MODGUARD: Identifying Integrity & Confidentiality Violations in Java Modules

Andreas Dann, Ben Hermann, and Eric Bodden

Abstract—With version 9, Java has been given the new module system Jigsaw. Major goals were to simplify maintainability of the JDK and improve its security by encapsulating modules’ internal types. While the module system successfully limits the visibility of internal types, it does not prevent sensitive data from escaping. Since the module system reasons about types only, objects are allowed to escape even if that module declares the type as internal. Finding such unintended escapes is important, as they may violate a module’s integrity and confidentiality, but is a complex task as it requires one to reason about pointers and type hierarchy.

We thus present MODGUARD, a novel static analysis based on Doop which complements the Java module system with an analysis to automatically identify instances that escape their declaring module. Along with MODGUARD we contribute a complete formal definition of a module’s entrypoints, i.e., the method implementations that a module actually allows other modules to directly invoke. We further make available a novel micro-benchmark suite MIC9BENCH to show the effectiveness but also current shortcomings of MODGUARD, and to enable comparative studies in the future.

Finally, we describe a case study that we conducted using Apache Tomcat, which shows that a migration of applications towards Jigsaw modules does not prevent sensitive instances from escaping, yet also shows that MODGUARD is an effective aid in identifying integrity and confidentiality violations of sensitive instances.

Index Terms—Java 9, Jigsaw, Module Systems, Security, Static Escape Analysis, Doop, Soot.

1 INTRODUCTION

With the release of version 9, the Java programming platform has introduced the module system Jigsaw, whose “primary goals are to make implementations of the Platform more easily scalable [...], improve security and maintainability” [1] by encapsulating internal, security-sensitive types [2]. With Java modules, developers decide which packages and classes will be exposed, and which will remain internal [2]. The Java platform has already been partitioned using the module system. For instance, java.lang.Class is exported publicly, whereas security-sensitive classes such as jdk.internal.misc.Unsafe1 are only available to their declaring module.

The issue we wish to address with this work is that, although the module system encapsulates internal types successfully, it does not prevent the unintentional escaping of security-sensitive data to the outside, e.g., secret keys. As we show in this work, reasoning about data flows between modules and which classes, methods, and fields a module actually exposes is complex:

While the module system prevents internal types from being visible outside of their declaring modules, instances of internal types can still escape a module, and all methods or fields of its exported supertypes can be invoked, accessed, or modified. Although escaping instances cannot be downcasted to the internal type, this behavior makes it complex for developers to reason about (unintended) data leaks or manipulation, as it requires reasoning about pointers and types, which is hard to discern in manual reviews.

As an example, consider CVE-2017-5648, a vulnerability in Apache Tomcat. The vulnerable part of the code fails to properly protect a critical object before passing it to untrusted code. This allows untrusted code to perform security-critical operations on the object, which in turn would lead to data leaks and privilege elevation.

CVE-2017-5648 was contained in a version of Tomcat, which did not yet use Java modules. In this work, we show that the module system misses analyses to identify escaping sensitive objects and data leaks. In particular, we show that Java’s module system encapsulation of internal types alone is insufficient to identify or avoid such vulnerabilities, as it reasons about declared types only, completely neglecting data flows resulting from pointers and virtual dispatch. To detect such vulnerabilities, developers would have to manually determine all potential interactions between modules, including exported types and instances of internal classes that might escape a module or not. This, however, involves complex reasoning about the module’s implementation.

To complement the Java module system with means to identify and analyze unintended data flows, we designed and implemented MODGUARD, a novel static analysis which automatically identifies instances, fields, or methods that might escape their declaring module. Thereby, MODGUARD enables developers to leverage the module system security-wise by identifying unintended data flows that the modularized application should prevent. We have implemented MODGUARD on top of Doop [3] and made it publicly available together with the results of all experiments described in this paper.

To assure the soundness [4] of our approach, we derived and present here a novel formal specification of what we
call module entrypoints, i.e., the set of method implementations a module defines, and which are invokable by other modules, either explicitly because their type is exported, or implicitly due to an exported supertype. Our specification—another major contribution of this paper—exactly captures the conditions under which classes, fields, and methods of a module become accessible.

As we are among the first to develop a static analysis devoted to Java modules, there exists no previous benchmark allowing systematic studies on the subject. To remedy this for future research, we make available MIC9BENCH, a novel open-source micro-benchmark for comparing analyses’ effectiveness for computing module entrypoints, and identifying integrity and confidentiality violations in Java modules. Experiments with MIC9BENCH show that MODGUARD identifies data flows causing integrity and confidentiality violations in Java modules effectively, but could benefit from future improvements for handling Java’s dynamic language support (i.e., java.lang.invoke.MethodHandle and VarHandle).

To demonstrate the efficiency of our analysis on a real-world application, we conduct a case study on Apache Tomcat 8.5.21. Due to the novelty of Java’s module system code using modules is still scarce, and there exist no multi-module projects on Maven Central. Thus, we ourselves migrated Tomcat to the module system, following the approach presented by Corwin et al. [5], and migrated each jar file to a separate module. Our case study shows that this naive migration fails to mitigate confidentiality and integrity violations. Developers instead must restructure the architecture to confine sensitive data properly utilizing the encapsulation the module system provides.

In our case study, we identified violations in 12 out of 26 Tomcat modules. Even if we retain only those modules’ export declarations that are required to compile Tomcat, violations remain in 6 modules. While the most effective reduction of violations occurs in the module catalina.ha, its violations are reduced by 35% only. To successfully confine sensitive data within a module, developers must introduce modules with care, also considering type hierarchy and pointers. In particular, the option to invoke methods of supertypes on instances of internal types makes it hard for developers to reason which entities can be accessed or manipulated. Our case study shows that MODGUARD supports such reasoning by pointing out unwanted interactions, and thus effectively supports migrations to Java’s module system, as well as refactorings of existing modules.

