Model Generation For Java Frameworks

Linghui Luo*¶, Goran Piskachev*¶, Ranjith Krishnamurthy†, Julian Dolby‡, Eric Bodden†§, Martin Schäf*
*Amazon Web Services, †Fraunhofer IEM, ‡IBM Research, §Paderborn University, ¶The work was done prior to Amazon.

***Abstract*—Modern applications often rely on rich frameworks to provide functionality. Android, for instance, handles many aspects of building a mobile app. But these frameworks also have costs. Given the importance of application security and tools to ensure it, one major cost is that framework complicate tools based on static analysis: (1) They hurt analysis quality by including large amounts of complex, dynamic, and native library code. (2) Frameworks like Android become the main program, making whole program analysis of the app problematic.**

Mechanisms such as Averroes have been developed to handle unknown library code for Java, and have proven effective for some analyses. However, they have two main limitations in the context of our complications: (1) They do not provide the precision required for security analysis. (2) They assume a main program, which is not the case for frameworks. To address this, we present GenCG, which extends Averroes to support taint analysis for Android and Spring. Evaluation with real-world Android applications shows that call graphs using the models generated by GenCG cover significantly more code of the app, improves recall of a client security analysis, and, at the same time, does not introduce more false positives.

***Index Terms*—static analysis, call graph, framework modeling**

I. Introduction

Static program analysis is used in a wide variety of applications, such as optimization [1], [2] and security analysis tools [3], [4]. Many techniques exist, and many of them rely on a call graph, i.e., a graph of which function is called at each call site. Call graphs are often built using a fixed-point algorithm: start with code known to be called, e.g., the main method in Java, and add code that it calls. This continues until no more code is added. Once a call graph has been obtained, many analyses are built on top of it.

Framework applications complicate such analysis in at least two ways: by comprising relevant code that is intractable for analysis, and by muddying the notion of what the program actually is. Many modern frameworks use dynamic features, e.g., constructing a string and then using it as a class name in Java. Unless the analysis is able to precisely interpret the operations that construct the string, it will not be able to understand what class is being used and hence will not be able to determine the relevant code. Furthermore, many frameworks use code in multiple languages, e.g., JNI calls to C code from Java. Most analysis frameworks only handle one language. And, for the code can be determined, there is often just too much to be analyzed in a scalable fashion. Thus, most static analysis tools model frameworks rather than analyze.

Modern frameworks also complicate the notion of a program: many modern programs are, in effect, clients of a framework, e.g., think of a servlet that is installed into Tomcat Web server. These programs have entry points that expect to be called from the framework, and finding these entry points depends on the framework. Sometimes, as with Spring [5], the framework expects user code to be annotated in specific ways that will cause the framework to call it. Sometimes, there are specific interfaces that must be implemented by user code to signal that specific code should be called from the framework. Often the user code makes a call into the framework, and then the framework calls back into the user code in a similar way. Thus the program becomes a collection of pieces knitted together by the framework. A prominent example of such a framework is Android [6].

A common approach to handling this kind of framework is to build a custom model that attempts expose the relevant semantics of the framework with respect to the program. For Android, for example, such a model will add entry points to all methods that implement appropriate interfaces. But this is never-ending, since there are many frameworks and because each framework itself keeps changing. As a result, the model has a constant struggle to model the semantics, and thus the analysis can easily miss many real issues. Mechanisms such as Averroes [7] have been developed to tackle this situation in dealing with Java libraries by avoiding much manual modeling of the library code. Averroes analyzes the original library to create a very coarse model that attempts to over approximate library behavior. Potentially this results in a conservative analysis. There are compromises needed to not look at the framework, and, while effective for some analyses, such mechanisms have not been designed to support the precision required for security analysis, especially the need for flow-sensitivity and field-sensitivity. While frameworks usually provide rich library functionalities, they are different than libraries. Unlike libraries, frameworks commonly use the *inversion of control* principle which Averroes fails to handle because it involves the framework being the main program.

To support security analyses in the context of handling modern Java frameworks, we present GenCG, which extends Averroes with improvements to its model to model execution order as needed for flow-sensitivity and, at the same time, restricting its model to give up on analysis completeness and focus on code that is more likely to result in analysis issues. An evaluation with two real-world benchmark suites—TaintBench [8] and F-Droid [9]—for Android taint analysis shows that call graphs using the model generated by GenCG cover significant more code (up to 4 times). As a result, it greatly enhances recall of true analysis results on a client taint analysis for Android, namely FlowDroid [10], compared to FlowDroid's hard-coded model. At the same

```
class MainActivity extends Activity {
  Msg msg = new Msg();
  public void onStart() {
    Location loc = lm.getLastKnownLocation("network"); // source
    msg.setContent(loc.toString());
  }

  public void onPause() {
    super.onPause();
    Intent intent = new Intent(this, TaskService.class);
    intent.putExtra("data", this.msg);
    startService(intent);
  }
}

class TaskService extends Service {
   LooperThread looperThread;
   Handler handler;
   public int onStartCommand(Intent intent, int flags, int startId) {
      handler = looperThread.handler;
      Msg m = (Msg) intent.getSerializableExtra("data");
      Message msg = handler.obtainMessage(1000, m);
      handler.sendMessage(msg);
      return super.onStartCommand(intent, flags, startId);
   }
}

class LooperThread extends Thread {
  Looper looper;
  Context context;
  Handler handler;
  public void run() {
    handler = new PushMessageHandler(context, looper);
  }
}

class PushMessageHandler extends Handler {
  String url = "http://103.30.7.178/upMsg.htm";
  public void handleMessage(Message msg) {
    List<NameValuePair> pars = new ArrayList<>();
    pars.add(new BasicNameValuePair("loc", new
         Gson().toJson(msg.obj)));
    httppost.setEntity(new UrlEncodedFormEntity(pars, "UTF-8"));
    httpclient.execute(httppost); // sink
  }
}
```

Listing 1. Motivating Example

time, our approach does not introduce more false positives. To demonstrate GENCG's applicability to other frameworks, we experiment with the Spring framework [5] and shows its effectiveness with evaluating on a micro benchmark suite with 42 Spring applications contributed by us along with the paper. Our artifacts are publicly available under https://doi.org/10.5281/zenodo.7553965.

