ModGuard: Identifying Integrity & Confidentiality Violations in Java Modules (Short Summary)

Andreas Dann[1] Ben Hermann[1] Eric Bodden[1,2]

Abstract: This short paper[3] presents a static analysis for the novel challenge of analyzing Java modules. Since modules have only been recently introduced with Java 9, we point out the impact of modules both from the security and the static code analysis perspective. In particular, we introduce a static analysis that allows developers to assess if a module successfully encapsulates internal data, along with a formal definition of a module's entrypoints.

Keywords: Static Code Analysis; Module System; Java 9

1 Overview

With the release of version 9, Java introduced the module system Jigsaw. It enables developers to explicitly declare which packages and types are exposed and which are internal [Or15].

Although modules can encapsulate internal types, they do not prevent the unintentional leak of security-sensitive data, e.g., secret keys, giving rise to integrity and confidentiality violations. Confining data by leveraging the module system requires reasoning about data flows between modules and which classes, methods, and fields are actually accessible from outside the module.

To complement Java's module system with means to identify unintended data leaks, we present ModGuard[4], a novel static analysis to identify escaping instances, fields, or methods. Further, we clarify if existing applications may benefit from the guarantees provided by the module system by conducting a case study on Apache Tomcat.

2 What are modules?

Like "traditional" JAR files, modules assemble related packages, native code, and resources. Additionally, modules further contain a static module descriptor file (`module-info.java`). A descriptor file specifies the module's unique name, the exported packages, and the other modules it requires.

Up to Java 8, every public class was visible to any other on the classpath. In Java 9, a class contained in a module (*java.desktop*) may only access another module's (*java.xml*) class if it requires that module, and the module exports the package [Or14] (cf. Figure 1).

[1] Heinz Nixdorf Institute, Paderborn University, Germany <firstname>.<lastname>@uni-paderborn.de
[2] Fraunhofer IEM, Germany
[3] The full-paper is available online [DHB19]
[4] `https://github.com/secure-software-engineering/modguard`

doi:10.18420/SE2021_04

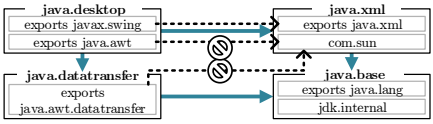

Fig. 1: Blue: Module Dep.; Dashed: Exported Pkg.

But *crucially instances* of internal types can still escape their module. Further, the methods and fields inherited from exported supertypes can be invoked, e.g., $java.desktop$ may invoke exported methods on instances escaping $com.sun$.

3 How to identify data leaks & escaping instances?

Identifying unintended data flows using static analyses on individual modules is challenging. The analysis must be conducted on open code much alike call-graph construction for libraries [Re16]: a module can be linked to any other, and the analysis must foresee all ways in which those other modules may invoke it.

To cope with this challenge, we define all potential interactions with a module in the form of a so-called entrypoint model using Datalog-based analysis rules extending Doop [SB11; SKB14]. The model distinguishes between *explicit* and *implicitly* entrypoints. Explicit entrypoints are methods whose declaring type is exported and can be invoked directly. Also, they may grant access to the so-called *implicit* entrypoints. Implicit entrypoints are methods that inherit, implement, or override methods of exported supertypes but are declared by an internal type whose instances may escape.

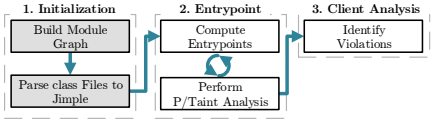

Fig. 2: ModGuard's Analysis Steps.

Based on the entrypoint model, we designed the analysis ModGuard (cf. Figure 2). After computing the entrypoints, ModGuard checks which fields, returned values, and classes became accessible as a result of invoking of the entrypoints by computing their points-to set. To identify violations, ModGuard intersects the points-to set with the point-to set of security-sensitive types and fields. If the intersection is non-empty, ModGuard reports a violation.

4 Can Modules help to confine data?

To clarify if applications may benefit from the guarantees provides by the module system, we exemplary studied Apache Tomcat 8.5.21. Since Tomcat not yet uses modules, we migrated every JAR to a module, following Corwin et al. [Co03]. Our case study shows that such a naïve migration fails to mitigate confidentiality and integrity violations, as ModGuard found violations in 12 out of 26 Tomcat modules.

5 Conclusion

ModGuard may help developers to leverage the module system security-wise by identifying the exposure of security-sensitive data or objects. The Apache Tomcat example shows that to confine sensitive data successfully, developers must introduce modules with care.

References

[Co03] Corwin, J.; Bacon, D. F.; Grove, D.; Murthy, C.: 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, ACM, pp. 241–254, 2003, ISBN: 1581137125, URL: http://doi.acm.org/10.1145/949343.9493.

[DHB19] Dann, A.; Hermann, B.; Bodden, E.: ModGuard: Identifying Integrity Confidentiality Violations in Java Modules. IEEE Transactions on Software Engineering/, pp. 1–1, 2019, URL: http://dx.doi.org/10.1109/TSE.2019.2931331.

[Or14] Oracle Corporation: JEP 261: Module System, 2014, URL: http://openjdk.java.net/jeps/261.

[Or15] Oracle Corporation: JEP 260: Encapsulate Most Internal APIs, 2015, URL: http://openjdk.java.net/jeps/260.

[Re16] Reif, M.; Eichberg, M.; Hermann, B.; Lerch, J.; Mezini, M.: Call Graph Construction for Java Libraries. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE 2016, ACM, Seattle, WA, USA, pp. 474–486, 2016, ISBN: 978-1-4503-4218-6, URL: http://doi.acm.org/10.1145/2950290.2950312.

[SB11] Smaragdakis, Y.; Bravenboer, M.: Using Datalog for Fast and Easy Program Analysis. In: Proceedings of the First International Conference on Datalog Reloaded. Datalog'10, Springer-Verlag, Oxford, UK, pp. 245–251, 2011, ISBN: 978-3-642-24205-2, URL: http://dx.doi.org/10.1007/978-3-642-24206-9_14.

[SKB14] Smaragdakis, Y.; Kastrinis, G.; Balatsouras, G.: Introspective Analysis: Context-sensitivity, Across the Board. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '14, ACM, Edinburgh, United Kingdom, pp. 485–495, 2014, ISBN: 978-1-4503-2784-8, URL: http://doi.acm.org/10.1145/2594291.2594320.