Comparative Evaluation of Architectural and Code-Level Approaches for Finding Security Vulnerabilities

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

ABSTRACT

During architectural risk analysis, Security Information Workers (SIWs) reason about security-relevant architectural flaws using a high-level representation of the system's structure instead of directly reading the code as in during a code review. It is still hard to extract from the code a high-level representation that is sound, conveys design intent, and enables expressive constraints that can find security vulnerabilities. As a result, architecture-level approaches are less mature than code-level ones that extract low-level representations that are not directly intended for use by SIWs.

In this paper, we compare an architecture-level approach with a code-level approach in terms of effectiveness (precision and recall) across test cases with injected vulnerabilities that range from coding bugs to architectural flaws. The evaluation shows that an architecture-level approach can uncover some security vulnerabilities with better precision and recall than a code-level approach. Moreover, it shows that the effectiveness of the approaches varies greatly based on whether the security vulnerability is a coding bug or an architectural flaw. These results may help SIWs select the right tools for the job of securing their systems.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented programming*

1. INTRODUCTION

Security vulnerabilities fall along a continuum ranging from architectural flaws to coding bugs [20]. Coding bugs are defects such as hard-coded passwords and can be found using a variety of techniques including type systems, static analysis, and dynamic analysis, among others. Examples of architectural flaws are the disclosure of sensitive information, or the bypassing of authentication components.

One technique for finding architectural flaws, Architectural Risk Analysis (ARA) has been recognized as one of the three key pillars of securing software, together with code review and penetration testing [20]. In particular, during ARA, rather than looking directly at the code, Security Information Workers (SIWs) study a diagram that represents an abstraction of the system's structure and look for ways in which the system may be vulnerable.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

SIW '14, November 07 2014, Scottsdale, AZ, USA
ACM 978-1-4503-3152-4/14/11/$15.00
http://dx.doi.org/10.1145/2663887.2663905

Finding architectural flaws tends to require a more global, high-level view of the system, and unfortunately has less mature tool support. One of the reasons for this is that it is still is an open problem to extract a high-level view of a system, one that conveys architectural abstraction suitable for SIWs and that is also *sound*. By sound, we mean that the high-level view reflects all possible communication in the system, for any execution.

Broadly speaking, approaches to find security vulnerabilities build some representation of a system, typically a graph, then look for problematic paths through the graph. For example, in taint analysis, a node in the graph is marked as a source, sink or sanitizer, then an analysis checks that there is no path from a source to a sink that does not go through a sanitizer. In this paper, we consider an approach that extracts a low-level graph that is used only internally by the tools to be *code level*. For example, FlowDroid [2] is a code-level approach since it extracts a low-level representation. We consider an approach that extracts a high-level representation intended for SIWs to be *architecture level*. For example, Scoria [30] is an architecture-level approach since it extracts a global, hierarchical, object graph that can be compared to a diagram drawn by a SIW [29].

It is unclear, however, how an architecture-level approach that makes a set of tradeoffs compares to a code-level approach that makes different tradeoffs. In particular, it is unclear if an architecture-level approach identifies security vulnerabilities that are closer to being architectural flaws, or if a code-level approach identifies vulnerabilities that are closer to being coding bugs.

Contributions. This paper contributes the following:

- A comparative evaluation of one architecture-level approach, Scoria [30], and one code-level approach, FlowDroid [2], in terms of effectiveness (precision and recall) at finding injected security vulnerabilities;
- An Architectural Flaw Index (AF-index) to classify security vulnerabilities along a continuum ranging from coding bugs to architectural flaws;
- A carefully curated benchmark with test cases with injected vulnerabilities that we hand-selected from different sources or designed ourselves.

Outline. The rest of this paper is organized as follows. Section 2 provides some background on the tools being evaluated. Section 3 describes our evaluation method. Section 4 presents our preliminary results. Section 5 discusses some lessons learned that are relevant to SIWs. Section 6 gives an overview of the most closely related work.

2. BACKGROUND

Section 2.1 lists some considerations for an architecture-level approach. Section 2.2 discusses the specifics of the Scoria approach. Section 2.3 discusses one of the test cases in the benchmark. Section 2.4 summarizes the key characteristics of the code-level approaches that this paper evaluates.

2.1 Approach Considerations

We identify a number of considerations to support SIWs.

Code- vs. Architecture-level approach. An architecture-level approach is based on a higher-level representation than the code and can help SIWs understand the system better, without reading every single line of code.

The representation approximates a runtime architecture. There are different possible high-level representations of a system such as module views, deployment views, and runtime views. For reasoning about security, a runtime view is often used, focusing on dataflow communication between components.

Separate the constraints from the extraction analysis. Separating the constraint checking (security policy) from the extraction analysis (mechanism) has several advantages such as independent evolution, including changing the constraints or the abstraction without changing the static analysis. The policy is expressed against the high-level representation.

SIW-assisted approach. The SIWs write constraints as logic predicates to check for invariants or unexpected communication paths on the representation to find security vulnerabilities. The constraints can enforce general, platform-specific, system-specific, or domain-specific rules.