II. BACKGROUND

Client analyses are the best judge of the utility of a given call graph, and we focus on taint analyses, which are effective in detecting many security vulnerabilities [11], [12], [13], [8]. Here, we introduce our chosen taint analysis tool, FlowDroid [10], which is primarily designed for detecting privacy leaks in Android apps. FlowDorid performs a field- and flow-sensitive data-flow analysis using the IFDS solver [14] built on top of the Soot framework [15]. The solver propagates

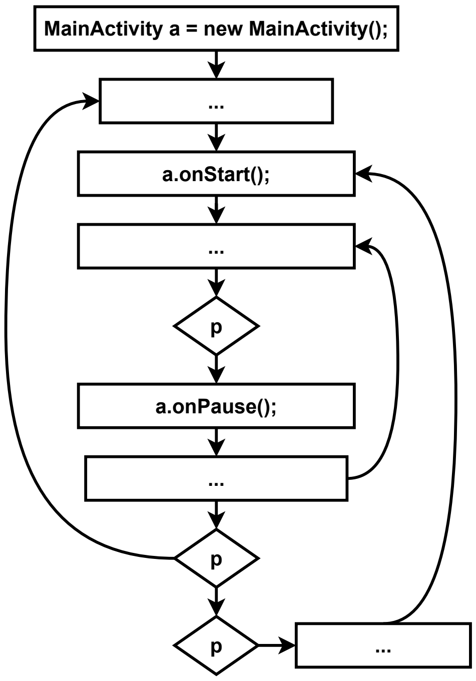

Fig. 1. CFG of `MainActivity` in FlowDroid's `dummyMainMethod()`

access paths throughout the app's inter-procedural control-flow graph. For inter-procedural analysis, by default, FlowDroid computes the call graph via the SPARK algorithm built-in the Soot framework [2], which is the most precise call graph algorithm that is available in Soot [16]. SPARK computes points-to sets to resolve the runtime type of the receiver of a polymorphic call, hence the call graphs are more precise than naive approaches such as Class Hierarchy Analysis (CHA) or Rapid Type Analysis (RTA).

FlowDroid constructs a hard-coded model of the interactions between the app and the Android framework. For calculating alias information, FlowDroid reuses the same infrastructure for the data-flow analysis by encoding the alias algorithm as an IFDS problem propagating access paths. As any taint analysis, FlowDroid starts tracking the taints created as source statements, and reports them as sensitive statements, called sinks. The list of sources and sinks are pre-defined Android API calls relevant for detecting privacy leaks, e.g., the source `getLastKnownLocation()` in Listing 1 returns the sensitive information, i.e., a user's last location, and if this data reaches the sink `HTTPClient.execute()`, i.e., sending HTTP Post, it should be reported as a data leak. The list of sources and sinks in FlowDroid is configurable. Hence, users may use the tool to detect other vulnerabilities in Android apps, e.g., SQL injections.

Due to its modular and open architecture, many of the components in FlowDroid can be exchanged with other Soot-compatible components. This includes new call graph algorithms, alias algorithms, source/sink specifications, etc. We use this capability to run GENCG with FlowDroid's taint analysis in our evaluation (Section IV).

We introduce the FlowDroid model with a motivating example. Listing 1 shows a non-trivial data leak from a malware app. Two Android components are involved in this leak: an activity `MainActivity` and a service `TaskService`. In this leak, Inter-Component Communication (ICC) exchanges sensitive data between the activity and the service.

The activity `MainActivity` first reads the user's last known location in the lifecycle method `onStart()` and then stores it into the field `msg` (Line 4-5). Later, in the lifeycle method `onPause()` of the activity, the `msg` containing the location is passed to an intent that starts the service `TaskService` (Line 11-12). `TaskService` uses the handler `PushMessageHandler` to perform asynchronous tasks. The `msg` containing the location is read from the intent and is encapsulated in a `Message` object for the handler. This `Message` object is sent by `sendMessage(Message)` (Line 23), and then processed later when `handleMessage(Message)` is called by the Android framework. In the `handleMessage(Message)` method, the last known location is leaked to a malicious server via a HTTP Post (Line 43).

To detect this leak, analysis tools need a sound call graph that captures all the calling relationships on the data-flow path of this leak. It must also model control flow of the lifecycle callback methods (e.g. `onStart()`, `onCreate()`) inside each Android component and the data exchange supported by ICC. To do so, FlowDroid creates a `dummyMainMethod()` that models the lifecyle for each Android component detected in the app. For example, Figure 1 shows the control-flow graph (CFG) of `MainActivity` in FlowDroid's `dummyMainMethod()`. Note the methods `onStart()` and `onPause` involved in the leak are called in the appropriate order. FlowDroid models control flow using an opaque predicate p, i.e., a predicate which cannot be evaluated statically. Therefore, at each branch on p, the analysis follows both possible paths. This is how FlowDroid models direct interactions between Android and the app. However, there are indirect interactions too: Android invokes callback from the "outside", such as `handleMessage(Message)`. Not modeling such callbacks leads to missing edges in the call graph. This is why in the example FlowDroid fails to detect the data leak.

As illustrated above, FlowDroid's model is often incomplete, but it is also hard to keep it current with development of the Android framework itself. Moreover, the approach is limited to the modeled framework and often tailored only for one specific analysis tool. Other analysis tools would need to either build on top of FlowDroid or implement their own model generation. And Android is not the only popular Java framework. In the enterprise domain, there are Java EE, Spring, Apache Struts. These frameworks are rarely supported by static analysis tools due to their complexities [17]. Thus, one really desires a *reusable* approach to modeling Java frameworks. Such an approach requires two components:

1) a framework-independent core that can be easily reused across several frameworks, and

```
class Library extends AbstractLibrary {
  static {
    Library library = new Library();
    AbstractLibrary abstractLibrary = AbstractLibrary.instance;
    abstractLibrary.libraryPointsTo = library;
  }
  void doItAll(){
    Library library = AbstractLibrary.instance;
    /* class instantiation */
    TaskService r1 = new TaskService();
    library.libraryPointsTo = r1;
    MainActivity r2 = new MainActivity();
    library.libraryPointsTo = r2;
    PushMessageHandler r3 = new PushMessageHandler();
    library.libraryPointsTo = r3;
    LooperThread r4 = new LooperThread();
    library.libraryPointsTo = r4;
    //...
    /* libarary callbacks */
    Handler r5 = (Handler) library.libraryPointsTo;
    Message r6 = (Message) library.libraryPointsTo;
    r5.handleMessage(r6);
    Service r7 = (Service) library.libraryPointsTo;
    r7.onCreate();
    Service r8 = (Service) library.libraryPointsTo;
    Intent r9= (Intent) library.libraryPointsTo;
    r8.onStartCommand(r9, 1, 1);
    Activity r10 = (Activity) library.libraryPointsTo;
    r10.onPause();
    Activity r11 = (Activity) library.libraryPointsTo;
    r11.onStart();
    //...
  }
}
```

Listing 2. `Library` class generated by original Averroes with extended Android front end.

