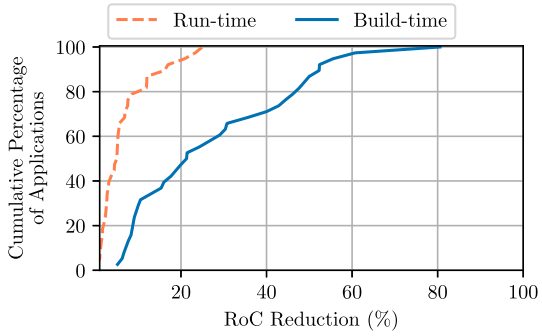


Fig. 6. Cumulative distribution function (CDF) of the Ratio of Coupling (RoC) reduction at compile- and run-time.

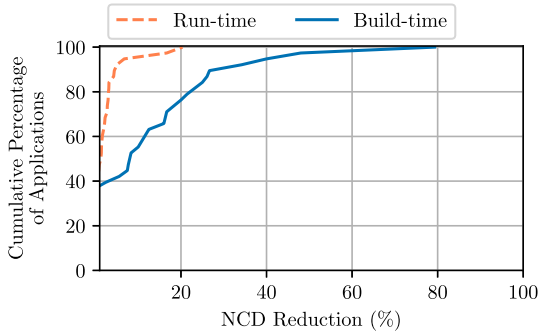


Fig. 7. Cumulative distribution function (CDF) of the Normalized Cumulative Component Dependency (NCD) reduction at compile- and run-time.

TABLE VII
RESULTS FOR SOFTWARE-BLOAT REDUCTION

Application Name	Runtime Memory Reduction (%) Compile-time	Run-time
Ballerina-lang	8.29%	0.09%
BunnyHop	54.72%	20.55%
Eclipse Jetty	16.83%	0.59%
Java-SPI	6.04%	1.51%
JavaUtils	21.87%	19.82%
Practical-Security	0.76%	0.19%
Quasar	4.55%	1.14%
Rahmnathan-utils	0.12%	0.04%
RecolInline-Server	50.70%	7.27%
Sense-nine	0.63%	0.22%
Avg. Memory Reduction	16.45%	5.14%

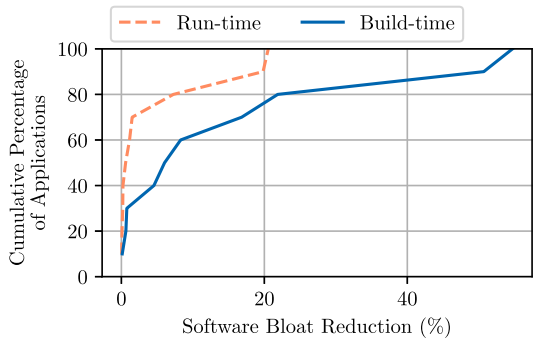


Fig. 8. Cumulative distribution function (CDF) of software bloat reduction at compile- and run-time.

of RoC by an average of about 7% and up to about 25% per module in the whole application. The repairs by DARCY also reduced the amount of NCD in 24 applications at run-time by an average of about 4% and up to 21%. Note that the RoC and NCD change at run-time ratios are less than those of compile-time, since at run-time we only measure the effect of repairing a single module on the whole application. These results indicate that DARCY can successfully enhance the encapsulation of Java 9+ applications by a significant amount at both compile- and run-time.

Fig. 6 shows the cumulative distribution function (CDF) of the RoC reduction followed by DARCY's repairs at both compile- and run-time. The figure indicates that, at compile-time, in about 70% of applications, DARCY was able to reduce the RoC up to 40%. The figure also shows that, at run-time, in about 80% of applications, DARCY was able to reduce the RoC of the entire application by about 10% by resolving the inconsistencies of a single loaded module.

Fig. 7 demonstrates the cumulative distribution function (CDF) of DARCY's NCD reduction rate at both compile- and run-time. At compile-time, the figure shows that resolving the inconsistencies in 80% of the applications could reduce the NCD up to 22%. Furthermore, the figure shows that resolving the inconsistencies of a module loaded at run-time can reduce the amount of NCD in 90% of the applications up to about 10%.

F. RQ6: Software Bloat

To answer this research question, we measured the memory usage of each application before and after DARCY's repair phase at both compile- and run-time. Recall that in Java 9+, with the JDK being modularized, we can create a lightweight custom Java Runtime Environment (JRE), reducing software bloat. More specifically, the size of a custom JRE may be reduced after a repair if the application has inconsistent dependencies of type *requires JDK* (Equation 2 of Section III).