General purpose approach. The approach should be general purpose and not be specific to a platform.

Extensibility. Power users are able to define their own properties to set on nodes and edges in the graph, predicates, queries and constraints on the graph to customize the security analysis.

Support for legacy code. The approach should support analyzing legacy code that has high business value, and not require SIWs to rewrite their programs using a radical programming language. For example, the approach could use annotations in the code, or store additional security design intent such as the list of sources and sinks in external files. This also enables using existing development environments.

2.2 Architecture-Level Approach: Scoria

To use Scoria, a SIW works as follows:

1. **Add annotations** to the code to express design intent;
2. **Run the static analysis** to extract a high-level representation in the form of an abstract object graph (Fig. 1) that is guaranteed to be sound and over-approximate any runtime object or relation.
3. **Write constraints** to query the high-level representation and find security vulnerabilities.

Next, we discuss each step in turn.

Annotations enable imposing a hierarchy on the objects.

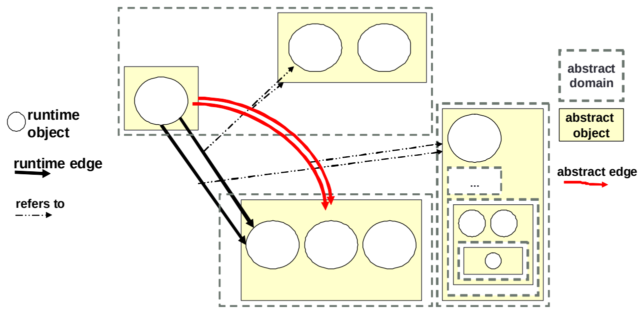

Figure 1: Abstraction of runtime objects and edges into abstract objects and edges in an abstract object graph.

In turn, the constraints can make use of this hierarchy.

- *Express design intent:* the annotations express local, modular hints about architectural tiers, logical containment and strict encapsulation; the SIW can refine the annotations to increase the precision of the extraction analysis, as needed;
- *Support analyzing legacy code:* the SIW adds only annotations to legacy code and any libraries in use, while using existing tools and development environments.

Static analysis. The extraction handles several challenges:

- *Soundness:* the SIW can rest assured that the object graph shows all possible objects and edges that may occur in any possible program run, modulo inevitable unsoundness for any static analysis [18], such as reflection and dynamic code loading;
- *Aliasing:* the SIW does not encounter multiple abstract objects in the object graph that represent the same runtime object, so he cannot assign multiple security properties to the same runtime entity;
- *Precision:* while a sound object graph is inherently going to have false positives, ideally, the SIW does not frequently investigate too many false positive objects or edges that do not exist in any program run;
- *Summarization:* the abstraction is precise if the SIW does not need to distinguish between runtime objects that have the same corresponding abstract object in the object graph;
- *High-level view:* the SIW can visually inspect the hierarchically laid out object graph in order to better understand how to refine the annotations, set the properties, or refine the constraints;
- *Traceability:* the SIW can select a node or an edge on the object graph and go to the relevant lines of code.

Dataflow communication. When reasoning about security, SIWs often track dataflow communication between components. In Scoria, we abstract the dataflow communication as abstract objects rather than types, When dataflow communication edges refer to abstract objects in the object graph, additional expressiveness in the constraints is possible, as discussed below. In this paper, we use *dataflow communication* to mean a particular kind of relation between abstract objects due a method invocation, field read, or field write. We use *information flow* to mean a *data flow* between variables.

Constraints support the following expressiveness:

a) *Object provenance:* check if the same abstract object that flows from a first source to a first destination also flows from a second source to a second destination;

b) *Object transitivity:* track the flow of potentially shared objects through the system;
c) *Object hierarchy:* identify protected data that is nested within abstract objects that ought to carry only unprotected data;
d) *Object reachability:* identify when protected data may be reachable via unexpected communication paths;
e) *Traceability:* by using the traceability between abstract objects and edges and the code, the constraints can use information such as subtyping between classes, method names of interest, or file names, among others;
f) *Indirect communication:* identify when protected data is sent to an abstract object nested within an untrusted sink;
g) *Security property:* enhance the abstract object graph with security design intent that the extraction analysis does not extract from the code.

The details of the object graph augmented with properties and the query language are discussed elsewhere [30]. This rest of this paper evaluates empirically whether this expressiveness enables such constraints to find security vulnerabilities with high precision and recall. Before we do so, we discuss one test case from the benchmark in more detail.

2.3 Sample Test Case: ACryptographic

ACryptographic. If the code does not implement a required step in a cryptographic algorithm that is used to protect confidential data, the implementation of the algorithm is broken. Such an example is architecture-level because there is no information flow from a source to a sink, like in a taint analysis problem.

In Fig. 2, the code[1] is missing a required cryptographic step. The method `missingStep()` (line 3) encrypts a hash input, and uses the object `md:MsgDigest` to implement the encryption algorithm. One of the required steps to implement an encryption algorithm is to call `md.update(...)`. If the implementation does not call the `update(...)` method, there is no data in the hash.