2) a standalone tool that generates a model for a certain type of static analysis tool, e.g., taint analysis tools.

In the next section, we introduce how our approach realizes these two components.

III. Approach

GenCG is inspired by Averroes [7], which generates a placeholder library that over-approximates the original Java library. As Figure 2 shows, GenCG takes both the application code and the framework code as input. It outputs two files: `instrumented-app.jar` contains classes in the original application as well as the instrumentation which will be introduced later in this section in Improvement 4; `placeholder.jar` is the model of the framework. One can then take either the `instrumented-app.jar` or the original application app (if there is no instrumentation happened as described in Improvement 4) together with the `placeholder.jar` as input for call graph construction and further analysis. In the following, we first briefly introduce Averreos, then we focus on describing the limitations of Averroes and how GenCG addresses them.

Averroes relies on the *separate compilation assumption*, that is, all library classes are compiled in the absence of the application classes, and hence cannot reference them explicitly, and this limits how the library classes can interact with the application. For example, it limits what kinds of

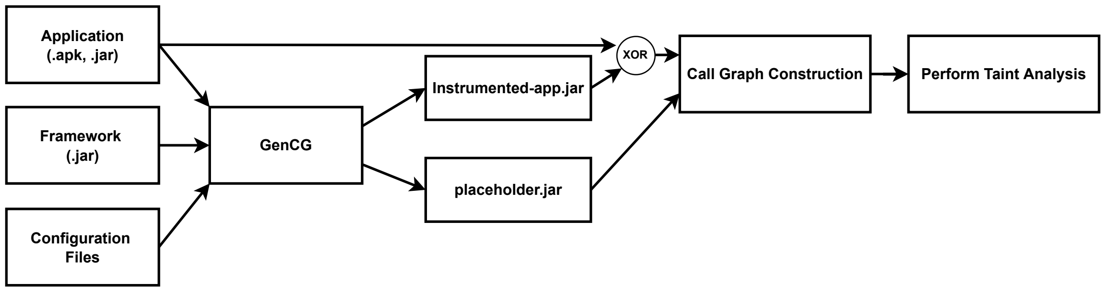

Fig. 2. Overview of applying GENCG for taint analysis.

```
public class averroes.AbstractLibrary extends java.lang.Object{
        public java.lang.Object libraryPointsTo;
        public static averroes.AbstractLibrary instance;
        public abstract void doItAll(); //...
}
```

Listing 3. The `AbstractLibrary` class generated by original Averroes.

```
public class averroes.AbstractLibrary extends java.lang.Object{
        public android.os.Handler LPT_1;
        public android.os.Message LPT_2;
        public android.app.Service LPT_3;
        public android.content.Intent LPT_4;
        public android.app.Activity LPT_5;
        public android.os.Bundle LPT_6;
        public java.lang.Thread LPT_7;
        public java.lang.Runnable LPT_8;
        //...
        public static averroes.AbstractLibrary instance;
        public abstract void doItAll(); //...
}
```

Listing 4. The `AbstractLibrary` class generated by GENCG.

```
public class Handler{
    void sendMessage(Message m){
        AbstractionLibrary lib = AbstractLibrary.instance;
        lib.libraryPointTo = this;
        // taint the libraryPointTo field
        lib.libraryPointTo = m;
        lib.doItAll();
    }
}
```

Listing 5. Placeholder method for each original library method.

```
public class Library extends AbstractLibrary{
    public void doItAll(){
        // ...
        Message m1 = new Message();
        Library.libraryPointsTo = m1 ; // strong update!
        // ...
        Message m2 = (Message) Library.libraryPointsTo;
        handler.handleMessage(m2);
    }
}
```

Listing 6. Problem caused by repeatedly created objects.

objects a library class can hold references to, which methods in the application can be called by the library and under which circumstance. The authors of Averroes formulate eight constraints in the paper. In the following, we introduce two most important constraints:

a) Local Variables: Local variables in the library can point to objects of application classes that are (1) instantiated by the application and then passed into the library (2) stored in fields accessible by the library code, or (3) the runtime type of which is a subtype of `java.lang.Throwable`. Based on this constraint, Averroes uses a single field to abstract all the local variables in the library.

b) Method Calls: A method can be called if it is non-static, overrides a method of a library class, and the library can reference an object of its class or a subclass. This constraint basically says that there should be a subtyping relation between the application class and the library class.

Averroes builds a class hierarchy based on both application classes and the original library classes. It uses this class hierarchy to identify classes that satisfy the constraints and generates a `Library` class with a `doItAll()` method. This `doItAll()` method implements potential library behaviors such as class instantiation and library callbacks. For example, any application class that satisfies the *Local Variables* constraint would be initialized in the `doItAll()` method and assigned to the `libraryPointsTo` field. Averroes calls library callback methods in `doItAll()` based on the subtyping relationship between the application classes and library classes described in the *Method Calls* constraint.

Averroes also generates placeholder code: It (1) assigns each callable method parameter to the `libraryPointsTo` field; (2) calls doItAll() method of the `Library` class. This is possible, as in the placeholder library, a piece of generated code initializes the `Library` class to be a singleton (see Listing 2); (3) returns `libraryPointsTo` if the original library method has a return type. Listing 5 shows the placeholder method for a library method `Handler.sendMessage(Message)`.

Averroes does address the two requirements we introduced in last section, but it ultimately is not precise enough for frameworks and a good taint analysis. Next, we explain the specific limitations of Averroes when using the placeholder library as replacement for the Android framework for taint analysis. For each limitation, we introduce the improvement we make in GENCG:

Limitation 1. Averroes uses a single `libraryPointsTo` field to represent all objects that the library references, which is too imprecise for a field-sensitive taint analysis. Once the `libraryPointsTo` field is tainted, it will be propagated everywhere and potentially result in many false positives, since the `Library.doItAll()` method is called in every placeholder method.

Improvement 1. To address this limitation, we introduce typed `libraryPointsTo` fields: instead of only using one field (`libraryPointsTo` in Listing 3) for all objects, for each type of object that library could point to, we create a field of this type in the `AbstractLibrary` class. We name such typed fields with names starting with the prefix `LPT`. Listing 4 shows our version of `AbstractLibrary` class for the motivating example. This can be unsound if objects are casted to other types in the application, i.e., a typed `LPT` field might hold references of other types of objects. But it is more precise than in the original Averroes, as it does not pollute objects of other types when a typed `LPT` gets tainted.

