StubDroid: Automatic Inference of Precise Data-flow Summaries for the Android Framework

Steven Arzt
Secure Software Engineering Group
EC SPRIDE, Technische Universität Darmstadt
Darmstadt, Germany
steven.arzt@cased.de

Eric Bodden*
Software Engineering Group
Paderborn University & Fraunhofer IEM
Paderborn, Germany
eric.bodden@uni-paderborn.de

ABSTRACT

Smartphone users suffer from insufficient information on how commercial as well as malicious apps handle sensitive data stored on their phones. Automated taint analyses address this problem by allowing users to detect and investigate how applications access and handle this data. A current problem with virtually all those analysis approaches is, though, that they rely on explicit models of the Android runtime library. In most cases, the existence of those models is taken for granted, despite the fact that the models are hard to come by: Given the size and evolution speed of a modern smartphone operating system it is prohibitively expensive to derive models manually from code or documentation.

In this work, we therefore present StubDroid, the first fully automated approach for inferring precise and efficient library models for taint-analysis problems. StubDroid automatically constructs these summaries from a binary distribution of the library. In our experiments, we use StubDroid-inferred models to prevent the static taint analysis FlowDroid from having to re-analyze the Android runtime library over and over again for each analyzed app. As the results show, the models make it possible to analyze apps in seconds whereas most complete re-analyses would time out after 30 minutes. Yet, StubDroid yields comparable precision. In comparison to manually crafted summaries, StubDroid's cause the analysis to be more precise and to use less time and memory.

Categories and Subject Descriptors

F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages – Program analysis

Keywords

Static analysis, summary, library, framework model, model inference

*During the time this research was conducted, Eric Bodden was at Fraunhofer SIT & Technische Universität Darmstadt.

ICSE '16, May 14-22, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2884781.2884816

1. INTRODUCTION

Over the past years, the Android operating system has become the most prevalent platform in the smartphone market, with a market share of more than 79% [7]. Applications for the platform are provided by many vendors. As of now, there are more than one million applications available through the official Google Play store alone, and numerous other app markets exist as well. With these numbers, the quality and security of Android applications has become a major concern which can no longer be addressed by manual code inspection. Indeed, researchers have already proposed various static code-analysis tools for Android, mostly addressing data-flow problems [1–6, 8, 12, 13, 20]. Analyzing the application alone, however, is insufficient, as applications make heavy use of the Android framework's application programming interface. In version 4.2, Android offers over 110,000 public methods and the number constantly grows with every new release. For an analysis to be precise, it must not only analyze the application code, but also consider the effect of these library methods on data flows inside the application.

Existing approaches to static analysis deal with library methods in one of three ways. The first class of analyses precisely models a subset of these framework methods manually [1,3–5,8,13,20], while the second class analyzes the entire android framework together with every application [12]. The third class uses simple rules of thumb such as "taint return value of call if one or more arguments are tainted", which are expected to cover the most common cases [6]. All of these approaches exhibit serious drawbacks.

Providing hand-written models for the framework is cumbersome to implement and requires manual re-evaluation of (and possibly changes to) the model for every new framework version, which is a prohibitive effort given the size of the code to be understood and modeled. CHEX [13], for instance, opts to not analyze the framework together with the application, and instead resorts to externally-defined models. However, the source of the models is mostly left open, effectively burdening the user with this non-trivial task.

Including the framework in the automated code analysis evades this manual effort, but it introduces an often prohibitive overhead on the code analysis in terms of the overall code size to be analyzed. The Android operating system consist of millions of lines of code, thereby greatly exceeding the size of usual applications to be analyzed, causing the analysis to spend significantly more time in analyzing library code than in analyzing application code, i.e., the code that is of actual interest to most analyses. In previous work on FlowDroid [1], we found such an approch to induce a sig-

nificantly increased analysis time—for every single app, over and over again.

Applying rules-of-thumb instead of a proper library implementation or external model evades the problems of high runtime overhead and high manual effort. However, it remains unclear to what extent these rules actually cover all important libraries, or whether false-negatives may occur due to missing taint propagations. Worse, if these rules are designed to avoid false-negatives, they necessarily need to over-approximate the behavior of method calls and can thus lead to an increased number of false-positives. Recall that these rules are oblivious to the individual callees, but aim at a generic model for all library calls.

In conclusion, none of the approaches presented so far solves the challenge of handling libraries during static analysis to full satisfaction. In this work, we thus present StubDroid, the first fully automated approach for inferring taint-flow models from binary library distributions such as the Android framework. StubDroid first performs a taint analysis on selected public methods. We demonstrate this for the collections API, which is the most commonly used part of the framework. StubDroid then generalizes the resulting source-to-sink data-flow mappings and stores them in a summary file. When a static client-side taint analysis later processes the invocation of a framework method within the code of an Android app, the analysis can simply plug in the information from the summary file, and short-circuit further analysis of the framework call and all its transitive callees. While seemingly simple, a significant challenge lies in establishing summaries that are field sensitive and handle aliasing, and in treating correctly the various callbacks that the framework code might use to interact with the application. Furthermore, note that the generated summaries are independent of any concrete client application. They must thus abstract from all possible state or call sequences while still retaining maximum precision.

We show that on commodity hardware it is usually possible to generate library summaries in under three minutes per framework class—a one-time effort. Using StubDroid's summaries for conducting a taint analysis on client applications with the FlowDroid taint tracking tool [1] can improve the analysis performance by over 90%. For many applications, the use of summaries even *enables* the analysis, as the full analysis of those apps would time out after spending 30 minutes and consuming tens of gigabytes of memory if summaries were not used. StubDroid avoids these blowups because, opposed to client analyses, its analysis can focus on one framework entry-point method at a time. To summarize, this paper presents the following original contributions:

- StubDroid, a method for automatically generating correct and precise models of the Android framework methods with respect to taint analysis,
- a full open-source implementation of the above,
- as an artifact, taint summaries for various important Android APIs, and
- an evaluation of the peformance effect of summary usage in the FlowDroid taint-analysis tool.

StubDroid's full source code and the library summaries generated with it are publicly available as an open-source project at: http://blogs.upb.de/sse/tools/stubdroid/

The remainder of this paper is structured as follows. In Section 2 we motivate why simple rules of thumb are not sufficient to handle library methods in a taint analysis. Section 3 shows what taint summaries for a method look like and also introduces the concept of access paths which StubDroid uses to model field references. Section 4 explains StubDroid's architecture for generating and applying summaries, before we report on our summary-computation process in detail in Section 5. Section 6 focuses StubDroid's callbacks handling, before we go into the process of applying summaries in Section 7. In Section 8, we report on the performance and correctness of StubDroid. Section 9 presents related work and Section 10 concludes the paper.