In summary, this paper makes the following contributions:

- a formal specification of module entrypoints, which defines which classes, fields, and methods of a module will be accessible (Section 3),
- a new static analysis, MODGUARD, to identify integrity and confidentiality violations in Java modules (Section 4),
- a benchmark, MIC9Bench, for static analyses targeting Java modules (Section 5),
- a case study on the application Tomcat (Section 5).

3. https://github.com/secure-software-engineering/mic9bench

2 JAVA MODULE SYSTEM DESIGN AND EXAMPLE

Java 9 introduces modules to the platform as first-class constructs [6]. A module encapsulates its internal API, preventing access to internal types during compile-time and run-time [2]. In the following, we provide a basic introduction to the design of Java’s module system and the resulting consequences for the visibility of types, methods, and fields. To motivate our confidentiality and integrity analysis for Java modules, we present Tomcat’s vulnerability CVE-2017-5648 which illustrates that the module system guarantees the encapsulation of internal types but fails to prevent violations due to unintended data leaks or manipulation of escaping instances.

2.1 Design of the Module System

Java modules assemble packages, classes, native code, and further resources, like simple JAR files. Yet, the new modules contain a static module descriptor which specifies the module’s unique name, its dependency on other modules, its exported packages, and a definition of re-exported dependencies. The module descriptor is processed by the Java compiler as well as the Java Virtual Machine (JVM), causing them to check and prevent access to the internal types of a module both at compile- and run-time. The dependencies between modules, as specified in the module descriptors, form an acyclic module graph. This module graph is used to resolve references between classes, replacing the previous class-loading based on the linear classpath.

![Example Module Graph](https://example.com/module-graph)

Figure 1: Example Module Graph. Blue Arrows: Module Dependencies; Dashed Arrows: Visibility Relations.

Up to Java 8, every public class was visible to any other class on the classpath loaded by the same ClassLoader. In Java 9, a class contained in a module may only access the types of another module if there exists a corresponding dependency relation in the module graph [7]. A class contained in module A is allowed to access types of another module B only if A depends on B.

Figure 1 shows, as an example, the subset of the module graph of the Java Development Kit (JDK) module java.desktop. The module java.desktop depends on the modules java.datatransfer, java.xml, which comprises the exported package java.xml and the internal package com.sun, and transitively on java.base. The root of each module graph is the module java.base, which contains essential Java classes. Due to the dependency relations, in Figure 1, the module java.desktop can access only exported classes of modules it depends on. Consequently, the module java.datatransfer cannot access any package in the module java.xml.

In particular, a module can only access types of modules it depends on whose package are exported and that are
declared public. For instance in Figure 1, only the types of the exported package java.xml are visible to the module java.desktop, whereas types declared in the internal packages com.sun are invisible. Thus, all types that are neither explicitly declared public nor declared in an exported package are invisible outside of their declaring module.

But crucially modules can invoke not only methods on object instances of exported types. Classes of a module A can directly invoke all methods that are declared public (static or not), protected and static, or implement a method from an exported supertype on object instances of a dependent module B, disregarding whether the types are declared in exported packages or not. The only condition is that classes of A must obtain access to an object instance to invoke the method on.


Listing 1 shows a simplified example in which the access to an internal object instance results in a confidentiality violation. The module my.mod exports its package api, making the types KeyProvider and Key visible to other modules, but keeping the type SecretKey internal. The module’s api creates an instance of the internal type SecretKey, returning it as the exported type Key. Although SecretKey is internal and stores its key material in the private field keyMaterial, classes outside the module my.mod can access the stored key using the inherited method Key.getKey(). Since the Java module system confines types only, invoking methods of exported supertypes is permitted. The actual problem, in this example, is the leak of the internal keyMaterial. For detecting such leaks, developers need to reason about complex pointers and type hierarchy. Even for this simple example a developer needs to reason that the constructor of the internal class SecretKey hands the internal, sensitive keyMaterial to the constructor of its exported superclass Key, which assigns it to the field key, which can be retrieved by the exported, inherited method Key.getKey(). The inherited method SecretKey.getKey() is, in our terminology, an entrypoint of the declaring module (in addition to the directly exported entrypoints KeyProvider.getKey() and Key.getKey()). Obviously, this manual reasoning becomes virtually impossible in real-world scenarios with more complex pointers and types, as the Tomcat example in Section 5 shows.

Excursion: Open modules By default, the JVM denies run-time access to internal types via reflection, or Java’s dynamic-language API [2], [7]. However, modules can be declared as open. This grants compile-time access to exported packages only, but run-time access throughout [6]. Likewise, packages can be declared as open, thereby granting reflective access to all types. For the remainder of this paper, we assume that neither modules nor packages are opened since they do not, as the name implies, encapsulate internal types.

2.2 Motivating Example - Tomcat CVE-2017-5648

To motivate our approach with a real-world example, we describe a vulnerability in Tomcat’s component catalina reported as CVE-2017-5648. The vulnerability enables “an application [running with a SecurityManager], to retain a reference to the request object and thereby access and/or modify information associated with another web application.”

Listing 2 Fix for CVE-2017-5648 - Tomcat rev. 1785776

This vulnerability shows that programming mistakes involving the unintentional leak of sensitive objects occur in real-world applications and that they may have a significant impact on an application’s security.

The corresponding fix for the vulnerability is shown in Listing 2. Lines 9, 12 show the vulnerable code, whereas lines 10, 13 show the fix.

In the vulnerable lines 9, 12 any Request received by the FormAuthenticator was dispatched directly to the context object representing the web application, which one must assume to be attacker-controlled. Since the dispatched Request object grants access to sensitive methods, any web application could abuse it.