A more fine-grained separation such as allocation sites is not possible without a deep analysis of both the application and library code, because allocation sites of objects that library can point to can be either in the application code or framework code. The Android SDK with stub methods would not be sufficient. Technically, it is also very hard to generate the placeholder library methods, as a method could be called on different objects and with different arguments. Although our type-based separation might still introduce false positives, it turns out to work well for detecting real-world taint flows later in our evaluation. Additionally, the Android SDK with stub methods is sufficient for GENCG to create a model, as it allows to build class hierarchy and resolve method signatures.

Limitation 2. Framework-based applications usually do not have a main method, as the application's flow of control is usually dictated by the framework, which is known as *inversion of control*. One could use the `Library.doItAll()` method generated by Averroes as the main entry point for analysis. However, `Library.doItAll()` contains no control flow at all, instead callbacks are invoked in arbitrary order, e.g., the method `onStart()` containing the source is called after `handleMessage()` containing the sink is Listing 2. An inter-procedural flow-sensitive taint analysis would miss the data leak in the motivating example.

Improvement 2. To address **Limitation 2**, we introduce control flow in the `doItAll()` method as shown in the CFG Figure 3 for the motivating example. To make our approach as general as possible for multiple frameworks, we do not precisely model which methods should be called in which order, as it requires to study each specific framework documentation carefully. Our goal is to add control-flow edges such that every possible call sequence is covered. This means also unrealizable call sequences, which could potentially introduce false positives. This is the trade-off for independence from framework details. We introduce three kinds of edges with if-statements using an opaque predicate *p*: skip-method edges (blue), skip-class edges (red) and loop edges (green). Usually, not every callback method is called every time when a library method is called as the placeholder method does, the skip-method edges allow cases that a callback method is not called. Similarly, all calls to methods of a class can be skipped by the skip-class edges. To simulate the effect that a method can be called multiple times in the framework, we introduced the loop edges. The combination of skip-method edges and loop edges ensures that all possible orders of method calls are captured, including the lifecycle of Android component classes. For instance, in the part where service is simulated in Figure 3, although the `onStartCommand()` appears first in the control-flow graph, the path along labeled edges `A-B-F-G-H-J-A-D-E-F-C` represents the execution path in which `onCreate()` is called before `onStartCommand()` as defined in the service's lifecycle. The data leak in our example is on one of the paths reflected in the CFG of `doItAll()` generated by GENCG.

Limitation 3. Repeatedly created objects could lead

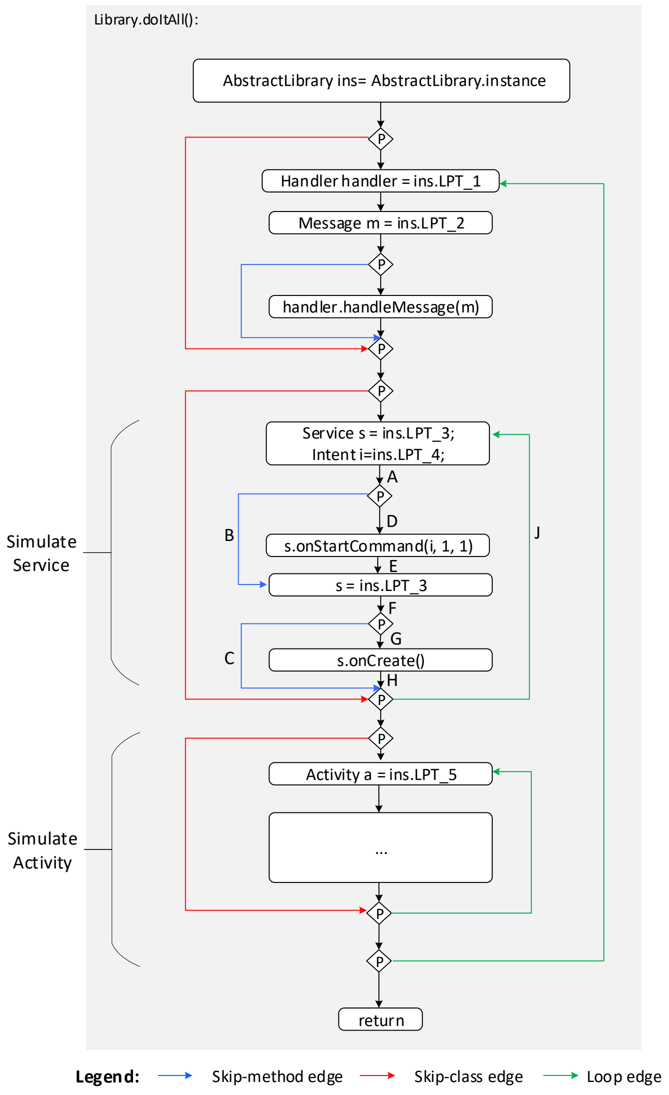

Fig. 3. The CFG of `Library.doItAll()` by GENCG.

to false negatives. Listing 5 shows the placeholder method generated by the Averroes for the library method `Handler.sendMessage`. As in the motivating example, a field-sensitive taint analysis will taint the parameter of `Handler.sendMessage`, then the `libraryPointsTo` field ought to be tainted when it analyzes the placeholder method. This tainted field will be then propagated into the `Library.doItAll()` method as it is called in every placeholder method. In Listing 6, the `libraryPointsTo` field needs to stay tainted when it is assigned to a local variable `m2` in `Library.doItAll()` such that `m2` is tainted when it is passed to the call `handler.handleMessage` (the sink is in this method). However, in `Library.doItAll()`, the `libraryPointsTo` field is overwritten by a newly created `Message` object. This means the points-to relationship needs to be removed and the `libraryPointsTo` field should not be tainted any more. This is the so called strong update [18], a precise taint analysis usually considers this. As a result, the data leak would be undetected.

Improvement 3. To address this, we move class instantiation from `doItAll()` to a separate `main()` method in the `Library` class to avoid unnecessary strong updates. The `doItAll()` method only contains calls library callbacks. The `main()` method will be taken by popular analysis frameworks by default as entry point and is only called once in call graph construction. In Android, most objects of application classes are only created once by the framework.

Limitation 4. Averroes does not consider Java annotations and results in missing edges in call graphs. In Java enterprise frameworks such as the Spring framework, annotations are commonly used to declare application code that will be called reflectively. One can use annotations to declare entry point methods (e.g. `PostMapping`) and even class fields that need to be initiated (e.g. `Autowired`) by the framework.

Improvement 4. To address this, GENCG handles class, method and field annotations that are supported by the framework. These annotations (annotation class signatures) are stored in configuration files used by GENCG and can be extended easily. For annotated methods and classes, GENCG creates artificial interfaces and instruments annotated classes to be subtypes of these interfaces. These artificial interfaces declare annotated methods and can be seen as a replacement for the annotations. This way it reduces the problem to the resolution of subtyping relationship that can be handled by the original Averroes. GENCG will generate invocations of those annotated methods in the library callbacks part of the `doItAll()` method.