2. MOTIVATING EXAMPLE

Using summaries, static analyses can gain considerable performance improvements. A summary, plugged in at a call site, not only renders unnecessary the analysis of any direct callees, but also of all of the methods called transitively. A summary thus truncates a whole call tree and replaces it by a single leaf containing the summarized data-flow facts. Even the summary for a simple method such as `HashSet.add()` will shortcut the analysis of about a dozen methods. When analyzing the client program, the whole tree is flattened into a single rule: "For a HashSet `s` and an element `x`, if `x` is tainted, then `s` is tainted after executing `s.add(x)`." Given that `HashSet.add()` is used many times in many applications, this can save considerable analysis time.

It is important, however, that summary rules are sufficiently precise, as imprecisions can carry over into the analysis result and can again degrade analysis performance. Ad-hoc rule sets often fail to distinguish different fields of the same object or different parameters of the same method call. A simple generic rule as above generally taints, whenever invoking any method `o.m(x)` with a tainted parameter `x`, the base variable `o`. As we show in Listing 1, such a coarse model can easily yield imprecisions, and therefore false positives. In this example, upon processing the constructor call at line 16, the analysis would taint the variable `p`, and implicitly all fields reachable through it. In the example, this would cause a false positive at line 17: as the return value of `getO1()` is retrieved from the tainted reference `p`, this return value is (falsely) considered tainted as well.

In some cases, too simplistic rules can also lead to false negatives such as in the example in Listing 2. In this example, `oos.writeObject(..)` (line 4) writes data to `out`, via an internal field reference. The standard rule set would mark `oos` as tainted as well as all fields reachable through it, but not `out`. But the data is leaked through `out`, causing analyses using such a summary to miss the flow to the `sink`. A sound summary must thus be able to encode that the call to `oos.writeObject(..)` has an implicit side-effect on (internal fields of) `out`.

As these examples show, too simplistic rules can lead to serious cases of over and under-tainting. Nevertheless, simplistic rules are the current state of the art [1, 5, 13, 20]. The only simple alternative to summaries, however, would be to analyze the complete Android SDK together with every app being inspected. Past experience as well as our experiments in Section 8.3 show, however, that the performance penalty of such an approach is prohibitive. StubDroid thus seeks to pre-compute precise summaries ahead of time, in an automated one-time effort. These summaries can then speed up

```
1  public class Pair {
2    private Object o1, o2;
3
4    public Pair(Object p1, Object p2) {
5      this.o1 = p1;
6      this.o2 = p2;
7    }
8    public Object getO1() {
9      return this.o1;
10   }
11   public void setComplex(Data a) {
12     this.o1 = a.b.c;
13   }
14   public static void main(String[] args) {
15     String s = source();
16     Pair p = new Pair("not tainted", s);
17     sink(p.getO1());
18   }
19 }
20 public class Data {  public Data2 b;  }
21 public class Data2 { public Object c; }
```

Listing 1: Complex Data Structure and Method

```
1  public void doLeak() {
2    ByteArrayOutputStream out = new
         ByteArrayOutputStream();
3    ObjectOutputStream oos = new
         ObjectOutputStream(out);
4    oos.writeObject(source());
5    oos.close();
6    sink(out.toByteArray());
7  }
```

Listing 2: Complex Data Structure and Method

any subsequent analysis runs without jeopardizing precision, and while reducing memory consumption.

3. SUMMARY MODEL

Assume the `Pair` class from Listing 1 to be part of a library. The summary for the `Pair` class' constructor needs to model that data flows from the first parameter to the field `this.o1`, i.e., that `this.o1` inherits the taint state of the first parameter. The same connection exists between the second parameter and the field `this.o2`. There is, however, no connection between the two fields or between the first parameter and field `this.o2`.

The taint summaries generated by StubDroid take the form of rules. Given a certain incoming taint, they model the effect of a certain method call on this taint. The constructor of the `Pair` class can therefore be described using two rules:

1. `this.o1` is tainted if `parameter 1` is tainted. (R1)
2. `this.o2` is tainted if `parameter 2` is tainted. (R2)

When applying a summary, the rules are used to perform a fixed-point iteration on the set of tainted variables at the call site that invokes the summarized method. Every taint state that is not explicitly changed by a summary rule is kept unchanged. (There are no strong updates.)

In Listing 1, which calls the constructor of `Pair` with only `Parameter 2` tainted, summary rule (R2) is applicable, while rule (R1) is not. Thus, `o2` is marked as tainted for the current instance of `Pair` while `o1` remains untainted.

Note that summaries can also model effects on private fields of objects. While those fields are invisible to application code, they can be used to model object state, akin to ghost fields in JML [11]. This makes summaries field sensitive, which is important for maintaining precision. The `Pair` class in the example contains a method `getO1()` returning `this.o1`. The return value of this method thus inherits the taint state of `this.o1`, which is distinct from the taint state of `this.o2`. Without field sensitivity, one could not distinguish between the taint states for the two fields, and a false positive would occur at line 17.

Highly precise data-flow analysis tools such as FlowDroid or Andromeda [19] work by tracking not only fields but so-called *access paths*. An access path is of the form $l.f.g$ where l is a local variable or parameter and f and g are field accesses. Access paths can have different lengths up to a user-customizable maximum, at which they are truncated. An access-path of length 0 is a simple local variable or parameter, e.g., l. Truncated access paths act as placeholders for all runtime objects reachable through them, and end with an asterisk, e.g., $l.f.*$ for all objects reachable through $l.f$ (including $l.f$ itself, but also $l.f.g$, $l.f.h$, ...).

To retain this level of precision, StubDroid's summaries are also based on access paths, with a customizable maximal length. A taint-summary rule thus always taints an acess path given that a certain incoming access path is tainted. One would thus write the above rules more precisely as:

1. `this.o1.*` is tainted if `parameter 1.*` is tainted.
2. `this.o2.*` is tainted if `parameter 2.*` is tainted.

In all rules, the asterisk character acts as a placeholder. Assume an application calls the constructor of `Pair` as in:

```
Data d = new Data(); d.f.g = source();
Pair p = new Pair("not tainted", d);
```

In this example, rule 2 applies. It will, however, not simply taint `this.o2.*`, but copy over the suffix of the access path, tainting instead `this.o2.f.g`. More formally, the asterisk on both sides of a rule references the same universally quantified variable ($\forall x$: `this.o1.`x is tainted if `parameter 1.`x is tainted). In result, StubDroid's summaries can retain the client analysis' precision as long as the library summary was generated with at least the same maximal access-path length that is also used to analyze the application. By default, access paths are truncated at length 5, the same default that also the FlowDroid client analysis uses. Longer access paths can require significantly more computation time during summary generation, but are possible by simply changing the StubDroid configuration. In previous work, we found FlowDroid's default length of 5 to be sufficiently precise in practice [1].