Table VII shows the reduction of software bloat in terms of the reduction in the size of the JRE image of affected applications after removing inconsistent *requires JDK* dependencies at both compile- and run-time repair. According to the results for compile-time, the reduction is about 16% in 10 applications and up to 55% once the repair is executed. For run-time, the table also shows that DARCY reduces the application's memory consumption by an average of 5% up to about 40% after repairing a loaded module.

Fig. 8 demonstrates the cumulative distribution function (CDF) of DARCY's software bloat reduction rate at both compile- and run-time. At compile-time, the figure shows that in 80% of the applications DARCY could reduce the size of the JRE image up to about 22% by resolving the module inconsistencies. Furthermore, the figure shows that resolving the inconsistencies of a module loaded at run-time can reduce the size of JRE image in 80% of the applications up to about

TABLE VIII
RESULTS FOR EXECUTION TIME

Component	Avg. Execution Time (ms)	
	Compile-time	Run-time
Class Dependency Analyzer (Classycle)	7428	-
Java Reflection Analysis	328	131
ServiceLoader Usage Analysis	315	101
Java Inconsistency Analysis	250	27
Repair	453	163
Total	8774	422

7%. Such results are particularly substantial for deployment and scalability goals in microservices or IoT devices that are memory constrained.

G. RQ7: Performance

As described in Section IV, DARCY builds on three tools, Classycle [14], Soot [16], and ANTLR [18]. As a result, to assess DARCY's performance we answer RQ7 in terms of these underlying tools as well as DARCY's execution time. We ran all the evaluation experiments on a MacBook Pro 2013 (2.3 GHz Intel Core i7, 16 GB, MacOS 10.14.).

Table VIII shows the average execution times of DARCY at both compile- and run-time. Results for Classycle are shown separately from the results for other components, since Classycle dominates the execution time. Note that Classycle is only executed once before an application is executed, and DARCY stores the collected package dependencies for further analysis. Therefore, it only appears in compile-time execution time in Table VIII. On average, DARCY takes under 9 seconds to execute at compile-time, which is reasonably efficient for both detection and repair. At run-time, DARCY takes only about 0.4 seconds on average to detect and repair the inconsistencies of a dynamically loaded module, which poses a minimal overhead to the system's execution.

VI. THREATS TO VALIDITY

In terms of accuracy, the main threat to internal validity is the risk of false positives or negatives of the static analysis tools used in the implementation. False positives or negatives in the results of the static analysis tools may cause DARCY to miss some inconsistencies in the detection phase or report false inconsistencies, which may lead to compilation errors or harming functionality of the application after the repair phase. Since DARCY takes Classycle's results as an input for the Java inconsistency analysis, it inherits all of Classycle's limitations. The accuracy of detecting the inconsistent dependencies is affected by the accuracy of the static analysis tool we use. However, Classycle has been used and in development for over 11 years and leveraged by other state-of-the-art tools for software architecture and anti-pattern analysis [27], [28], [29], [30], [31], [32], [33]. A similar threat to internal validity exists for our use of Soot; however, Soot is a widely used [34], [35] and actively maintained framework [36] for static analysis of Java programs. We further manually determine whether every identified inconsistency is correct to ensure that any unforeseen issues with underlying static analyses do not compromise DARCY's accuracy.

The main threat to external validity is that DARCY may identify an architectural inconsistency that is intentionally defined by developers. For example, in the case of Java libraries, a developer may export some packages that might not be used internally by the library itself. In these situations, DARCY cannot be aware of the developer's intention. Consequently, the evaluation results in this paper report the potential architectural inconsistencies and the gain of repairing them. To mitigate this threat, we keep the developers in the loop and let them override DARCY's decision before the repair phase, as explained in Section IV.

Another threat to external validity is the selection and number of Java applications in the evaluation dataset. To mitigate this threat, we selected open source Java 9+ applications from many developers and thousands of repositories in Github, one of the largest and most widely used open-source repositories online. Another threat to external validity is whether the types of inconsistencies we identify comprehensively cover those that may exist. To alleviate this threat, we considered the architectural inconsistencies based on all types of module directives defined in Java.