In an architectural-level approach, the goal is to use a high-level representation that makes visually obvious problematic or missing communication. Using an object graph extracted using Scoria, the SIW can notice the missing edge (shown in red) between the object `cs:CryptoStep` and the object `md:MsgDigest`, because the method `missingStep()` declared in the class `CryptoStep` is missing a required step, namely the method invocation `md.update(...)`. The SIW can write a constraint using object provenance to find the missing edge. The constraint checks if there is an edge from `md:MsgDigest` to `cs:CryptoStep`, then there must be an edge from `cs:CryptoStep` to `md:MsgDigest` that represents the method invocation `update(...)`. A Scoria constraint can check that if there is an edge as follows:

$$(\texttt{MsgDigest}, \texttt{cs.OWNED}) \xrightarrow{\texttt{hashInput:String}} (\texttt{CryptoStep}, \texttt{main.LOGIC})$$

then there must be an edge as follows:

$$(\texttt{CryptoStep}, \texttt{main.LOGIC}) \xrightarrow{\texttt{hashInput:String}} (\texttt{MsgDigest}, \texttt{cs.OWNED})$$

Code-level approaches such as FlowDroid often check only for the presence of information flow from sources to sinks.

[1]Adapted from a test case in the SAMATE Reference Dataset [22], and based on one of the Common Weakness Enumerations (CWE-325 [25]).

```
1  @Domains({"OWNED"})
2  class CryptoStep {
3    void missingStep() {
4      @Domain("OWNED") MsgDigest md = new ... ;
5      @Domain("OWNED") IO io = new ...;
6      @Domain("unique") String hashInput = new ...;
7      // Injected vulnerability: missing required step
8      // md.update(hashInput);
9      io.writeLine(io.toHex(md.digest()));
10   }
11 }
12 class IO {
13   String toHex(String str) {... }
14   void writeLine(String line) {...}
15 }
16 class MsgDigest {
17   MsgDigest(String algorithm) { ... }
18   void update(String string) { ... }
19 }
20 @Domains({"LOGIC", "INOUT"})
21 class Main {
22   public static void main(String[] args) {
23     Main m = new Main();
24     m.run();
25   }
26   void run(){
27     @Domain("INOUT") IO io = new ...;
28     @Domain("LOGIC") CryptoStep cs = new ...;
29     cs.missingStep();
30   }
31 }
```

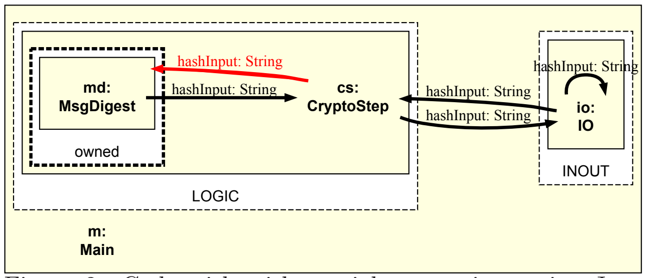

Figure 2: Code with with partial annotations using Java 1.5 annotations. Some defaults are assumed. The method invocation `md.update(hashInput)` is commented out.

If SIWs want to check for the absence of communication, it would be difficult because the graphs used by code-level approaches have too many low-level details. For our example, FlowDroid extracts a graph with more than 40,000 edges (because of the `for` loop in the `toHex` method of `IO`) compared to the 11 edges that Scoria extracts in the underlying graph (not shown). Moreover, there is no any tainted information that flows from a source to a sink. FlowDroid is not intended to be an architecture-level approach, so it cannot report the vulnerability.

2.4 Code-Level Approaches

We evaluate the Scoria architecture-level approach against the following code-level approaches that we discuss next. We compare directly against a state-of-the-art research tool, FlowDroid [2], and indirectly against the commercial tools Fortify [11] and AppScan [12] since Artz et al. compared FlowDroid to those tools on the common test cases. We consider FlowDroid to be state-of-the-art since it was recently published at a prominent venue and other approaches, e.g., [13], are building on FlowDroid and appearing at venues on the state of the art in program analysis.

FlowDroid. FlowDroid reasons about information flow at the level of variables. The static analysis creates an information flow graph which has nodes representing variables and edges representing assignments. Finding a vulnerability means checking that no path from a source to a sink exist. FlowDroid considers aliasing and requires a precomputed call graph, and relies on the existing static analysis framework, SOOT [14].

Fortify. Fortify SCA [11] is a commercial tool that provides several types of analyses. The information flow analysis in Fortify is based on finding a path in the control flow graph from a source (a method that returns confidential information) to a sink (a method that discloses information received as arguments). Fortify uses a set of secure coding rules to find feasible paths from sources to sinks. Another analysis Fortify provides is a control flow analysis in which constraints restrict invocations of methods in a certain order. The control flow analysis can find, for example, if an XML reader is properly configured before it is used.