4. ARCHITECTURE

Figure 1 shows the general workflow involving StubDroid. StubDroid generates one summary file per library class, from a binary distribution of the respective library. At this stage, no application code is present. We assume that applications only interact with libraries through public methods and fields, rather than applying reflection to access private members. Therefore, all public methods (and only those) need to be summarized.

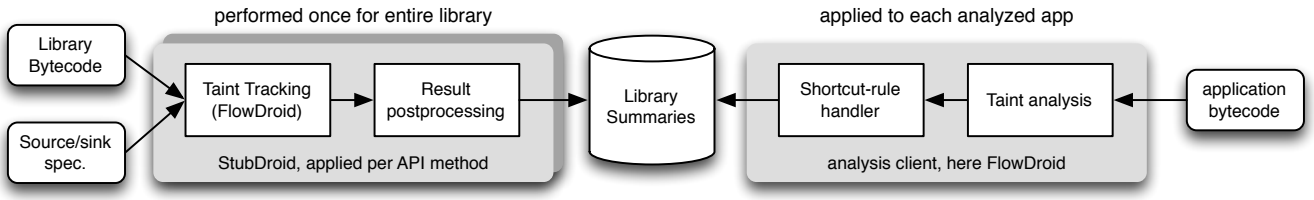

Figure 1: StubDroid's Process and Architecture

The summary generator cannot anticipate in which order or with which parameters the library method will later be called inside the application code. The generated summaries must work in all possible apps and usage contexts. Therefore, StubDroid must assume all possible call sequences, must abstract from all library state and parameters before a method call, and must then analyze the effect of the method on these abstract descriptions as explained in Section 3. StubDroid can thus analyze the effects of every API method in isolation, which helps the tool to keep its memory requirements low despite the analysis' high precision.

StubDroid generates summaries as XML files, one file per library class. The generated summary files can afterwards be used during the analysis of an arbitrary number of applications. At this stage, the library code no longer needs to be available, as the summaries are self-contained. When using the libraries during an analysis, the XML files are loaded on demand. A client-side summary storage keeps track of all classes for which summaries are available. Only when a taint can potentially reach a field or method of a certain class, the summary file for this class is requested from the storage and loaded into memory. This greatly reduces memory consumption if summaries for large (or many different) libraries are available on disk.

In our particular implementation, StubDroid uses the FlowDroid open-source data-flow tracker [1] for generating and as a client applying library summaries. Nevertheless, the summaries are encoded in generic XML, which makes them usable also for other taint-analysis clients. We based StubDroid on FlowDroid, because it is precise (context-, field-, object-, and flow-sensitive) as well as easily extendable. Further, as FlowDroid is based on Soot [10], it can be run on Java source code, Java bytecode, and Android's Dalvik bytecode[1]. StubDroid inherits these capabilities.

To integrate library summaries into its taint analysis, FlowDroid offers the concept of so-called *Taint Wrappers*. Those wrappers implement shortcut rules that circumvent the analysis of certain callee methods. To enable FlowDroid to process StubDroid's summaries, we implemented shortcut-rule handling as a taint wrapper which obtains the summary rules directly from the summary storage.

5. SUMMARY GENERATION

StubDroid generates data-flow summaries and must thus conduct a data-flow analysis on the library's bytecode or source code. As explained before, StubDroid analyzes the library by focusing on one API method at a time. Given such a method, StubDroid conducts a taint analysis starting at a number of different potential data sources. In general, every access path within the method in question must be considered a source. This includes all access paths involving parameters, the `this` reference for instance methods, but also all visible static fields. Every access to one of these access paths becomes the left-hand side of a summary rule. For example, in Listing 1, the constructor of the `Pair` class induces rules for the access paths `this.*`, `p1.*` and `p2.*`.

Further, note that `this.*` is used as a source instead of the combination of `this.o1.*` and `this.o2.*`. StubDroid starts with such more abstract top-level access paths only, and reconstructs on demand which fields have actually been used. We will detail this later. Using longer access paths to start with would require StubDroid to consider all access paths reachable through the above base variables, which could cause a combinatorial blowup.

StubDroid, further considers every method return as a sink at which the summarized flow is concluded.[2] The summary rules are generated by comparing the taint state of access paths before and after the execution of the method to be summarized. Every taint derived from one of the source access paths that reaches the end of a method must be translated into a summary rule. In the example from Listing 1, a flow derived from `p1.*` reaches the end of the constructor method as a taint `this.o1.*`. This leads to the generation of rule (R1) from Section 3. Note that identity flows are not encoded, e.g., no rule is generated for the flow that starts with `o1.*` and ends with the same `o1.*` at the end of the method. This is because generally StubDroid does not support strong updates to kill flows. All taints that existed at the beginning of a method are implicitly assumed to still exist also after the summarized method's invocation. We will consider strong updates in future work.

As the sources are only top-level access paths, rule generation is not trivial. Assume that at the end of method `setComplex` from Listing 1, an access path `this.o1.*` is tainted. As taint is linked to the source `a.*`, not `a.b.c.*`, this can lead to all of the following rules:

- `this.o1.*` is tainted if `a.*` is tainted.
- `this.o1.*` is tainted if `a.b.*` is tainted.
- `this.o1.*` is tainted if `a.b.c.*` is tainted.

To restrict the rules it needs to generate, StubDroid must find the concrete fields that were accessed during taint tracking. This is achieved by analyzing the concrete taint-propagation path backwards. The taint analysis is configured to record every statement that influenced a taint, i.e., for which the tainted access path changed during the propagation. For method `setComplex`, this generates the following taint-propagation path. (The temporary variables are artifacts of the Jimple-representation [10] that StubDroid operates on.)

[1]The new ART runtime uses the same bytecode as Dalvik.

