Comparative Evaluation of Static Analyses that Find Security Vulnerabilities

Ebrahim Khalaj **Radu Vanciu** **Marwan Abi-Antoun**

March 2014

Department of Computer Science
Wayne State University
Detroit, MI 48202

Abstract

To find security vulnerabilities, many research approaches and commercial tools use a static analysis and check constraints. Previous work compared using a benchmark several approaches where the static analysis and constraints are combined, and the evaluation focused on corner cases in the Java language. This paper augments the comparative evaluation of these approaches with one approach that separates the constraints from the static analysis. We also extend the benchmark to cover more classes of security vulnerabilities. Approaches that combine the static analysis and constraints work well for vulnerabilities that are sensitive to the order in which statements are executed. The additional effort required to write separate constraints is rewarded by a better recall in dealing with dataflow communication and better precision for callback methods that are common in applications built on frameworks such as Android.

This work was supported in part by the Army Research Office under Award No. W911NF-09-1-0273.

Keywords: comparative evaluation, static analysis, security vulnerabilities

Contents

1 Introduction

Over the past several years, static analysis has emerged as an important tool in finding security vulnerabilities. Static analysis tools complement dynamic testing because the quality of the output does not depend heavily on having a good test coverage. Static analysis is also particularly relevant for finding security vulnerabilities, since it can cover all possible executions of a program, thus enabling a worst—rather than an average—case analysis.

Several static analysis approaches and tools, both commercial and research-based, have been proposed. There is no good understanding, however, of the differences in the expressiveness power of the various approaches, i.e., if the approaches find the same or different types of vulnerabilities, or if they have roughly the same or different rates of false positives or false negatives.

One reason for this state of affairs is the lack of good benchmarks on which the various approaches can be compared. There are several benchmarks [11, 20], but they are not particularly focused on security vulnerabilities.

Moreover, some features such as callbacks used by frameworks tend to complicate the detection of security vulnerabilities by static analysis. As a result, it is important for a benchmark to include test cases that use those features, as opposed to focusing on the popularity of specific frameworks in widely deployed applications.

In this paper, we took an existing benchmark and evaluated it by running an additional tool, and comparing the tool against other tools in terms of true positives, false positives, and false negatives. This comparison is instructive because the previously evaluated tools use only static analysis, but the tool being compared combines annotations, a static analysis to extract a high-level representation, and human-specified constraints on the high-level representation. The evaluation uncovered some classes of vulnerabilities that the original benchmark lacked, so we extended the benchmark with some new test cases.

Contributions. This paper makes the following contributions:

- An empirical comparative evaluation of several static analysis approaches with one approach that separates the extraction from the constraints;
- An extended benchmark with new equivalence classes that highlight additional conceptual issues of security flaws;

Outline. The rest of this paper is organized as follows. Section **??** describes in what ways this contribution constitutes science. Section 2 summarizes the key characteristics of the static analyses that this paper evaluates. Section 3 describes our evaluation method. Section 4 presents our results. Section 5 discusses some threats to validity and limitations. Section 6 introduces the most closely related work.

2 Background

The evaluation in this paper covers the following commercial and research tools, AppScan [16], Fortify [14], FlowDroid [11] and Scoria [30]. We first discuss these tools then discuss their characteristics and those of related and similar tools in the solution space.

2.1 Evaluated Approaches

Fortify. Fortify SCA [14] 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) [1]. 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.

[1] We use information flow instead of data flow terminology to distinguish a data flow between variables from dataflow communication between objects.

AppScan. Similarly to Fortify, AppScan Source [16] 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: it consists of a source, a sanitizer, and a sink and checks that no information flows from a source to a sink without passing through the sanitizer [28].

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. An Example 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 architects should consider using other approaches. This paper focuses on analyses that find non-local vulnerabilities.

FlowDroid. Similar to AppScan and Fortify, FlowDroid [11] 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 [17].

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 [27] and is object-, field-, and context-sensitive, while AppScan uses WALA [28]. 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, the architects 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 [23]. If the architects 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.

Scoria is an approach that separates constraints from static analysis [30], and does not require a precomputed may-alias analysis or a call-graph. Instead, Scoria relies on ownership types that are added to the code using existing language support for annotations. The annotations implement the Ownership Domains type system [1], where a domain is a named, conceptual group of objects. Two variables in the same domain may alias, but two variables in different domains may not alias.

The architects add annotations to reference variables in the code, then use a typechecker to ensure that the annotations are consistent with each other and with the code. Otherwise, the typechecker reports a warning which architects attempt to resolve. Once the program has annotations that typecheck, the architects use a static analysis to extract an object graph, where a node represents an abstract object, while an edge represents dataflow communication between objects.

The extracted object graph is hierarchical, with an object having child domains, and in turn, each domain having child objects. If the extraction analysis is not able to extract an edge or an object, it reports a warning. Domains allow architects to express design intent and push objects that represent implementation details underneath other objects that are more architecturally relevant. Object hierarchy provides precision and prevent the analysis from merging objects excessively. Scoria allow architects to distinguish between two objects of the same type in different domains. Architects can fine-tune the precision of the analysis using annotations. By placing objects in different domains, the architects can prevent the extraction analysis from merging those two abstract objects. The architects can then assign different security properties values to those different objects. The object graph is sound such that every runtime object has a unique representative abstract object in the graph. Also, the graph shows all possible relations such that every runtime edge between two runtime objects has a corresponding abstract edge between the representatives of the two objects. Not having unique representatives could lead to misleading results; in that case, architects may be able to assign different security property values to one runtime object that has multiple representative abstract objects.

Based on the object graph, architects query the graph, write constraints on the query results or assign security properties to objects and edges. The constraints are thus separated from the extraction analysis. A dataflow communication edge refers to objects and allows architects to write constraints in terms of object

provenance, where the same object that flows from a source to a destination, cannot flow from a second source to a second destination (Fig. 1a). Since the object graph is hierarchical, constraints can also be written in terms of indirect communication (Fig. 1c), where a confidential object flows to a descendant of an untrusted sink. The architects can then write constraints in terms of object reachability (Fig. 1d), where confidential data is reachable from the object that a dataflow with an untrusted destination refers to. Object reachability can consider various types of edges. In addition to dataflow communication edges, the object graph can also have points-to edges that correspond to field declarations in the code.

2.2 Other Approaches

We discuss briefly other approaches for finding security vulnerabilities that we do not include in this evaluation.

Android-specific approaches such as Blue Seal [13] use an information flow static analysis to find vulnerabilities in applications that abuse the permissions that the users provide, for example, to allow an application to access the GPS information.

Other approaches such as Program Query Language (PQL) [21] and Object Query Language (OQL) [31] also separate constraints from the static analysis. In PQL and OQL, the extracted object graphs are flat and lack the precision provided by object hierarchy and the expressiveness of constraints in terms of object provenance and indirect communication.

When reasoning about security, architects often track instances instead of types, Scoria extract an approximation of a runtime architecture. Other architectural-level approaches use a static analysis to extract an architectural representation from code [26]. The representation is often a code architecture [4]; as a result, the architects cannot distinguish between different instances of the same type in order to assign to them different values of security properties.

Other researchers also explored the idea of finding vulnerabilities using constraints [2]. Almorsy et al. [2, 3] write constraints on various UML models such as sequence and class diagrams extracted from the program's Abstract Syntax Tree (AST). The static analysis that extracts the sequence diagram is an AST visitor that does not consider aliasing.

Other static taint analyses [12] reason about information flow at the level of variables. ScanDroid [12] proposes an Android-specific static analysis built using the WALA framework [15] that is context-sensitive, flow-insensitive analysis and incorporates constraints.

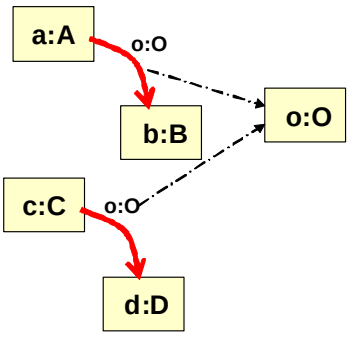
Specialized type systems can find security vulnerabilities at compile time. For example, Tainting Checker [6] is a type system that allows architects to use taint annotations on types. This approach thus avoids using a separate may-alias analysis. Although the Tainting Checker requires fewer annotations than Scoria, the constraints are hard-coded into the type system.

Another approach to find security vulnerability is by monitoring applications at runtime [7], or by analyzing execution traces. Dynamic analysis approaches provide a higher precision but do not cover all possible executions. Achieving good results requires a high-coverage test suite with test cases that trigger the vulnerabilities. In addition to the static analysis, AppScan provides a dynamic analysis and supports a combination of static and dynamic analysis.

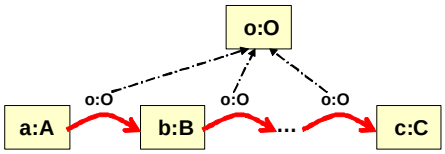
3 Method

The comparison of FlowDroid [11] with two commercial tools focuses on the evaluation of information flow approaches for finding security vulnerabilities. The evaluation is based on the DroidBench benchmark consisting of 39 small Android applications. This paper uses the same benchmark and compares different types of approaches without rerunning the information flow approaches. Our evaluation excludes tests from the same equivalence class that target the same behavior of a static analysis and tests that focus on code constructs specific to Java such as anonymous classes or static initialization blocks (Table 1).