AppScan. Similarly to Fortify, AppScan Source [12] supports both an information flow analysis and a control flow analysis based on the call graph. AppScan supports customizing the rules to enable developers to extend the possible sources or sinks, or to specify methods that sanitize data or that can propagate confidential data. A constraint has a fixed form, consisting of a source, a sanitizer, and a sink; it then checks that no information flows from a source to a sink without passing through a sanitizer [27].

Both AppScan and Fortify can find security vulnerabilities that are local and can thus be found by analyzing one class or one method at a time. Examples of these vulnerabilities are a hard-coded password or an absolute path in the code. The analysis scales to large systems because it does not consider aliasing or call-graph construction. If only local vulnerabilities are detected, it is likely that unidentified vulnerabilities exist, so SIWs should consider using other approaches that find non-local vulnerabilities, such as Scoria.

All these approaches use a precomputed call graph and rely on existing static analysis frameworks to compute a may-alias relation. FlowDroid uses recent results in aliasing analysis [26] and is object-, field-, and context-sensitive, while AppScan uses WALA [27]. It is unclear which implementation Fortify uses. The static analysis frameworks allow increasing the precision by turning on sensitivity flags. If the approach reports a false positive, SIWs need to understand the inner workings of the static analysis, which is non-trivial. Turning on sensitivity flags may also increase the analysis time and reduce scalability. If SIWs notice a false negative, they may change the list of secure coding rules, but it is a lot harder to change a constraint without changing the static analysis.

2.5 Other Approaches Not Evaluated Here

We discuss briefly other approaches for finding security vulnerabilities that are not yet included in this evaluation. In future work, we hope to broaden the evaluation to include some of these approaches.

Static analysis. SIWs can also use static analysis tools such as JOANA [10]. JOANA is sound but extracts a low-level representation that is not intended for use by SIWs.

Type systems for security. SIWs can also use a type system such as Tainting Checker [5] or Java Information Flow [21]. Some of these type systems also impose an annotation overhead, like Scoria. But the annotations tend to be more complex than the Scoria annotations that simply place objects into groups and are also amenable to type inference.

Approaches at the level of a code architecture. Some architecture-level approaches reverse-engineer diagrams that represent a code architecture. Berger et al. [3] extract an approximation of the runtime architecture for two web applications that is rather a code architecture since it does not distinguish between components of the same type that are used in different contexts.

3. METHOD

We compare Scoria to FlowDroid in terms of precision and recall. To measure precision and recall, we design an extended benchmark where the number of vulnerabilities for each test case is known.

Extended benchmark. In the extended benchmark, we include hand-selected test cases from several benchmarks proposed by other researchers [2, 19, 22], where the test cases are both general Java applications and Android applications. The benchmark is divided into equivalence classes that target the specifics of static analyses such as callbacks, collections, object-sensitivity, inter-application communication, and lifecycle of objects created by the Android framework. Other test cases are related to authentication, encryption, and injection attacks (Table 2). From DroidBench [2], we exclude test cases from the same equivalence class that target the same behavior of a static analysis and test cases that focus on code constructs specific to Java such as anonymous classes or static initialization blocks. From the SAMATE Reference Dataset (SRD) [22], we select two test cases related to misuse of encryption and from CERT Oracle Secure Coding Standard for Java [19], one test case related to command injection and one related to missing encryption. Then, from the list of Common Weaknesses Enumeration (CWE) [25], we propose four test cases related to bypassing authentication, exploitable service and violation of least privilege principle, for which we were unable to find test cases in the investigated benchmarks. The extended benchmark is available [1].

Evaluator. The evaluator, this paper's middle author, received instruction on Scoria. The evaluator then followed the Scoria approach (Section 2.2), by adding annotations, running the extraction analysis and writing constraints. The evaluator did not rerun FlowDroid on the test cases that were reused from the original DroidBench benchmark, but did run FlowDroid on the additional test cases that we selected from other benchmarks or that we designed ourselves.

Measures. We compare the tools in terms of: true positives (TP), a real vulnerability in the test case 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 in the test case that is missed by the tool. A higher TP is better; a lower FP is better; and a lower FN is better. Based on TP, FP, and FN, we compute Precision and Recall for each test case using the standard definitions: Precision = (TP)/(TP+FP) and Recall = (TP)/(TP+FN). For both Precision and Recall, a higher number is better.

Missing measures. Unfortunately, since the evaluator was still learning the approaches, we did not measure more user-oriented metrics such as the effort spent adding annotations, writing constraints, or fine-tuning tool parameters.

Architectural Flaw Index (AF-index). After an initial tabulation of the results, we refined our analysis to classify the vulnerability in the test case along the continuum of coding bug to architectural flaw, to compute precision and recall per category of vulnerability.

We assign to each test case an Architectural Flaw index (AF-index) that attempts to measure how close is a security vulnerability to an architectural flaw as opposed to a coding bug. To compute the AF-index, we assign a weight to each Scoria feature that is used to find the vulnerability in the test case, then compute the weighted sum. We assign higher weights to features such as object provenance and indirect communication (3) since they are more difficult to reason about, and lower weights for features that can be observed in the code such as traceability (1). For the test cases in the extended benchmark, the index ranges from 1 to 10. The higher the index, the more likely is that the vulnerability is an architectural flaw. An AF-index of 0 corresponds to vulnerabilities related to implicit flows, which both FlowDroid and Scoria cannot handle.