We briefly explain how GENCG could be applied for supporting beans in the Spring framework. In Spring, objects that are created and managed by the framework are called beans. Beans can be defined both via XML configuration, or in Java via annotations and code. Spring uses mainly three annotations to resolve dependencies: `@Autowired` (defined in Spring), `@Inject` (defined in JSR-330) and `@Resource` (defined in JSR-250). They work similarly. They can be used to annotate class properties (fields), constructors and setters. Spring will then create objects for those annotated fields. To handle beans, GENCG scans fields and methods in application classes that are declared with these autowiring annotations and do the following:

- For each field that is autowired, GENCG instruments the default constructor to instantiate the field. If the type of the field is a concrete class, an object of this type will be created with default property values and assigned to the annotated field. If the type of the field is an abstract class or interface, for each concrete subclass, an object will be created and assigned. Such subtying relationship is obtained from the class hierarchy GENCG computes.
- The annotated constructor and setter are regarded as annotated callback methods by GENCG. Invocations of them are generated in the `Library.doItAll()` method To stay general for multiple frameworks, GENCG does not search the concrete property values of each bean defined with the annotation `@Bean` or in the XML configuration. It uses some default values for primitive types and `null` for reference types. We lose precision by doing this, however, call edges to methods of autowired objects will be captured in the call graph construction using the instrumented app.

In addition to these four main improvements, we remove the array elements writes and exception handling parts from the `doItAll()`. Our version of `doItAll()` only keeps the library callbacks part, which is the main feature of Java frameworks. This restricts the model to give up on analysis completeness, yet let us focus on code that is more likely to result in taint analysis issues.

IV. EVALUATION

We evaluate GENCG using the taint analysis of FlowDroid as a client. We evaluate how the generated model for Android, i.e., the `placeholder.jar` of the Android SDK, can be used to improve call graphs constructed by Soot's SPARK algorithm and its impact on FlowDroid's taint analysis. We do not compare GENCG to Averroes because Averroes does not support Android at all. Our evaluation answers the following research questions:

RQ1. How good are the call graphs using the model generated by GENCG in comparison to using FlowDroid's model?

RQ2. How does the model generated by GENCG impact FlowDroid's taint analysis?

A. Experimental Settings

We compare GENCG's model to FlowDroid's model using two real-world benchmark suites:

- TaintBench [8]: an Android malware benchmark suites for benchmarking static taint analysis. It consists of 39 Android malware apps and 203 precisely documented true-positive and 46 false-positive taint flows. For each benchmark app, there exists also a list of sources and sinks for taint analysis tools to configure.

- F-Droid [9]: A dataset of 30 open-source real-world apps from F-Droid used by Mordahl et al. in their evaulation of Android taint analysis tools [9]. This dataset also contains a documentation of 63 true-positive and 693 false-positive taint flows.

For each app, we do the following:

- FlowDroid (version 2.7.1): the app's apk file and the Android SDK platform jar are used as input for FlowDroid.
- GenCG-FlowDroid: we first use GenCG to generate the `placeholder.jar` file as replacement for the Android SDK platform jar. The `placeholder.jar` file and the app's apk file are then used as input for FlowDroid. We modify FlowDroid to use these two files to construct call graph without generating its own `dummyMainMethod()`.

FlowDroid is configured with the sources and sinks provided by each benchmark suite, which consists of sources and sinks appeared in its ground truth. Other configuration options of FlowDroid are kept default. By default, FlowDroid uses Soot's SPARK algorithm to construct call graph. We modified FlowDroid to dump a serialized call graph once it is constructed for each app. We collect the following metrics for comparison:

- **Model Generation Time** (T_{model}): the time used by GenCG to generate the `placeholder.jar` of the framework model for each benchmark app.
- **Analysis Time** ($T_{analysis}$): the time used by FlowDroid to analyze either the original benchmark app with the Android SDK or the `placeholder.jar` generated by GenCG.
- **Percentage of static reachable methods over concrete methods** ($P_{concrete}$): This is the proportion of concrete (non-abstract) methods present in the call graphs. For each suite, $P_{concrete} = C_{callgraph}/C_{suite}$, where $C_{callgraph}$ is the number of concrete methods in the serialized call graphs and C_{suite} is the number of concrete methods of all classes in the suite. In TaintBench, there are 73,225 (C_{suite}) concrete methods, while it is 301,750 in F-Droid. We could also find the source code of 28 of the 30 F-Droid apps to identify the application classes[1]. The number of concrete methods of application classes (App-only C_{suite}) in these 28 apps is 17,755.
- **Percentage of static reachable methods over executed methods** ($P_{executed}$): This is the proportion of executed methods that are present in the call graphs. $P_{executed} = E_{callgraph}/E_{suite}$, where $E_{callgraph}$ is the number of methods in the serialized call graphs which are also executed at runtime and E_{suite} is the number of executed methods of all apps in the suite. $P_{executed}$ is only collected for F-Droid, but not for TaintBench, since many of the apps in TaintBench cannot be executed with their malware behavior as the services with which the apps used to communicate are nowadays disabled. To collect the executed methods of the F-Droid apps, the first three authors explore each app separately and use the code coverage tool ACVTool [19] to collect methods that are executed at runtime during the exploration. Finally, the executed methods collected by three authors are merged together for each app. ACVTool crashed for 7 apps, therefore, we could only collect the executed methods for 23 of the 30 F-Droid apps. The total number of executed methods of these 23 F-Droid apps are 17,582 (E_{suite}) as Table I shows, of them 5,759 are from application classes.
- **Percentage of static reachable methods over ground-truth methods** ($P_{groundTruth}$): This is the proportion of methods that contains the sources and sinks of true-positive taint flows (we call these methods *ground-truth methods*) that are present in the call graphs. $P_{groundTruth} = G_{callgraph}/G_{suite}$, where $G_{callgraph}$ is the number of methods in the call graphs which are also ground-truth methods and G_{suite} is the number of the ground-truth methods in the suite.
- **Number of true positives, Number of false positives, Precision, Recall, F-Measure**: these are standard metrics for evaluating static analysis tools related to the detected and true/false taint flows based on the ground truth when running a client taint analysis.

B. ***RQ1.*** *How good are the call graphs using the model generated by* GenCG *in comparison to using FlowDroid's model?*

To answer this question, we look at the call graphs constructed with Soot's SPARK algorithm (FlowDroid's default call graph algorithm) and refer to three metrics, $P_{concrete}$, $P_{executed}$, and $P_{groundTruth}$, in the following. We discuss them individually for each benchmark.