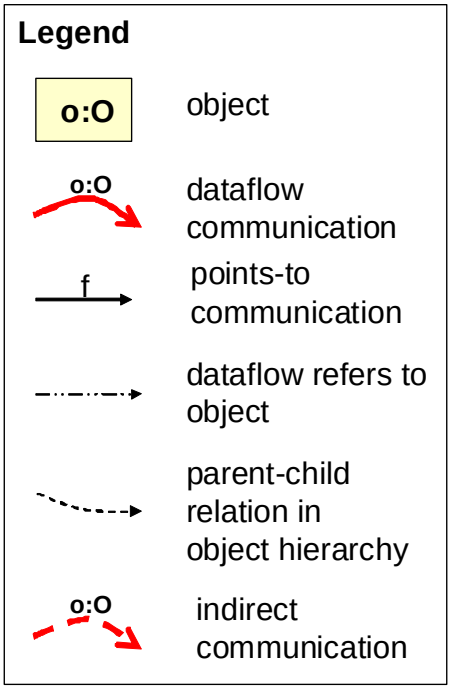
Study Design. We used participant who had no experience on the evaluated tools. There were two major phases, annotating the code and writing constraints. We gave the participants several training session on

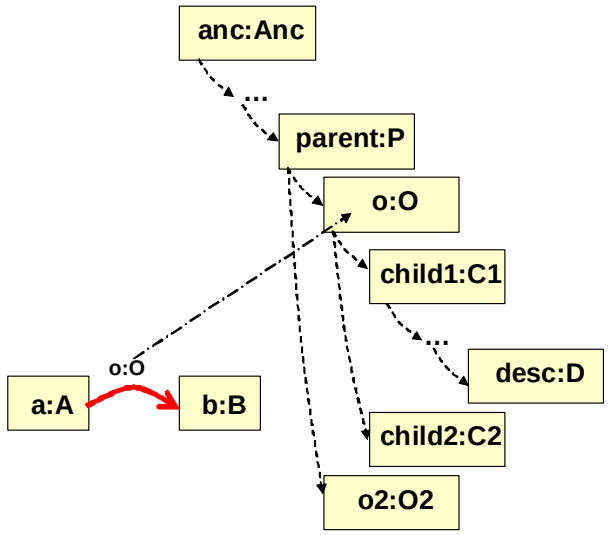


(a) Object Provenance: The same `o:O` that flows from `a:A` to `b:B` should not flow from `c:C` to `d:D`

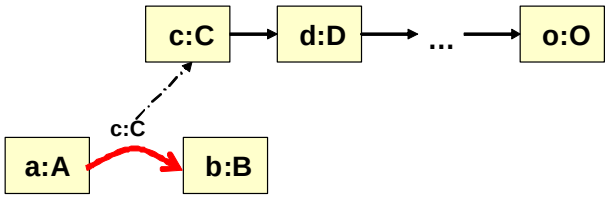


(b) Object Transitivity: All dataflow edges in a path refer to the same `o:O`

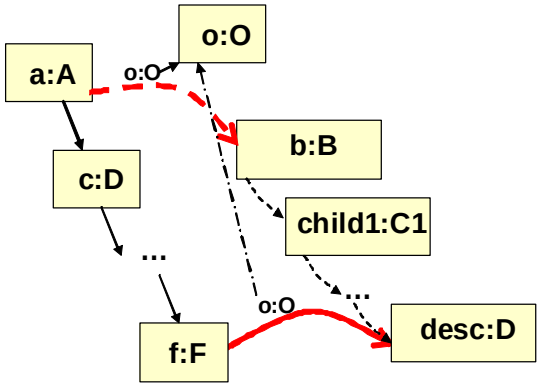




(c) Object Hierarchy: A dataflow object `o:O` has ancestors and descendants



(d) Object Reachability: `o:O` is reachable from dataflow object `c:C`



(e) Indirect Communication: `a:A` is connected to `b:B` through descendants and reachable objects.

Figure 1: Features of Scoria [30]. The extracted object graph is hierarchical and has dataflow communication edges (thick red or blue edges) that refer to objects.

the concepts of annotating and on how to annotate the code. We assigned 13 test cases to each participant and they annotated the given test case. We observed that for the early test cases, the participants had to get feedback from us more frequent. But they managed to do most of the work of the next test cases. We also prepared two test cases, IntentSink1 and ObjectSensitivity1 as reference for the participant. We saw that the writing constraint phase was more convenient for the participants. Because of the nature of the

Table 1: Selected tests from DroidBench

	Selected		DroidBench	
Equivalence class	**tests**	**vulns.**	**tests**	**vulns.**
Arrays and Lists	1	0	3	0
Callbacks	5	9	6	10
Field and Object Sensitivity	6	2	7	2
Inter-App Communication	3	3	3	3
Lifecycle	5	5	6	6
General Java	3	2	5	4
Android-Specific	5	3	5	3
Implicit Flows	4	8	4	8
TOTAL	32	32	39	36

test cases of DroidBench, the participants managed to reuse the written constraints across the assigned test cases.

Participants. We hired two graduate students as the participants. The participants have not had any experience about Scoria and the used tools and concepts such as type checker, static analysis and object graph. We held several learning sessions about ownership domains and the participants learned how the annotations work via several concrete examples. The goal was to teach them how to annotate the code. Then we gave them the instructions on how to use the analysis to extract the object graph. They learned how to use the object graph to visually see vulnerabilities in the code. Annotating the code was the first phase of the evaluation project. On the second phase we thought the participants how to write constraints on the object graph.

Tasks. First task of participants was to annotate the code. They had to inspect the given test case to understand it.Next task was to run the analysis on the annotated code and visually inspect the extracted object graph. The participants had to go back and refine the annotations if the object graph does not show what they wanted to see on it. The last task of the participants was to write a constraint as a JUnit test to find the vulnerability in the test case.

Tools and Instrumentation. We asked participants to use either Eclipse 3.5.x or 3.7.x. They could use all of the features of Eclipse. Other plugins such as Android SDK, Android development tools and subversion client adaptor had to be installed on Eclipse.

Procedure. First the participants inspected the code to understand it. Then they tried to expose the design intents of the test case in the annotations. The participants used type checker to make sure annotations be consistent to each other. Sometimes the participants had to go back and refine the annotations to resolve conflicts. For some type checker warnings, the participants had to do minor refactoring on the code. Extracting local variables, converting anonymous classes to nested classes, and adding default constructor were the common refactoring. Next, the participants extracted the object graph using the static analysis tool. The participants used object graph to visually confirm vulnerabilities in the test case. At the end, the participants write constraints in the form of JUnit tests. As a result, dataflow edges that represent vulnerabilities in the code were reported by the tool.

Analysis. Our analysis started with accumulating the results of the assigned test cases to each participant in a table (Table 2). Each step in the procedure has a corresponding column in the table such as remaining type checker and remaining extraction warnings. We reported the results of running Scoria and FlowDroid on a test case in terms of true positives, false positives and false negatives. We analyzed the results to see which tool did better on which test case and we investigated the reasons. We also had special attention to the used features of Scoria because mainly they were the reason that Scoria did better on some test cases.

4 Results

Quantitative results. The results indicate that the research approaches FlowDroid and Scoria outperform the commercial tools in terms of both precision and recall (Table 3). Compared to FlowDroid, Scoria has higher recall, precision and F-measure (the weighted harmonic mean of precision and recall).

The results provide answers to the research questions and identify gaps in the solution space. By using global reasoning and constraints, the architects can improve the recall and precision. Security approaches that handle implicit flows and additional forms of design intent such as the state of objects, may have higher recall. To improve precision, security approaches may support flow-sensitive analysis. Another source of imprecision is code that is never executed (unreachable code), such as a method that is never invoked. Determining if a method may be invoked is non-trivial in the presence of frameworks, where the callback methods are invoked by the framework based on configuration files. One solution is to parse the configuration files and attempt to determine the methods that may be invoked. Another solution is to assume that all the methods declared on a class that is instantiated may be invoked. The rest of the section focuses on qualitative results and discusses the most interesting test cases.

IntentSink1. In Android, objects of type `Intent` are used for intra- and inter-application communication. Since other applications can access such objects, they should not refer to confidential information. The test case checks if the device identification number (imei) flows to an object of type `Intent`, which is then passed as an argument to the method `setResult` (Fig. 2). FlowDroid is the only approach that fails to find the vulnerability because it cannot track information flow through intermediate values in the framework. Finding the vulnerability requires more than a simple constraint on the information flow. In Scoria, the constraint uses security properties for two objects. The architects set the object `imei:String` as confidential and `i:Intent` as untrusted. The constraint then uses object reachability and edge traceability. The constraints checks if a dataflow edge with an untrusted destination refers to a confidential object and the destination is in turn referred from another edge that traces to code to the method `setResult` (Fig. 2).

ObjectSensitivity1. This test case has two objects of type `LinkedList` where only `l1:LinkedList` has elements that represent confidential data. Since only the first element of `l2:LinkedList` is passed as an argument to the method `sendTextMessage` of the class `SmsManager`, no information disclosure exists (Fig. 3). None of the evaluated approaches report any false positives. The information flow based approaches use the two object allocation sites to distinguish between the objects, while Scoria uses domains. One list refers to objects of type `String` in the domain `DATA` while the second one refers to objects in `SHARED`. The constraint checks that no objects in `DATA` flow to an object of type `SmsManager`.

Button2. The test case has three vulnerabilities that are triggered by pressing a few buttons in a particular order. The device identifier (imei) is disclosed either to the phone log or via a text message sent to the attacker. Only FlowDroid and Scoria report all three vulnerabilities. FlowDroid has less precision compared to Scoria reporting one false positive, which is due to the variable `imei` being assigned `null` before concatenated to a constant and the result is disclosed (Fig. 4). In Scoria, the architect inspects the code and annotates the result of concatenation in `SHARED` while the object `imei:String` is in the domain `DATA`. The architect then assigns security properties. The objects of type `SmsManager` and `Log` are untrusted, and `imei:String` is confidential. The constraint checks if a dataflow edge with an untrusted destination refers to a confidential object. It reports three edges corresponding to the three security vulnerabilities in the code.

ListAccess1. The test case has a list containing the confidential device identifier (imei) and non confidential constant strings. Since only non confidential data is retrieved from the list and sent via text message to the attacker, there are no security vulnerabilities. All approaches return one false positive because they do not distinguish between specific elements of the list.