To fix this vulnerability, the authors expose a RequestFacade to the web application, instead of the original Request object. The RequestFacade wraps the Request object, denying any access to its security-critical methods.

CVE-2017-5648 was contained in a version of Tomcat that did not yet use Java modules. Yet, while the module system can encapsulate the internal Request object, its encapsulation is too weak to prevent such vulnerabilities since it reasons about types only: Both classes Request and RequestFacade implement the interface ServletRequest. Crucially, this interface has to be exported to be usable by web applications. Thus, any code outside of the component catalina may invoke any method on any object of type RequestFacade, but also on Request, as long
as the method is defined in the interface ServletRequest. Unfortunately, ServletRequest defines a number of such methods, e.g., getParameter, getLocale, etc., and because ServletRequest is exported, attackers can invoke those methods on the unprotected Request object even if the type Request is declared as internal.

The lesson learned is thus that vulnerabilities of this kind cannot be remedied solely by relying on the encapsulation of types, but also require reasoning about escaping instances and data flows between modules.

The vulnerability shows that the module system needs to be complemented with analyses that detect if security-critical objects like Request escape. This is generally non-trivial, however, as an analysis must reason about pointers while being aware of a module’s entrypoints and boundaries. In the example, the context object represents the untrusted web application that runs on top of Tomcat, and outside of the component catalina. Crucially, the analysis has to recognize that values passed to context.fireRequestInitEvent(..) in line 9 and fireRequestDestroyEvent(..) in line 12 resemble a confidentiality violation: because they are passed to a context object residing outside the module.

An analysis has to take into consideration that there are two internal subtypes of the exported interface ServletRequest: one which is security-sensitive (Request) and one which is safe to be used outside the module (Request-Facade). As the information which of these two classes is sensitive is domain specific, an analysis has to be provided with a list of sensitive entities.

3 Definition of Module Entrypoints

Implementing a static analysis on individual modules is challenging, as the analysis must be conducted on open code much alike call-graph construction for libraries [8]: a module can be linked to any other, and the analysis must foresee all ways in which those other modules can invoke that module’s functionality. MODGUARD constructs an entrypoint model that precisely defines those possible invocations.

3.1 Explicitly vs. Implicitly Reachable Entrypoints

As the example in Listing 1 shows, invocations crossing module boundaries are not restricted to explicitly exported types. Instead, external code may interact with many methods the module defines. We call those methods “entrypoints”. Entrypoints comprise two kinds of methods: explicitly and implicitly reachable methods. Initially, external code can only invoke explicitly reachable methods. They are methods whose declaring type is exported. In Listing 1, the methods KeyProvider.getKey() and Key.get() are explicitly reachable, as their types Key and KeyProvider are declared in the exported package api. The use of explicitly reachable methods may grant access to further methods as well, which we call implicitly reachable. Implicitly reachable methods are either declared by internal types, inherit, implement, or override explicitly reachable methods of supertypes.

In the example, SecretKey.getKey() is an implicitly reachable method: its declaring type SecretKey is kept internal, yet the method can be directly invoked on the object that is returned by KeyProvider.getKey(). Note that implicit reachability is different from the usual notion of indirect reachability, i.e., the ability to invoke a method indirectly through a chain of calls.

3.2 Design Choices

The union of all explicitly and all implicitly reachable methods constitutes an upper bound of a module’s entrypoints. This upper bound is naïve since it ignores whether objects of internal types, which declare the implicit method, actually escape the module or not.

To gain a tighter bound, MODGUARD identifies which instances of internal types actually escape a module utilizing points-to analysis.

Internal types may escape through different paths: they can be returned from an accessible method, can be referenced by a (static-) field of another escaping object, can be assigned to the field of a parameter, etc. Computing all escaping objects results in scalability issues since a points-to analysis has to be carried out for all these potential paths before the actual client analysis starts. Hence, MODGUARD computes the entrypoints for a subset of the escaping paths only, and postpones the identification of further escapes to client analyses.

MODGUARD’s initial entrypoint set is based on the following design decisions. First, the entrypoints are sound w.r.t. explicitly reachable methods. Second, the entrypoint computation should be scalable, and thus the entrypoint model may only contain a subset of the implicitly reachable methods.

To be sound w.r.t. the explicit methods, our entrypoint model computes all explicit methods of the exported types and supertypes. To be scalable, the entrypoint model only computes implicit methods of internal types that escape directly through an exported type. The initial entrypoint model computes the implicit methods of escaping types only that are returned from an accessible method or that are referenced by (static-) fields of exported classes. Additionally, the entrypoint model also includes the exported supertypes’ methods of escaping instances.

Limiting the entrypoint set to implicit methods of objects that are (directly) returned from the module bounds the set to a reasonable size. Without these constraints, the size of the initial entrypoint set would explode quickly, for instance, consider that the class Object is a supertype of every class, and thus all its public methods would be entrypoints, too. Consequently, all types returned by these entrypoints would be also an entrypoint, e.g., the method getClass(), which returns an instance of type Class, which in turn defines further public methods would be entrypoints also, and so on. Thus, MODGUARD postpones the computation of entrypoints not directly declared in the module to client analyses, which may extend the initial set based on their objective.

3.3 Entrypoint Model: logic-based specification

We realized our entrypoint definition in a model that is a logic-based specification in the syntax of declarative, Datalog-based analysis rules extending Doop [3], [9], shown
in Figure 2. In the following, we introduce our entrypoint model in detail to make our description of the previous sections precise and showcase its generality.

Datalog rules establish facts about derived relations (the head) from the conjunction of previously established facts (the body), separated by the left arrow symbol (←).

Some of the relations in Figure 2 are functions, written as RELATION[DOMVAR] = VAL. This notation is equivalent to RELATION(DOMVAR, VAL) but required by Datalog, which throws an error if a computation yields multiple values for the same domain variable [10].

