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.Unsafe`[1] 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.

• *The authors are with the Chair for Software Engineering, Heinz Nixdorf Institute, Paderborn University, Germany.*
E-mail: ⟨First Name⟩.⟨Last Name⟩@uni-paderborn.de

Manuscript received July, 2018;

1. The replacement for `sun.misc.Unsafe` in earlier versions.

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.[2]

To assure the soundiness [4] of our approach, we derived and present here a novel formal specification of what we

2. https://github.com/secure-software-engineering/modguard

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,[3] 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 naïve 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.

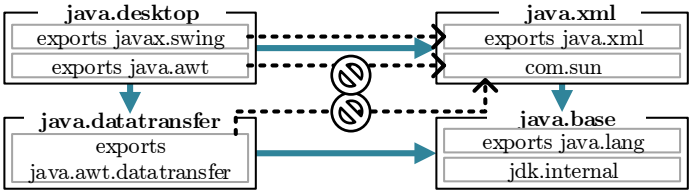

Figure 1: Example Module Graph. Blue Arrows: Module Dependencies; Dashed Arrows: Visibility Relations.

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.

```
1 module my.mod{ exports api; requires java.base; }
2
3 package api;
4 public class KeyProvider {
5  public static Key getKey() {
6    return new SecretKey();}}
7
8 abstract class Key {
9  private byte[] key;
10 protected Key(byte[] key){this.key = key;}
11 public getKey(){return key;} }
12
13 package internal;
14 public class SecretKey extends Key {
15 private static byte[] keyMaterial = {1,2,3,4};
16 public SecretKey() { super(keyMaterial); }}
```

Listing 1 Violation. Green: Exported types and methods. Yellow: Internal types. Red: Sensitive field `keyMaterial`.

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 subsection 2.2 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 [4] "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."

```
1 class Request implements HttpServletRequest{
2  public HttpServletRequest getRequest(){
3   return new RequestFacade(this); } }
4
5 class RequestFacade implements HttpServletRequest
6
7 class FormAuthenticator{
8
9 - if(context.fireRequestInitEvent(request)){
10 + if(context.fireRequestInitEvent(request.
       getRequest())){
11     disp.forward(request.getRequest(), response);
12 - context.fireRequestDestroyEvent(request);
13 + context.fireRequestDestroyEvent(request.
       getRequest());}}}
```

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

4. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5648

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 (`RequestFacade`). 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.getKey()` 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 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].

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:DECLARINGTYPE[$method$:M] = $type$:T
METHOD:MODIFIER($modifier$:$String$, $method$:M)
RETURNVAR(var:V, $method$:M)
CLASSMODIFIER($modifier$:$String$, $class$:T)
SUPERTYPEOF($supertype$:T, $type$:T)
OVERRIDESMETHOD($method$:M, $type$:T, $supertype$:T)
IMPLEMENTSINTERFACE($method$:M, $type$:T, $supertype$:T)
MODULEDECLTYPE($module$:D, $type$:T)
MODULEEXPORTS($fromModule$:D, $package$:$String$, $toModule$:D)
EXPORTEDTYPE($type$:T)
MODULEFORANALYSIS($module$:D)
VARPOINTSTO(var:V, ctx:C, $heap$:H, $hctx$:HC)
FLDPOINTSTO($base$:H, $baseCtx$:HC, fld:F, $heap$:H, ctx:HC)

Output Relations:
EXPLICITMETHOD($class$:T, $method$:M)
IMPLICITMETHOD($class$:T, $method$:M)
ENTRYPOINT($method$:M)
CLASSHASPOSSIBLEENTRYPOINT($class$:T)

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

Domain 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]: RETURNVAR represents a *method's* return variable, METHOD:DECLARINGTYPE represents its declaring type, METHOD:MODIFIER and CLASSMODIFIER represent the modifier of a *method* or *class*, and SUPERTYPEOF represents all *supertypes* of a *type*. The new input relations IMPLEMENTSINTERFACE and OVERRIDESMETHOD represent every *method* of a *type* that implements or overrides a method of an exported *supertype*.