TaintBench: There are in total 73,225 concrete methods in the 39 benchmark apps. The call graphs based on the model generated by GenCG cover 41.23% of these concrete methods, while with FlowDroid's model $P_{concrete}$ is only 10.86% as Table II shows. We see an increase of $P_{concrete}$ using GenCG for all TaintBench apps.

Table IV shows the $P_{groundTruth}$ values. In the case of TaintBench, we see that using the model generated by GenCG, the call graphs have a better coverage of the methods in the ground truth than FlowDroid, i.e., 95.32% vs. 68.23%, meaning that call graphs using the model generated by GenCG are likely to enable a client taint analysis to detect true taint flows in the ground truth. We will report more on this when answering RQ2.

F-Droid: Among all 30 apps, there are 301,750 concrete methods, of which 17,755 are from application classes as Table I shows. Table II shows the $P_{concrete}$ values: 14.23% for GenCG-FlowDroid and 4.51% for FlowDroid. Similarly, if we only count application-only methods covered in the call graphs, the values are 68.0% for GenCG-FlowDroid and 26.97% for FlowDroid. Overall, $P_{concrete}$ of GenCG-FlowDroid is about three times higher than FlowDroid. Individually per app, GenCG-FlowDroid shows higher value on all 30 apps when considering all classes in app's apk

[1]The source code of the apps net.osmand.plus_355 and eu.kanade.tachiyomi_41 is not available on F-Droid any more.

TABLE I
BENCHMARK SUITES

	#Apps	C_{suite}	App-only C_{suite}	E_{suite}	App-only E_{suite}
TaintBench	39	73,225	N/A	N/A	N/A
F-Droid	30	301,750	17,755	17,582	5,759

TABLE II
% STATIC REACHABLE METHODS OVER **CONCRETE** METHODS ($P_{concrete}$)

	FlowDroid	GENCG-FlowDroid	#Apps with higher %
TaintBench	10.86	41.23↑	39 (of 39)
F-Droid	4.51	14.23↑	30 (of 30)
F-Droid (app-only)	26.97	68.0↑	26 (of 28)

TABLE III
% STATIC REACHABLE METHODS OVER **EXECUTED** METHODS ($P_{executed}$)

	FlowDroid	GENCG-FlowDroid	#Apps with higher %
F-Droid	32.82	72.38↑	21 (of 23)
F-Droid (app-only)	48.98	83.73↑	20 (of 23)

TABLE IV
% STATIC REACHABLE METHODS OVER **GROUND-TRUTH** METHODS ($P_{groundTruth}$)

	FlowDroid	GENCG-FlowDroid
TaintBench	68.23	95.32↑
F-Droid	88.71	91.94↑

file, and in 26 of 28 apps (of which application source code is available) when considering application-only methods. We sample the missing methods in the call graphs constructed with our approach in these two apps. We find out that they are mostly UI callback which are specified the layout XML files of the apps. Modeling such method calls from Android requires parsing the layout XML files. Because such XML configuration is different in every framework and our approach is designed to be reusable across several frameworks, we do not model this specifically for Android. This could be done, of course, though, with appropriate engineering effort.

Based on 23 apps from F-Droid for which we are able to dynamically explore the apps by running ACVTool and compile a list of executed methods at runtime, we report $P_{executed}$ in Table III. In both variants, i.e., considering all methods or application-only methods, GENCG-FlowDroid has significant higher $P_{executed}$, 72.38% vs. 32.82% for all methods and 83.73% vs. 48.98% for application-only methods.

Finally, considering the $P_{groundTruth}$ values in Table IV, GENCG-FlowDroid also achieves a better coverage than FlowDroid (91.94% vs. 88.71%).

Compared to FlowDroid's model, call graphs constructed using the model generated by GENCG show to have a much better percentage of static reachable methods over the concrete methods, executed methods, and the methods from the ground truth of both TaintBench and F-Droid.

*C. **RQ2.** How does the model generated by* GENCG *impact FlowDroid's taint analysis?*

To judge the accuracy impact of our approach on FlowDroid's taint analysis, we use the precision, recall, and F-measure metrics to answer this question. For performance impact, we look into the time used for generating the model by GENCG T_{model} and the analysis time used by FlowDroid $T_{analysis}$. The time overhead of GENCG is measured by T_{model}, which is performed only once for each app and can be reused for multiple client analysis tools that analyze Java byte code.

Table VI shows the comparison between FlowDroid and GENCG-FlowDroid evaluated on TaintBench. As we can see in this table, Using the call graphs based on the Android model generated by GENCG, FlowDroid detects 19 (67 vs. 48) more true positives with even less false positives (9 vs. 14). As the call graphs constructed with our approach enables FlowDroid's taint analysis to analyze more code, the recall based on TaintBench's ground truth is improved from 0.24 to 0.33. Although a generic approximation like our approach can be noisy, the evaluation results show that it does not affect the client taint analysis in detecting real-world malicious taint flows in TaintBench. Even the precision is increased from 0.77 to 0.88. As a result, the F-measure improves 11% (0.37 vs. 0.48).

Regarding F-Droid benchmarks, we have two fewer true positive as Table VII shows; this is because having extraneous sources and sinks unexpectedly impacts FlowDroid's taint computation. This was previously discovered by Luo, Pauck et al. [8]. When configuring FlowDroid with only the sources

TABLE V
TIME IN SECONDS

	FlowDroid	GENCG-FlowDroid	
	Analysis ($T_{analysis}$)	Analysis ($T_{analysis}$)	Model Generation (T_{model})
TaintBench	161.81	1444.8	311.92
F-Droid	1051.83	7205.74	428.27

TABLE VI
PRECISION, RECALL, AND F-MEASURE REGARDING THE TAINT FLOWS

		FlowDroid	GENCG-FlowDroid
TaintBench	#True Positives	48	67
	#False Positives	14	9
	Precision	0.77	0.88
	Recall	0.24	0.33
	F-measure	0.37	0.48

TABLE VII
TRUE-POSITIVE, FALSE-POSITIVE, AND UNCLASSIFIED TAINT FLOWS

		FlowDroid	GENCG-FlowDroid
F-Droid	#True Positive	11	9
	#False Positive	16	1
	#Unclassified	350	594

and sinks in these two true-positive flows, FlowDroid could detect them using the model generated by GENCG. Moreover, FlowDroid produces fewer false positives based on existing ground truth when using our model. GENCG also enables FlowDroid to detect more taint flows, however, these are not documented in the ground truth. As it is infeasible to manually triage these flows due to the large number and the complexity of the apps, hence we mark them as unclassified in Table VII. Note that we only count a taint flow as true/false positive if it matches a true-positive/false-positive in the ground-truth documentation of F-Droid. Because there are two many unclassified flows, we can not judge the impact on precision, recall and F-measure.