FieldSensitivity4. In this test case, an object has a field which is disclosed to an untrusted sink. After disclosure a confidential value is assigned to the field. Fortify and FlowDroid have higher precision compared to AppScan and Scoria each reporting one false positive. Scoria is flow-insensitive and does not consider the order of statements; hence, it reports a false positive. The constraint uses object transitivity and finds a dataflow edge in the object graph that flows from an object of type `TelephonyManager` to an object of type

Table 2: Results of comparing approaches that find security vulnerabilities.

	provenance	transitivity	hierarchy	reachability	traceability	indirect communication	object security properties	edge security properties	typechecker warnings	extraction warnings	AppScan			Fortify			FlowDroid			Scoria		
											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
Callbacks																						
Button1							2	0	1	6	0	0	1	1	0	0	1	0	0	1	0	0
Button2							2	0	1	8	1	0	2	1	0	2	3	1	0	3	0	0
LocationLeak1							2	0	6	4	0	0	2	0	0	2	2	0	0	2	0	0
LocationLeak2							2	0	6	2	0	0	2	0	0	2	2	0	0	2	0	0
MethodOverride1		✓					0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0
Field and Object Sensitivity																						
FieldSensitivity1		✓					0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0
FieldSensitivity3		✓					0	0	0	3	1	0	0	1	0	0	1	0	0	1	0	0
FieldSensitivity4		✓					0	0	1	3	0	1	0	0	0	0	0	0	0	0	1	0
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
ObjectSensitivity2		✓					2	0	2	4	0	1	0	0	0	0	0	0	0	0	2	0
Inter-App Communication																						
IntentSink1				✓	✓		2	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0
IntentSink2							2	0	0	6	1	0	0	1	0	0	1	0	0	1	0	0
ActivityCommunication1	✓						0	0	1	4	1	0	0	1	0	0	1	0	0	1	0	0
Lifecycle																						
BroadcastReceiverLifecycle1							2	0	0	3	1	0	0	1	0	0	1	0	0	1	0	0
ActivityLifecycle1		✓			✓		0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0
ActivityLifecycle2							2	0	1	4	0	0	1	1	0	0	1	0	0	1	0	0
ActivityLifecycle3							2	0	0	3	0	0	1	0	0	1	1	0	0	1	0	0
ServiceLifecycle1							2	0	0	3	0	0	1	0	0	1	1	0	0	1	0	0
General Java																						
Loop1							2	0	2	3	1	0	0	0	0	1	1	0	0	1	0	0
SourceCodeSpecific1		✓					0	0	2	3	1	0	0	1	0	0	1	0	0	1	0	0
UnreachableCode							2	0	1	1	0	0	0	0	1	0	0	0	0	0	1	0
Miscellaneous Android-Specific																						
PrivateDataLeak1		✓					0	0	3	7	0	0	1	0	0	1	1	0	0	1	0	0
PrivateDataLeak2							2	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0
DirectLeak1							2	0	0	4	1	0	0	1	0	0	1	0	0	1	0	0
InactiveActivity							2	0	0	1	0	1	0	0	1	0	0	0	0	0	1	0
LogNoLeak							2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Implicit Flows																						
ImplicitFlow1							0	0	0	0	0	0	2	0	0	2	0	0	2	0	0	2
ImplicitFlow2							0	0	0	0	0	0	2	0	0	2	0	0	2	0	0	2
ImplicitFlow3							0	0	0	0	0	0	2	0	0	2	0	0	2	0	0	2
ImplicitFlow4							0	0	0	0	0	0	2	0	0	2	0	0	2	0	0	2
Extended Benchmark																						
ACipher	✓						0	0	0	1							0	0	0	0	0	0
ACipher2	✓						0	0	0	0							0	1	0	0	0	0
AToken1			✓	✓		✓	2	0	0	6							3	0	0	3	0	0
AToken2			✓	✓		✓	2	1	0	6							1	2	0	1	0	0
CERT_ASocket			✓			✓	2	0	2	5							1	1	0	1	0	0
AActivity				✓			2	0	0	3							0	0	1	1	0	0
CERT_ARuntime					✓		1	0	0	6							1	0	0	1	0	0
ADatacontainer				✓			2	0	2	4							2	0	0	2	0	0
AChat	✓						0	0	0	2							0	1	1	1	0	0
SecretViewer2	✓						1	0	3	1							0	0	1	1	0	0

Table 3: Recall and precision based on tests from DroidBench and Extended Benchmark

	AppScan	Fortify	FlowDroid	Scoria
TP, higher is better	13	14	31	35
FP, lower is better	4	3	6	6
FN, lower is better	19	18	12	8
Precision	76%	82%	84%	85%
Recall	41%	44%	72%	81%
F-measure	0.53	0.57	0.77	0.83

```
class IntentSink1 extends Activity {
  TelephonyManager tm = ...;
  void onCreate(Bundle b) {
    String imei = tm.getDeviceId(); //source
    Intent i = this.getIntent();
    i.putExtra("secret", imei);
    this.setResult(RESULT_OK, i); //sink, leak
  }
}
```

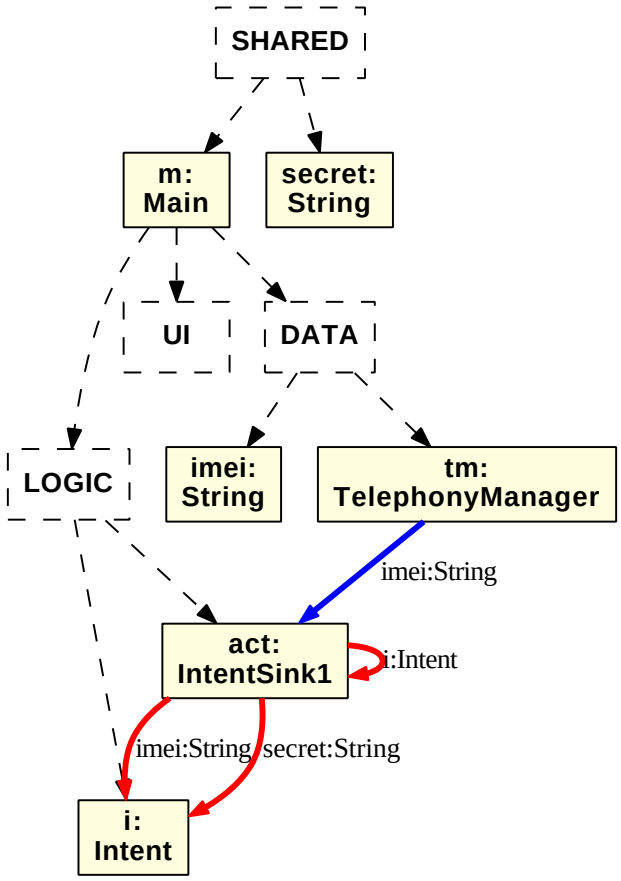


Figure 2: The object graph extracted by the static analysis of Scoria. The device id flows from an object of type `TelephonyManager` to an object of type `Intent` which is then disclosed.

```
class ObjectSensitivity1 extends Activity {
  TelephonyManager tm = ...;
  SmsManager sms =...
  void onCreate(Bundle b) {
    LinkedList<String> l1 = new LinkedList<String>();
    LinkedList<String> l2 = new LinkedList<String>();
    l1.add(tm.getSimSerialNumber());//source
    l2.add("123");
    sms.sendTextMessage("+49 1234", l2.get(0));//sink
  }
}
```

Figure 3: The device id flows from an object of type `TelephonyManager` to `l1:LinkedList`, but only the element of the other list `l2:LinkedList` is disclosed.

```
class Button2 extends Activity {
  String imei = null;
  void onCreate(Bundle b){
   Button b1 =...   b1.setListener(new Listener1());
   Button b2 =...   b2.setListener(new Listener2());
  }
  void onClick(View v){
    imei = tm.getDeviceId();
    Log.i("TAG", "Button3: " + imei); //sink, leak
  }
}
class Listener1 extends View.OnClickListener {
  void onClick(View v){
    sm.sendTextMessage("+49 1234", "sendIMEI: " + imei);  //sink, leak
    Log.i("TAG", "sendIMEI: " + imei);  //sink, leak
    imei = null;
  }
}
class Listener2 extends View.OnClickLisener {
 void onClick(View v) {
  imei = null;
  Log.i("TAG", "Button 2: " + imei); //sink, no leak
 }
}
```

Figure 4: The device id flows from an object of type `TelephonyManager` to an object of type `Log` and to an object of type `SmsManager`. In one case the value is assigned `null` before concatenation.

`FieldSensitivity4` which extends `Activity` and then from the object of type `FieldSensitivity4` to an object of type `SmsManager` (Fig. 5).

InactiveActivity. In this test case, a confidential value is disclosed to an object of type `Log` in the body of the class `InactiveActivity`. The object of type `InactiveActivity` is set disabled in a configuration file (Fig. 6). For this test case, FlowDroid has higher precision compared to Fortify, AppScan and Scoria each reporting one false positive. In Scoria, the constraint uses two security properties and finds a direct dataflow communication edge that refers to `imei:String` which is confidential, and has an untrusted destination of type `Log`. The dataflow edge is a false positive because the source is the object of type `InactiveActivity` that disabled for any execution of the test case. If the architect were to assign a security property, IsActive, and revise the constraint, Scoria would avoid the false positive. Being active or inactive is however a state of the object that can change at runtime, not a property.

Lessons Learned. Analyzing test cases across equivalence classes in DroidBench we observed that none of the approaches can handle implicit flows and collections. For Lifecycle, research tools outscore commercial tools because the precomputed call graph has missing edges. FlowDroid compensates the limitation by building the Activity lifecycle and provides entry points and asynchronously invoked methods to the

```
class FieldSensitivity4 extends Activity {
 void onCreate(Bundle b){
  String imei = tm.getDeviceId(); //source
  Datacontainer data1 = new Datacontainer();
  sms.sendTextMessage("+49 1234", data1.value); //sink
  data1.value = imei;
 }
}
class Datacontainer{
  String value = "android";
}
```

Figure 5: The device id flows from an object of type `TelephonyManager` to a an object `value:String` after this object flows to an object of type `SmsManager`. A flow-sensitive approach reports an information disclosure.

```
class InactiveActivity extends Activity {
 void onCreate(Bundle b){
  String imei = tm.getDeviceId(); //source
  Log.i("TAG", imei); //sink, no leak
 }
}
//AndroidManifest.xml
<activity
  android:name="de.ecspride.InactiveActivity"
  android:enabled="false" >
</activity>
```

Figure 6: The device id flows from an object of type `TelephonyManager` to a an object of type `LOG`. The activity is however disabled in the configuration file.

precomputed call graph. The major difference between commercial tools and research tools is for callbacks. Scoria find all vulnerabilities without reporting false positives, while AppScan and Fortify miss six out of the nine vulnerabilities.

4.1 Extensions of the Benchmark

DroidBench is designed to evaluate information-flow based approaches and tends to focus more on limitations and corner cases of static analysis. Test cases check if the analysis is able to track the flow of confidential data through fields or static variables or through concatenation of objects of type `String`. In most test cases, confidential information such as the device identification number flows directly to an object of type `Log` or to an object of type `SmsManager`. Indeed, in 30 out of the 32 selected tests, Scoria is able to find the vulnerabilities by using direct communication or object transitivity. This paper extends DroidBench with tests that focus on more expressive constraints and security properties. In particular, the tests focus on strengths of Scoria related to non-local reasoning such as hierarchy, and indirect communication (Table 2).