The input relations MODULEDECLTYPE, MODULEEXPORTS, and EXPORTEDTYPE are module specific: MODULEDECLTYPE represents in which unique *module* a *type* is declared, MODULEEXPORTS specifies which module *fromModule* exports which *package* to which other module *toModule*, and EXPORTEDTYPE represents every publicly exported *type*. The input relation MODULEFORANALYSIS represents the module for which the entrypoints should be computed, as a Java project typically consists of a set of modules.

VARPOINTSTO and FLDPOINTSTO encode points-to information: they link a return variable *var* or a field *fld* to an heap object *heap*.

Output Relations The output relations ENTRYPOINT and CLASSHASPOSSIBLEENTRYPOINT encode the computed entrypoint model. The relations EXPLICITMETHOD and IMPLICITMETHOD represent the *explicitly* and *implicitly* reachable methods.

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

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

The rule IMPLICITMETHOD represents all implicit methods; overriding, implementing, or inheriting a supertype's method. To express implicit methods in Datalog, the rule IMPLICITMETHOD 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 set by the rules ENTRYPOINT and METHODRETURNTYPE; requiring that an object of type *class* actually escapes by a return of an previously established *entrypoint*.

The second definition of rule IMPLICITMETHOD states that every *method* declared in an exported *supertype* constitutes an implicitly reachable method, too. Similar to the first definition, the rule constraints this to methods of types *retType* that actually escape through a previously established *entrypoint*, encoded by the METHODRETURNSTYPE(ENTRYPOINT, RETTYPE). Moreover, we limit implicit methods to those whose declaring types *retType* are defined in the module itself (MODULEDECLTYPE(_,RETTYPE), or whose declaring types are directly returned from another entrypoint of the module, encoded by the conjunction METHOD:DECLARINGTYPE and MODULEDECLTYPE.

Since both rules EXPLICITMETHOD and IMPLICITMETHOD recursively refer to the main rule ENTRYPOINT, 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 METHODRETURNSTYPE 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 static

fields of escaping objects. The rule is defined analogously, and thus omitted for brevity.

```
ClassHasPossibleEntryPoint(class),
EntryPoint(method) ←
  ExplicitMethod(class, method).

ClassHasPossibleEntryPoint(class),
EntryPoint(method) ←
  ImplicitMethod(class, method).

ExplicitMethod(class, method) ←
  Method:DeclaringType][method] = class,
  ClassModifier("public", class),
  ModuleForAnalysis(module),
  ModuleDeclType(module, class),
  ExportedType(class),
  (Method:Modifier("public", method);
  (Method:Modifier("protected", method),
  Method:Modifier("static", method))).

ImplicitMethod(class, method) ←
  Method:DeclaringType][method] = class,
  ModuleForAnalysis(module),
  ModuleDeclType(module, class),
  Method:Modifier("public", method),
  (OverridesMethod(method, class, supertype);
  ImplementsInterface(method, class, supertype)),
  ExportedType(supertype),
  EntryPoint(entrypoint),
  MethodReturnsType(entrypoint, class).

ImplicitMethod(supertype, method) ←
  SupertypeOf(supertype, retType),
  ExportedType(supertype),
  Method:DeclaringType[method] = supertype,
  Method:Modifier("public", method),
  ClassModifier("public", supertype),
  EntryPoint(entrypoint),
  ModuleForAnalysis(module),
  (ModuleDeclType(module, retType);
  (Method:DeclaringType[entrypoint] = classOfEntryPoint,
  ModuleDeclType(module, classOfEntryPoint)),
  MethodReturnsType(entrypoint, retType).

MethodReturnsType(method, type) ←
  ReturnVar(var, method),
  VarPointsTo(_, heap, _, var),
  Value:Type[heap] = type.
```

Figure 3: Datalog rules for module entrypoints.

As we have provided a formal definition of entrypoints, we will describe next how we use Doop to turn this definition into an actual static analysis tool. This tool ModGuard goes beyond merely computing entrypoints: it also computes all potential data flows between modules. Therefore, ModGuard checks for of all user-defined sensitive classes, methods, and fields if they may escape the module through invocations of its entrypoints, or may be manipulated by such calls.

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 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.