4. PRELIMINARY RESULTS

Next, we discuss our precision and recall results to date, first in aggregate, then grouped by equivalence class, then grouped by AF-index.

Precision and Recall in Aggregate. Table 1 (last row) shows that compared to FlowDroid, Scoria has higher recall, precision and a similar F-measure, the weighted harmonic mean of precision and recall. Based on the original test cases from DroidBench, Scoria finds one vulnerability and avoids one false positive of FlowDroid, but reports five additional false positives. The extended benchmark highlights cases where Scoria can do better than FlowDroid, which misses 5 vulnerabilities that Scoria finds, and reports 7 false positives.

Precision and Recall per Equivalence Class. Table 2 presents precision and recall *per equivalence class*: each row shows the equivalence class and the number of test cases in each equivalence class. For each approach in the columns, a row shows the number of true positives (TP), false positives (FP), and false negatives (FN). A more detailed table with the results for each test case is Table 3.

Table 3 has all the test cases used to compare AppScan, Fortify, FlowDroid and Scoria. The test cases are grouped into equivalence classes. The last column in the table indicates the origin of the test case.

The first few columns indicate which features of the Scoria constraints (Section 2.2) were used in the constraints in order to find the vulnerabilities in each test case. As the reader can see, most of the features were used, some (transitivity, provenance, reachability) more than others (hierarchy, traceability, indirect communication). The test cases that seem to involve architectural flaws (not originating from DB) tend to use more of the Scoria features than coding bugs (originating from DB). For AToken1, which is classified in the bypass authentication equivalence class that represents an architectural flaw, Scoria used hierarchy, reachability and indirect communication to find the vulnerabilities. Another example is SecretViewer2 that is categorized in the least privilege violation equivalence class and represents an architectural flaw. Scoria used provenance to find the vulnerability in SecretViewer2. If we used a feature in the constraint, we put a check mark on the corresponding cell of the table. We tried to be precise in selecting the used features. For example, transitivity is a special case of provenance and is implemented using provenance in Scoria constraints; but by convention, we did not select provenance in the test cases that used only transitivity.

After the feature columns, the next two columns indicate the number of security property types that were used for objects and edges in the Scoria constraint of corresponding test case. For example, if we used *TrustLevel* and *IsConfidential*, we put 2 in the corresponding cell. It might be the case that we have set values for these two security property types on multiple objects or edges while writing the constraints.

Next, the two columns before the results display the remaining warnings from the typechecker and the extraction analysis for each test case. Most of the remaining typechecker warnings were caused by static code. The main reason of unresolved extraction warnings was also static code.

Due to space limits, a detailed discussion of the individual test cases are in a companion technical report [1]. For each test case, the technical report shows code snippets with the annotations and the object graphs extracted using Scoria, and discusses how Scoria or FlowDroid suffers from false negatives or false positives.

Precision and Recall per AF-index. Table 1 also groups the test cases per AF-index and compares the precision and recall of FlowDroid and Scoria *per AF-index*. In terms of precision and recall, Scoria is more effective at finding architectural flaws than FlowDroid, which has a better precision for finding vulnerabilities closer to coding bugs. By using reachability and transitivity, FlowDroid can find some vulnerabilities with a high AF-index, but fails to find 5 architectural flaws that require constraints that are more expressive.

The AF-index indicates that most of the test cases in DroidBench are likely closer to coding bugs. The test cases from CERT, SRD, and the ones we designed (US) have vulnerabilities that are closer to architectural flaws. The benchmark should be further extended to allow a more detailed comparison of approaches that find architectural flaws.

5. LESSONS LEARNED

We distill some high-level lessons from our evaluation.

A high-level representation helps a SIW understand the system more than reading the code. Our evaluator relied heavily on the high-level representation. He frequently inspected the hierarchically laid out object graph in order to better understand how to refine the annotations, set the properties, refine the constraints, or investigate the suspicious communication. From studying the object graphs, he was able to identify the sources and sinks to run FlowDroid on the new test cases.

Many tools focus on coding bugs, not enough tools focus on architectural flaws. Many tools, both established, commercial ones and cutting-edge research tools, find security vulnerabilities with good precision and recall in the low AF-index ranges, but suffer from lower precision and recall in the higher AF-index ranges.

Commercial tools and research tools differ widely in terms of precision and recall. SIWs who use only commercial tools may be missing a large number of security vulnerabilities due to their relatively low recall, compared to the research tools. SIWs cannot however be expected to use research tools on their own, since such tools often lack the level of support available to commercial tools. This points to

Table 1: Precision and recall for FlowDroid and Scoria *per AF-index* based on the extended benchmark.