ACipher. Abstract objects with different security properties may trace to code to the same object allocation site. In theory, an object-sensitive analysis uses a sequence of k object allocation expressions, but in practice the implementations are often parameterized with $k = 1$ to improve scalability. The identity of an abstract object should be more than just one object allocation expression in the code. The test case ACipher shows how this assumption may lead to insufficient precision. An object representing encrypted data and an object representing decrypted data are created for the same object allocation expression in the code of the class `Cipher` (Fig. 7). The encryption service uses two objects of type `Cipher`, one for encryption and one for decryption. Unless the analysis distinguishes between the two objects, the approach returns a false positive as decrypted data is stored in a file. Scoria uses object provenance and checks that the same object that flows from the decryption cipher to the encryption service does not flow to an object of type `FileOutputStream`.

```
class Service{
  Cipher eCipher = new Cipher();
  Cipher dCipher = new Cipher();
  String encrypt(String text){
    return eCipher.doFinal(text);
  }
  String decrypt(String text){
    return dCipher.doFinal(text);
  }
}
class Cipher{
  String doFinal(String text){
    String s = new String();
    return s;
   }
}
```

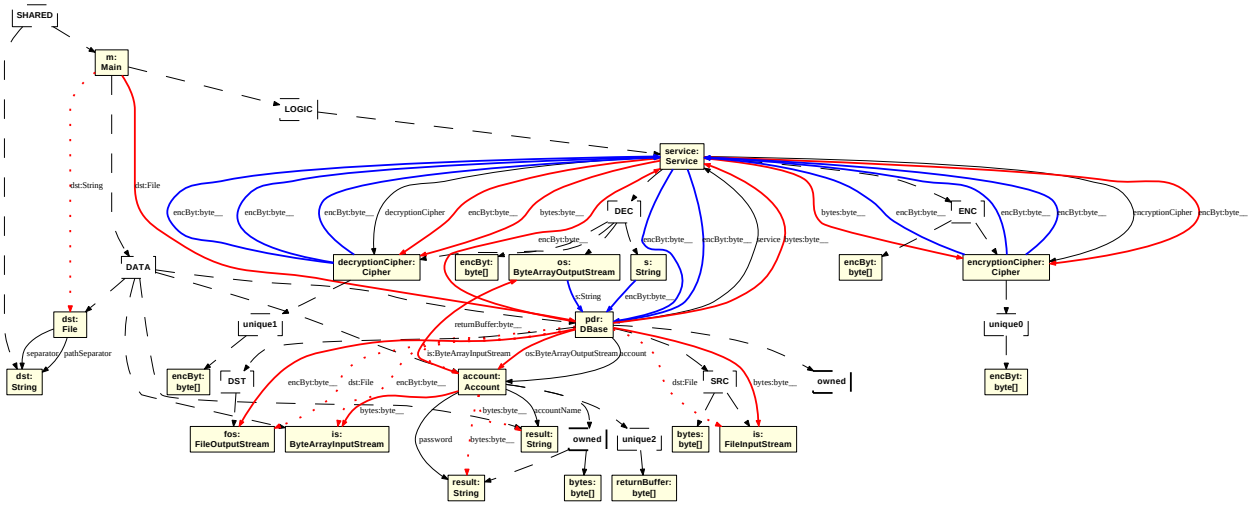


Figure 7: The object of type `Service` has two fields of the same type `Cipher`. Both encrypted and decrypted data are created for the same object allocation expression in the method `doFinal`. A precise static analysis should avoid reporting that decrypted data is disclosed into a file. By using object identity, Scoria avoids this false positive.

AToken1. In most DroidBench test cases, confidential data flows directly to an untrusted sink. But an object from which confidential data is reachable may also flow into an untrusted sink. Finding this new class of vulnerabilities requires a constraint that uses object reachability. In the test AToken1, an object of type `String` representing the authentication token is confidential (Fig. 8). Another object of type `Client` has a field that refers to the token. A vulnerability exists if the object of type `Client` and all the objects it refers to are serialized into an untrusted file. Scoria uses direct communication and object reachability through points-to edges to find the vulnerability.

AToken2. AToken2 is similar to Atoken1. The difference is that some fields can be marked as transient and they are not serialized. For this test case, the field representing the token is transient so no vulnerability exists. To avoid the false positive, Scoria can use the IsTransient property for the points-to edge from the object of type `Client` to the token. This property is independent of the serialization library the code uses.

ASocket. The test belongs to a new class of vulnerabilities when confidential data flows indirectly into an untrusted sink through a descendant of the sink. To find these vulnerabilities, Scoria can use a constraint based on object hierarchy and indirect communication. For example, confidential data representing the password in clear text is sent over the network. If the communication is provided by an object of type `Socket` that does not provide encryption, the confidential data is vulnerable to eavesdropping (Fig. 9). A

```
class View {
  Host h = new Host();
  Client c = new Client(h);
  void onCreate(String pwd){
    c.authenticate(pwd);
    File tFile = new File("client.tmp");
    OutputStream fos = new FileOutputStream(tfile);
    OutputStream out = new ObjectOutputStream(fos);
    out.write(c);
  }
}
class Client{
 //@Transient for AToken2
 String aToken;
 Host h;
}
```

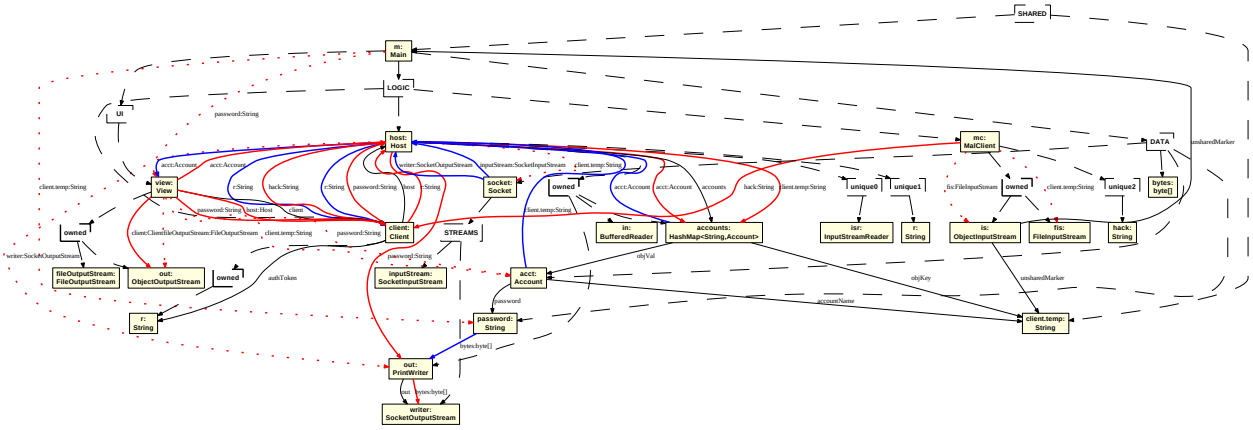


Figure 8: By using object reachability, Scoria reports that the authentication token is disclosed into a file because is reachable from an object of type `Client` (AToken1). The vulnerability does not exist if the field is marked as not being persisted using any persistence mechanism (Java or framework-specific) (AToken2).

constraint that checks direct communication to the object of type `Socket` does not find this vulnerability because the password is sent indirectly to a descendant of type `OutputStream`. FlowDroid assumes that all objects of type `OutputStream` are untrusted sinks and finds the vulnerability. This assumption leads however to a false positive for an object of type `OutputStream` that is a descendant of an object of type `SSLSocket`, which does provide encryption. Scoria uses object hierarchy to distinguish between the two objects of the same type `OutputStream` and avoids the false positive.

AActivity. Another class of vulnerabilities requires reasoning about the state of an object, which may change at runtime. An approach should avoid mixing the state with a security property that does not change at runtime. This may occur when the state is saved in a configuration file but can still be changed programmatically. Similarly to the test InactiveActivity in DroidBench, AActivity reads the state `disabled` of an object of type `Activity` from the configuration file. When the method `onClick` is invoked, the object changes the state of `SecondActivity` to `enabled` and `onCreate` method of `SecAct` discloses the device identification number (Fig. 10). FlowDroid reads the configuration file and assumes the state of object remains unchanged. This assumption allows FlowDroid to avoid the false positive in the test InactiveActivity. For AActivity, however, FlowDroid reports a false negative. Although Scoria does not model object states, it assumes the worst-case that the Activity is always enabled. The assumption then leads to a false positive for InactiveActivity.

ARuntime. All vulnerabilities presented so far are related to information disclosure. Other vulnerabilities such as tampering may allow attackers to tamper with the input that is passed to a method that executes

```
class Host{
  Socket socket = new Socket("localhost", 9999);
  SSLSocket sslSocket = ...;
  void sendPassword1(String pwd) {
    OutputStream sout = socket.getOutputStream();
    PrintWriter out = new PrintWriter(sout, true);
    out.println(pwd);
  }
  void sendPassword2(String pwd) {
    OutputStream sout = sslSocket.getOutputStream();
    PrintWriter out = new PrintWriter(sout, true);
    out.println(pwd);
  }
}
```

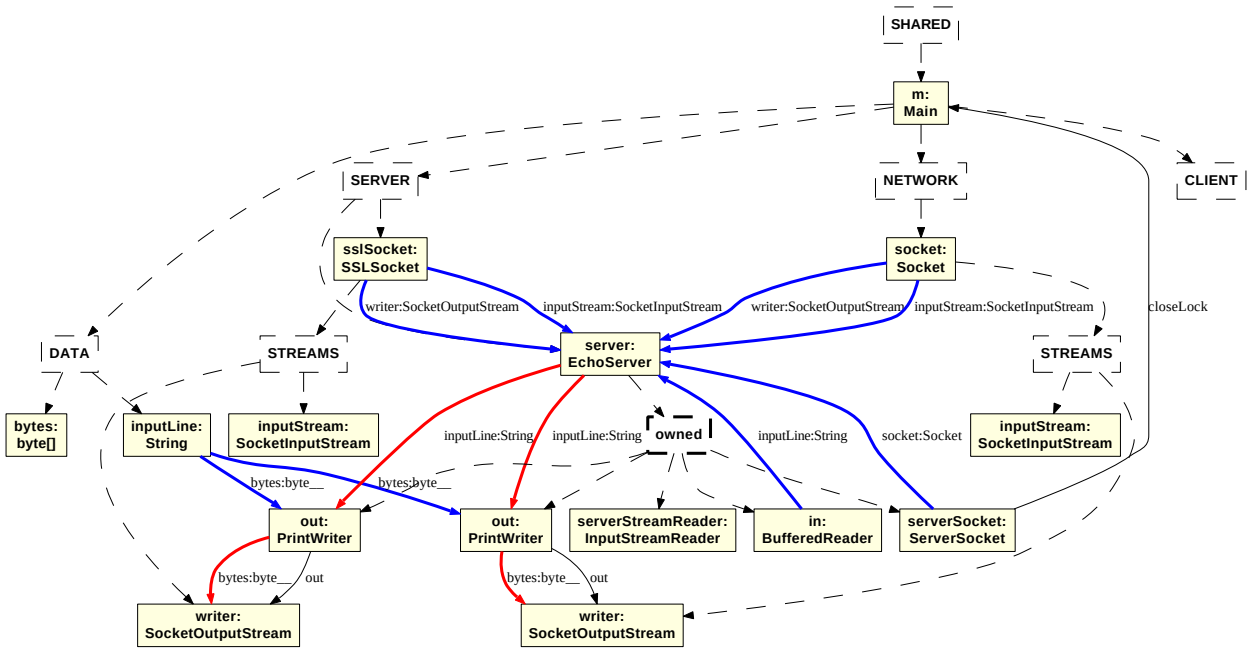


Figure 9: By using indirect communication, Scoria reports that sending the password to a descendant of an object of type `Socket` makes the password vulnerable to eavesdropping. The password should be sent using an object of type `SSLSocket`.