[2]Exceptional method returns are ignored in the current implementation, but do not pose any specific further challenges aside from the implementation effort.

```
<method id="test.Pair: void
    setComplex(test.Data)">
  <flows>
    <flow isAlias="true">
      <from sourceSinkType="Parameter"
          ParameterIndex="0"
        BaseType="test.Data"
        AccessPath="[test.Data: test.Data2
            b, test.Data2:
            java.lang.Object c]"
        AccessPathTypes="[test.Data2,
            java.lang.Object]" />
      <to sourceSinkType="Field"
        BaseType="test.Pair"
        AccessPath="[test.Pair:
            java.lang.Object o1]"
        AccessPathTypes="[java.lang.Object]"
            taintSubFields="true" />
    </flow>
  </flows>
</method>
```

Listing 3: Taint Summary for `Pair.setComplex()`

```
tmp$0 = this.a;
tmp$1 = tmp$0.b;
tmp$2 = tmp$1.c;
this.o1 = tmp$2;
```

As the path shows, the taint `this.o1.*` was derived from `tmp$2.*` which in turn was derived from `tmp$1.c.*` etc. Note that this backwards-analysis makes taints more precise. Whenever an assignment defines the base variable of the current taint, its right-hand side replaces the current base variable of the access path. On the second statement, the taint `tmp$1.c.*` is mapped to `tmp$0.b.c.*`, because the second statement defines `tmp$1` as `tmp$0.b`. In other words, StubDroid uses assignments to extend the access path with the field information from the right-hand side of the assignment. With the first statement, the final result of `this.a.b.c.*` is reconstructed. This is the most precise source access path for the new rule. Therefore the actual rule that StubDroid computes is: `this.o1.*` is tainted if `a.b.c.*` is tainted.

5.1 XML File Storage

StubDroid stores its summary facts in one XML file per class. This allows clients to load summaries on demand. Only when a taint reaches a method declared in a specific class, that class' summary file must be loaded. Listing 3 shows the summary of the `setComplex()` method from the example class in Listing 1. A summary file contains `method` elements which in turn contain `flow` elements, one per pair of source and sink between which the method causes a data flow. For each flow, StubDroid stores the source (`from`) and the sink (`to`) in terms of the method's interface. Sources and sinks differentiate between method parameters, fields, and return values. Method parameters are referenced by index. Fields are represented by their full signature. If not the parameter or field itself, but an access path starting at the respective element is referenced, the fields on the access path are stored as an ordered list of full field signatures.

Additonally, StubDroid further stores the propagated types of all fields referenced in a flow summary. While the normal field signature only contains the declared type of the field, the taint analysis also provides StubDroid with an (often more precise) type propagated along with the respective taint. When applying the summary, this allows StubDroid to inject these precise runtime types back into the client's taint analysis. This is useful as some clients use type information to refine call-graph information on the fly. A library method, for instance, might be declared to return `java.lang.Collection`, but always return a more precise type `HashSet`. In this case, the taint analysis can make use of this information: when seeing a call `c.add(..)` on a tainted collection `c`, if the client knows that `c` is a `HashSet` then it can resolve the `add`-call precisely to `HashSet.add(..)`.

```
int charToInt(char c) {
  int[] vals = new int[]  { 0, 1, 2, 3, 4,
      ... };
  int idx = (int) c;
  return vals[idx - 48];
}
```

Listing 4: Implicit Flow Sample Code

5.2 Summaries and Implicit / Native Flows

The computation of library summaries is based on a taint analysis which can either only be performed for explicit data flows through assignments, or also for implicit flows through control-flow dependencies. Listing 4 shows an implicit data flow within a method for converting characters containing digits to integers. The conversion functions in the Oracle JDK and the Android platform are implemented in a similar fashion. Method `charToInt` does not directly assign the parameter value to the result value, but still the result depends on the parameter, and a summary should contain a rule which taints the result value if the parameter is tainted.

StubDroid offers different possibilities to handle such cases. The FlowDroid data-flow tracking tool that StubDroid uses internally supports computing implicit flows just like "normal" explicit flows if the respective option is enabled. In many cases, however, implicit flows do not fully leak the data in question but rather leak information about that data. Determining whether this information leak is problematic is a hard semantic problem [9]. Given that tracking of implicit flows is also expensive, we thus recommend not to track them and instead specify manual summaries for the small number of relevant data-type conversion methods by hand. These are basic JDK methods such as `Integer.toDecimalString()` that are hardly ever extended or changed. This is how we use StubDroid in our daily work and how we conduct our experiments.

Similar problems occur when libraries reference native code. In the Oracle JDK's `ConcurrentHashMap` implementation, for instance, native code is used for fast non-blocking access to the underlying data structures. Since the FlowDroid, and thus StubDroid, cannot analyze native code, we created summaries for such methods manually. Note that StubDroid does not support sanitizers or flows across external resources such as files. We leave this to future work.

6. CALLBACKS

In general, Java (and Android) methods can contain callbacks in which a library method invokes client code. We distinguish generic callbacks such as `toString()` from the special case of Android lifecycle callbacks.

```
1 public String append(String inStr,
      IAppender appender) {
2   return "Hi" + appender.getString(inStr);
3 }
```

Listing 5: Callback Example

6.1 Generic Callbacks

Every library method can invoke methods on objects passed in from application code to call back into the client code, see Listing 5. A correct summary of method `append(..)` requires knowledge about the concrete implementation of `IAppender` (more concretely: of `IAppender.getString()`) passed to `append(..)`. Without this knowledge, one has to manually choose an approximation. An under-approximation of the effects of the call to `getString()` would assume its return value never to be tainted. An over-approximation, though, would assume the return value always to be tainted. A probably more useful approximation would be to assume the return value of `getString()` to only be tainted if the `inStr` parameter is tainted when `append(..)` is called. Even such an approximation might be incorrect, though, depending on the concrete possible implementations of `IAppender`.

To handle such callbacks precisely, StubDroid adopts the principle of component-level analysis introduced by Rountev et. al. [16]. During the analysis of a method to be summarized, one may reach call sites for which the library itself contains no callees. We call these call sites *gaps*. The summary rules presented so far connect interface components of the respective library method, e.g., link a parameter to a return value. With callbacks, flows may start and end at gaps as well. A gap can be thought as a "hole" inside a summarized method flow. StubDroid cannot make any assumptions as to what transformations are made to a taint abstraction inside a gap. It can only summarize the part of the flow that is inside the library method until it reaches the gap. For all possible outgoing taints, it can then again summarize how they flow further inside the library method. In other words, a call to a gap method is treated like an additional interface of the method to be summarized. In the example of Listing 5, this leads to the following rules:

- `<Gap1>Parameter0.*` is tainted if `inStr.*` is tainted
- `Return.*` is tainted if `<Gap1>Return.*` is tainted

Note that StubDroid must generate rules for every possible outgoing taint of the gap method. In the example, only the return value of the gap method `getString` is used within `append`. If, however, a gap method is, for instance, called with a heap object as a parameter and that object is used later on, this must also be represented by a flow. This flow would then account for gap implementors that taint fields inside the heap object they received as a parameter. In short, all possible ways in which a taint can be passed back from a callee to its caller can lead to new flow rules.