AF-index	**#Test cases**	**FlowDroid** TP	FP	FN	**Scoria** TP	FP	FN	**FlowDroid** Precision	**Scoria** Precision	**FlowDroid** Recall	**Scoria** Recall	**FlowDroid** F-measure	**Scoria** F-measure
0	4	0	0	8	0	0	8			0%	0%		
1	0	0	0	0	0	0	0						
2	26	22	2	0	22	6	0	92%	79%	100%	100%	0.96	0.88
3	5	2	2	1	3	0	0	**50%**	**100%**	**67%**	**100%**	0.57	1.00
4	4	2	0	3	5	0	0	100%	100%	**40%**	**100%**	0.57	1.00
5	1	0	0	1	1	0	0		100%	**0%**	**100%**		1.00
6	0	0	0	0	0	0	0						
7	1	1	1	0	1	0	0	**50%**	**100%**	100%	100%	0.67	1.00
8	0	0	0	0	0	0	0						
9	1	3	0	0	3	0	0	100%	100%	100%	100%	1.00	1.00
10	1	1	2	0	1	0	0	**33%**	**100%**	100%	100%	0.50	1.00
TOTAL	43	31	7	13	36	6	8	82%	86%	**70%**	**82%**	0.76	0.84

Table 2: Precision and recall *per equivalence class* based on the extended benchmark. FlowDroid numbers above the middle line are reused from DroidBench [2]. The ones below the line are ours.

Equivalence class	**#Test cases**	**FlowDroid** **TP**	**FP**	**FN**	**Scoria** **TP**	**FP**	**FN**	**FlowDroid** Precision	**Scoria** Precision	**FlowDroid** Recall	**Scoria** Recall	**FlowDroid** F-measure	**Scoria** F-measure
Arrays and Lists	1	0	1	0	0	1	0	0%	0%				
Callbacks	5	9	1	0	9	0	0	90%	100%	100%	100%	0.95	1.00
Field and Object Sensitivity	7	4	0	0	4	3	0	100%	57%	100%	100%	1.00	0.73
Inter-App Communication	3	2	0	1	3	0	0	100%	100%	67%	100%	0.80	1.00
Lifecycle	5	5	0	0	5	0	0	100%	100%	100%	100%	1.00	1.00
General Java	3	2	0	0	2	1	0	100%	67%	100%	100%	1.00	0.80
Android-Specific	6	3	0	1	4	1	0	100%	80%	75%	100%	0.86	0.89
Implicit Flows	4	0	0	8	0	0	8			0%	0%		
Bypass Authentication	2	4	2	0	4	0	0	67%	100%	100%	100%	0.80	1.00
Missing Encryption	4	1	2	1	2	0	0	33%	100%	50%	100%	0.40	1.00
Command Injection	1	1	0	0	1	0	0	100%	100%	100%	100%	1.00	1.00
Least Privileged Violation	1	0	0	1	1	0	0		100%	0%	100%		1.00
Exploitable Service	1	0	1	1	1	0	0	0%	100%	0%	100%		1.00
TOTAL	43	31	7	13	36	6	8	82%	86%	70%	82%	0.76	0.84

a need to improve the commercial offerings currently available on the market.

Unsound tools have low recall. Some approaches favor unsoundness for precision, but this tradeoff significantly lowers recall. There are many sources of unsoundness. For example, in one test case (SecretViewer2), programming to interfaces leads to an unsound call graph for FlowDroid and unsound results. For business-critical applications that have to be free from vulnerabilities, it may be worthwhile to use more heavyweight approaches to find all the vulnerabilities as opposed to using a lighter analysis that may find only 70-80% of the vulnerabilities relatively easily. In that case, finding the remaining 20-30% of the vulnerabilities may be hard and may require using additional tools.

Soundiness is possible and works. Soundiness, i.e., sound modulo inevitable unsoundness [18], is possible and works, and does not lead to poor precision. The rate of false positives (FP) is within an acceptable range.

6. RELATED WORK

We discuss related security benchmarks and other evaluations of approaches for finding security vulnerabilities.

Security benchmarks. A test case in the benchmark is usually small in size and checks if an approach finds a few vulnerabilities while it avoids false positives. Livshits et al. proposed the benchmark SecuriBench Micro [16] for evaluating approaches that find vulnerabilities in web-applications. The test cases focus on vulnerabilities such as injection attacks and check if the static analyses handles aliasing, collections, and dataflow communication. FlowDroid performs well on SecuriBench Micro but does not find two vulnerabilities related to dataflow communication. Since both FlowDroid and Scoria can handle web applications, it would be interesting to extend the evaluation to SecuriBench Micro.

MalGenome [31] is a collection of 1200 malware Android applications. Our evaluation focuses on finding vulnerabilities in legitimate applications. The approaches may not detect malware, which often avoids detection by taking advantage of known limitations of static analyses.

The CERT secure coding standards [19] provide guidance and include code snippets for developers to learn how to avoid introducing security vulnerabilities. Unfortunately, the code snippets are not available as stand-alone tests on which to readily evaluate security approaches. The CERT researchers include in the description tools such as Fortify [11], Coverity [4], and FindBugs [28] that can automatically enforce the rule. However, for some rules automatic detection is unavailable. Constraints in Scoria can implement such rules [30]. Some of the test cases from the extended benchmark, such as ASocket and ARuntime, are inspired from these rules.