sensitive operations. For example, the method `exec` of the class `Runtime` executes the strings passed as arguments as operating system commands (Fig. 11). If the argument is unsanitized, an attacker can inject and execute arbitrary operating system commands. To find such a vulnerability with Scoria, the architects assign the object of type `Runtime` as trusted and the object of type `String` as unsanitized. Scoria can use a constraint based on direct communication to check if an unsanitized object flows to a trusted sink. The constraint can be more precise and report those dataflow edges that trace to code to an invocation of the method `Runtime::exec`. AppScan and Fortify may be able to find this vulnerability and avoid false positives if `dir` is sanitized because they allow architects to specify which methods sanitize the input using secure coding rules. FlowDroid does not currently support sanitization.

AChat. Even if a program is defect free, sometimes abuse of an existing feature leads to information disclosure. We see evidence of this possible abuse in AChat. In this test case, users are able to find other users by searching their phone number. There is an object of type `FFService` that accepts a list of telephone numbers and returns a list of usernames of the users as response. An object of type `ChatActivity` sends list of telephone numbers to the service. This is the normal scenario to find other users. A malicious client

```
class AActivity {
 void onClick(View view) {
  Intent intent = new Intent(this,SecAct.class);
  startActivity(intent);
 }
}

class SecAct {
  void onCreate(savedInstanceState) {
    TelephonyManager tm = ...;
    String imei = tm.getDeviceId(); //source
    Log.i("TAG", imei); // sink, leak
  }
}
```

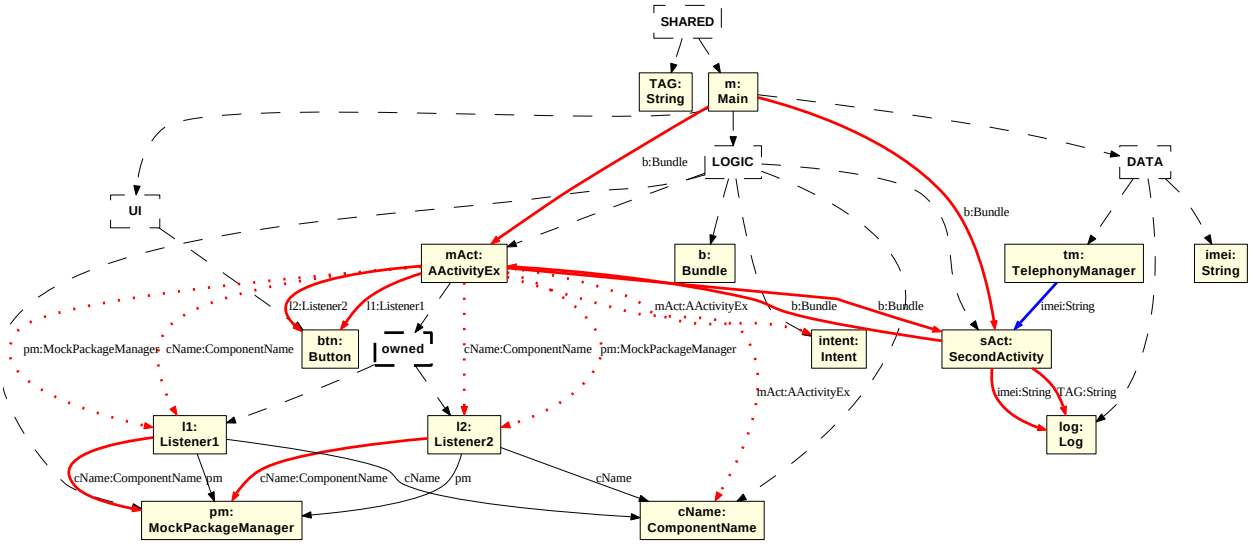


Figure 10: By assuming the worst-case analysis that the `Activity` is always enabled, Scoria manages to report the information disclosure.

is able to abuse this feature by connecting to the service and sending a list of randomly generated telephone numbers(Fig. 12). As a result, the service may disclose confidential information of users to the malicious client. Having hard-coded constraints is not enough to handle this situation. The analysis should consider if an object flows form an object of type `ChatActivity` to an object of type `FFService`, the same object should not flow from an object of type `FFService` to an object of type `Client`. The constraint of FlowDroid does not consider this situation so it misses the information disclosure. By configuring FlowDroid's sources and sinks to make it more precise, FlowDroid reports a false positive too. In Scoria, the architect can write a specific constraint according to the test case so for AChat. A constraint that uses object provenance finds the information disclosure.

ADatacontainer. In ADatacontainer, an object of type `Datacontainer` has a field with a default value. The object is instantiated in `onCreate` method of `Activity`. There is a button and in the `onClick` method of the button, the field is disclosed to an untrusted sink. After disclosure, a confidential value is assigned to the filed. There is no information disclosure if the button is clicked once, but by clicking more than one time, the confidential data is disclosed to the sink. Another method that has the same situation is `onResume`. The value of the field is disclosed to another untrusted sink as the return value of a method. After disclosure a confidential value is assigned to the field. Invoking two or more times of the `onResume` leads to disclosure of the confidential data. Although by invoking for the first time, the default value is sent to the untrusted sink (Fig. 13). This method is a part of `Activity` lifecycle and the body of `onResume`

```
class Dir {
  public static void main(String[] args) {
    String dir = System.getProperty("dir");
    Runtime rt = Runtime.getRuntime();
    rt.exec("/system/bin/ls"+dir);
  }
}
```

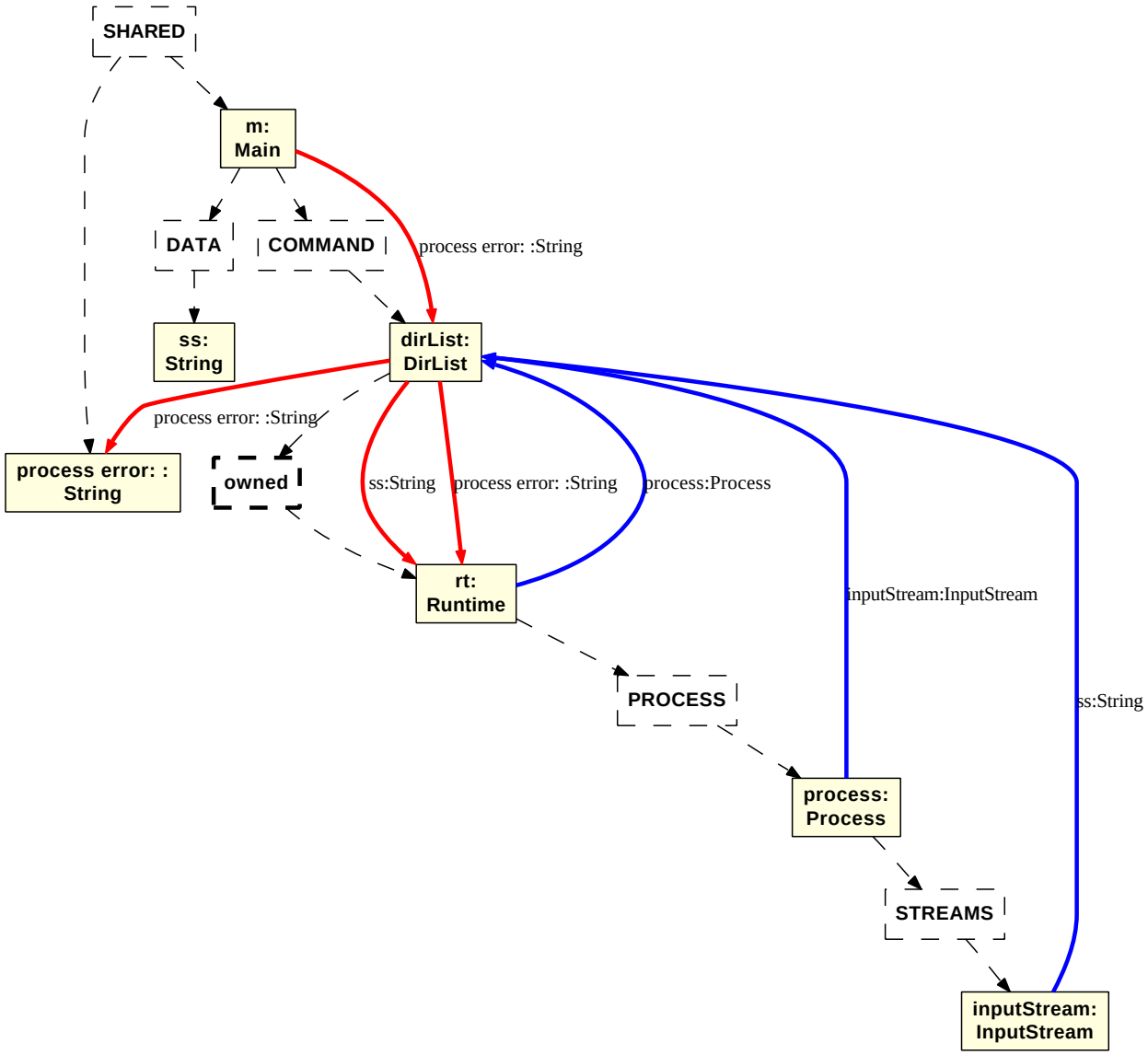


Figure 11: By using direct communication and security properties, Scoria reports that untrusted data is passed as argument to the method `exec`, which gives an attacker the possibility to tamper with the input and execute arbitrary operating system commands.

```
class FFService{
  List<...> telNumbers = null;
  List<...> userNames = null;
  List<...> findByNumber(List<...> givenNumbers){
    ...
   return userNames;
}

class Client{
  sentNumbers = null;
  void clientCall() {
    FFService srv = new FFService();
    usernames = srv.findByNumber(sentNumbers);
 }
}

class ChatActivity{
  ContactsProvider cp = ...;
  List<String> tNumbers = null;
  void onCreate(savedInstanceState){
  tNumbers = cp.getPhoneNumbers();
  FFService srv = new FFService();
  srv.findByNumber(tNumbers);
  }
}
```