When the summary is later applied to a target program, the gaps must be filled with either a different summary or with the results of analyzing client code. StubDroid first attempts to find other summaries that can fill the gaps in question without any interaction with the client analysis. If unsuccessful, it passes the taint information at the gap's call site to the client analysis to find additional implementations of the gap method in client code. This allows the client analysis to focus on the client code alone when filling gaps. The fill-ins applied to gaps can in turn have new gaps which must be filled using the same principle. The fixed-point iteration stops if there are no open (i.e., unfilled) gaps remaining.

6.2 Android Lifecycle Methods

Lifecycle methods allow the Android platform to control applications. In general, the Android OS is much more tightly coupled with its applications than a normal Java VM with its programs. Android applications do not contain a `main` method, but instead derive classes from certain pre-defined system classes and overwrite so-called *lifecycle methods* which allow the the Android middleware to, for instance, start, pause, or resume applications when necessary.

Currently Android comprises four different types of *components*: Activities, Services, Broadcast Receivers, and Content Providers. All of them have distinct lifecycles, defining ways for the operating system to influence the execution of the respective component. Nevertheless, the semantics of all four lifecycles is rather self-contained and well documented. The framework essentially just calls the implemented fraction of the lifecycle methods in a predefined, well-known order. The most complex lifecycle is the one of the activity component type, and even this one contains fewer than a dozen methods. We therefore leave the simulation of lifecycle-induced call-backs to the summary client. In our experiments, we use the concrete client FlowDroid, which handles lifecycle methods by simulating their effects through a generated dummy main method that simulates the lifecycle of the Android application [1, Section 3].

Notifications are special callbacks that allow the Android operating system to notify applications of system events like a battery shortage or an incoming text message. Sensors like GPS are also modeled through callbacks in the application: When the user moves around, a special interface in the application is called with the new coordinates of devices. These callbacks can be invoked by the operating system at any time while the respective host component is running, and they are provided through about 200 special-purpose interfaces. With our implementation we provide a list of these callback interfaces. To obtain a complete and precise list of callback methods, it is therefore sufficient to match all methods contained in interfaces in this list against the list of interface methods implemented in the target program. Clients must then simulate a call to all such methods at a well-known point in the lifecycle, at which the app is known to be in its *running* state. These callbacks only depend on external events like incoming SMS messages, and not on the behavior of the application under analysis. Thus, they can occur at any time while the app is running.

These observations conveniently reduce the problem of modeling the framework to analyzing the effects of framework methods called by the application without impeding correctness or precision of the obtained analysis results. Our previous publication on FlowDroid describes in detail how one can generate sound dummy-main methods that respect callbacks in Android [1]. A similar methodology must be followed for other Android client analyses.

7. APPLYING SUMMARIES IN CLIENTS

After the summaries have been computed once, they can be used in an arbitrary number of taint analyses on client programs or Android apps. The library code is then no longer required. StubDroid integrates into FlowDroid us-

```
1 void doLeak() {
2   Data data = new Data();
3   data.b.c.d = source();
4   Pair p = new Pair("foo", "bar");
5   p.setComplex(data);
6   sink(p.getO1().d);
7 }
```

Listing 6: Client Program for Pair Class

```
1 public void leakWithAliasing() {
2   Pair p = new Pair(new Data(), null);
3   Data d1 = (Data) p.getO1();
4   d1.b.c = source();
5   Data d2 = (Data) p.getO1();
6   leak(d2.b.c);
7 }
```

Listing 7: Library Summary Client with Aliases

ing the concept of *Taint Wrappers*. Taint wrappers are handlers for shortcut rules. In FlowDroid, they model external domain knowledge through an interface exposed by the taint-tracking engine. Whenever a method call is processed, the registered wrapper is asked whether it contains an explicit taint-propagation rule for the callee. StubDroid provides FlowDroid with a specialized taint wrapper implementation to inject the summary data during analysis. While this concept is specific to FlowDroid, other tools like CHEX [13] have similar extension points for explicit (library) method models and could thus use StubDroid's summaries in a similar fashion.

Assume the pair class from the motivating example in Listing 1 to be used in the program in Listing 6. This user code constructs an object of type `Data` and taints its field `b.c.d`. The data object is then passed to the `setComplex()` library method. This method is not part of the user code, but requires one to apply a library summary. Conceptually, the method copies the contents of the field `b.c` to `this.o1`. Therefore, the object containing the tainted data is returned by `getO1()` and leaked in line 6. Note that in this example the actual tainted data is stored in a sub-field `d` which is never touched by the library implementation.

Recall the summary for the `setComplex()` method computed in Sec. 5: `this.o1.*` is tainted if `Parameter 0.b.c.*` is tainted. As explained in Section 3, the asterisk serves as a placeholder. When applying the summary rule to the example client code, StubDroid will therefore match the asterisk with the actual fields of the longer incoming access path and derive the more precise rule `this.o1.d.*` is tainted if `Parameter 0.b.c.d.*` is tainted.

This rule can then easily be matched against the incoming taint. FlowDroid will query the taint wrapper for the access path `data.b.c.d.*`. StubDroid matches the argument variable `data` against `Parameter 0` and then applies the rule. It reports `p.o1.d.*` back to the taint analysis. Note that the rules directly create new taints on access paths. If the client code direct read the field `p.o1` instead of calling a getter method, this would be captured by the taint on `p.o1.d.*`.

7.1 Aliasing

Library methods and user-code methods may both taint heap objects which may, in turn, have aliases both inside and outside of the library. The example in Listing 7 uses a `Pair` object to store a `Data` object. The code then retrieves this data object as `d1`, and taints one of its inner fields. Afterwards, the same object is retrieved again as `d2` before the data is read out and leaked. If the `Pair` class were not a library class but a part of the client code, FlowDroid could directly find the leak. In FlowDroid, whenever a tainted value is assigned to a heap object, the data-flow engine automatically starts a backwards tracking to find aliases. In the example, it would inter-procedurally propagate the access path `d1.b.c.*` backwards to check whether this access path or some prefix of it is referenced on the right side of an assignment. All discovered aliases are then forward-propagated as first-class taints.

In Listing 7, however, the class `Pair` is abstracted away using a library summary. Therefore, FlowDroid cannot determine that `d1.b.c.*` aliases with `p.o1.b.c.*`. The relationship between `p.o1` and `d1` is encoded in `getO1()`, but when the backwards propagation reaches the call to `getO1()`, there is no callee to process. Therefore, such aliasing relationships must also be encoded in the summaries. For this reason, taint wrappers in FlowDroid provide an extension for aliasing. Alias summaries in StubDroid are just like flow summaries, only with an inverse propagation rule. For `Pair.getO1()`, there is already a rule: `return.*` is tainted if `this.o1.*` is tainted. Since the rule deals with heap objects, however, it can also be applied backwards: If `return.*` is tainted afterwards, `this.o1.*` may have been tainted before. In this direction, it encodes a may-alias relationship.