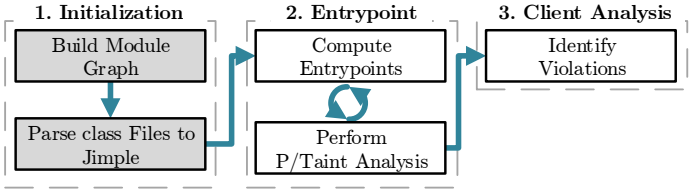

Figure 4: Overview of ModGuard. Gray: Steps in Soot. White: Steps in Doop.

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

entrypoints' 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 SENSITIVEFIELDPOINTSTO.

The remaining input relations represent the points-to sets of further escaping objects. VISIBLEFIELDPOINTSTO 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. Similarly, 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.

Client Analysis Input Relations:

SENSITIVEFIELD($field$:F)
SENSITIVEMETHOD($identifier$:M)
SENSITIVECLASS($class$:T)

SENSITIVEFIELDPOINTSTO(f : $field$, $heap$:H, ctx:HC)
VISIBLEFIELDPOINTSTO(f : $field$, $heap$:H, ctx:HC)
RETURNVARPOINTSTO(var : :V, $type$:T, $heap$:H, ctx:HC)
MOCKOBJECTPOINTSTO($type$:T, $heap$:H)
ARGUMENTOFCALLBACKPOINTSTO($type$:T, $heap$:H)

CHECKVIOLATIONFIELD($field$, $fieldValue$) ←
  SENSITIVEFIELDPOINTSTO($field$, $fieldValue$, ctx),
  VISIBLEFIELDPOINTSTO($field$, $fieldValue$, ctx).

CHECKVIOLATIONFIELD($field$, $fieldValue$) ←
  SENSITIVEFIELDPOINTSTO($field$, $fieldValue$, ctx),
  RETURNVARPOINTSTO(var, _, $fieldValue$, ctx).

CHECKVIOLATIONFIELD($field$, $fieldValue$) ←
  SENSITIVEFIELDPOINTSTO($field$, $fieldValue$, _),
  MOCKOBJECTPOINTSTO(_, $fieldValue$).

CHECKVIOLATIONFIELD($field$, $fieldValue$) ←
  SENSITIVEFIELDPOINTSTO($field$, $fieldValue$, _),
  ARGUMENTOFCALLBACKPOINTSTO(_, $fieldValue$).

Figure 5: MODGUARD Client Analysis: Input Relations & Queries for Checking Field Violations.

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 SecrectKey()`. 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 supertype `Key`. Third, MODGUARD binds the created mock-object `new SecrectKey()` 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 sets 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 SecrectKey()`, 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 and compute points-to information 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 Sălcianu 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 Are integrity and confidentiality violations introduced 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 (**m**odule, **i**ntegrity, **c**onfidentiality 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

	Test Scenario	Result
Accessible Field	Integrity/Confidentiality primitive field	✓
	Integrity/Confidentiality non-primitive field	✓
	Integrity/Confidentiality field array, collection	✓
	Getter/Setter for field	✓
Invokable Method	Access to explicit method	✓
	Access to implicit interface/ abstract method	✓
Parameter	Entity added to parameter array, collection	✓
	Static method returns internal field	✓
Callback	Entity/Class referencing Entity as argument	✓
Exception	Declared/Undeclared exception	✓
Reflection & Invoke API	Referenced by `VarHandle`/ `MethodHandle`	✗
	Access to privileged `MethodHandles.Lookup`	✗
	Return field via reflection	✓
Side-Effect	Pass entity to native code	✗

✓true positives, ✗false negatives; no false positives observed; Intel i7 2.60GHz, ∅ per module: Runtime 22.4 min,RAM 3.4 GB (incl. JDK); excl. Doop Reflection: 7.4 min, 3.1 GB

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 detects 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 *naïve* 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 *naïve* 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.

