Static Analysis of Android Apps: A Systematic Literature Review

Li Li	University of Luxembourg / SnT / SERVAL, Luxembourg
Tegawendé F. Bissyandé	University of Luxembourg / SnT / SERVAL, Luxembourg
Mike Papadakis	University of Luxembourg / SnT / SERVAL, Luxembourg
Siegfried Rasthofer	TU Darmstadt / Fraunhofer SIT, Germany
Alexandre Bartel	TU Darmstadt / Fraunhofer SIT, Germany
Damien Octeau	University of Wisconsin and Penn. State University, USA
Jacques Klein	University of Luxembourg / SnT / SERVAL, Luxembourg
Yves Le Traon	University of Luxembourg / SnT / SERVAL, Luxembourg

20 April 2016

ISBN 978-2-87971-150-8 TR-SNT-2016-3

Static Analysis of Android Apps: A Systematic Literature Review

Li Li[a,1], Tegawendé F. Bissyandé[a], Mike Papadakis[a], Siegfried Rasthofer[b], Alexandre Bartel[b], Damien Octeau[c], Jacques Klein[a], Yves Le Traon[a]

[a]Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg
[b]Fraunhofer SIT, Technische Universität Darmstadt, Germany
[c]University of Wisconsin and Pennsylvania State University

Abstract

Context: Static analysis approaches have been proposed to assess the security of Android apps, by searching for known vulnerabilities or actual malicious code. The literature thus has proposed a large body of works, each of which attempts to tackle one or more of the several challenges that program analyzers face when dealing with Android apps.

Objective: We aim to provide a clear view of the state-of-the-art works that statically analyze Android apps, from which we highlight the trends of static analysis approaches, pinpoint where the focus has been put and enumerate the key aspects where future researches are still needed.

Method: We have performed a systematic literature review which involves studying around 90 research papers published in software engineering, programming languages and security venues. This review is performed mainly in five dimensions: problems targeted by the approach, fundamental techniques used by authors, static analysis sensitivities considered, android characteristics taken into account and the scale of evaluation performed.

Results: Our in-depth examination have led to several key findings: 1) Static analysis is largely performed to uncover security and privacy issues; 2) The Soot framework and the Jimple intermediate representation are the most adopted basic support tool and format, respectively; 3) Taint analysis remains the most applied technique in research approaches; 4) Most approaches support several analysis sensitivities, but very few approaches consider path-sensitivity; 5) There is no single work that has been proposed to tackle all challenges of static analysis that are related to Android programming; and 6) Only a small portion of state-of-the-art works have made their artifacts publicly available.

Conclusion: The research community is still facing a number of challenges for building approaches that are aware altogether of implicit-Flows, dynamic code loading features, reflective calls, native code and multi-threading, in order to implement sound and highly precise static analyzers.

1. Introduction

Since its first commercial release in September 2008, the Android mobile operating system has witnessed a steady adoption by the manufacturing industry, mobile users, and the software development community. Just a few years later, in 2015, there were over one billion monthly active Android users, meanwhile its official market (Google Play) listed more than 1.5 million apps. This adoption is further realised at the expense of traditional mobile systems, such as iPhones, since Android occupied 83.1% of the mobile device sales in the third quarter of 2014 [1], driving a momentum which has created a shift in the development community to place Android as a "priority" target platform [2].

Because Android apps now pervade all user activities, ill-designed and malicious apps have become big threats that can lead to damages of varying severity (e.g., app crashes, financial losses with malware sending premium-rate SMS, reputation issues with private data leaks, etc). Data from anti-virus vendors and security experts regularly report on the rise of malware in the Android ecosystem. For example, G DATA has reported that the 560,671 new Android malware samples collected in the second quarter of 2015 revealed a 27% increase, compared to the malware distributed in the first quarter of the same year [3].