Table V shows the values for T_{model} used by GENCG and $T_{analysis}$ for both approaches evaluated on both benchmark suites. As previously seen, using GENCG's models, FlowDoid detects more taint flows and hence a better recall, which causes an increased analysis time. In the TaintBench apps, the increase is from 161.81 to 1444.8 seconds, whereas for the F-Droid apps, the increase is about 7 times.

GENCG improves accuracy of FlowDroid's taint analysis in both precision and recall, when evaluated on TaintBench. GENCG also enables FlowDroid's taint analysis to detect more taint flows in F-Droid apps. Due to the larger call graphs using models generated GENCG, FlowDroid's taint analysis takes longer on those benchmarks. Additionally, it takes time to generate the model with GENCG.

V. APPLICATION TO THE SPRING FRAMEWORK

To evaluate how effectively GENCG generates models for Spring applications, we develop a benchmark suite called CGBench with ground-truth documentation. CGBench consists of 42 Spring apps classified into 6 categories as shown in Table VIII. 39 of them are micro benchmark apps to demonstrate specific Spring features with one or two built-in taint-style vulnerabilities such as SQL Injection, XSS, Log Injection etc. The vulnerabilities in these apps are all true taint flows. 3 of the 42 apps are bigger apps that are built to demonstrate vulnerabilities. These 3 apps contain multiple Spring features and taint-style vulnerabilities. The ground truth documentation CGBench consists of 60 true taint flows and 10 false taint flows (which imprecise tools could detect). We modify FlowDroid to analyze Spring applications, configure it with the sources and sinks in each benchmark app, and analyze the `instrumented-app.jar` file and `placeholder.jar` file generated by GENCG for each app. Table VIII shows the evaluation results on CGBench. In total, our modified version of FlowDroid detects 45 true-positive taint flows out of 60 and 4 false-positive taint flows out of 10, which makes the precision 0.92, the recall 0.75 and the F-measure 0.83.

VI. THREATS TO VALIDITY

Since our approach is designed to be general, we only consider language-level concepts (e.g. subtyping, annotations) and explicitly did not model framework behaviors that require parsing configuration files (e.g. XML, HTML files). Client analyses which need such information must add this support. Our type-based model is more precise than Averroes for taint analysis as we show in our evaluation, yet it might not work well for code which contains a lot of type casting code, as it pollutes the type-based pointers. Our approach mainly focuses on modeling callback invocations, control-flows and object creations by frameworks to enable construction of sound application-only call graphs (do not contain calls between framework methods). Aliasing through assignments inside the framework code and its side effects are not in the scope of this work. A threat to the external validity is that our evaluation results have limited generalizability to other client analyses. Because our focus is constructing call graphs that allow taint analyses to effectively find more real-world issues, we only evaluated our approach with the taint analysis in FlowDroid. FlowDroid uses StubDroid [20] to generate summaries for handling the taint propagation through common library methods. These summaries cover many methods from the Android framework. Our modified version of FlowDroid uses existing summaries in FlowDroid and only analyzes placeholder methods if no summaries are available. Other taint analysis approaches that do not model taint propagation

TABLE VIII
EVALUATION RESULTS ON CGBENCH.

No.	Category	#True Taint Flows	#False Taint Flows	#True Positive	#False Positive
1	HTTP Request Handlers	9	0	8	0
2	Component Classes	5	0	4	0
3	Handler Interceptors	6	0	6	0
4	Parameter Sources	19	0	16	0
5	Configuration	6	1	0	0
6	Demo Apps with Mixed Features	15	10	10	4
	$\sum$	60	10	45	4
	Precision				0.92
	Recall				0.75
	F-measure				0.83

through library methods, might still produce imprecise results with our approach.

VII. RELATED WORK

Many previous approaches have addressed the challenge of analyzing apps within certain frameworks. A significant number of these approaches focus on the Android framework. FlowDroid [10] precisely models the Android lifecycle and UI callback handling by creating a dummy main method. Amandroid [13] and IccTA [21] extend this model by introducing control and data dependencies between Android components such that inter-component communications are also captured. These analyses use the Android model and do not consider the implementation of the Android framework. On the other hand, DroidSafe [22] manually crafts framework with stub implementations. Similar to the placeholder library used in our approach, these stub implementations are analyzed as replacements of the original framework implementation. However, as the authors themselves pointed out, implementing these stubs is labor-intensive and requires expertise in Android. In these tools the model is hard-coded in the tool's implementation which makes it hard to reuse in other analyses. Both Droidel [23] and our approach automatically create app-specific stubs of the Android framework with a single entry point. While the authors of Droidel acknowledged that their approach is not suitable for flow-sensitive analyses, our approach with the adapted FlowDroid's taint analysis is flow-sensitive. Droidel still requires a one-time manual modification of the original Android framework source code to replace the usage of reflection with the Droidel's own interfaces. In comparison, our approach only needs the stub version of the framework and is not specific to Android. We could not compare our approach to Droidel in an experiment, as the original Android framework source code used by Droidel was not available and even the authors do not have it anymore.

There are few approaches that focus on the generating a model for the Android callbacks. Perez et al. [24] proposed a method for generating, so called, Predicate Callback Summaries (PCSs), which are control flow–based graph representations of the callback API implementations. PCSs can further be integrated in other analyses. While Perez et al. focus on non-GUI-based Android callbacks, Yang et al. [25] proposed a method for generating the model of the GUI-based callbacks. This model is a context-sensitive graph-based representation. Later this model can be incorporated in other analyses, for example to construct a full GUI model of a given app for program understanding or test generation.

In the area of Java web frameworks, IBM's TAJ [26] and its follow-up work F4F [27] are among the best-known approaches targeting Java enterprise applications. TAJ is a taint analysis that has partly modeled the Apache Struts framework and Enterprise Java Beans in its analysis engine. Adding a new framework support requires engineering effort in the analysis engine which is similar to most Android taint analyses, such as FlowDroid and Amandroid. The follow-up work F4F improves this by proposing a specification language WAFL that integrates with the engine where different frameworks can be modeled and enables reusability. However, to support new frameworks, the WAFL specifications need to be written manually which requires high engineering effort. In our approach, the generated placeholder library can be processed by any existing Java analysis frameworks (e.g., Soot, WALA) and only the configurable lists of APIs (e.g., entry point classes/methods/annotations, annotations for dependency injection) should be extended. Similar to F4F, recent work JackEE [17] also introduces a rule-based specification that covers general concepts for modeling Java enterprise framework behaviors. JackEE leverages Doop and its model of a new framework is a collection of logic rules, which can be understood by Doop.