The final taint propagation in the example hence works as follows: When `p.getO1()` is called first, nothing inside the `p` object is tainted yet, so the rule does not yet apply. When FlowDroid queries StubDroid's taint wrapper for aliases of `d1.b.c.*`, StubDroid can apply the inverse of the rule and return a taint on `p.o1.b.c.*` to FlowDroid. This taint is then propagated forward. In Line 5, the normal summary rule for `getO1()` then applies and `d2.b.c.*` gets tainted. Therefore, FlowDroid can now detect the leak in Line 6.

To allow for more flexibility and some corner-cases (such as strings which are immutable), StubDroid stores a flag alongside every summary rule that indicates whether the flow may be inverted in order to answer alias queries.

7.2 Handling Incomplete Summaries

Note that taint wrappers can also be used with incomplete summaries. This is useful if libraries cannot be fully analyzed since, for instance, they depend on native code. In this case, analyzing the Java-based parts is still valuable, though it needs to be complemented with additional approximations. The FlowDroid client, for instance, supports two modes that determine how calls to methods are handled for which no summary is available.

In the so-called *conservative mode*, the return value of a method call is always considered as tainted if the base object on which the method is invoked (or any field inside it) is tainted. In the example this would lead to a sound analysis even if the summary for the `getData()` method was missing, but may come at the cost of reduced precision. Similarly, the `hashCode()` and `equals()` methods can also be over-approximated with simple rules even if there is no summary for them for a certain class.

FlowDroid also supports the `exclusive` flag which allows a taint-wrapper implementation to claim the taints it gener-

ates for a specific call site and incoming taint as complete. If the wrapper declares itself as exclusive, the analysis will not consider the callee's implementation even if it is available. By default, the StubDroid taint wrapper is exclusive for all methods for which it has at least one summary fact in its input XML file, because this indicates that the respective class has been fully analyzed by StubDroid and thus all existing data flows have been summarized.

8. EVALUATION

The summaries generated by StubDroid are maximally useful if they substantially reduce the time required to run the target analysis on a client program, if they do not reduce the precision of the analysis result (thus avoiding false positives), and if they are sound, i.e., do not introduce any false negatives. Our setup for computing the performance measures is explained in Section 8.1. The time required to compute a library summary is evaluated in section 8.2. Section 8.3 addresses the performance gains of using summaries and shows that StubDroid substantially reduces the time required for performing static analysis. In Section 8.4, we finally discuss the soundness and precision of our approach.

8.1 Experimental Setup

All performance experiments were carried out on a computation server featuring 40 virtual Xen CPU cores backed by Intel Xeon E5-4640 cores in physical hardware. The server was running Ubuntu 14.04 and Oracle's JVM version 1.7 in its default settings. Only the maximum heap size was set to 15 GB for summary generation and to 150 GB for analyzing apps. The large heap size for the app analysis was chosen to allow for a fair comparison with approaches that analyze the full Android library together with every app. Note that this may make some analyses perform less aggressive garbage collection than usual and thus report higher memory values than they would in more constrained scenarios.

To analyze the Android platform, we used an `android.jar` file manually built from a Galaxy Nexus device running Android 4.3. This is because the `android.jar` files included in the Android SDK as distributed by Google contain stub implementations only, which raise `NotImplementedException`s in every method. (They are only used to allow Android apps to link against the library interfaces.) As StubDroid is not only applicable to Android apps, but also to normal Java programs, we also evaluated it on the Java 8 (version 1.8.0_05) runtime library.

To evaluate the performance of StubDroid both in terms of summary application and summary generation, we computed summaries for the Android and the JDK implementations of the Java collections API. These classes are widely used in almost all applications, which is why modeling them is of high priority. Furthermore, these classes are rather large, and are thus suitable for assessing the scalability of the approach. The same holds for the string-processing classes, especially `java.lang.StringBuilder` which is used, e.g., to concatenate strings. With summaries for these two types of libraries, FlowDroid can successfully analyze most applications. All reported timings were averaged over 10 runs. The raw data is available on our project web page.

8.2 Summary Generation Performance

Generating the library summaries is a one-time effort that only needs to be repeated when the library is updated, which is rare in comparison to how often client applications using the library are analyzed. Nevertheless it is important that the summary generation is practically feasible. In Table 1, we report performance numbers on both the Android SDK and the Oracle JDK.

Our results show that StubDroid usually finishes in under three minutes per class for common Java collection APIs. For some of the concurrent collection implementations, the generation can take up to slightly over ten minutes. In these cases, the summaries generated by StubDroid are also considerably larger than those for their non-concurrent counterparts. This happens because the concurrent implementations need to assign additional internal synchronization fields which also become part of the summary and increase the complexity of the data flows in the respective methods.

Even though summaries could be centrally pre-computed on large servers, the memory requirements of StubDroid are modest. All summaries could be created within the 15 GB of heap space allotted, most of them using much less memory than available. This even applies to the large concurrent-collection classes such as the skip-lists.

Note that the Android and Oracle implementations of a particular class only share the specification, but not any source code. The versions shipped with the Android OS are especially optimized for resource-constrained devices. The differences are especially apparent in the number of flows generated for each class in Table 1. For `java.util.PriorityQueue`, StubDroid detects more than twice as many flows in the JDK implementation than in the one from Android.

8.3 Analysis Performance

We next evaluate how StubDroid can impact the performance of a client taint analysis. We therefore ran FlowDroid on a number of Android test apps in three different modes:

- Full analysis mode. In this mode, the full Android library was placed on the classpath and analyzed together with the app. Library summaries were not used.
- Hand-written summaries. In this mode, FlowDroid is run with its default hand-written summaries. This mode acts as a base-line as it corresponds to running the original FlowDroid implementation.
- StubDroid mode. In this mode, only library stubs were placed on the classpath and StubDroid's summaries were used to model the taint propagation over library call sites.

The hand-written summaries that FlowDroid uses by default not only cover the collection APIs but also a few other Android APIs such as cursors or intents. To allow for a fair comparison we generated StubDroid summaries for those APIs as well. The results in Table 2 show that StubDroid offers a performance that is comparable to FlowDroid's default hand-written summaries with a similarly high memory consumption; both analysis modes have comparable cost. In comparison to analyzing the full library implemenation together with every app, summaries provide a major decrease in runtime and memory consumption. In many cases, it even makes the analysis feasible; analyzing the app together with the full library implementation times out after 30 minutes.