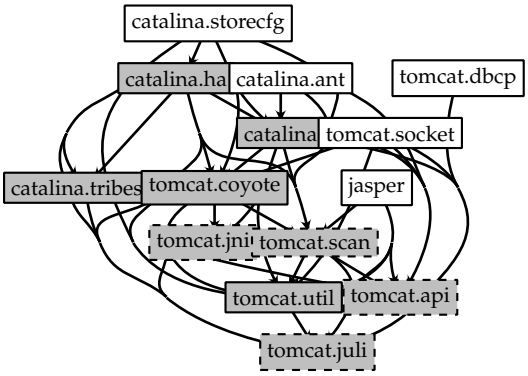

Figure 6: Tomcat 8.5.21's Module Graph. In the strict version, white modules do not export any packages. Dashed modules do not differ in the naïve and strict version.

To determine sensitive methods in Tomcat, which MODGUARD 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 the 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.

Tomcat Module	#Violations strict / Δ naïve version			Run-Time (min.)
	Methods	Fields	Classes	
catalina	2556/−236	8/−21	31/−17	11:13
catalina.ant⊖	0/–	0/–	0/−7	00:43
catalina.ha	1081/−391	3/−1	15/–	03:48
catalina.storeconfig⊖	0/−79	0/–	0/–	01:49
catalina.tribes	0/–	3/–	4/−1	01:13
jasper⊖	0/−6	0/–	0/–	01:30
tomcat.coyote	2020/−294	0/−3	20/−14	01:25
tomcat.dbcp⊖	0/–	0/−3	0/–	00:44
tomcat.jni⊙	1/–	2/–	0/–	00:38
tomcat.util	78/–	0/–	1/–	00:39
tomcat.util.scan⊙	449/–	0/–	0/–	00:44
tomcat.websocket⊖	0/–	0/–	0/−6	00:46

Table 2 shows that in the strict modularization, no violations occur in the 5 modules `catalina.ant`, `catalina.storeconfig`, `jasper`, `tomcat.dbcp`, and `tomcat.websocket`, since they are at the top of the module graph (cf. Figure 6) all their export statements are deleted, and thus no data flows exist, which could leak sensitive entities.

Conversely, the violations remain unchanged in the 3 modules `tomcat.api`, `tomcat.jni`, 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.

An instance of a data flow violating confidentiality of the sensitive methods of class `StandardSession` in module

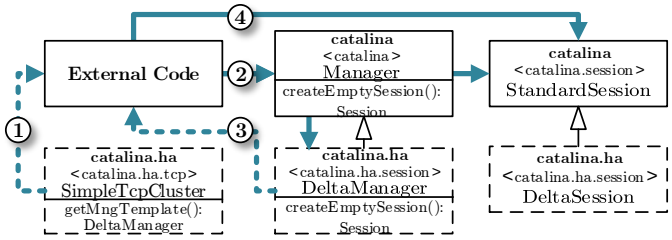

Figure 7: Example Violation in Module `catalina.ha`. Types marked with a dashed line are internal.

`catalina` is depicted in Figure 7. The violation occurs in the strict version, as follows: First, external code acquires an instance of `DeltaManager`, returned from the public method `getManagerTemplate()`. Second, external code invokes the implicit method `createEmptySession` on the acquired instance, which is overridden in `DeltaManager` from its exported supertype `Manager`. Third, the method `createEmptySession` returns an instance of `DeltaSession` to external code. Fourth, on the such acquired instance of `DeltaSession` external code can invoke all overridden sensitive methods of the supertype `StandardSession` (exported by module `catalina`). This exposes the internal, sensitive methods of `DeltaSession` in `catalina.ha` to external code.

> Integrity and confidentiality violations are indeed a problem when modularizing real-world applications. MODGUARD identifies data flows leading to violations, and thus, helps developers to assess how successfully a module confines sensitive entities.

In result, Table 2 shows that neither the naïve nor the strict modularization of Tomcat effectively limits that sensitive entities escape through complex (unintended) data flows. While an automatic migration using `jdeps` is possible without major effort, the resulting modules do not benefit from the module system security-wise. Although internal types are encapsulated, modularizing w.r.t. forbidden data flows can prevent data leaks, and thus results in a security benefit. To limit violations, it is insufficient to simply limit package exports. Even modules whose exports we were able to be reduce (`catalina.ha`, `catalina.tribes`, and `tomcat.coyote`), effectively confine a small subset of sensitive entities only. Thus, applications must instead be migrated with care to prevent data flows leading to integrity and confidentiality violations. For this migration, developers should be supported by appropriate tools. Our case study shows that MODGUARD can support the migration to Java modules, as well as refactorings, by revealing integrity or confidentiality violations.