DARCY's evaluation on only one programming language, i.e., Java, is another threat to external validity. This threat is alleviated by the fact that Java is one of the most widely used languages in the world [37], [38]. Furthermore, the general idea behind DARCY can be extended to any other languages with modular programming constructs that utilize provides and requires interfaces advocated by software architecture-based development and design [39], [40], [41].

VII. RELATED WORK

The most closely related literature to DARCY bridges the gap between the prescriptive and descriptive architecture. There are a variety of different types of strategies to address this issue: focusing only on the descriptive architecture by reverse engineering it; obtaining the descriptive architecture and the prescriptive architecture, followed by checking their conformance; ensuring that early in the software lifecycle that the descriptive and prescriptive architectures conform by providing architectural constructs in code; and approaches that ensure conformance of the descriptive and prescriptive architecture from the beginning and into maintenance.

Many approaches address the architecture-implementation mapping issue by ignoring the prescriptive architecture and simply trying to obtain the most accurate descriptive architectures possible [31], [32], [42], [43], [44], [45], [46], [47], [48], [49]. A large number of these approaches rely on software clustering to determine components from implementations [42], [43], [50], [51], [52]. More recently, Hammad et al. [53] have developed a tool-assisted approach that relies on component recovery techniques and converts object-oriented Java apps, i.e., pre-Java-9, to component-based Java 9+ apps that properly use the JPMS constructs with the least privileged architecture specification. Furthermore, Shi et al. [54] studied C++20

modules and proposed H2M, an approach to convert header-based C++ applications to module-based C++ applications with better compiling performance. While these approaches mostly focus on recovering the descriptive architecture, DARCY tries to find inconsistencies between the prescriptive and descriptive architectures.

A series of approaches detect inconsistencies between architecture and implementation by reverse engineering the descriptive architecture from the code and comparing it with the prescriptive architecture [55], [56], [57], [58], [59], [60], [61], [62], [63], [64], [65], [66], [67], [68]. Murphy et al. introduced the software reflexion method which helps an engineer compare prescriptive and descriptive architectures in a manual manner [55]. A number of these approaches extend the reflexion method with automated architecture recovery techniques [65], [66], [67]. Moreover, Buckley et al. [69] proposed JITTAC, a tool that uses a real-time Reflexion Modeling approach to provide developers with Just-In-Time architectural consequences of their implementation actions and promote consistency between the prescriptive and descriptive architecture. However, the main difference between these approaches and DARCY is that they require the software architectures to manually specify the prescriptive architecture, whereas in DARCY the prescriptive architecture is automatically obtained by leveraging the module declarations in JPMS.

Other approaches provide implementation-level constructs that represent architectural elements (e.g., customizable programming-language classes representing components) that help ensure architectural conformance from a forward-engineering perspective [2], [70], [71], [72], [73], [74], [75], [76]. Many of these approaches support various notions of software architectural connectors or interfaces, rather than just components, but, unlike DARCY, they do not address a mainstream programming language widely used by many developers such as Java.

Certain approaches achieve architecture-implementation mapping from both a forward-engineering (e.g., code generation) and reverse-engineering perspective, i.e., round-trip engineering [77], [78], [79]. 1.x-way mapping [77] allows manual changes to be initiated in the architecture and a separated portion of the code, with architecture-prescribed code updated solely through code generation. 1.x-line mapping [78] extends 1.x-way mapping to product-line development. Song et al. [79] introduce a runtime approach for architecture-implementation mapping from a roundtrip-engineering perspective.

Another series of approaches focus on detecting or fixing different types of dependency conflicts in software systems, e.g., detecting security issues [80], or conflicts in referencing third-party libraries [81], [82], [83], [84], [85], [86], [87]. Dann et al. introduced ModGuard [80], a static analysis based approach that automatically identifies instances involving unintentional leaks of sensitive objects in real-world applications which may have a significant impact on an application's security. ModGuard and Darcy are similar in improving the security of modular Java applications, however, they focus on two different perspectives. ModGuard focuses on leaking packages' internal data without exposing them, while Darcy detects unnecessarily exposed packages which may lead to security issues.

On the other hand, Wang et al. focused on the semantic conflicts caused by the dependency conflicts and proposed an automated testing technique that attempts to identify inconsistent behaviors of the APIs in conflicting library versions [82] as well as creating the corresponding stack trace [84]. They also proposed techniques to monitor dependency conflicts for the Python library ecosystem [85] and detect dependency conflicts in Golang, as well as suggesting proper fixes [86]. Furthermore, Lambers et al. [83] developed a technique that detects all dependency conflicts on multiple granularity levels based on graph transformation. Additionally, Li et al. [87] proposed an automated technique to repair dependency issues where changes in software systems violate package dependency constraints.