The first two apps in Table 2 were taken from the DroidBench [1] micro-benchmark suite. (We chose those two apps of the suite that actually use library methods.) These apps are rather small, so the discrepancy between the size of the

Class	Generation Time (s)		Number of Flows		Memory (MB)	
	Oracle JDK	Android	Oracle JDK	Android	Oracle JDK	Android
java.util.ArrayDeque	45.82	52.76	82	88	1,086.52	2,203.27
java.util.ArrayList	37.95	42.52	72	72	1,086.52	2,323.40
java.util.HashMap	34.38	27.86	92	78	3,125.93	1,910.82
java.util.HashSet	51.46	45.04	140	81	3,125.93	1,910.82
java.util.LinkedHashMap	35.45	31.77	81	74	3,213.93	1,983.74
java.util.LinkedList	61.87	66.55	130	120	2,688.00	2,468.69
java.util.PriorityQueue	65.36	37.17	731	246	2,415.65	1,984.09
java.util.Stack	57.46	65.49	86	87	1,663.25	1,980.43
java.util.Vector	54.86	62.77	87	98	1,841.91	2,101.82
java.util.[...].ConcurrentHashMap	88.24	71.18	116	144	4,027.54	2,323.40
java.util.[...].ConcurrentLinkedQueue	64.23	35.58	36	40	1,509.77	1,761.86
java.util.[...].ConcurrentLinkedDeque	74.67	71.72	883	887	4,072.54	3,372.84
java.util.[...].ConcurrentSkipListMap	352.56	766.33	2206	2262	5,765.34	3,968.78
java.util.[...].ConcurrentSkipListSet	397.02	960.48	2234	3221	4,966.98	3,644.88
java.util.[...].DelayQueue	147.63	86.32	1745	807	4,864.34	3,106.15
java.lang.StringBuffer	92.06	81.73	397	309	1,212.16	2,621.19
java.lang.StringBuilder	86.35	81.72	443	305	1,617.32	2,425.11
Average	102.79	152.18	562	525	2,760.80	2,475.96

Table 1: Summary Generation Times for Android and JDK APIs

app and the size of the library is significant. If no summaries are used, the analysis spends most of its time in the library. The other apps in the table are real-world applications taken from the Google Play Store.

Data-flow analysis is especially useful to find privacy violations in potentially malicious applications. Many malware apps steal the user's unique device identifier (IMEI), his phone number, or other personally-identifiable values. To assess how StubDroid can help with finding such data thefts, we assess how it impacts the performance of FlowDroid on 258 apps from four different malware families inside the well-known Malware Genome Project [21]. Table 3 shows the average runtimes for each malware family. Due to space constraints, we only report values for the largest and most prevalent malware families. The column TO states the number of apps for which the analysis timed out after five minutes. In cases where all runs timed out, measurements are naturally not available (n/a). In cases where some runs timed out but not others, the numeric values indicate the average for the runs that did not time out.

The data in Table 3 indicates that analyzing the complete library together with every single app is infeasible and leads to timeouts in almost all cases. Using summaries, on the other hand, allows all but a handful of apps to be analyzed in under one minute. These time and memory savings, however, also depend on the precision of the summaries. For the *DroidDream Light* malware there are cases in which the hand-written summaries incur a higher memory consumption than analyzing the full library. This is because the imprecise summaries lead to severe over-tainting during the analysis. The precise summaries computed by StubDroid do not show this memory explosion. Thus, StubDroid can considerably save time and memory in comparison to a full analysis as well as in comparision to the hand-written rule sets currently used by most static data-flow analyses.

Note that FlowDroid cannot complete the analysis of large applications when configured to analyze the full library implementation together with the app. In the very same hardware configuration, computing the summaries and then analyzing the app with these summaries does, however, complete in a reasonable amount of time. This is primarily because the summaries are computed with one API method as entry point after the other, drastically reducing the peak memory requirements.

8.4 Relative Soundness and Precision

A summary approach such as StubDroid can only be as precise and as sound as the analysis on which it is based. We thus rather check that StubDroid preserves analysis results, by comparing the data-flow results of two setups: (1) FlowDroid applied to apps including the complete library implementation and (2) FlowDroid applied to apps without the runtime library but with StubDroid's summaries instead. Ideally the same results should be achieved. Recall that StubDroid creates summaries that are applicable to arbitrary client programs. It therefore abstracts from concrete call sequences and states. It is important to evaluate whether this generalization leads to a loss of precision or soundness. We performed these comparisons on all apps from Tables 2 and 3 on which the full analysis terminated. We confirmed that all flows detected by FlowDroid that involved StubDroid-summaries were equal to a full analysis of the target app plus the library. This means that replacing the library implementation with StubDroid's summaries does not incur any penalty in precision or soundness.

In the Hamburg Casino app, FlowDroid's hand-written summary rules caused 15 false positives out of 36 flows in total (41,7%) due to a single overly aggressive rule. Another rule caused 1 false positive out of 3 flows (33,3%) in the Broncos News app. The StubDroid summaries avoided all of these false positives.

9. RELATED WORK

In this Section, we compare StubDroid to existing approaches for summarizing library behavior (Section 9.1) and show a variety of existing work that can directly benefit from the summaries computed by StubDroid (Section 9.2).

9.1 Existing approaches for library summaries

The IFDS [15] framework, on which also FlowDroid is based, already computes low-level method summaries to improve efficiency if the same method is called multiple times in

Application	Full		Hand-Written		StubDroid	
	Time (s)	Memory (MB)	Time (s)	Memory (MB)	Time (s)	Memory (MB)
ArrayAccess1	21.13	458.19	5.75	128.10	5.63	132.66
HashMapAccess1	21.37	493.88	5.99	173.70	5.96	174.75
Alipay	45.10	6,271.49	5.57	1,727.78	5.51	1,640.99
Avira Antivirus	Timeout	Timeout	48.72	3,908.80	38.18	2,662.60
Broncos News	Timeout	Timeout	4.90	1,571.53	4.86	1,373.82
Hamburg Casino	Timeout	Timeout	57.73	3,352.54	48.17	3,856.88
OpenTable	Timeout	Timeout	81.51	7,596.06	78.01	5,669.13
Wikipedia	46.87	3,884.95	1.48	270.01	1.59	445.81

Table 2: Summary Application Performance (Benign Applications)

		Full			Hand-Written			StubDroid		
Family	Apps	TO	Time (s)	Mem (MB)	TO	Time (s)	Mem (MB)	TO	Time (s)	Mem (MB)
ADRD	22	22	n/a	n/a	0	6.70	3,669.00	0	1.84	1,004.86
BaseBridge	121	115	25.03	1,210.95	7	17.53	525.03	0	6.54	311.30
DroidDream Light	46	45	63.30	1,222.58	0	7.41	7,309.21	0	5.26	451.32
Geinimi	69	69	n/a	n/a	0	29.79	849.36	1	6.91	281.52

Table 3: Summary Application Performance (Malware), TO = # of apps where analysis timed out

the same program. These summaries, however, are linked to the concrete context of the client analysis. Therefore, there is no easy way to serialize and re-use these summaries for multiple analysis runs, let alone different analyses. Naeem and Lhoták present a method for summarizing alias-analysis information [14], but no further data-flow relationships. StubDroid handles aliasing along with data flows. Aliases of objects that get tainted inside a library are modeled as first-class taints which avoids a separate summary concept. Aliasing relationships with objects in the client program must be computed anew for every client when applying the library summary, so no further summarization is possible.