To deal with the aforementioned threats, the research community has investigated various aspects of Android development, and proposed a wide range of program analyses to identify syntactical errors and semantic bugs [4], to discover sensitive data leaks [5, 6], to uncover vulnerabilities [7, 8], etc. In most cases, these analyses are performed statically, i.e., without actually running the Android app code, in order not only to ensure scalability when targeting thousands of apps in stores, but also to guarantee a traversal of all possible execution paths. Unfortunately, static analysis of Android programs is not a trivial endeavour since one must account for several specific features of Android, to ensure both soundness and completeness of the analysis. Common barriers to the design and implementation of performant tools include the need to support Dalvik bytecode analysis or translation, the absence of a main entry point to start the call graph construction and the constraint to account for event handlers through which the whole Android program

Email address: `li.li@uni.lu` (Li Li)
[1]Corresponding author.

works. Besides these specific challenges, Android carries a number of challenges for the analysis of Java programs, such as how to resolve reflective calls and deal with dynamic code loading. Thus, despite much efforts in the community, state-of-the-art tools are still challenged by their lack of support for some analysis features. For example, the state-of-the-art FlowDroid [5] taint analyzer cannot track data leaks across components since it is unaware of the Android Inter-Component Communication (ICC) scheme. More recent tools which focus on ICC calls may not account for reflective calls.

Because of the variety of concerns in static analysis of Android apps, it is important for the field, which has already produced substantial amount of approaches and tools, to reflect on what has already been done, and on what remains to do. Although a recent survey [9] on securing Android has mentioned some well-known static analysis approaches, a significant part of current works has been skipped from the study. Furthermore, the study only focused on general aspects of Android security research, neglecting basic characteristics about the static analyses used, and missing an in-depth analysis of the support for some Android-specific features (e.g., *XML Layout*, or *ICC*).

This paper is an attempt to fulfill the need of a comprehensive study for static analysis of Android apps. To reach our goal, we performed a systematic literature review of such approaches. After identifying thoroughly the set of related research publications, we perform a trend analysis and provide a detailed overview on key aspects of static analysis of Android apps such as the characteristics of static analysis, the Android-specific features, the addressed problems (e.g. security or energy leaks) and also some evaluation-based empirical results. Finally, we summarize the current limitations of static analysis of Android apps and point out potential new research directions.

The main contributions of this paper are:

- We build a comprehensive and searchable repository of research works dealing with static analysis for Android apps. These works are categorized following several criteria related to their support of common analysis characteristics as well as Android-specific concerns.
- We analyze in detail key aspects of static analysis to summarize the research efforts and the reached results.
- We further enumerate the limitations of current static analysis approaches (on Android) and provide insights for potential new research directions.
- Finally, we provide a trend analysis on this research field to report on the latest focus as well as the level of maturity in key analysis problems.

2. Background Information on Android and Static Analysis

We now provide to the reader the preliminary details which are necessary to understand the purpose, techniques and key concerns of the various research work that we have reviewed. Mainly, we summarize the different aspects of static analysis in general in Section 2.1 before revisiting some details of the Android programming model in Section 2.2.

2.1. Concepts of Static Program Analysis

Static program analysis generally involves an automated tool that takes as input the source code (or object code in some cases) of a program, examines this code without executing it, and yields results by checking the code structure, the sequences of statements, and how variable values are processed throughout the different function calls. The main advantage of static analysis is that it can reveal errors (or vulnerabilities) that do not manifest themselves (or are not exploited) until long after the software is released to the public. A typical static analysis process is run as follows: 1) first, a call graph (CG) is built from the analyzed program by representing an abstraction of the calling relationships between the subroutines (e.g., methods in Java) of the program; 2) then, a Control Flow Graph (CFG) can be built to include more fine-grained details of the structure of the whole program, e.g., by making explicit all the paths inside a subroutine; 3) finally, other information, such as the values of variables at different points of the CFG, can be collected to allow the static analysis to support more in-depth verification, e.g., through data-flow or alias analysis.

In the following, we detail essential concepts of static analysis, as we mentioned above, summarizing the main techniques that are implemented (in Section 2.1.1), and covering the construction of the CG and CFG (in Section 2.1.2) and their enrichment with context information (in Section 2.1.3). This section only summarizes key aspects about static analysis. For more details, interested readers are encouraged to refer to the doctoral dissertation of Alexandre Bartel [10], a co-author of this literature review.

2.1.1. Analysis Techniques

Control-flow analysis. A control-flow analysis is a technique to show how hierarchical flow of control within a given program are sequenced, making all possible execution paths of a program are analyzable. Usually, the control sequences are expressed as a control-flow graph (CFG), where each node represents a basic block of code (statement or instruction) while each directed edge indicates a possible flow of control between two nodes.