Regarding run-time architectural conformance, in the most closely related work, Yan et al. [61] introduce DiscoTect, a technique that observes running software systems and constructs an architectural view of them. Using this dynamic architectural discovery, developers and architects can map the architectural styles to implementation styles which helps them with building systems that are consistent with their architectural design. However, their approach does not provide an automated technique to enforce the conformance between prescriptive and descriptive architecture as DARCY does.

Despite other techniques, DARCY is the first approach that supports architectural-implementation conformance, at both compile- and run-time, in a mainstream programming language using architectural constructs built directly into the programming language by its creators. Furthermore, our approach includes repairing non-conforming architectures rather than just determining inconsistencies. In addition, DARCY is the only approach for architecture-implementation mapping that focuses on software bloat and attack-surface reduction.

The only existing framework similar to JPMS is OSGI [88]. The major differences between OSGI and JPMS are as follows. OSGI was not able to modularize the JDK, preventing the construction of customized runtime images with a minimized JDK, which JPMS enables. Additionally, OSGI cannot handle reflective access to modules' internal packages. Similar dependency-analysis facilities for OSGI are limited to removing unused dependencies of type import, which represents the require dependency, and cannot cover the other 7 types of inconsistencies in JPMS applications previously introduced in Section III. Therefore, there is no similar facility for OSGI that repairs all types of inconsistent dependencies as DARCY does.

VIII. CONCLUSION

This paper formally defines 8 types of architectural inconsistencies in Java 9+ applications and introduces DARCY, an approach for automatic detection and repair of these types of inconsistencies both statically at compile-time and dynamically at run-time. DARCY leverages custom static analysis, state-of-the art static analysis tools, and a custom parser generator in its

implementation to effectively detect and robustly repair architectural inconsistencies. The results of our evaluation indicate a pervasive existence of architectural inconsistencies among open-source Java 9+ applications. According to our experiment, DARCY's automatic repair results in a significant reduction of the attack surface, enhancement of encapsulation, and reduction of memory usage for Java 9+ applications. Possible future directions of this research include (1) providing run-time architectural visualization showing the architectural inconsistencies as they occur, (2) expanding DARCY as an interactive framework that modifies the repair decisions based on the developer's feedback in real-time, and (3) providing an implementation of DARCY as a plug-in for popular Integrated Development Environments (IDE).

IX. DATA AVAILABILITY

Darcy's research artifacts are publicly available on Zenodo.[3] The artifacts include the source code of the Darcy tool, the source code of the subject applications in Table II, the detailed information of the subject applications, and the GitHub crawling script for selecting the candidate modular Java applications.

ACKNOWLEDGMENT

We thank the anonymous reviewers of this paper for their detailed feedback, which helped us improve the work.