VIII. CONCLUSION

We present GENCG—a general approach to modeling Java frameworks. GENCG produces a placeholder jar file that can be used as a sound replacement of the original framework by precise call graph construction algorithms and further client analyses. We demonstrate its generalization with both the Android and the Spring framework. A throughout evaluation with two real-world Android taint analysis benchmark suites shows our approach is especially effective in enabling a precise flow-, field- and context-sensitive taint analysis in detection of more real-world issues without introducing much noise. We constructed a micro benchmark suite—CGBench—consisting of common taint-style vulnerabilities in Spring-based web applications. We evaluate our approach using this suite and show the applicability of our approach on Spring framework.

References

[1] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam, "Array-data flow analysis and its use in array privatization," in *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 2–15. [Online]. Available: https://doi.org/10.1145/158511.158515

[2] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of CASCON*. IBM, 1999.

[3] G. Piskachev, R. Krishnamurthy, and E. Bodden, "Secucheck: Engineering configurable taint analysis for software developers," in *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2021, pp. 24–29.

[4] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri, "Andromeda: Accurate and scalable security analysis of web applications," in *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, 2013, pp. 210–225. [Online]. Available: https://doi.org/10.1007/978-3-642-37057-1_15

[5] V. Tanzu, "Java spring framework," https://spring.io/, online; accessed 27 September 2022.

[6] Google, "Android framework," https://www.android.com/.

[7] K. Ali and O. Lhoták, "Averroes: Whole-program analysis without the whole program," in *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, ser. Lecture Notes in Computer Science, G. Castagna, Ed., vol. 7920. Springer, 2013, pp. 378–400. [Online]. Available: https://doi.org/10.1007/978-3-642-39038-8_16

[8] L. Luo, F. Pauck, G. Piskachev, M. Benz, I. Pashchenko, M. Mory, E. Bodden, B. Hermann, and F. Massacci, "Taintbench: Automatic real-world malware benchmarking of android taint analyses," *Empir. Softw. Eng.*, vol. 27, no. 1, p. 16, 2022. [Online]. Available: https://doi.org/10.1007/s10664-021-10013-5

[9] A. Mordahl and S. Wei, "The impact of tool configuration spaces on the evaluation of configurable taint analysis for android," in *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, C. Cadar and X. Zhang, Eds. ACM, 2021, pp. 466–477. [Online]. Available: https://doi.org/10.1145/3460319.3464823

[10] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, M. F. P. O'Boyle and K. Pingali, Eds. ACM, 2014, pp. 259–269. [Online]. Available: https://doi.org/10.1145/2594291.2594299

[11] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "Taj: Effective taint analysis of web applications," *SIGPLAN Not.*, vol. 44, no. 6, p. 87–97, jun 2009. [Online]. Available: https://doi.org/10.1145/1543135.1542486

[12] G. Piskachev, J. Späth, I. Budde, and E. Bodden, "Fluently specifying taint-flow queries with fluenttql," *Empirical Softw. Engg.*, vol. 27, no. 5, sep 2022. [Online]. Available: https://doi.org/10.1007/s10664-022-10165-y

[13] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, G. Ahn, M. Yung, and N. Li, Eds. ACM, 2014, pp. 1329–1341. [Online]. Available: https://doi.org/10.1145/2660267.2660357

[14] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 49–61. [Online]. Available: https://doi.org/10.1145/199448.199462

[15] S. Arzt, S. Rasthofer, and E. Bodden, "The soot-based toolchain for analyzing android apps," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2017, pp. 13–24.

[16] O. Lhoták and L. J. Hendren, "Scaling java points-to analysis using SPARK," in *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, ser. Lecture Notes in Computer Science, G. Hedin, Ed., vol. 2622. Springer, 2003, pp. 153–169. [Online]. Available: https://doi.org/10.1007/3-540-36579-6_12

[17] A. Antoniadis, N. Filippakis, P. Krishnan, R. Ramesh, N. Allen, and Y. Smaragdakis, "Static analysis of java enterprise applications: frameworks and caches, the elephants in the room," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, A. F. Donaldson and E. Torlak, Eds. ACM, 2020, pp. 794–807. [Online]. Available: https://doi.org/10.1145/3385412.3386026

[18] A. De and D. D'Souza, "Scalable flow-sensitive nalysis for java with strong updates," in *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, ser. Lecture Notes in Computer Science, J. Noble, Ed., vol. 7313. Springer, 2012, pp. 665–687. [Online]. Available: https://doi.org/10.1007/978-3-642-31057-7_29

[19] A. Pilgun, O. Gadyatskaya, Y. Zhauniarovich, S. Dashevskyi, A. Kushniarou, and S. Mauw, "Fine-grained code coverage measurement in automated black-box android testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–35, 2020.

[20] S. Arzt and E. Bodden, "Stubdroid: automatic inference of precise data-flow summaries for the android framework," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 725–735. [Online]. Available: https://doi.org/10.1145/2884781.2884816

[21] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. D. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, A. Bertolino, G. Canfora, and S. G. Elbaum, Eds. IEEE Computer Society, 2015, pp. 280–291. [Online]. Available: https://doi.org/10.1109/ICSE.2015.48

[22] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015. [Online]. Available: https://www.ndss-symposium.org/ndss2015/information-flow-analysis-android-applications-droidsafe

[23] S. Blackshear, A. Gendreau, and B. E. Chang, "Droidel: a general approach to android framework modeling," in *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2015, Portland, OR, USA, June 15 - 17, 2015*, A. Møller and M. Naik, Eds. ACM, 2015, pp. 19–25. [Online]. Available: https://doi.org/10.1145/2771284.2771288

[24] D. D. Perez and W. Le, "Generating predicate callback summaries for the android framework," in *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '17. IEEE Press, 2017, p. 68–78. [Online]. Available: https://doi.org/10.1109/MOBILESoft.2017.28

[25] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in android applications," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. IEEE Press, 2015, p. 89–99.

[26] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "Taj: Effective taint analysis of web applications," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 87–97. [Online]. Available: https://doi.org/10.1145/1542476.1542486

[27] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg, "F4F: taint analysis of framework-based web applications," in *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, C. V. Lopes and K. Fisher, Eds. ACM, 2011, pp. 1053–1068. [Online]. Available: https://doi.org/10.1145/2048066.2048145