Is There Value in Reasoning about Security at the Architectural Level: a Comparative Evaluation

Ebrahim Khalaj Radu Vanciu Marwan Abi-Antoun
Department of Computer Science, Wayne State University
{mekhalaj, radu, mabiantoun}@wayne.edu

ABSTRACT

We propose to build a benchmark with hand-selected testcases from different equivalence classes, then to directly compare different approaches that make different tradeoffs to better understand which approaches find security vulnerabilities more effectively (better recall, better precision).

1. INTRODUCTION

For more than a decade, Architectural Risk Analysis (ARA) has been recognized as one of the three key pillars of securing software, together with code review and penetration testing. The goal of ARA is to look for architectural flaws, such as the disclosure of sensitive information, or the bypassing of authentication components, rather than implementation-level defects or coding bugs such as hard-coded passwords. Unlike finding coding bugs based a code-oriented view of the system, finding architectural flaws tends to require reasoning based on an architectural, high-level view of the system.

There is no shortage of static and dynamic program analysis tools focused on coding bugs. ARA, however, is much less mature and lacks compelling tool support. Organizations are spending tens of millions of dollars on penetration testing, code review and the tools associated with them. Yet, organizations are still mostly finding the same issues year after year, and missing whole classes of architectural flaws. Some of those architectural flaws end up being more serious and more important than the coding bugs that the current tools are finding. Since coding bugs and architectural flaws are split roughly evenly, it makes sense to focus on tools that support ARA, and to study their value, if any.

In recent work, Vanciu and Abi-Antoun [11] proposed an approach, Scoria, to support ARA. Scoria requires annotations that express design intent, a static analysis to extract a sound, high-level representation of the system's abstract runtime structure, and human-specified constraints on the high-level representation. Broadly speaking, a first characteristic of Scoria is that it promotes reasoning about security at the architectural level, which makes it suitable to be compared to other, more code-oriented approaches. In this ongoing work [5], we propose to better understand the value of reasoning about security at the architectural level by comparing Scoria to other approaches that are more code-oriented, and as a result, may find security vulnerabilities that are closer to coding bugs than architectural flaws.

A second characteristic of Scoria is separating the constraints from the extraction, unlike other static analyses that combine the constraints into the extraction analysis, so a comparison of such tools is likely to lead to additional insights into the kinds of constraints that are needed to find architectural flaws. Moreover, any approach necessarily makes different tradeoffs between soundness, precision, scalability, among others, but the impact on these tradeoffs on the approach's effectiveness in practice, defined in terms of precision and recall, has received little study.

This project aims to make the following contributions:

- An empirical evaluation comparing several approaches for finding injected security vulnerabilities, in terms of their effectiveness;
- An extended, open-source benchmark with new equivalence classes that were not previously considered, on which existing and future approaches can be evaluated.

2. THE SCIENCE

Security science looks for objective methods to compare approaches to each other. One way to do so is by defining a benchmark, namely a suite of test cases from different equivalence classes, and run each approach on the benchmark. In this case, the benchmark must be focused on different equivalence classes of security vulnerabilities.

We propose to answer the following research question:

Research Question: Is an approach that makes one or more of the following tradeoffs more effective overall (better recall, better precision) than another approach that makes a different tradeoff? (a) Sound and possibly less precise vs. Unsound and possibly more precise; (b) Flow-sensitive vs. Flow-insensitive; (c) Whole program vs. Modular; (d) Design intent-based vs. Code-only; (e) Analyst-assisted approach vs. More automated approach; (f) General purpose constraints vs. Special purpose, property-specific constraints; (g) Separate extraction and constraints vs. Combined extraction and constraints; (h) High-level representation of the system vs. Code-oriented view of the system.

There are, of course, other considerations such as the effort involved in learning and applying an approach.

HotSoS '14, April 08 - 09 2014, Raleigh, NC, USA
ACM 978-1-4503-2907-1/14/04
http://dx.doi.org/10.1145/2600176.2600206

Table 1: Some equivalence classes. Some numbers for FlowDroid are reused from the original study [3].

equivalence classes	tests	FlowDroid			Scoria		
		TP	FP	FN	TP	FP	FN
Arrays and Lists	1	0	1	0	0	1	0
Callbacks	5	9	1	0	9	0	0
Field and Object Sensitivity	7	4	0	0	4	3	0
Inter-App Communication	3	2	0	1	3	0	0
Lifecycle	5	5	0	0	5	0	0
General Java	3	2	0	0	2	1	0
Android-Specific	6	3	0	1	4	1	0
Implicit Flows	4	0	0	8	0	0	8
Missing Encryption	3	1	2	0	1	0	0
Bypass Authentication	2	4	2	0	4	0	0
Command Injection	1	1	0	0	1	0	0
Exploitable Service	1	0	1	1	1	0	0
Least Privilege Violation	1	0	0	1	1	0	0

3. METHOD

In this research, we are designing an extended benchmark [1] that includes hand-selected testcases from several benchmarks proposed by other researchers [3, 7], then evaluating several representative approaches from different parts of the solution space. Furthermore, we are augmenting the benchmark with testcases from additional equivalence classes that were not previously considered.