REFERENCES

[1] R. N. Taylor and N. Medvidovic, *Software Architecture: Foundations, Theory, and Practice*. Hoboken, NJ, USA: Wiley, 2009.
[2] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: Connecting software architecture to implementation," in *Proc. 24th Int. Conf. Softw. Eng.*, New York, NY, USA: ACM, 2002, pp. 187–197, doi: 10.1145/581339.581365.
[3] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead Jr., and J. E. Robbins, "A component- and message-based architectural style for GUI software," in *Proc. 17th Int. Conf. Softw. Eng.*, New York, NY, USA: ACM, 1995, pp. 295–304, doi: 10.1145/225014.225042.
[4] "Project Jigsaw." OpenJDK. Accessed: Jul. 23, 2023. [Online]. Available: http://openjdk.java.net/projects/jigsaw/
[5] N. Ghorbani, J. Garcia, and S. Malek, "Detection and repair of architectural inconsistencies in Java," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 560–571.
[6] "Darcy++ web page." Google. [Online]. Available: https://sites.google.com/view/darcy-plus-plus/homess
[7] K. Sharan, "The module system," in *Java 9 Revealed*. Berkeley, CA, USA: Apress, 2017, pp. 7–30.
[8] M. Reinhold, "JSR 376: Java platform module system," Oracle Corporation, Austin, TX, USA, Tech. Rep. Accessed: Jul. 23, 2023. [Online]. Available: https://openjdk.org/groups/web/crServer.html
[9] P. Deitel, "Understanding Java 9 modules." Oracle. [Online]. Available: https://www.oracle.com/corporate/features/understanding-java-9-modules.html
[10] P. J. Deitel and H. M. Deitel, *Java 9 for Programmers*. Englewood Cliffs, NJ, USA: Prentice Hall, 2017.
[11] S. Mak and P. Bakker, *Java 9 Modularity: Patterns and Practices for Developing Maintainable Applications*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2017.
[12] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, and D. Smith, "The Java® language specification Java SE 9 edition," Oracle Corporation, Austin, TX, USA. Techn. Rep., [Online]. Available: https://docs.oracle.com/javase/specs/jls/se9/html/index.html
[13] "API specification for the Java platform, standard edition: Class ServiceLoader." Oracle Corporation. Accessed: Jul. 23, 2023. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/ServiceLoader.html
[14] F.-J. Elmer, "Classycle: Analysing tools for Java class and package dependencies," Classycle. Accessed: Jul. 23, 2023. [Online]. Available: https://classycle.sourceforge.net/
[15] J. Garcia, M. Hammad, and S. Malek, "Lightweight, obfuscation-resilient detection and family identification of android malware," *ACM Trans. Softw. Eng. Methodol.,* vol. 26, no. 3, 2018, Art. no. 11.
[16] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A Java bytecode optimization framework," in *Proc. CASCON 1st Decade High Impact Papers*, Indianapolis, IN, USA: IBM Press, 2010, pp. 214–224.
[17] "ServiceLoader (Java SE 9 & JDK 9)," Oracle Help Centre. Accessed: Jul. 23, 2023. [Online]. Available: https://docs.oracle.com/javase/9/docs/api/java/util/ServiceLoader.html
[18] ANTLR. Accessed: Jul. 23, 2023. [Online]. Available: http://www.antlr.org
[19] GitHub. Accessed: Jul. 23, 2023. [Online]. Available: https://github.com
[20] "Beautiful Soup," Segfault. Accessed: Jul. 23, 2023. [Online]. Available: https://www.crummy.com/software/BeautifulSoup/
[21] P. Manadhata and J. M. Wing, "Measuring a system's attack surface," School of Computer Science, Carnegie-Mellon Univ., Pittsburgh, PA, USA, Tech. Rep., 2004. Available: https://www.cs.cmu.edu/~wing/publications/tr04-102.pdf
[22] P. K. Manadhata, K. M. Tan, R. A. Maxion, and J. M. Wing, "An approach to measuring a system's attack surface," School of Computer Science, Carnegie-Mellon Univ., Pittsburgh, PA, USA, Tech. Rep., 2007.
[23] P. K. Manadhata and J. M. Wing, "An attack surface metric," *IEEE Trans. Softw. Eng.*, vol. 37, no. 3, pp. 371–386, May/Jun. 2011.
[24] E. Bouwers, A. van Deursen, and J. Visser, "Quantifying the encapsulation of implemented software architectures," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME),* , Piscataway, NJ, USA: IEEE Press, 2014, pp. 211–220.
[25] L. C. Briand, S. Morasca, and V. R. Basili, "Measuring and assessing maintainability at the end of high level design," in *Proc. Conf. Softw. Maintenance (CSM),* Piscataway, NJ, USA: IEEE Press, 1993, pp. 88–87.
[26] J. Lakos, *Large-Scale C++ Software Design*. Reading, MA, vol. 173, pp. 217–271, 1996.
[27] M. R. Shaheen and L. du Bousquet, "Quantitative analysis of testability antipatterns on open source Java applications," in *QAOOSE Proc.*, 2008, p. 21.
[28] R. Yokomori, N. Yoshida, M. Noro, and K. Inoue, "Extensions of component rank model by taking into account for clone relations," in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evolution, Reeng. (SANER),* , vol. 3, Piscataway, NJ, USA: IEEE Press, 2016, pp. 30–36.
[29] E. Constantinou, G. Kakarontzas, and I. Stamelos, "Towards open source software system architecture recovery using design metrics," in *2011 15th Panhellenic Conf. Inform.*, Sept 2011, pp. 166–170.
[30] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural change in open-source software systems," in *Proc. IEEE/ACM 12th Work. Conf. Mining Softw. Repositories (MSR)* , Piscataway, NJ, USA: IEEE Press, 2015, pp. 235–245.
[31] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Piscataway, NJ, USA: IEEE Press, 2013, pp. 486–496.
[32] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidović, and R. Kroeger, "Comparing software architecture recovery techniques using accurate dependencies," in *Proc. 37th Int. Conf. Softw. Eng.* vol. 2, Piscataway, NJ, USA: IEEE Press, 2015, pp. 69–78.
[33] E. Constantinou, G. Kakarontzas, and I. Stamelos, "Open source software: How can design metrics facilitate architecture recovery?" 2011, *arXiv:1110.1992*.
[34] L. Hendren, "Uses of the soot framework," Sable Research Group. Accessed: May 8, 2024. [Online]. Available: http://www.sable.mcgill.ca/~hendren/sootusers/
[35] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for Java program analysis: A retrospective," in *Proc. Cetus Users Compiler Infrastructure Workshop (CETUS)*, vol. 15, 2011, p. 35.
[36] "Soot GitHub issue." GitHub. Accessed: Jul. 23, 2023. [Online]. Available: https://github.com/Sable/soot/issues