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.

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 will 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. Similarly, 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 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.

Burias 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 *Resion* 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. Similarly, 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 invokable 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 module system still makes it hard for developers to reason about data flows and potential interactions between modules. Although the 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

[1] Oracle Corporation, "JEP 200: The Modular JDK," https://openjdk.java.net/jeps/200. [Online]. Available: https://openjdk.java.net/jeps/200

[2] ——. (2015) Jep 260: Encapsulate most internal apis. http://openjdk.java.net/jeps/260.

[3] Y. Smaragdakis and M. Bravenboer, "Using datalog for fast and easy program analysis," in *Proceedings of the First International Conference on Datalog Reloaded*, ser. Datalog'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 245–251. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24206-9_14

[4] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: A manifesto," *Communications of the ACM*, vol. 58, no. 2, pp. 44–46, 2015. [Online]. Available: http://doi.acm.org/10.1145/2644805

[5] J. Corwin, D. F. Bacon, D. Grove, and C. Murthy, "MJ: a rational module system for Java and its applications," in *OOPSLA '03 Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, vol. 38, no. 11. ACM, 2003, pp. 241–254. [Online]. Available: http://doi.acm.org/10.1145/949343.9493

[6] Oracle Corporation, "The Java Language Specification Java SE 9 Edition," Oracle Corporation, Tech. Rep., 2017. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se9/jls9.pdf

[7] ——. (2014) JEP 261: Module System. http://openjdk.java.net/jeps/261.

[8] M. Reif, M. Eichberg, B. Hermann, J. Lerch, and M. Mezini, "Call graph construction for java libraries," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 474–486. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950312

[9] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, "Introspective analysis: Context-sensitivity, across the board," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 485–495. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594320

[10] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: Understanding object-sensitivity," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: ACM, 2011, pp. 17–30. [Online]. Available: http://doi.acm.org/10.1145/1926385.1926390