**Input Relations:**
- METHOD:DECLARING TYPE[method:M] = type:T
- METHOD:MODIFIER(modifier: String, method:M)
- RETURN VAR(var: V, method:M)
- CLASS MODIFIER(modifier: String, class: T)
- SUPERTYPEOF(superType:T, type: T)
- OVERRIDE METHOD (method:M, type:T, supertype:T)
- IMPLEMENT INTERFACE (method:M, type:T, supertype:T)
- MODULE DECLARE Type (module:D, type:T)
- MODULE EXPORTS (fromModule:D, package: String, toModule:D)
- EXPORT TYPE(type:T)
- MODULE FOR ANALYSIS (module:D)

**Output Relations:**
- EXPPLICIT METHOD (class:T, method:M)
- IMPLICIT METHOD (class:T, method:M)
- ENTRY POINT (method:M)
- CLASS HAS POSSIBLE ENTRY POINT (class:T)

Figure 2: Entrypoint model: domain, input, and output relations. Doop's [3], [9] default rules are gray.

**Domain:**
- T set of class types
- D set of module descriptors
- M set of method identifier
- H set of heap abstractions (e.g., allocation sites)
- F set of fields
- V set of program variables
- HC set of heap contexts

**Input Relations:**
- METHOD:DECLARING TYPE [method:M] = type:T
- METHOD:MODIFIER [modifier: String, method:M]
- RETURN VAR [var: V, method:M]
- CLASS MODIFIER [modifier: String, class: T]
- SUPERTYPEOF [superType:T, type: T]
- OVERRIDE METHOD [method:M, type:T, supertype:T]
- IMPLEMENT INTERFACE [method:M, type:T, supertype:T]
- MODULE DECLARE Type [module:D, type:T]
- MODULE EXPORTS [fromModule:D, package: String, toModule:D]
- EXPORT TYPE [type:T]
- MODULE FOR ANALYSIS [module:D]

**Output Relations:**
- EXPPLICIT METHOD [class:T, method:M]
- IMPLICIT METHOD [class:T, method:M]
- ENTRY POINT [method:M]
- CLASS HAS POSSIBLE ENTRY POINT [class:T]

Figure 2: Entrypoint model: domain, input, and output relations. Doop's [3], [9] default rules are gray.

To incorporate Java modules into Doop, we added module descriptors to the domain, which represents a module, its name, its (re-exported) dependencies, its exported packages, and internal packages.

**Input relations** The input relations of our entrypoint model, shown in Figure 2, are logically grouped: relations representing Doop’s intermediate representation, Java modules, and points-to information. The built-in relations represent the program as Datalog facts [10]: RETURN VAR represents a method’s return variable, METHOD:DECLARING TYPE represents its declaring type, METHOD:MODIFIER and CLASS MODIFIER represent the modifier of a method or class, and SUPERTYPEOF represents all supertypes of a type. The new input relations IMPLEMENT INTERFACE and OVERRIDE METHOD represent every method of a type that implements or overrides a method of an exported supertype.

The input relations MODULE DECLARE Type, MODULE EXPORTS, and EXPORT TYPE are module specific: MODULE DECLARE Type represents in which unique module a type is declared, MODULE EXPORTS specifies which module fromModule exports which package to which other module toModule, and EXPORT TYPE represents every publicly exported type. The input relation MODULE FOR ANALYSIS represents the module for which the entrypoints should be computed, as a Java project typically consists of a set of modules.

VAR POINTS TO and FLD POINTS TO encode points-to information: they link a return variable var or a field fld to an heap object heap.

**Output Relations** The output relations ENTRY POINT and CLASS HAS POSSIBLE ENTRY POINT encode the computed entrypoint model. The relations EXPPLICIT METHOD and IMPLICIT METHOD represent the explicitly and implicitly reachable methods.

**Entrypoint Logic** The entrypoint model definition is shown in Figure 3. The main rules CLASS HAS POSSIBLE ENTRY POINT and ENTRY POINT (in duplicate) state that the set of entrypoints constitutes every implicitly and explicitly reachable method.

The rule EXPPLICIT METHOD states that a method is explicitly reachable if it has the modifiers public or static protected, and its declaring class is exported.

The rule IMPLICIT METHOD represents all implicit methods; overriding, implementing, or inheriting a supertype’s method. To express implicit methods in Datalog, the rule IMPLICIT METHOD is defined twice; once for overridden methods, and once for inherited methods. Note that Datalog executes multiple definitions of the same rule independently and merges the results.

The first definition states that every method implementing or overriding a method of an exported supertype constitutes an implicitly reachable method. In correspondence with our design choices, we constrain this by the rules ENTRY POINT and METHOD RETURN TYPE; requiring that an object of type class actually escapes by a return of an previously established entrypoint.

The second definition of rule IMPLICIT METHOD states that every method declared in an exported supertype constitutes an implicitly reachable method, too. Similar to the first definition, the rule constrains this to methods of types retType that actually escape through a previously established entrypoint, encoded by the METHOD RETURN TYPE (ENTRY POINT, RET TYPE). Moreover, we limit implicit methods to those whose declaring types retType are defined in the module itself (MODULE DECLARE TYPE, RET TYPE), or whose declaring types are directly returned from another entrypoint of the module, encoded by the conjunction METHOD:DECLARING TYPE and MODULE DECLARE TYPE.

Since both rules EXPPLICIT METHOD and IMPLICIT METHOD recursively refer to the main rule ENTRY POINT, they are repeatedly applied to newly discovered entrypoints, until no further entrypoints are found. Thus, both rules compute entrypoints that become transitively reachable, e.g., if an internal type’s method grants access to further internal types.