Zuhu et al. [22] analyze implementations of clients to automatically infer library specifications. As they state, however, this requires an external oracle (e.g. a user consulting the documentation) to verify every generated specification candidate since the library code is not regarded at all. StubDroid on the other hand is fully automatic and directly analyzes the library implementations.

Rountev et al. [16] construct library summaries for IDE [17] data-flow analyses by first conducting a data-flow analysis on the library and then abstracting away redundant data-flow facts that are internal to the library. Rountev's approach, however, does not discuss how summaries can actually be abstracted in such a way that they can be persisted and can become useful for different clients. In fact, it appears that in their experiments the implementation stores and reuses summaries only within one and the same analysis process.

F4F [18] by Sridharan et al. is a system for performing taint analyses on framework-based web applications. It provides a specification language for modelling both the framework behavior and information from configuration files. Sample generators for such specifications are given for a number of web application frameworks. While F4F focuses on dispatch logic between web pages and accesses to user controls on them, StubDroid analyzes and summarizes data flows in basic framework methods and is fully automated. No specialized specification generator must be developed and maintained when the target framework changes.

9.2 Approaches benefiting from summaries

Library summaries are required for various analysis tools such as CHEX [13], which scans applications for potential cases of datamisuse, e.g., when security vulnerabilities allow unauthorized access to an application's internal data. Apposcopy [3] detects Android malware based on semantic signatures describing data flows and inter-component communcations. This requires a precise flow detection which in turn needs precise library models. FlowDroid [1] is a data-flow tracker that can also analyze the library code together with the target application, but gains massive performance benefits from using library summaries as shown in this paper. AppSealer [20] is a tool for automatically patching component hijacking vulnerabilities in Android applications. It combines static and dynamic data-flow analyses to find vulnerable components to be patched. Library methods are handled using a coarse-grained default rule ("return value of method call is tainted if at least one parameter is tainted") with a few hand-written exceptions. Scandal [8] statically detects leaks of privacy-sensitive data in Android applications. The authors manually modeled the behavior of 220 commonly used Android framework methods to maintain precision. StubDroid automatically computes the library summaries required by these tools without the need for manual inspection of the library code. DroidSafe [5] requires the analyst to manually develop library stubs as Java code that are simplified versions of the original implementations. The authors have created 550 stub classes by hand which is a considerable effort that could be evaded by using StubDroid's data-flow summaries instead.

10. CONCLUSIONS

We have presented StubDroid, the first fully automated approach for inferring library specifications from binary distributions, e.g. of the Android operating system. The approach handles callbacks and aliasing. We have shown that the generated summaries are complete, precise, and help static taint analyses to save substantial computation time and memory. As future work, we we plan to further increase the performance of StubDroid and to apply it to other commonly-used libraries for Android app development.

Acknowledgements.

This work was supported by the BMBF within EC SPRIDE and by the DFG priority program Reliably Secure Software Systems (RS^3).

11. REFERENCES

[1] S. Arzt, S. Rasthofer, C. Fritz, 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 *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 29. ACM, 2014.

[2] W. Enck, P. Gilbert, B. gon 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*, pages 393–407, 2010.

[3] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware. Technical report, Stanford University, 2013. submitted for publication.

[4] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. Scandroid: Automated security certification of android applications. *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/avik/projects/scandroidascaa*, 2(3), 2009.

[5] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information-flow analysis of android applications in droidsafe. In *Proc. of the Network and Distributed System Security Symposium (NDSS). The Internet Society*, 2015.

[6] W. Huang, Y. D. A. Milanova, and J. Dolby. Scalable and precise taint analysis for android. Technical report, Technical report, Department of Computer Science, Rensselaer Polytechnic Institute, 2015.

[7] International Data Corporation. Worldwide quarterly mobile phone tracker 3q12, Nov. 2012. http://www.idc.com/tracker/showproductinfo.jsp?prod_id=37.

[8] J. Kim, Y. Yoon, K. Yi, and J. Shin. ScanDal: Static analyzer for detecting privacy leaks in android applications. In H. Chen, L. Koved, and D. S. Wallach, editors, *MoST 2012: Mobile Security Technologies 2012*, Los Alamitos, CA, USA, May 2012. IEEE.

[9] D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In R. Sekar and A. Pujari, editors, *Information Systems Security*, volume 5352 of *Lecture Notes in Computer Science*, pages 56–70. Springer Berlin Heidelberg, 2008.

[10] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[11] G. T. Leavens, A. L. Baker, and C. Ruby. Jml: A notation for detailed design. In *Behavioral specifications of Businesses and Systems*, pages 175–188. Springer, 1999.

[12] S. Lortz, H. Mantel, A. Starostin, T. Bähr, D. Schneider, and A. Weber. Cassandra: Towards a certifying app store for android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, pages 93–104. ACM, 2014.

[13] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM.

[14] N. A. Naeem and O. Lhoták. Faster alias set analysis using summaries. In J. Knoop, editor, *Compiler Construction*, volume 6601 of *Lecture Notes in Computer Science*, pages 82–103. Springer Berlin Heidelberg, 2011.

[15] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95*, pages 49–61, 1995.

[16] A. Rountev, M. Sharp, and G. Xu. Ide dataflow analysis in the presence of large object-oriented libraries. In L. Hendren, editor, *Compiler Construction*, volume 4959 of *Lecture Notes in Computer Science*, pages 53–68. Springer Berlin Heidelberg, 2008.

[17] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *TAPSOFT '95*, pages 131–170, 1996.

[18] 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 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 1053–1068, New York, NY, USA, 2011. ACM.

[19] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *FASE 2013*, pages 210–225, 2013.

[20] M. Zhang and H. Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. *Proceedings of the 21st Network and Distributed System Security (NDSS) Symposium*, 2014.

[21] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *SP '12*, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.

[22] H. Zhu, T. Dillig, and I. Dillig. Automated inference of library specifications for source-sink property verification. In C.-c. Shan, editor, *Programming Languages and Systems*, volume 8301 of *Lecture Notes in Computer Science*, pages 290–306. Springer International Publishing, 2013.