Data-flow analysis. A data-flow analysis [11] is a technique to compute at every point in a program a set of possible values. This set of values depends on the kind of problem that has to be solved using data-flow analysis. For instance, in the *reaching definition problem*, one wants to know the set of definitions (e.g., statements such as `int x = 3;`) reachable at every program point. In that particular problem, the set of possible values at program point `P` is the set of definitions that reaches `P` (i.e., the variable is not redefined before it reaches `P`).

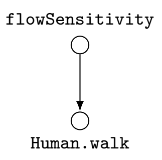
Points-to analysis. Points-to analysis consists of computing a static abstraction of all the data that a pointer expression (or just a variable) can point to during program run-time.

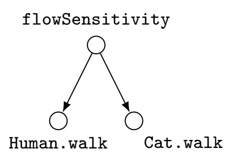
2.1.2. Call-Graph Construction

Because Android supports the object-oriented programming scheme with the Java language, in the remainder of this section

Code Snippet | Sensitive Call-Graph | Insensitive Call-Graph

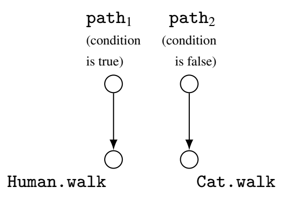
```java
public void flowSensitivity() {
  Animal a = new Human();
  a.walk();
  a = new Cat();
}
```

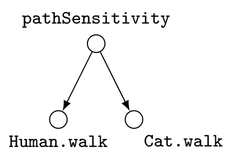




(a) Flow Sensitivity

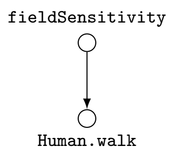
```java
public void pathSensitivity() {
  Animal a = null;
  if (condition) {
   a = new Human();
  } else {
   a = new Cat();
  }
  a.walk();
}
```

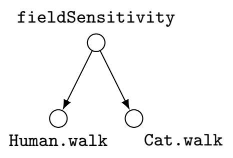




(b) Path Sensitivity

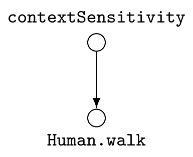
```java
public void fieldSensitivity()
     {
  C c1 = new C();
  C c2 = new C();
  c1.f1 = new Human();
  c2.f1 = new Cat();
  c1.f1.walk();
}
public class C {
  Animal f1;
}
```

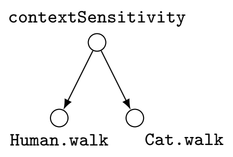




(c) Field Sensitivity

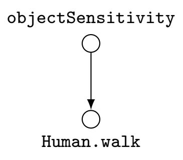
```java
public void
     contextSensitivity() {
  Human h = new Human();
  Cat c = new Cat();
  Animal a = method(c);
  a = method(h);
  a.walk();
}
public Animal method(Animal a)
     {
  return a;
}
```

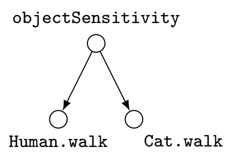




(d) Context Sensitivity

```java
public void
     objectSensitivity() {
  Contains c1 = new Contains();
  Contains c2 = new Contains();
  c1.setAnimal(new Human());
  c2.setAnimal(new Cat());
  c1.animal.walk();
}
public class Contains {
  Animal animal;
  public void setAnimal(Animal
     a) {
    this.animal = a;
  }
}
```





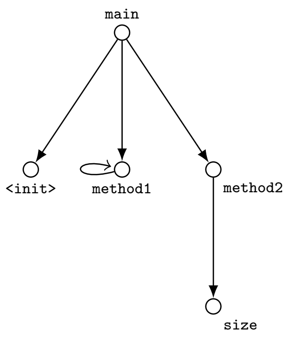
(e) Object Sensitty

Figure 1: Five Examples of Sensitivity cases in Static Analysis of Object-Oriented Programs.

```java
public class MyOjbect {
  public static void main(String[]
       args) {
    MyOtherObject o = new
       MyOtherObject();
    if (args.length == 2) {
      o.method1(2);
    } else {
      o.method2("hi!");
    }
  }
}

public class MyOtherObject {
  int a = 0;
  public MyOtherObject() {
    this.a = 3;
  }
  public void method1(int i) {
    this.a += i;
    if (i == 55)
      this.method1(55)
  }
  public void method2(String s) {
    this.a += s.size();
  }
  public void method3(int j) {
    this.method2(j);
    this.method2(j);
  }
}
```

(a) A Java program



(b) Corresponding Call Graph

Figure 2: Source Code of a two-Classes Java program and its Call Graph Generated from the main Method

we focus on the analysis of Object-Oriented programs. Such programs are made of classes, each representing a concept (e.g. a car) and including a set of fields to represent other objects (e.g., wheels) and a set of methods containing code to manipulate objects (e.g, drive the car forward). The code in a class method can call other methods to create instances of objects or manipulate existing objects.