The rule METHOD RETURN TYPE determines the possible concrete run-time types of objects flowing into method returns. The rule states that a method grants access to a type if the return variable var points-to a heap object heap of the type. Our implementation comprises a similar rule computing concrete type information also for instance and...
computes module entrypoints and points-to information using the static analysis frameworks Doop [3] and Soot [11].

Each usage scenario of MODGUARD’s client analysis might consider a different set of classes, methods, and fields as sensitive, represented by the input relations SENSITIVEFIELD, SENSITIVEMETHOD, and SENSITIVECLASS in Figure 5. We expect this information to be provided as user input either as our annotation @Critical or as a text file.

We have observed that sensitive information is often stored using primitive types, or arrays of those. For instance, secret keys are stored in byte arrays, while passwords are stored in char arrays. To track such primitive-typed data in addition to regular pointers, we use P/Taint [12] an extension to Doop that augments its points-to analysis with additional rules for a context-sensitive, flow-insensitive propagation of primitive-typed data.

Figure 4 gives an overview of MODGUARD consisting of three steps, which we introduce in the following.

4.1 Module Analysis Initialization
To compute the module entrypoints one first needs to construct the module graph from the information contained in the module descriptors. We adapted Soot to parse the descriptor of the module under analyses and its transitive dependencies using the information from requires, exports, and opens declarations.

MODGUARD then uses Soot to transform the bytecode from all modules into the Jimple intermediate representation [11], on which Doop operates in later stages of the analysis. Furthermore, MODGUARD marks primitive-typed sensitive entities as tainted for Doop’s P/Taint analysis [12].

4.2 Precise Modeling of Module Entrypoints
Modules do not have a single entrypoint (e.g., a main method). Modules instead compromise many entrypoints – the explicitly and implicitly – reachable methods, as described in Section 3. We implemented the rules for our module’s entrypoint model in Doop, directly based on the Datalog rules shown in Section 3. To model the module’s entrypoints precisely, MODGUARD’s confidentiality and integrity analysis completes the initial entrypoint set. Therefore, MODGUARD identifies objects that escape transitively through static or instance fields of escaping objects, arguments passed into the module, or arguments passed to callback methods outside the module. To detect such escapes, MODGUARD generates mock-objects representing the arguments of entrypoints’ methods, which it simulates to be passed to the corresponding entrypoints. Since MODGUARD does not require client code, such mock-objects simulate the allocation of the

4 Design and Implementation of MODGUARD
The examples presented in Section 2 show a clear motivation for an analysis to detect escaping objects that leak or manipulate sensitive data. For this purpose, we created MODGUARD, an analysis to detect such escaping objects in Java modules. To detect escaping objects, MODGUARD
entripoints’ arguments by client code for Doop’s points-to analysis.

Figure 5 shows the input relations and queries of MODGUARD’s client analysis to identify whether sensitive fields can be accessed or changed from outside the module under analyzes, the queries for sensitive methods and classes are defined analogously.

To identify data flows that lead to integrity and confidentiality violations, MODGUARD checks if the points-to sets of sensitive entities and the points-to sets of reachable and escaping entities intersect. If the two points-to sets intersect, a sensitive entity escapes the module, marking a confidentiality violation, or an object that has been passed into the module has been assigned to a sensitive entity, marking an integrity violation. In Figure 5, the input relations represent the points-to sets of such escaping objects and the sensitive entities. The input relation SENSITIVEFIELD represents the fields that should not be leaked or manipulated from outside of the module. The points-to set of these sensitive fields is represented by the relation SENSITIVEFIELDPOINTS TO.

The remaining input relations represent the points-to sets of further escaping objects. VISIBLEFIELDPOINTS TO represents the points to set of all fields that are either public or protected and reside in classes the module exports. The relation RETURNVARPOINTSTO represents the points-to set of the return variables of all reachable implicit and explicit methods. Similar, the relations MOCKOBJECTPOINTSTO and ARGUMENTOFCALLBACKPOINTSTO represent the points-to set of receiver mock-objects on which an entrypoint method is invoked, of mock-objects that are passed as arguments into the module, and of arguments that are passed to callback methods outside the module.

4.3 Module Integrity & Confidentiality Analysis

To identify whether the values of sensitive fields can leak or be manipulated, MODGUARD intersects the points-to set of sensitive fields and escaping objects in the query CHECKVIOLATIONFIELD in Figure 5. Each definition of the query CHECKVIOLATIONFIELD intersects the points-to set of the sensitive fields SENSITIVEFIELDPOINTSTO with one of the input relations that represents the points-to set of the escaping objects. If a query results in a non-empty intersection, MODGUARD reports a violation of the corresponding sensitive field.

For the example in Listing 1, MODGUARD’s confidentiality and integrity analysis works as follows. First, MODGUARD computes the entrypoint model for the module. As defined by the rules in Section 3 the entrypoint model contains the explicit method KeyProvider.getKey(). To represent the return value of this method, MODGUARD creates a mock-object in Doop of the internal type SecretKey representing the returned object new SecretKey(). Second, MODGUARD’s entrypoint definition detects, utilizing the entrypoint rule IMPLICITMETHOD, that the returned object’s type SecretKey constitutes the implicit entrypoint getKey(), declared in its public exported super-type Key. Third, MODGUARD binds the created mock-object new SecretKey() to the this parameter of the implicit method getKey() and creates an additional mock-object representing the returned key key, which aliases with the sensitive field keyMaterial. Finally, MODGUARD checks if the points-to set of the private field keyMaterial and the points-to set of the mock-objects intersect, utilizing the rule CHECKVIOLATIONFIELD. Since MODGUARD bound the this parameter of the method getKey() to the returned mock-object new SecretKey(), the points-to set of the returned object and sensitive field intersect, and MODGUARD successfully detects the confidentiality violation.