[11] P. Lam, E. Bodden, O. Lhotak, L. Hendren, O. Lhoták, and L. Hendren, "The Soot framework for Java program analysis: a retrospective," in *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, no. 15, Galveston Island, TX, oct 2011, pp. 35–42.
[12] N. Grech and Y. Smaragdakis, "P/taint: Unified points-to and taint analysis," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 102:1–102:28, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133926
[13] Y. Smaragdakis, G. Balatsouras, G. Kastrinis, and M. Bravenboer, "More Sound Static Handling of Java Reflection," in *Programming Languages and Systems: 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*, X. Feng and S. Park, Eds. Cham: Springer International Publishing, 2015, pp. 485–503. [Online]. Available: https://doi.org/10.1007/978-3-319-26529-2{_}26
[14] A. Sălcianu and M. Rinard, "Purity and Side Effect Analysis for Java Programs." Springer, Berlin, Heidelberg, 2005, pp. 199–215. [Online]. Available: http://link.springer.com/10.1007/978-3-540-30579-8{_}14
[15] J. Whaley and M. Rinard, "Compositional pointer and escape analysis for java programs," in *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '99. New York, NY, USA: ACM, 1999, pp. 187–206. [Online]. Available: http://doi.acm.org/10.1145/320384.320400
[16] J. Späth, L. N. Q. Do, K. Ali, and E. Bodden, "Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Krishnamurthi and B. S. Lerner, Eds., vol. 56. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 22:1–22:26.
[17] S. Rasthofer, S. Arzt, and E. Bodden, "A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks," *Proceedings 2014 Network and Distributed System Security Symposium*, no. February, pp. 23–26, 2014. [Online]. Available: http://www.internetsociety.org/doc/machine-learning-approach-classifying-and-categorizing-android-sources-and-sinks
[18] P. Parrend and S. Frénot, "Supporting the secure deployment of osgi bundles," in *2007 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, June 2007, pp. 1–6.
[19] ——, "Security benchmarks of osgi platforms: toward hardened osgi," *Software: Practice and Experience*, vol. 39, no. 5, pp. 471–499, 2009. [Online]. Available: http://dx.doi.org/10.1002/spe.906
[20] F. Goichon, G. Salagnac, P. Parrend, and S. Frénot, "Static vulnerability detection in java service-oriented components," *J. Comput. Virol.*, vol. 9, no. 1, pp. 15–26, Feb. 2013. [Online]. Available: http://dx.doi.org/10.1007/s11416-012-0172-1
[21] N. Geoffray, G. Thomas, B. Folliot, and C. Clément, "Towards a new isolation abstraction for osgi," in *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, ser. IIES '08. New York, NY, USA: ACM, 2008, pp. 41–45. [Online]. Available: http://doi.acm.org/10.1145/1435458.1435466
[22] N. Geoffray, G. Thomas, G. Muller, P. Parrend, S. Frénot, and B. Folliot, "I-jvm: a java virtual machine for component isolation in osgi," in *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, June 2009, pp. 544–553.
[23] K. Gama and D. Donsez, *Towards Dynamic Component Isolation in a Service Oriented Platform*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 104–120. [Online]. Available: https://doi.org/10.1007/978-3-642-02414-6{_}7
[24] C. C. Huang, P. C. Wang, and T. W. Hou, "Advanced osgi security layer," in *Advanced Information Networking and Applications Workshops, 2007, AINAW '07. 21st International Conference on*, vol. 2, May 2007, pp. 518–523.
[25] D. Gay and B. Steensgaard, "Fast escape analysis and stack allocation for object-based programs," in *Proceedings of the 9th International Conference on Compiler Construction*, ser. CC '00. London, UK, UK: Springer-Verlag, 2000, pp. 82–93.
[26] F. Vivien and M. Rinard, "Incrementalized pointer and escape analysis," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, ser. PLDI '01. New York, NY, USA: ACM, 2001, pp. 35–46. [Online]. Available: http://doi.acm.org/10.1145/378795.378804
[27] J. Whaley and M. Rinard, "Compositional pointer and escape analysis for java programs," in *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '99. New York, NY, USA: ACM, 1999, pp. 187–206. [Online]. Available: http://doi.acm.org/10.1145/320384.320400
[28] IBM T.J. Watson Research Center, "Watson Libraries for Analysis (WALA)," http://wala.sourceforge.net/wiki/index.php, 2006.
[29] P. Buiras, D. Vytiniotis, and A. Russo, "Hlio: Mixing static and dynamic typing for information-flow control in haskell," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP 2015. New York, NY, USA: ACM, 2015, pp. 289–301. [Online]. Available: http://doi.acm.org/10.1145/2784731.2784758
[30] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J.Sel. A. Commun.*, vol. 21, no. 1, pp. 5–19, Sep. 2006. [Online]. Available: https://doi.org/10.1109/JSAC.2002.806121
[31] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 393–407. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924971
[32] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo, "Hails: Protecting data privacy in untrusted web applications," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 47–60.
[33] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Improving application security with data flow assertions," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 291–304. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629604

Andreas Dann is a Ph.D. student in the secure software engineering research group at Paderborn University. Andreas Dann's current research interests are the secure design of software leveraging Java 9 's module systems, as well as the detection of security vulnerabilities in open-source dependencies using static analysis.

Ben Hermann is a postdoctoral researcher at Paderborn University and the Heinz-Nixdorf-Insitute. His research focuses on static analysis and software security. He worked on several static analysis frameworks including Soot and OPAL and has significant experience in engineering these frameworks and the analyses build on top of them. He received his doctorate degree from the University of Darmstadt for his work on Java security.

Eric Bodden is a full professor for Secure Software Engineering at the Heinz Nixdorf Institute of Paderborn University, Germany. He is further the director for Software Engineering at the Fraunhofer Institute for Engineering Mechatronic Systems. Prof. Bodden has been recognized several times for his research on program analysis and software security, most notably with the German IT-Security Price and the Heinz Maier-Leibnitz Price of the German Research Foundation, as well as with several distinguished paper and distinguished reviewer awards.