A program usually starts with a single entry point referred to in Java as the `main` method. A quick inspection of the main method's code can list the method(s) that it calls. Then, iterating this process on the code of the called methods leads to the construction of a directed graph (e.g., see Figure 2), commonly known as the *call graph* in program analysis. Although the concept of a call graph is standard in static analysis approaches, different concerns, such as precision requirements, may impact the algorithms used to construct a program's call graph. For Java programs, a number of popular algorithms have been proposed, including CHA [12], RTA [13], VTA [14], Andersen [15], Steensguard [16], etc., each being more or less sensitive to different properties of program executions. We detail some of the main properties below to allow a clear differentiation between research works in the literature. These properties are illustrated in Figure 1 with example code snippets and the corresponding call graphs extracted in both cases where the property holds and where it does not.

Flow Sensitivity. A flow-sensitive CG is a CG that is aware of the order of program statements. In the illustrative example of Figure 1a, a *Human* instance is first created and referred to by the *Animal* reference `a`. Then, method `walk` is called on `a`. At execution time, only method `Human.walk` is called at this point (line 3). Subsequently, the program associates the variable `a` to a new instance of *Cat*. In the construction of the CG we are interested in the building a directed graphs between method calls, i.e., `flowSensitivity` (line 1) and `walk` (line 3) in our case. When the CG is flow-sensitive, it contains a single edge since, at line 3, `a` can only refer to a *Human* object . If the CG is flow-insensitive, then, switching the order of positions of statements can be switched between lines 3 and 4. Thus, the CG must consider the case where `a` refers to a *Cat* object when `a.walk` is called. The flow-insensitive CG therefore contains two edges: one from `flowSensitivity` to `Human.walk` and another to `Cat.walk`. Obviously, while in some cases a flow-insensitive CG may be sufficient (e.g., computation of the depth of a CG), in other cases, it brings imprecision which will necessarily lead to false positives in the analysis.

Path Sensitivity. A path-sensitive CG takes the execution path into account. In the illustrative example of Figure 1b, depending on the value of the condition in line 3, when the execution reaches line 8, `a` may refer to a *Human* object or a *Cat* object. Thus, when path-sensitivity is taken into account, two graphs must be produced, one for each path: in path$_1$, at line 8, `a` points to a *Human* object and thus method `Human.walk` is the one included in the CG. On the other hand, in path$_2$, `a` points to a *Cat* object and thus method `Cat.walk` is in the call graph. In contrast, when building a path-insensitive CG, at line 8, `a` points to both a *Human* object and a *Cat*, and the graph would thus contain both method `Human.walk` and method `Cat.walk`. Overall, path-sensitivity brings a scalability challenge for large programs where there can be an exponential of execution paths.

Field Sensitivity. A field-sensitive approach models each field of each object. Take the code snippet of Figure 1c as an example. At lines 2 and 3, `c1` and `c2` are separately assigned to new *C* objects, which contain a *Animal* field. At line 4, the field of `c1`, (i.e., `c1.f1`), points to a new *Human* object while at line 5, the field of `c2`, (i.e., `c2.f1`), points to a new *Cat* object. As a result of field-sensitive analysis, at line 6, the model of `c1.f1` can only point to a *Human* object and only method `Human.walk` is in the field-sensitive call graph. On the other hand, a field-insensitive approach, which only models each field of each class of objects[2]. This means that in the example field `c1.f1` and `c2.f1` have the same model. Thus, at line 5 `f1` points to a *Human* object and a *Cat* object and both method `Human.walk` and `Cat.walk` are in the field-insensitive call graph.

2.1.3. Graph Enrichment

During, or after, call-graph construction, the static analysis purposes may require supplementary information about the *context* in which the different methods are called. In particular, this context can be modeled by considering the call site (i.e., *context sensitivity*) or by modeling the allocation site of method objects (i.e., *object sensitivity*).