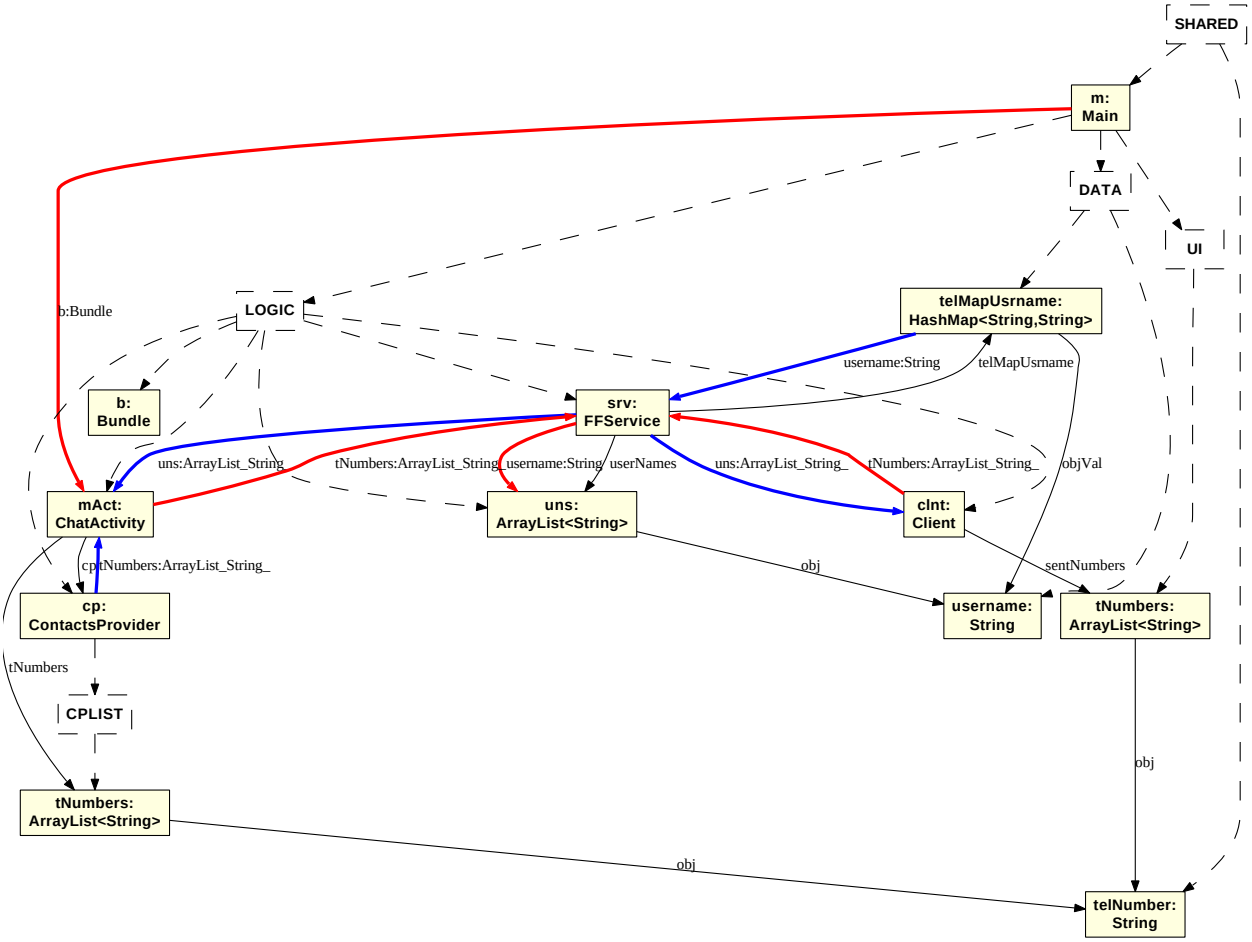


Figure 12: The list of telephone numbers flows from `clnt:Client` to `srv:FFService`, and a list of usernames is disclosed to `clnt:Client`.

method combines flow-sensitivity with `Activity` lifecycle. FlowDroid reports two information disclosures in this test case, because FlowDroid is flow-sensitive and it tries to implement the `Activity` lifecycle. So the information disclosure in `onResume` can be found because this method is a part of the modeled `Activity` lifecycle by FlowDroid. To deal with `onClick` method, FlowDroid uses configuration files to build the call graph so `onClick` method is a part of call graph too. Even if `onClick` is removed from the configuration files and an inner class is used to implement the `OnClickListener` of the button, FlowDroid manages to find the information disclosure using reachability. Scoria does the worst-case analysis and it considers every single method may be executed. So it reports both security vulnerabilities in `onClick` and `onResume`. In the constraint the architect may use security properties and finds the confidential data that is flowed to the untrusted sink.

ACipher2. To find security vulnerabilities, the analysis should be aware of the context that the vulnerability happens. It is not enough to only consider security properties. In fact, Specifying sources and sinks using method declarations may lead to insufficient precision. Therefore, there should be a context that adds extra meaning to the security properties. We see evidence of this in ACipher2. There is a class named `DBase` that has a method that copies data of a source file to a destination file. To do that `DBase` method calls `run` method in an object of type `Service`. The `run` method uses a method in an object of type `Cipher` that returns encrypted text if the input data is decrypted text and decrypted text if the input data is encrypted text. Then an object of Type `PrintWriter` writes the return value of the `run` method in the destination file. Information disclosure happens only when decrypted text is written to the destination file (Fig. 14). So in ACipher2 different contexts for the method `run` are sending decrypted text and sending encrypted text. FlowDroid reports a false positive for this test case because it cannot distinguish between different contexts of `run` method. In fact, for FlowDroid there is no difference between sending an encrypted text and sending a decrypted text. When the encrypted text is sent, FlowDroid reports a false positive. Scoria puts different contexts in different domains so it can distinguish them. In a constraint, architects can place two objects of type `Service` in different domains and checks if an object, which is placed in decryption domain flows from an object of type `Cipher` to an object of type `Service`, the same object should not flow from an object of type `DBase` to an object of type `PrintWriter`.

SecretViewer2. Bell-LaPadula Model [18] is a security model that implements access control by using several levels of security clearance, which are organized in a lattice, and several categories of documents. Each specific clearance level has access to a group of categories. For example, there are clearance levels A, B, C. Clearance level A is higher than B, and clearance level B is higher than C. Categories may be Unclassified, Confidential and Secret. There are also rules that indicate which clearance level has access to which category. If a clearance level has access a category, all the higher levels will have the access too. In contrast, lower levels may not have the access. SecretViewer2 implements a simplified version of Bell-LaPadulla Model using the Document-View architecture. There is an object of type `Document` that has two types of Viewer, `PubViewer` and `SecViewer`. The object of type `Document` sends two types of messages to its viewers. `MsgDtoSV` that is a message form the object of type `Document` to `SecViewer` and `MsgDtoPV` that is a message form the object of type `Document` to `PubViewer`. In SecretViewer, viewers can be considered as clearance levels in Bell-LaPadulla model and different types of messages from `Document`, can be mapped to categories. In `Msg` class that is the base class, an object of type `InputStream` is created. In `BaseViewer` that is the base class for viewers, there is a method named `readStream` which prints the content of the message in a file using an object of type `PrintWriter`. In the the object of type `Document`, when both message types are instantiated, an object of type `InputStream` is passed to them via a method named `getInputStream`. In this method, the confidential data in stored in the `InputStream` object. So if an object of type `MsgDtoPV` is instantiated with the passing `InputStream` object, the object of type `PubViewer` have access to the confidential data and it is disclosed to the object of type `PrintWriter` which is an untrusted sink. The important point is `SecViewer` would be able to see both types of messages but `PubViewer` can only have access the `MsgDtoPV` messages (Fig. 15). FlowDroid misses the information disclosure in SecretViewer because it generates an unsound call graph. If the `getInputStream()` of an object of type `java.net.URLConnection` is set as a source, because of the unsound call graph, FlowDroid would not be able to find the source and the analysis will be aborted. If the `getInputStream()` of an object of type `Document` is set as a source, which is an

```
class Datacontainer{
 String value = "android";
 void setValue(String v){
  value = v;
 }
 String getValue(){
  return value;
 }
}
class ADatacontainer {
 DataContainer d1 = new Datacontainer();
 TelephonyManager tm = ...;
 void onClick(View v) {
  String imei = tm.getDeviceId(); //source
  Log.i("INFO",d1.value);
  d1.value = imei;
 }
 void onResume() {
  super.onResume();
  String imei = tm.getDeviceId(); //source
  SmsManager sms = ...;
  sms.sendTextMessage(d1.getValue()); //sink, no leak
  d1.setValue(imei);
}
```