4.4 Limitations of MODGUARD

MODGUARD shares some inherent limitations with other static analyses. For instance, MODGUARD does not identify violations resulting from using MethodHandles.Lookup, MethodHandle, or VarHandle from Java’s dynamic-language API. Consequently, MODGUARD currently under-approximates w.r.t. Java’s dynamic-language API leading to false negatives. For dealing with reflection we use Doop’s extensions to resolve reflective calls in conjunction with our rules and queries [13].

Additionally, MODGUARD fails to detect violations occurring in native code, resulting in false negatives. Detecting violations in arbitrary native methods requires analyses for languages such as C and C++, or for native binaries, which is out of scope. However, for commonly used native method like System.arraycopy(), MODGUARD’s analysis can be supplemented with hand-crafted summaries (Datalog facts) as proposed by Sălicingu and Rinard [14] that specify the impact of these native methods on data flows between modules.

5 Evaluation

Our evaluation addresses the following research questions: RQ1 How effectively can MODGUARD identify confidentiality and integrity violations in Java modules?
**RQ2** Occur integrity and confidentiality inter-module violations during the migration of real-world applications and can MODGUARD properly identify these violations?

### 5.1 RQ1: Confidentiality and Integrity Violations in Java Modules – Micro MIC9Bench

While there exist benchmark suites for detecting Java vulnerabilities [15] or evaluating points-to analyses [16], there currently exists no benchmark specific to Java modules. Existing benchmarks suites cannot be used to assess the effectiveness of Java module analyses, as they comprise no module declarations, nor any ground truth for such analyses.

Specifically for Java 9 module analyses, we developed the test suite MIC9BENCH (module, integrity, confidentiality for Java 9). The test suite contains 22 small hand-crafted modules each violating either integrity or confidentiality.

Table 1 presents the results of MODGUARD applied to MIC9BENCH. The results show that MODGUARD successfully detects whether sensitive entities referenced by *fields can be accessed or modified from external code. Moreover, MODGUARD correctly detects if external code can gain access to sensitive *methods, i.e., if packages are erroneously exported.

**Table 1: MIC9Bench Test Results**

<table>
<thead>
<tr>
<th>Test Scenario</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessible Field</td>
<td>Integrity/Confidentiality primitive field ✓</td>
</tr>
<tr>
<td>Invokable Method</td>
<td>Access to explicit method ✓</td>
</tr>
<tr>
<td>Parameter</td>
<td>Entity added to parameter array, collection ✓</td>
</tr>
<tr>
<td>Exception</td>
<td>Declared/Undeclared exception ✓</td>
</tr>
<tr>
<td>Reflection &amp; Invoke API</td>
<td>Referenced by VarHandle/MethodHandle x</td>
</tr>
<tr>
<td>Side-Effect</td>
<td>Pass entity to native code x</td>
</tr>
</tbody>
</table>

MODGUARD identifies whether sensitive entities are disclosed through parameters which are passed into the module, i.e., if sensitive entities are assigned to a parameter or added to a collection, which is accessible to external code. Similarly, MODGUARD successfully detects if sensitive entities leak as arguments of callback methods or exceptions.

To detect if an exposed collection discloses sensitive entities MODGUARD uses Doop’s partially 1-object-sensitive points-to-analysis context-insensitive++.

MODGUARD does not detect if a module exposes references to internal types, methods, and fields in the form of Method-, VarHandle, or MethodHandles.Lookup objects. Since the access checks for *Handle instances are made at creation-time rather than at run-time, external code can invoke any operation freely on such escaping instances circumventing access checks. Additionally, MODGUARD does not detect violations in side-effects occurring in native code. As discussed in subsection 4.4, MODGUARD currently has only limited support for these features.

**MODGUARD effectively finds confidentiality and integrity violations of sensitive entities, unless Java’s dynamic-language API is used.**

### 5.2 RQ2: Confidentiality and Integrity Violations in Real-World Application

To assess MODGUARD on a real-world application, we applied it to two different modularizations of Apache Tomcat 8.5.21. Since code using modules is still scarce in October 2018 as Java’s modularity feature has only been introduced recently, we ourselves modularized Tomcat to Java modules to assess MODGUARD. In fact, we searched Maven Central for artifacts containing a module-info.class, which yield 215 different artifacts. Only 83 of the 215 artifacts actually include a module-info file in their source-jar, all other artifacts accidentally include a module-info.class due to rebundling a beta version of the library org.slf4j. These 83 artifacts export all packages publicly and do not contain any internal packages, which would render the analysis pointless. Since no client code using these artifacts exists, we were unable to construct reasonable a modularization for these artifacts, simply because we do not know which of the packages should be internal. In contrast, Tomcat consists of multiple modules that depend on each other, and thus we know, at least, which packages each module must export to compile Tomcat successfully – allowing us to create reasonable modules.

To assess MODGUARD on Tomcat, we first modularize Tomcat using the default JDK tool jdeps, resulting in a na¨ıve modularization, which exports all packages publicly. Thereby, we check how many sensitive entities Tomcat exposes using Java’s default visibility restrictions. Second, we modularize Tomcat by restricting the na¨ıve modularization, retaining exactly those export statements which Tomcat requires to compile, resulting in a strict modularization.

MODGUARD identifies integrity and confidentiality violations of sensitive entities in both modularizations despite the reduced exports in the strict version. Comparing the number of integrity and confidentiality violations in Table 2 shows that the module system’s encapsulation of internal types can limit violations to a small extent only. To reduce violations effectively, data flow analyses must complement the module system identifying data leaks or manipulations.

#### 5.2.1 Setup - Tomcat Modularization

To modularize Tomcat, we chose to follow the approach presented by Corwin et al. [5]. Thereby, we created for each of Tomcat’s 26 JAR files a separate module, maintaining the original grouping of classes into logical units.