Applications with injected vulnerabilities. Several web applications with injected vulnerabilities are available to teach developers security lessons and allow evaluation of security approaches [23]. SecuriBench [17] is a collection of several web applications that span from hundreds to tens of thousands of lines of code. Each application has several injected vulnerabilities common in web application such as SQL injection. The benchmark is used by its designers to evaluate approaches such as Program Query Language. Other researchers such as Liu and Milanova [15] use SecuriBench to evaluate their approach that handles both explicit and implicit information flow.

InsecureBank [24] has several injected vulnerabilities including information disclosure to an external memory card. FlowDroid is able to find the vulnerabilities on the client side which is an Android application, but not on the server side, which is implemented in Python.

Case studies on real-world applications. Enck et al. [7]

Table 3: Overall results. On common test cases, the numbers for AppScan, Fortify and FlowDroid are reused from [2].

Feature used (Section 2.2)	(a) provenance	(b) transitivity	(c) hierarchy	(d) reachability	(e) traceability	(f) indirect comm.	(g) object sec. property	(g) edge sec. property	typechecker warnings	extraction warnings	AppScan			Fortify			FlowDroid			Scoria			Origin
AF index weight	3	2	2	2	1	3	1	1			TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	
Arrays and Lists																							
ListAccess1		✓					0	0	0	3	0	1	0	0	1	0	0	1	0	0	1	0	DB
Callbacks																							
Button1							2	0	1	6	0	0	1	1	0	0	1	0	0	1	0	0	DB
Button2							2	0	1	8	1	0	2	1	0	2	3	1	0	3	0	0	DB
LocationLeak1							2	0	6	4	0	0	2	0	0	2	2	0	0	2	0	0	DB
LocationLeak2							2	0	6	2	0	0	2	0	0	2	2	0	0	2	0	0	DB
MethodOverride1		✓					0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0	DB
Field and Object Sensitivity																							
FieldSensitivity1		✓					0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	DB
FieldSensitivity3		✓					0	0	0	3	1	0	0	1	0	0	1	0	0	1	0	0	DB
FieldSensitivity4		✓					0	0	1	3	0	1	0	0	0	0	0	0	0	0	1	0	DB
InheritedObjects1							2	0	1	3	1	0	0	1	0	0	1	0	0	1	0	0	
ObjectSensitivity1		✓					0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	DB
ObjectSensitivity2		✓					2	0	2	4	0	1	0	0	0	0	0	0	0	0	2	0	DB
ADatacontainer				✓			2	0	2	4							2	0	0	2	0	0	US
Inter-App Communication																							
IntentSink1				✓	✓		2	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	DB
IntentSink2							2	0	0	6	1	0	0	1	0	0	1	0	0	1	0	0	DB
ActivityCommunication1	✓						0	0	1	4	1	0	0	1	0	0	1	0	0	1	0	0	DB
Lifecycle																							
BroadcastReceiverLifecycle1							2	0	0	3	1	0	0	1	0	0	1	0	0	1	0	0	DB
ActivityLifecycle1		✓			✓		0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0	DB
ActivityLifecycle2							2	0	1	4	0	0	1	1	0	0	1	0	0	1	0	0	DB
ActivityLifecycle3							2	0	0	3	0	0	1	0	0	1	1	0	0	1	0	0	DB
ServiceLifecycle1							2	0	0	3	0	0	1	0	0	1	1	0	0	1	0	0	DB
General Java																							
Loop1							2	0	2	3	1	0	0	0	0	1	1	0	0	1	0	0	DB
SourceCodeSpecific1		✓					0	0	2	3	1	0	0	1	0	0	1	0	0	1	0	0	DB
UnreachableCode							2	0	1	1	0	0	0	0	1	0	0	0	0	0	1	0	DB
Miscellaneous Android-Specific																							
PrivateDataLeak1		✓					0	0	3	7	0	0	1	0	0	1	1	0	0	1	0	0	DB
PrivateDataLeak2							2	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	DB
DirectLeak1							2	0	0	4	1	0	0	1	0	0	1	0	0	1	0	0	DB
InactiveActivity							2	0	0	1	0	1	0	0	1	0	0	0	0	0	1	0	DB
AActivity				✓			2	0	0	3							0	0	1	1	0	0	US
LogNoLeak							2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	DB
Implicit Flows																							
ImplicitFlow1							0	0	0	0	0	0	2	0	0	2	0	0	2	0	0	2	DB
ImplicitFlow2							0	0	0	0	0	0	2	0	0	2	0	0	2	0	0	2	DB
ImplicitFlow3							0	0	0	0	0	0	2	0	0	2	0	0	2	0	0	2	DB
ImplicitFlow4							0	0	0	0	0	0	2	0	0	2	0	0	2	0	0	2	DB
Missing Encryption																							
ACipher	✓						0	0	0	1							0	0	0	0	0	0	SRD
ACipher2	✓						0	0	0	0							0	1	0	0	0	0	SRD
ASocket			✓			✓	2	0	2	5							1	1	0	1	0	0	CERT
ACryptographic	✓				✓		0	0	2	2							0	0	1	1	0	0	SRD
Bypass Authentication																							
AToken1			✓	✓		✓	2	0	0	6							3	0	0	3	0	0	US
AToken2			✓	✓		✓	2	1	0	6							1	2	0	1	0	0	US
Command Injection																							
ARuntime					✓		1	0	0	6							1	0	0	1	0	0	CERT
Exploitable Service																							
AChat	✓						0	0	0	2							0	1	1	1	0	0	US
Least Privilege Violation																							
SecretViewer2	✓						1	0	3	1							0	0	1	1	0	0	US