[3]https://doi.org/10.5281/zenodo.10574309

[37] "TIOBE Index for August 2018." TIOBE. Accessed: Jul. 23, 2023. [Online]. Available: https://www.tiobe.com/tiobe-index/

[38] "The State of the Octoverse 2017." Dribbble. Accessed: Jul. 23, 2023. [Online]. Available: https://octoverse.github.com/

[39] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala component model for consumer electronics software," *Computer*, vol. 33, no. 3, pp. 78–85, Mar. 2000.

[40] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Trans. Softw. Eng.*, vol. 26, no. 1, pp. 70–93, Jan. 2000.

[41] K. Lau and Z. Wang, "Software component models," *IEEE Trans. Softw. Eng.*, vol. 33, no. 10, pp. 709–724, Oct. 2007.

[42] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *IEEE Trans. Softw. Eng.*, vol. 35, no. 4, pp. 573–591, Jul./Aug. 2009.

[43] R. Koschke, "Architecture reconstruction," in *Softw. Eng.*, Springer-Verlag, 2006, pp. 140–173.

[44] I. Ivkovic and M. Godfrey, "Enhancing domain-specific software architecture recovery," in *Proc. 11th IEEE Int. Workshop Program Comprehension,* Piscataway, NJ, USA: IEEE Press, 2003, pp. 266–273.

[45] M. W. Godfrey and E. H. Lee, "Secrets from the monster: Extracting Mozilla's software architecture," in *Proc. 2nd Symp. Constructing Softw. Eng. Tools (CoSET)*. Citeseer, 2000, pp. 15–23.

[46] M. Mirakhorli, Y. Shin, J. Cleland-Huang, and M. Cinar, "A tactic-centric approach for automating traceability of quality concerns," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE),* Piscataway, NJ, USA: IEEE Press, 2012, pp. 639–649.

[47] A. Shahbazian, Y. K. Lee, D. Le, Y. Brun, and N. Medvidovic, "Recovering architectural design decisions," in *Proc. IEEE Int. Conf. Softw. Archit. (ICSA)*, Piscataway, NJ, USA: IEEE Press, 2018.

[48] E. Constantinou, G. Kakarontzas, and I. Stamelos, "An automated approach for noise identification to assist software architecture recovery techniques," *J. Syst. Softw.*, vol. 107, pp. 142–157, 2015.

[49] D. Qiu, Q. Zhang, and S. Fang, "Reconstructing software high-level architecture by clustering weighted directed class graph," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 25, no. 4, pp. 701–726, 2015.

[50] M. Shtern and V. Tzerpos, "Clustering methodologies for software engineering," *Adv. Soft. Eng.*, vol. 2012, pp. 1: 1–1:1, Jan. 2012, doi: 10.1155/2012/792024.

[51] O. Maqbool and H. Babri, "Hierarchical clustering for software architecture recovery," *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 759–780, Nov. 2007.

[52] R. A. Bittencourt and D. D. S. Guerrero, "Comparison of graph clustering algorithms for recovering software architecture module views," in *Proc. 13th Eur. Conf. Softw. Maintenance Reeng. (CSMR)* , Piscataway, NJ, USA: IEEE Press, 2009, pp. 251–254.

[53] M. M. Hammad, I. Abueisa, and S. Malek, "Tool-assisted componentization of java applications," in *Proc. IEEE 19th Int. Conf. Softw. Archit. (ICSA)*, Piscataway, NJ, USA: IEEE Press, 2022, pp. 36–46.