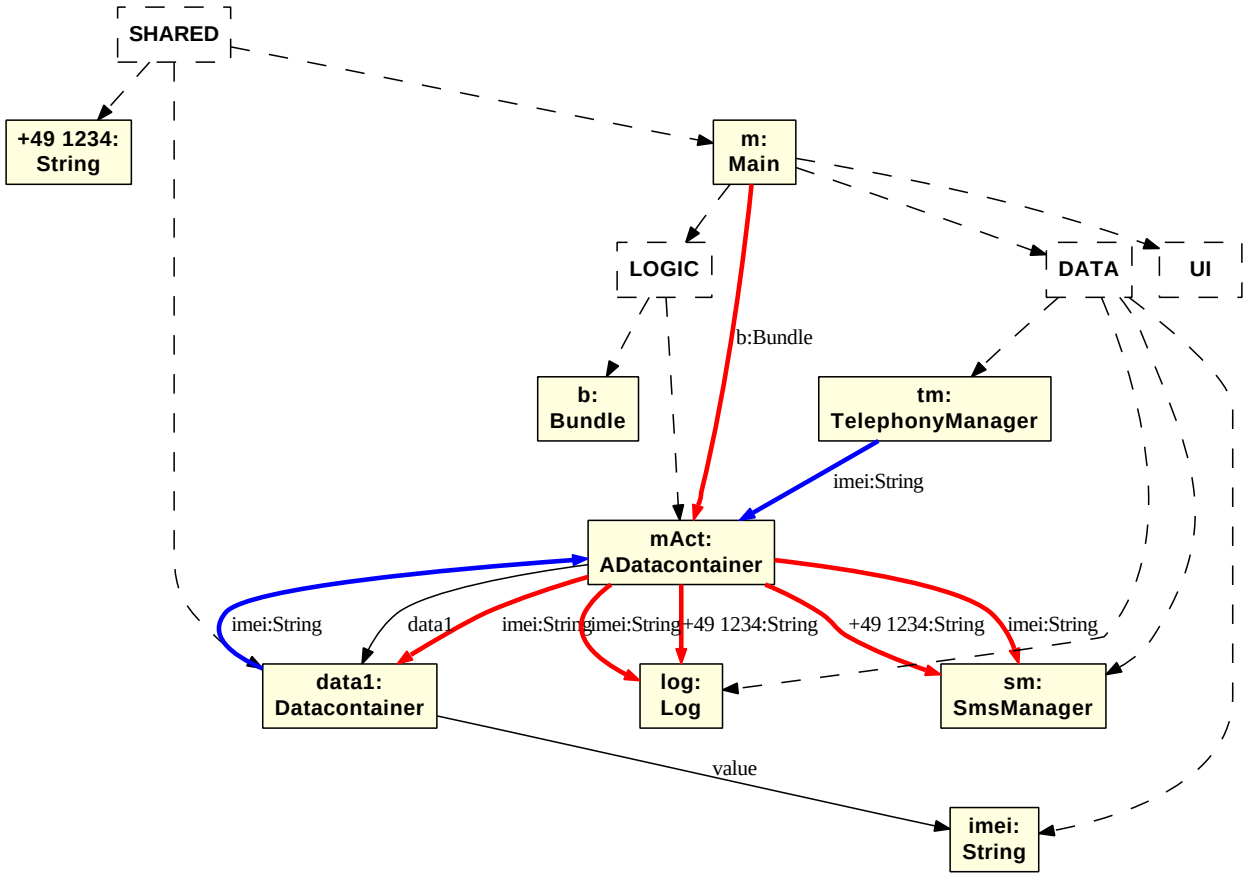


Figure 13: The imei flows to `Log` and `SmsManager` by invoking `onClick()` and `OnResume()` more than one time.

```
class DBase {
  Account account;
  Service eService = ...;
  Service dService = ...;
  void copy(File src, File dst){
    String mydata = getBytesFromFile(src);
    String dd = dService.run(mydata);
    InputStream is = new ByteArrayInputStream(dd.getBytes());
    account = new Account(is); //stream to account
    ByteArrayOutputStream os = new ByteArrayOutputStream();
    PrintWriter pw = new PrintWriter(dst);
    account.fPack(os); //account to stream
    String data = os.toString();
    String eData = eService.run(data);
    pw.println(eData);
  }
}

class Service{
    Cipher cipher;
    Service(String cType){
    cipher = new Cipher(cType);
  }
  String run(String text){
    String retVal = cipher.doFinal(text);
    return retVal;
  }
}

class Cipher{
  String ctype;
  Cipher(String cType){
    ctype = cType;
  }
  String doFinal(String bytes){
    String enc = new String();
    return enc;
  }
}
```

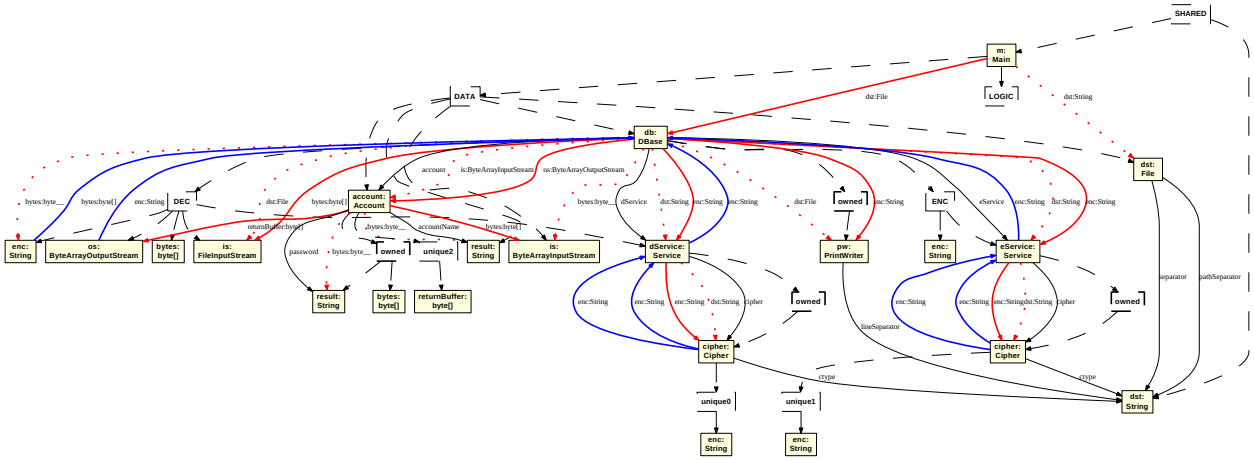


Figure 14: `doFinal` Method of `Cipher` encrypts decrypted data and decrypts encrypted one.

application specific source, FlowDroid manages to find the source but it is not able to find the vulnerability. Using indirect communication, Scoria is able to track an object of type `byte[]`, which flows from the object of type `PubViewer` to an object of type `FileInputStream`. This data is confidential because `SecViewer` also writes on the object of type `FileInputStream`. So in a constraint Scoria makes the object confidential. Then through creation edges of the object of type `PubViewer`, Scoria checks if any object of type `String` is created and flows to the object of type `PrintWriter` which is the untrusted sink. This is the case in SecretViewer and Scoria finds the vulnerability.

5 Discussion

Unsound call-graph reduces recall

The results for the commercial approaches indicate a lower recall for the Callbacks and Lifecycle equivalence classes. These approaches use a precomputed call graph. Extracting a call graph is challenging in the presence of callbacks that are invoked by a framework. The framework invokes these callbacks in a predefined order based on the lifecycle of an object. Security vulnerabilities exist for example when one callback stores confidential data into a field, and another callback reads the value of the field and discloses the confidential data into an untrusted sink. To overcome this limitation, FlowDroid models the lifecycle of an object of type `Activity` by adding the callbacks as entry points in the call graph and assumes that all the callbacks are invoked. This solution is specific to applications developed for the Android framework and the approach needs to be redesigned to be used on a different framework. So in ActivityLifecycle and LocationLeak test cases FlowDroid has a better recall than AppScan and Fortify. FlowDroid reports, however, a false negative for SecretViewer2, which uses the Model-View architecture. Instead of a precomputed call graph, the Scoria extraction analysis assumes that every method declared on an instantiated class may be invoked. This assumption increases the recall and reduces the computational complexity. The assumption is, however, a source of imprecision as the UnreachableCode test case highlights.

By filtering, approaches gains precision, but reduce recall

In some cases, FlowDroid sacrifices recall to gain more precision. First, by using hard-coded properties from configuration files, FlowDroid incorporates values read from configuration files to improve control flow sensitivity. If a class that extends `Activity` has the state `enabled` assigned `false` in the configuration file, FlowDroid cuts the portion of the control flow graph related to this class from the analysis. By using such heuristics, FlowDroid avoids a false positive in InactiveActivity, but fails to find the vulnerability in AActivity, where the state `enabled` is changed at runtime. Scoria is more conservative and assumes that all such classes may be instantiated at runtime. For some cases (e.g., InactiveActivity), Scoria returns false positives.

Some classes of security vulnerabilities are hard to find

Also there are some classes of security vulnerabilities that are hard to find. Four evaluated approaches are unable to handle ImplicitFlow test cases. In an implicit flow test case, there is a dependency between the the predicate of a conditional statement and the expression that is going to be execute base on the value of the predicate. For example, if an application allows unlimited login attempts and returns a binary response, by using brute-force attack based on a dictionary a malicious user can guess the passwords although the there is no explicit flow of the confidential password to an untrusted sink.

Summarization increases recall

By summarizing multiple runtime objects that have the same conceptual purpose with one conceptual representative, Scoria achieves higher recall. Otherwise to find such vulnerabilities, the value flow through library code or through files should be kept by the analysis. For Loop1 test case, there is an object of type `String`

```
class Document {
  void notifyObservers(){
    MsgDtoSV dToSV = new MsgDtoSV(getInputStream());
    //by mistake!
        MsgDtoPV dToPV = new MsgDtoPV(getInputStream());
        Listener ls = secListeners.get(0);
    ls.update(dToSV);
    ls.update(dToPV);
        Listener lp = pubListeners.get(0);
        l.update(dToPV);
  InputStream getInputStream(){
    try {
          URL oracle = new URL("http://www.oracle.com/");
          URLConnection conn = oracle.openConnection();
          InputStream inputStream = conn.getInputStream();
          return inputStream;
}

class PubViewer extends BaseViewer{
  void update(Msg msg){
    readStream(msg);
  }
}

class SecViewer extends BaseViewer{
  void update(Msg msg){
    readStream(msg);
  }
}

class Msg {
  InputStream inputStream;
  public Msg(InputStream is){
    inputStream = is;
  }
}
```

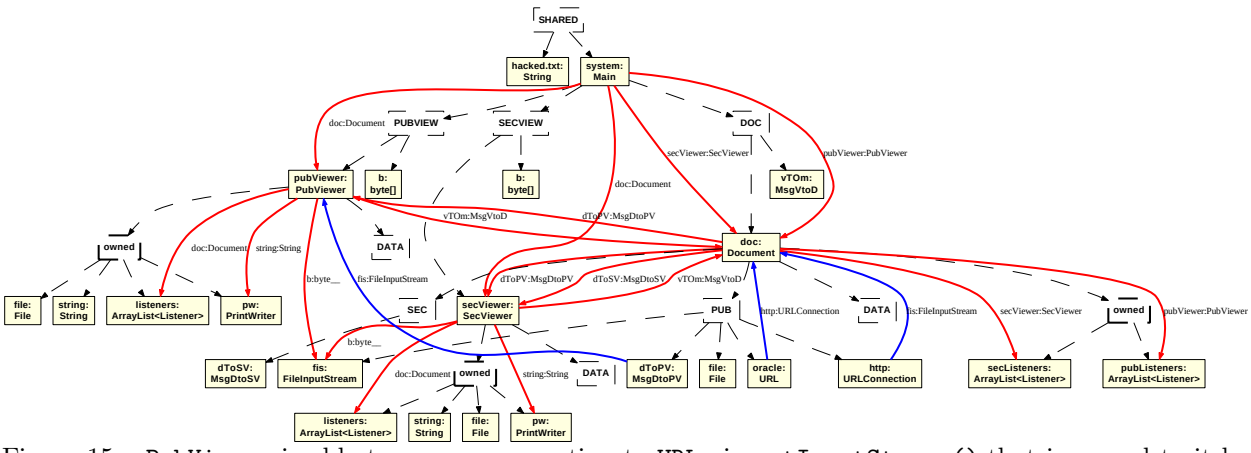


Figure 15: `PubViewer` is able to open a connection to `URL` via `getInputStream()` that is passed to it by mistake.

which contains the device identification number. This object is obfuscated in another object by adding an intermediate character between each two characters of initial object. Then the obfuscated object is sent to an object of type `SMSManager`, which is an untrusted sink. Because the obfuscated object still represents confidential data, the architect decides to put the initial object and obfuscated object in the same domain. By assigning `IsConfidential.True` to the representative of both objects, Scoria reports a true positive. FlowDroid and AppScan keep track of the transformations by propagating the security property from the argument of the method `concat` of the class `String` to the return value and report the vulnerability, while Fortify fails to report it.