evaluated Fortify on several popular Android applications available on the market. Fortify found several classes of vulnerabilities including the misuse of the device identification number or the GPS location. Other approaches [6, 9] use dynamic analysis to find vulnerabilities by monitoring applications. Their goal is to identify and prevent vulnerabilities at runtime. These approaches were evaluated on hundreds of applications selected based on their popularity.

Using popular apps in an evaluation may not be enough. An approach may report the same vulnerability in applications that have similar functionality but may miss other classes of vulnerabilities. For example, Scoria finds a vulnerability in an open-source Android application, Universal Password Manager [30]. Fahl et al. [8] analyzed more than 20 similar Android applications that store encrypted passwords. These applications share the same vulnerability, where the password in clear text is disclosed to the clipboard. In Android, the clipboard is available to apps without explicit permission, so the password can be exposed.

7. CONCLUSION

This paper contributes a comparative evaluation of two approaches for finding security vulnerabilities, an architecture-level approach and a code-level approach. It also contributes the Architectural Flaw Index as a way to classify test cases along the continuum of coding bug vs. architectural flaw. Our hope is that SIWs insist that commercial and research approaches be comparatively evaluated on common benchmarks. The paper also contributes a carefully curated benchmark consisting of test cases that we hand-selected from different sources or designed ourselves. We hope that SIWs contribute test cases to the benchmark, particularly ones focusing on architectural flaws or where widely deployed tools struggle.

Acknowledgments. This work was supported in part by the Army Research Office under Award W911NF-09-1-0273.

8. REFERENCES

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

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI*, 2014.

[3] B. Berger, K. Sohr, and R. Koschke. Extracting and Analyzing the Implemented Security Architecture of Business Applications. In *CSMR*, 2013.

[4] Coverity. Static Application Security Testing, 2013.

[5] W. Dietl, S. Dietzel, M. Ernst, K. Muslu, and T. Schiller. Building and using pluggable type-checkers. In *ICSE*, 2011.

[6] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI*, 2010.

[7] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *USENIX Conference on Security*, 2011.

[8] S. Fahl, M. Harbach, M. Oltrogge, T. Muders, and M. Smith. Hey, You, Get Off of My Clipboard - On How Usability Trumps Security in Android Password Managers. In *Financial Cryptography*, 2013.

[9] Y. Falcone, S. Currea, and M. Jaber. Runtime Verification and Enforcement for Android Applications with RV-Droid. In *Runtime Verification*, 2013.

[10] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Intl. Journal of Information Security*, 8(6), 2009.

[11] HP. Fortify Static Code Analyzer, 2013.

[12] IBM. AppScan Source, 2013.

[13] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. Android Taint Flow Analysis for App Sets. In *International Workshop on the State of the Art in Java Program Analysis (SOAP)*, 2014.

[14] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop*, 2011.

[15] Y. Liu and A. Milanova. Static information flow analysis with handling of implicit flows and a study on effects of implicit flows vs explicit flows. In *CSMR*, 2010.

[16] B. Livshits. Stanford SecuriBench Micro, 2006.

[17] B. Livshits and M. Lam. Finding Security Errors in Java Programs with Static Analysis. In *Usenix Security Symposium*, 2005.

[18] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Guyer, U. Khedker, A. Møller, and D. Vardoulakis. In Defense of Soundiness: a Manifesto. `http://soundiness.org/`, 2014.

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

[20] G. McGraw. *Software Security: Building Security In*. Addison-Wesley Professional, 2006.

[21] A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *POPL*, 1999.

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

[23] OWASP. WebGoat Project, 2013.

[24] Paladion. AppSec tools for Mobile Enthusiasts. InsecureBank, 2013.

[25] The MITRE Corporation. Common Weakness Enumeration (CWE). `https://cwe.mitre.org/`, 2013.

[26] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. ANDROMEDA: Accurate and Scalable Security Analysis of Web Applications. In *FASE*, 2013.

[27] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: effective taint analysis of web applications. In *PLDI*, 2009.

[28] University of Maryland. FindBugs: Find Bugs in Java Programs. `http://findbugs.sourceforge.net`, 2007.

[29] R. Vanciu and M. Abi-Antoun. Ownership Object Graphs with Dataflow Edges. In *WCRE*, 2012.

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

[31] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *IEEE Symposium on Security and Privacy*, 2012.