To avoid results that are too specific to a platform such as Android mobile applications, we are picking a variety of examples including web applications, and ones based on general rules for writing secure code such as the ones proposed by CERT [6]. Our goal is to keep the benchmark small, by including small-to-medium testcases that are focused on security and that are from different equivalence classes. This will enable the visual inspection of the results, in order to compute precision and recall.

We are first evaluating Scoria on the testcases. On the same testcases, we are then evaluating several selected approaches. Some of the approaches that we are considering include: (a) static analysis approaches that combine constraints into static analysis, e.g., [3, 4]; (b) type systems, e.g., [8]; and (c) other architectural-level approaches that extract different types of architectural views, e.g., [2, 9].

4. PRELIMINARY RESULTS

We used some initial testcases from DroidBench [3]. DroidBench contains 39 test cases with 36 injected vulnerabilities; of those, we selected 32 testcases that fall into 7 equivalence classes. We also used SAMATE Reference Dataset (SRD) [7], which contains test cases based on the Common Weakness Enumeration (CWE) [10]. We selected CWEs related to architectural flaws then looked for corresponding testcases in SRD. If we found an interesting CWE without a corresponding testcase, we designed a test case based on the CWE. For example CWE-319 is "Cleartext Transmission of Sensitive Information". We designed two testcases based on this CWE and added them to our benchmark. To design more testcases, we explored CERT [6] rules. For example, CERT rule IDS07-J states: do not pass untrusted, unsanitized data to the `Runtime.exec()`. So we designed a testcase with an injected vulnerability by passing unsanitized data to the method. Table 1 shows some of the equivalence classes in the benchmark.

We are comparing tools in terms of: true positives (TP), a real vulnerability that is reported by the tool; false positives (FP), a vulnerability that does not exist but is reported by the tool; and false negatives (FN), a real vulnerability that is missed by the tool. Table 2 shows how Scoria compares to FlowDroid, for example.

Table 2: Recall and precision for FlowDroid and Scoria based on tests from DB, SRD and CERT.

	FlowDroid	**Scoria**
TP, higher is better	31	35
FP, lower is better	6	6
FN, lower is better	12	8
Precision	84%	85%
Recall	72%	81%
F-measure	0.78	0.83

5. FUTURE WORK

Future work will continue evaluating more representative approaches, by adding more columns to the results table. Future work will also involve designing more test cases (adding more rows) focused on architectural flaws and that are insufficiently represented in existing benchmarks. We are encouraging practitioners and researchers to contribute to the benchmark, by evaluating their tools on at least some of the shared, common testcases, or by proposing additional testcases where the existing tools seem to struggle.

Acknowledgments. Funding was provided by the Army Research Office under Award No. W911NF-09-1-0273.

6. REFERENCES

[1] ScoriaBench. `www.cs.wayne.edu/~mabianto/sb`, 2014.

[2] M. Almorsy, J. Grundy, and A. Ibrahim. Automated Software Architecture Security Risk Analysis Using Formalized Signatures. In *ICSE*, 2013.

[3] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *PLDI*, 2014. to appear.

[4] J. Graf, M. Hecker, and M. Mohr. Using JOANA for Information Flow Control in Java Programs – A Practical Guide. In *Working Conference on Programming Languages (ATPS)*, 2013.

[5] E. Khalaj, R. Vanciu, and M. Abi-Antoun. Comparative Evaluation of Static Analyses that Find Security Vulnerabilities. Technical report, WSU, 2014.

[6] F. Long, D. Mohindra, R. C. Seacord, D. F. Sutherland, and D. Svoboda. *The CERT Oracle Secure Coding Standard for Java*. Addison-Wesley, 2011.

[7] National Institute of Standards and Technology. NIST SAMATE Reference Dataset Project. `http://samate.nist.gov/SRD/`, 2013.

[8] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA*, 2008.

[9] K. Sohr and B. Berger. Idea: Towards Architecture-Centric Security Analysis of Software. In *ESSOS*, 2010.

[10] The MITRE Corporation. Common Weakness Enumeration. `https://cwe.mitre.org/`, 2013.

[11] R. Vanciu and M. Abi-Antoun. Finding architectural flaws using constraints. In *ASE*, 2013.