[54] C. Shi, "Modularization of C++ applications based on C++ 20 modules," Ph.D. dissertation, Univ. California, Irvine, CA, USA, 2022.

[55] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software reflexion models: Bridging the gap between design and implementation," *IEEE Trans. Softw. Eng.*, vol. 27, no. 4, pp. 364–380, Apr. 2001.

[56] N. Medvidovic, A. Egyed, and P. Gruenbacher, "Stemming architectural erosion by coupling architectural discovery and recovery." in *Proc. Stray Relief Anim. Welfare*, vol. 3, 2003, pp. 61–68.

[57] A. Van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva, "Symphony: View-driven software architecture reconstruction," in *Proc. IEEE/IFIP Conf. Softw. Archit. (WICSA)* , Piscataway, NJ, USA: IEEE Press, 2004, pp. 122–132.

[58] L. O'Brien, D. Smith, and G. Lewis, "Supporting migration to services using software architecture reconstruction," in *Proc. 13th IEEE Int. Workshop Softw. Technol. Eng. Pract.,* Piscataway, NJ, USA: IEEE Press, 2005, pp. 81–91.

[59] J. B. Tran, M. W. Godfrey, E. H. Lee, and R. C. Holt, "Architectural repair of open source software," in *Proc. 8th Int. Workshop Program Comprehension (IWPC)*, Piscataway, NJ, USA: IEEE Press, 2000, pp. 48–59.

[60] M. Abi-Antoun, J. Aldrich, D. Garlan, B. Schmerl, N. Nahas, and T. Tseng, "Improving system dependability by enforcing architectural intent," in *Proc. ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, New York, NY, USA: ACM, 2005, pp. 1–7.

[61] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman, "DiscoTect: A system for discovering architectures from running systems," in *Proc. 26th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2004, pp. 470–479.

[62] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," in *Proc. ACM Sigplan Notices*, vol. 40, no. 10. New York, NY, USA, 2005, pp. 167–176.

[63] J. A. Diaz-Pace, J. P. Carlino, M. Blech, A. Soria, and M. R. Campo, "Assisting the synchronization of UCM-based architectural documentation with implementation," in *Proc. Joint Work. IEEE/IFIP Conf. Softw. Archit. (WICSA) & 3rd Eur. Conf. Softw. Archit. (ECSA)*, Piscataway, NJ, USA: IEEE Press, 2009, pp. 151–160.

[64] M. Sefika, A. Sane, and R. H. Campbell, "Monitoring compliance of a software system with its high-level design models," in *Proc. 18th Int. Conf. Softw. Eng.,* Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1996, pp. 387–396.

[65] R. Koschke and D. Simon, "Hierarchical reflexion models," in *Proc. 10th Work. Conf. Reverse Eng.*, Piscataway, NJ, USA: IEEE Press, vol. 3, 2003, pp. 186–208.

[66] A. Christl, R. Koschke, and M.-A. Storey, "Equipping the reflexion method with automated clustering," in *Proc. 12th Work. Conf. Reverse Eng.,* Piscataway, NJ, USA: IEEE Press, 2005, pp. 10–98.

[67] R. Koschke, P. Frenzel, A. P. Breu, and K. Angstmann, "Extending the reflexion method for consolidating software variants into product lines," *Softw. Qual. J.*, vol. 17, no. 4, pp. 331–366, 2009.

[68] A. Gurgel et al., "Blending and reusing rules for architectural degradation prevention," in *Proc. 13th Int. Conf. Modularity*, New York, NY, USA: ACM, 2014, pp. 61–72.

[69] J. Buckley, S. Mooney, J. Rosik, and N. Ali, "JITTAC: A just-in-time tool for architectural consistency," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2013, pp. 1291–1294.

[70] M. Caporuscio, H. Muccini, P. Pelliccione, and E. Di Nisio, "Rapid system development via product line architecture implementation," in *Proc. Int. Workshop Rapid Integration Softw. Eng. Techn.,* Berlin, Heidelberg: Springer-Verlag, 2005, pp. 18–33.

[71] J. Grundy, "Multi-perspective specification, design and implementation of software components using aspects," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 10, no. 6, pp. 713–734, 2000.

[72] S. Malek, M. Mikic-Rakic, and N. Medvidovic, "A style-aware architectural middleware for resource-constrained, distributed systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 3, pp. 256–272, Mar. 2005.