Figure 6 shows an excerpt of the resulting module graph. In the strict modularization: the modules on top
of the graph (in white) catalina.ant, catalina.-storeconfig, jasper, tomcat.dbcp, and tomcat.-websocket do not export any packages, the modules (marked with a dashed line) tomcat.api, tomcat.-jni, tomcat.juli, and tomcat.util.scan remain unchanged, and the exports of the remaining modules (marked in gray) are reduced to the minimum necessary to compile. Note that the modules on the top of the module graph do not export any packages since Tomcat does not contain any client code using them. Thus, these modules cannot be invoked externally - they are dead code. While this modularization is more strict than a regular one, we do not run the risks of exposing internal types unnecessarily.

Figure 6: Tomcat 8.5.21’s Module Graph. In the strict version, white modules do not export any packages. Dashed modules do not differ in the naïve and strict version.

To determine sensitive methods in Tomcat, which MOD-GUARD expects as user input, we used the machine-learning framework SuSi [17], an existing and established tool to determine sources and sinks in Java and Android binaries. We trained SuSi on methods of JDK 9 that throw SecurityExceptions, e.g., methods accessing or modifying CLASSLOADERS or the filesystem. The features implemented by these methods are guarded by permissions checks, triggered by calls to Java’s SecurityManager, which investigate the call stack to check whether all calling classes possess the required permissions. These methods are considered security-sensitive, and their set of calling classes is restricted. Thus, we use these methods as the training set for SuSi - which should identify similar security-sensitive methods in Tomcat to which the access should be restricted, too.

Using this training set, SuSi reported 3,300 sensitive methods in 12 Tomcat modules. We further included 90 classes and 25 fields as sensitive that are declared as internal in their JavaDoc comments. Note that access to these sensitive entities does not necessarily imply the existence of security vulnerabilities, yet it indicates privileged classes, fields, and methods whose caller may be limited, similar to I/O-methods in the JDK. Developers should investigate these warnings and revise the design and implementation such that these sensitive entities do not leak.

5.2.2 Integrity & Confidentiality Violations in Tomcat

Table 2 shows that even with the strict modularization, Tomcat allows data flows resulting in integrity or confidentiality violations for thousands of sensitive entities. Comparing the naïve and strict version shows that within the strict modularization the number of violations of sensitive entities does not change or is only slightly reduced. Since modules cross-reference types between each other the reported violations of one module intersect with the reported violations of other modules. For instance, method violations in an exported supertype in one module are reported for all dependent modules that declare escaping subtypes. Thus, removing the export of such supertype also reduces the violations in dependent modules. In fact, the majority of data flows violating confidentiality occur due to sensitive types that are declared in exported packages whose export cannot be removed without compilation errors.

Table 2: Violations in Tomcat. Modules ⊙ declare no exports, modules ⊖ do not differ in the strict and naïve version.

<table>
<thead>
<tr>
<th>Tomcat Module</th>
<th>#Violations strict / Δ naïve version</th>
<th>Run-Time (min.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods</td>
<td>Fields</td>
<td>Classes</td>
</tr>
<tr>
<td>catalina</td>
<td>2556/−236</td>
<td>8/−21</td>
</tr>
<tr>
<td>catalina.ant⊝</td>
<td>0/−</td>
<td>0/−</td>
</tr>
<tr>
<td>catalina.juli</td>
<td>1081/−391</td>
<td>3/−1</td>
</tr>
<tr>
<td>catalina.storeconfig⊝</td>
<td>0/−−90</td>
<td>0/−−3</td>
</tr>
<tr>
<td>catalina.tribes⊝</td>
<td>0/−−6</td>
<td>0/−−6</td>
</tr>
<tr>
<td>jasper⊝</td>
<td>2020/−294</td>
<td>0/−3</td>
</tr>
<tr>
<td>tomcat.dbcp⊝</td>
<td>0/−</td>
<td>0/−−6</td>
</tr>
<tr>
<td>tomcat.juli⊝</td>
<td>1/−</td>
<td>2/−</td>
</tr>
<tr>
<td>tomcat.util⊝</td>
<td>78/−</td>
<td>0/−−6</td>
</tr>
<tr>
<td>tomcat.util.scan⊝</td>
<td>449/−</td>
<td>0/−−6</td>
</tr>
<tr>
<td>tomcat.websocket⊝</td>
<td>0/−−6</td>
<td>0/−−6</td>
</tr>
</tbody>
</table>

Conversely, the violations remain unchanged in the 3 modules tomcat.api, tomcat.juli, and tomcat.-util.scan since their export statements remain unchanged. For the module tomcat.util the violations remain unchanged since the removed package exports do not contain any sensitive entity.

The strict modularization influences violations in the modules catalina, catalina.ha, catalina.tribes and tomcat.coyote only, excluding modules whose exports are removed completely or are unchanged.

In the module catalina, 18 export statements out of 30 are removed. However, the encapsulation of internal types only reduces violations by a small extent. Instead, violations still occur in packages whose exports are removed. For instance, the removed export of package org.apache.-catalina.webresources contains 140 internal, sensitive methods. 32 of these methods are still accessible in this strict modularization since WebResource, which declares the sensitive methods, is contained in the package org.-apache.catalina whose exports cannot be removed without compilation errors. The module catalina.ha does not benefit much from the stricter encapsulation of internal types either; 70% of its sensitive methods are still leaked and can be invoked from the outside.
Vulnerabilities in OSGi Bundles

Parrend and Frénot [18], [19] study vulnerability patterns in the OSGi platform and OSGi bundles, e.g., modifying a bundle’s private data through its API or shared variables. To detect such vulnerability patterns, Parrend and Frénot utilize points-to analyses in their following work [20], similar to MODGUARD. However, their analysis does detect escaping objects, but aims to detect if objects can be passed from untrusted code into a trusted bundle, thereby risking denial of service attacks when untrusted code is executed, and thus solves a different issue. In the future, we incorporate their vulnerability patterns into our analysis where applicable.