Separating constraints from extraction increases recall

Approaches such as FlowDroid attempt to keep track of all variables and assignments that may occur at every step of the execution. This is challenging and becomes a source of missed vulnerabilities in the presence of frameworks and libraries. For the test IntentSink1, the confidential information is first passed to an object `i:Intent` which is then passed as an argument to the sink through some variables in the framework. FlowDroid fails to find this vulnerability since it does not analyze the framework. By using constraints based on object traceability and object reachability, Scoria reports the vulnerability. The constraint checks if the same information that leaves the source is passed to the sink through some other objects. By having the domain annotations, it is easier for Scoria to keep track of information flow because it merges objects of the same type in the same domain.

Object provenance increases recall

By using Scoria, architects have the flexibility of writing constraints in terms of object provenance. Object provenance is used to find the vulnerability in AChat, where the same object that flows from the application to the service should not flow from an external client to the service. In AChat, The service allows a registered user to search for other users of the application according to their phone number, in a database. If a malicious client would be able to send a request to the service including fake numbers, it would have access to a list of valid telephone numbers of the users. The important point here is the service is not explicitly disclosing phone numbers and disclosure can be happened by abusing of an existing feature of it. To find such vulnerability, architects should have a degree of flexibility to write constraints which may not be considered by designer of a security approach like FlowDroid.

Soundness does not generate an excessive number of false positives

Scoria is domain-sensitive, but object- and flow-insensitive. The results of Scoria are comparable to the ones of FlowDroid which is object-sensitive and indicate that the precision provided by domains is sufficient to address the aliasing challenge. Domain-sensitivity compensates precision for Scoria and on average one out of five vulnerabilities is a false positive (85%). The imprecision of Scoria is due to flow-insensitivity. For the test FieldSensitivity4, the order in which statements are executed is important since the confidential data is written into a field after the value of the field is logged. Also for the test ObjectSensitivity2, the confidential value is assigned to a variable and a field and then a default value is overwritten into the variable and the field. Eventually the field and the variable are written to an untrusted sink. For other test cases, Scoria compensates by distinguishing between objects of the same type in different domains. Then, Scoria is more precise and avoids the false positive reported by FlowDroid. For Button2, a variable `imei` is assigned null before is concatenated to a constant. In Scoria, the architect inspects the code and fine-tunes the annotations such that the result of concatenation is in `SHARED` while the object `imei:String` is in the domain `DATA`.

Object provenance and indirect communication increase precision

Scoria allows architects to write constraints in terms of object provenance and indirect communication to increase the precision. To handle DroidBench test cases, it is enough to use object transitivity and object reachability. But Scoria has other features that make is able to handle more complicated test cases. For

example for ASocket, the architect writes a constraint using object provenance and indirect communication and checks that no confidential object flows to a descendant of an object of type `Socket`, which does not provide encryption. There are two objects of the same type `SocketOutputStream`, one of them is s descendant of an object of type `Socket` and the other one is a descendant of an object of type `SSLSocket`. Using object hierarchy, Scoria avoids a false positive that FlowDroid reports.

Application-specific security properties improve precision

FlowDroid requires the architects to supply the list of sinks and sources. In general, these are methods from the libraries and frameworks and the lists are reused for all test cases in DroidBench. In some particular cases in the extended benchmark, the sources are specific to the test case. For example, in the test case ACipher, the source is the method `decrypt` of the class `Service`. The selection of this source influences the result of the approach. If the architects select the method `doFinal` of the class `Cipher`, FlowDroid reports a false positive.

Using abstract objects to assign security properties leads to a more precise analysis

Using an abstract object to assign security properties instead of method declaration may provide higher precision. For the test case ASocket, FlowDroid sets `getOutputStream()` method of an object of type `OutputStream` as a sink. This assumption leads to a false positive for `getOutputStream()` method of the object of type `OutputStream` which is a descendant of `SSLSocket`. Scoria assigns security property to `Socket` and now it is distinguishable from `SSLSocket`. So Scoria avoids the false positive.

5.1 Threats to Validity

There are several threats to validity to our evaluation.

Android-specific. The test cases in the benchmark are all Android mobile applications, so the findings may not be broadly applicable to all kinds of applications. However in the approaches that are specific to mobile apps, one can think of these mobile apps as a framework that is built on Android framework. So the findings may generalize to most framework applications that use features such as callbacks.

Micro-benchmark. The test cases in the benchmark are all very small, consisting of fewer than 100 lines of non-comment, non-blank, lines of code, fewer than 5 classes, and fewer than 5 methods, so they do not create interesting networks of objects or interesting object hierarchies in the object graph. Having a richly connected object graph is particularly important for Scoria, which tends to bring out security vulnerabilities that involve networks of objects, reachability and indirect communication through the object hierarchy. Most of the approaches in this paper, including Scoria, were also separately evaluated on larger systems. To mitigate this threat, future work will consider including larger systems into the benchmark.

Experience of architects. The expertise and experience of the architects using the approaches to find security vulnerabilities plays a role. This issue is more pronounced with an approach such as Scoria that is not a push-button approach. In Scoria, human judgment is involved in adding annotations, writing constraints and interpreting the results. During this evaluation of Scoria, the architects were not very experienced but got frequent help and support from the original Scoria designers.

5.2 Limitations

Static analyses only. All the approaches evaluated in this paper use static analysis only. A future evaluation should include approaches based on dynamic analysis also, type systems.

Common limitations of a static analysis.

Handling reflection

In particular, none of the evaluated approaches handle reflection directly. In Scoria, the architects can summarize the effects of reflection using annotations by defining virtual object allocations or virtual fields that lead to additional objects or points-to edges between abstract objects.

Working with collections

None of the evaluated approaches distinguish between different elements of a collection. For the test case ListAccess1, all four approaches report a false positive. This is a common limitation of static analysis methods.

Measures of effort. One dimension that is lacking in this paper is measuring the effort involved in applying each of the approaches. This information was not provided in the original benchmark. Also, since the Scoria evaluators were still learning to use the approach and the tool, it did not make sense to measure the effort. In the case of Scoria, the effort of adding annotations and writing constraints can be significant, so is worth measuring. If Scoria outperforms some of the other approaches in some cases only, it would be interesting to study whether the improvement in terms of fewer false positives or higher precision is worth the extra effort.

6 Related Work

The section discusses related security benchmarks and the alternatives that researchers use to evaluate the 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 [20] 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 as well.

MalGenome [32] is a collection of 1200 malware Android applications. This paper focuses on approaches that find vulnerabilities in legitimate well-intended applications. The approaches may not be able to detect malware, which often avoid detection by taking advantage of known limitations of static analyses.

The CERT secure coding standards [22] 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 [14], Coverity [5], and FindBugs [29] 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 tests with which this paper extends DroidBench, like ASocket and ARuntime, are also inspired by 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 [24]. SecuriBench [21] 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 [19] use SecuriBench to evaluate their approach that handles both explicit and implicit information flow.

InsecureBank [25] 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. [8] 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 [7, 10] 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 a similar functionality, but miss other classes of vulnerabilities. For example, Scoria finds a vulnerability in an open-source Android application, Universal Password Manager [30]. Fahl et al. [9] 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 without explicit permissions, thus potentially exposing the password to eavesdropping.

7 Conclusion

In this paper, we compared one static analysis that uses annotations and constraints to other approaches that use only static analysis to find security vulnerabilities. Our findings are useful to security researchers when identifying gaps in the solution space, and to security practitioners when deciding which tools to use.

We also showed there is value in evaluating newly developed tools against previous, more established tools, and in extending an existing benchmark to cover more classes of security vulnerabilities. Future work will continue building on this work by evaluating more tools (adding more columns to Table 2) and more test cases (adding more rows).

Acknowledgements

The authors thank Sumukhi Chandrashekar and Dajun Lu for their early contributions to this project.

References

[1] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, 2004.

[2] M. Almorsy, J. Grundy, and A. Ibrahim. Supporting automated vulnerability analysis using formalized vulnerability signatures. In *ASE*, 2012.

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

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

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

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

[7] 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*, volume 10, 2010.

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

[9] 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.

[10] Y. Falcone, S. Currea, and M. Jaber. Runtime verification and enforcement for android applications with rv-droid. In *Runtime Verification*, 2013.

[11] 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.

[12] A. Fuchs, A. Chaudhuri, and J. Foster. SCanDroid: Automated Security Certification of Android Applications. Technical report, Univ. of Maryland, 2009. `http://babu.cs.umd.edu/~avik/projects/scandroidascaa/paper.pdf`.

[13] S. Holavanalli, D. Manuel, V. Nanjundaswamy, B. Rosenberg, F. Shen, S. Y. Ko, and L. Ziarek. Flow Permissions for Android. In *ASE*, 2013.

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

[15] IBM. T.J. Watson Libraries for Analysis (WALA). `http://wala.sourceforge.net`, 2012.

[16] IBM. AppScan Source, 2013.

[17] 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.

[18] L. LaPadula and D. Bell. Secure computer system: Unified exposition and multics interpretation. Technical report, Hanscom Air Force Base, 1976.

[19] 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*, pages 146–155, 2010.

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

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

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

[23] MathWorks. Polyspace, 2012.

[24] OWASP. WebGoat Project, 2013.

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

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

[27] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. ANDROMEDA: Accurate and Scalable Security Analysis of Web Applications. In *Conference on Fundamental Approaches to Software Engineering (FASE)*, 2013.

[28] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: effective taint analysis of web applications. In *PLDI*, pages 87–97, 2009.

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

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

[31] VisualVM. Analyzing a Heap Dump Using Object Query Language (OQL), 2013.

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