[73] J. Aldrich, C. Omar, A. Potanin, and D. Li, "Language-based architectural control," in *Proc. Int. Workshop Aliasing, Capabilities Ownership (IWACO)*, 2014, pp. 1–11.

[74] A. Radjenovic and R. F. Paige, "The role of dependency links in ensuring architectural view consistency," in *Proc. 7thWork. IEEE/IFIP Conf. Softw. Archit.*, Piscataway, NJ, USA: IEEE Press, 2008, pp. 199–208.

[75] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, "Abstractions for software architecture and tools to support them," *IEEE Trans. Softw. Eng.*, vol. 21, no. 4, pp. 314–335, Apr. 1995.

[76] N. Ubayashi, J. Nomura, and T. Tamai, "Archface: A contract place where architectural design and code meet together," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.Vol. 1*. New York, NY, USA: ACM, 2010, pp. 75–84.

[77] Y. Zheng and R. N. Taylor, "Enhancing architecture-implementation conformance with change management and support for behavioral mapping," in *Proc. 34th Int. Conf. Softw. Eng.*, Piscataway, NJ, USA: IEEE Press, 2012, pp. 628–638.

[78] Y. Zheng, C. Cu, and R. N. Taylor, "Maintaining architecture-implementation conformance to support architecture centrality: From single system to product line development," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 2, 2018, Art. no. 8.

[79] H. Song et al., G. Huang, F. Chauvel, Y. Xiong, Z. Hu, Y. Sun, and H. Mei, "Supporting runtime software architecture: A bidirectional-transformation-based approach," *J. Syst. Softw.*, vol. 84, no. 5, pp. 711–723, 2011.

[80] A. Dann, B. Hermann, and E. Bodden, "Modguard : Identifying integrity & confidentiality violations in Java modules," *IEEE Trans. Softw. Eng.*, vol. 47, no. 8, pp. 1656–1667, Aug. 2021.

[81] Y. Wang et al., "Do the dependency conflicts in my project matter?" in *Proc. 26th ACM joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 319–330.

[82] Y. Wang et al., "Will dependency conflicts affect my program's semantics," *IEEE Trans. Softw. Eng.*, vol. 48, no. 7, pp. 2295–2316, Jul. 2022.

[83] L. Lambers, D. Strüber, G. Taentzer, K. Born, and J. Huebert, "Multi-granular conflict and dependency analysis in software engineering based

on graph transformation," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 716–727.
[84] Y. Wang et al., "Could I have a stack trace to examine the dependency conflict issue?" in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 572–583.
[85] Y. Wang et al., "Watchman: Monitoring dependency conflicts for Python library ecosystem," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, 2020, pp. 125–135.
[86] Y. Wang et al., "Hero: On the chaos when PATH meets modules," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 99–111.
[87] Z. Li, Y. Wang, Z. Lin, S.-C. Cheung, and J.-G. Lou, "NuFix: Escape from NuGet dependency maze," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2022, pp. 1545–1557.
[88] "OSGI Alliance." OSGi. Accessed: May 8, 2024. [Online]. Available: https://www.osgi.org/resources/where-to-start/

Negar Ghorbani received the B.S. degree in computer software engineering from Sharif University of Tehran, and the M.S. and Ph.D. degrees in software engineering from the University of California, Irvine (UCI). Her research interests include software engineering, focusing on the applications of machine learning and natural language processing for improving developer productivity, and software development lifecycle.

Tarandeep Singh received the double bachelor's degree with a B.S. in software engineering and a B.A. in business economics from the University of California, Irvine. He currently works as a software engineer in the industry. His research interests include software engineering, particularly in software analysis.

Joshua Garcia (Member, IEEE) received the B.S. degree in computer engineering and computer science from the University of Southern California (USC) and the M.S. and Ph.D. degrees in computer science from USC. He is an Assistant Professor with the School of Information and Computer Sciences, University of California, Irvine. His current research interests include software engineering, focusing on software testing and analysis, software security, and software architecture. He is a member of the ACM and ACM SIGSOFT.

Sam Malek (Member, IEEE) received the B.S. degree in information and computer science from the University of California, Irvine, and the M.S. and Ph.D. degrees in computer science from the University of Southern California. He is a Professor with the Informatics Department, School of Information and Computer Sciences, University of California, Irvine. His research interests include software engineering, and to date his focus has spanned the areas of software analysis and testing, mobile computing, security, software architecture, and accessible computing.