Geoffray et al. [21], [22] introduce I-JVM, a JVM to isolate vulnerable or malicious OSGi bundles from each other. I-JVM executes each bundle in a separate thread containing a private copy of all static variables, strings, and java.lang.Class objects, thereby achieving isolation. Similar, Gamma and Donsez [23] propose to load untrusted OSGi bundles in separate sandboxes to achieve fault isolation. Analogously, Huang et al. [24] introduce an Advanced OSGi Security Layer to prevent malicious bundles from performing security-sensitive operations, e.g., modifying files on disk, or probing the API of other bundles, by inspecting the state of the JVM.

While the approaches isolate bundles, MODGUARD helps to improve the encapsulation within modules themselves. Therefore, MODGUARD statically analyzes module’s entrypoints to detect integrity or confidentiality violations.

Escape Analysis

Several approaches already have applied static analysis to determine whether objects escape a dedicated code region, e.g., if they become accessible outside of a method or a thread, to improve memory allocation [25], [26], [27], e.g., allocating objects on the stack or removing synchronization.

Similar to our analysis, escape analyses usually rely on points-to analyses but operate on a complete code base rather than on standalone modules.

Two of the most related approaches are contributed by Whaley et al. [27] and extended by Viven et al. [26]. They present an abstract inter-procedural points-to & escape analysis based on so-called points-to escape graphs. In an escape graph, nodes represent objects and edges represent references between them. The analysis separates the code under analysis into unanalyzed and analyzed regions and uses the escape graph to record escape paths into unanalyzed regions [26]. The presented approaches do not only cover objects escaping methods but also (static) fields, parameters, exceptions, and callbacks.

A state-of-the-art escape analysis is implemented in the Watson Libraries for Analysis (WALA) [28] framework. The algorithm respects fields, thread constructor parameters, and all objects transitively reachable from fields of escaping objects, but solely focuses on threads.

Current escape analyses check if objects escape methods or threads but do not consider escapes in larger contexts like modules. In addition, they analyze concrete implementations including callers, whereas our analysis, in the absence of such callers, analyzes all potential usage scenarios of a module. Finally, they ignore access restriction in the module.

6 RELATED WORK

As the Java module system has only been recently introduced, existing work on modules focuses solely on OSGi or on information-flow control in distributed systems.
system, and thus would report violations of entities which are actually inaccessible.

**Information-Flow Control**

To cope with unintended data flows several approaches [29], [30], [31], [32], [33] control information-flow using run-time-monitoring, static analysis, or language-based mechanisms.

Sabelfeld and Myers [30] state that visibility constraints and access controls, like in the Java module system, are insufficient to protect confidential data. Thus, they advocate the introduction of security-type systems into programming languages to enable the implementation of static security analyses and to enforce information-flow policies.

Buras et al. [29] introduce the library Hybrid LIO for the programming language Haskell to enforce information-flow policies both statically and at run-time. The authors extend Haskell’s type system to distinguish public and confidential data. Based on the extended type system, the library Hybrid LIO checks statically, and if required at run-time, if confidential data flows into public objects or methods.

In contrast to these approaches, MODGUARD does not introduce a security-type system for modules. Instead, MODGUARD analyzes all potential interactions between modules and checks if they leak or manipulate sensitive entities. Nevertheless, the presented security-type systems are more powerful as they enable to check for non-interference and data flows in distributed systems, e.g., web server or files.

Enck et al. [31] introduce TaintDroid a run-time monitor for Android to limit data flow between Android apps. TaintDroid instruments the Android VM and taints sensitive information on the level of variables, methods, files, and inter-application messages. TaintDroid traces the data flow at run-time and reports violations whenever data flows into a method, variable, or application with a lower confidentiality level.

Yip et al. [33] propose Resin a language run-time to prevent data leaks in web applications. Resin allows developers to specify at application-level data flow assertions, which are then enforced at run-time. Similar, Giffin et al. [32] present a novel web framework that allows the specification of data flow policies for sensitive entities, and enforces them at run-time.

Instead, MODGUARD is a static analysis focusing on single modules and cannot identify violations in distributed systems but supports the design of Java modules.

**7 CONCLUSION**

We have presented a novel static analysis to identify confidentiality and integrity violations of sensitive entities in Java modules. To model the possible usages of a module precisely, we introduced a formal definition of module entrypoints respecting transitively accessible types and methods. This formal definition not just serves our own analysis implementation but may also serve as a basis for future static analyses in the context of Java’s module system, since it specifies which methods or types of a module may become accessible, thereby computing the set of methods that are directly invokeable on any given module.

A case study of integrity and confidentiality violations in Tomcat 8.5.21 showed that using the default JDK tool jdeps for migrating to Java 9 provides modules, without any encapsulation. Yet, it shows that simply restricting modules by limiting export statements, only has a small effect on the number of violations. Even with a few exported types, a significant number of sensitive entities can leak. Hence, if one desires to also limit leakage or manipulation of sensitive entities, one might be forced to refactor the application’s type hierarchy or module boundaries.

Our static code-analysis MODGUARD can be helpful in identifying problematic data flows leading to confidentiality and integrity violations and can thus aid such refactorings. Yet, our study also shows that the Java’s module system still makes it hard for developers to reason about data flows and potential interactions between modules. Although Java module system successfully hides internal types, there exists no means to efficiently reason about or limit data flow between modules. MODGUARD complements the Java module system by analyzing the data flow between modules and pointing out unwanted and potentially security-relevant data flows and interactions.

**References**