[2]Theoretically, a field-insensitive analysis may not even take fields into consideration. However, this kind of analysis is unlikely to be used with object-oriented languages like Java/Android. Thus, in this work, we take all the cases that are not field-sensitivity as field-insensitivity.

Context Sensitivity. In a context-sensitive analysis, when analysing the target of a function call, one keeps track of the calling context. This information may allow to go back and forth to and from the original call site with precision, instead of trying out all possible call sites in the program. In the illustrative example of Figure 1d, at line 6, method `walk` is called by object `a`. Considering a context-sensitive analysis, each method call is modeled independently. That is, for the first method call (line 4), the model of the parameter points to `c` and the return value model points to `c`. For the second method call (line 5), the model of the parameter points to `h` and the return value model points to `h`. Thus, only method `Human.walk` is added to the call graph. On the other hand, a context-insensitive analysis has only a single model of the parameter and a single model of the return value for a given method. Consequently, in a context-insensitive analysis the model of the parameter points to `c` and `h` and the return value to `c` and `h`. Thus, a context-insensitive approach has both methods `Human.walk` and `Cat.walk` in the call graph.

Object Sensitivity. An object-sensitive approach is a context-sensitive approach that distinguishes invocations of methods made on different objects. Take the code snippet of Figure 1e as an example. At line two and three, two *Contains* objects are instantiated. Variables `c1` and `c2` refer to these objects. The class *Contains* has an instance field `animal` of type *Animal* and an instance method `setAnimal` to associate a value with field `animal`. At line four, method `setAnimal` is called on `c1` with a *Human* object as parameter. At line five, method `setAnimal` is called on `c2` with a *Cat* object as parameter. Finally, at line six, method `walk` is called on the `animal` field of object `c1`. At lines four and five, an object-insensitive approach would consider `c1` and `c2` as the same receiver. The result would be that the method calls at line four and six cannot distinguish between the receiver and model `c1` and `c2` as a unique object of type *Contains*. Thus, method `walk` called at line six is represented by two methods in the call graph: `Human.walk` and `Cat.walk`. On the other hand, an object-sensitive approach would have model `c1` and `c2` separately for each call of `setAnimal`. Thus, the call at line six would only be represented by method `Human.walk` in the call graph.

2.2. Static Analysis of Android Programs

Android apps are made up of components. Figure 3 illustrates the four different types of components and their possible interacctions:

1. an *Activity* represents the visible part of Android apps, the user interfaces;
2. a *Service*, which is dedicated to execute (time-intensive) tasks in the background;
3. a *Broadcast Receiver* waits to receive user-specific events as well as system events (e.g., the phone is rebooted);
4. a *Content Provider* acts as a standard interface for other components/apps to access structured data.

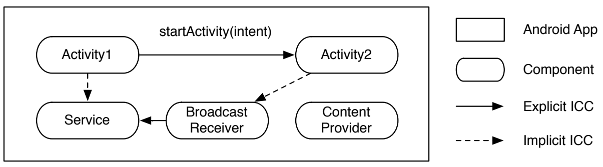


Figure 3: Overview of basic concepts of Android apps.

Android components communicate with one another through specific methods, such as *startActivity()*, which are used to trigger inter-component communications (ICC). ICC methods take an Intent object as a parameter which includes information about the target component that the source component wants to communicate with. There are two types of ICC interactions: *explicit* ICC where the intent object contains the name of the target component, and *implicit* ICC where the intent object specifies instead the capability/action that the target component must have (e.g., a web browser to open a url). In order for a component to be considered as a potential target of an implicit ICC, it must specify an *Intent Filter* in its Manifest configuration file, declaring what kind of Intents it is capable of handling, i.e., what kind of actions it can perform.

2.2.1. Android-specific Analysis Challenges

We now enumerate some challenges for static analysis that are mainly due to Android peculiarities.

Dalvik bytecode. Although Android apps are primarily developed in Java, they run in a Dalvik virtual machine. Thus, all app packages (apks) are distributed on markets with Dalvik bytecode, and only a relatively few are distributed with source code in open source repositories. Consequently, a static analyzer for Android must be capable of directly tackling Dalvik bytecode, or at least of translating it to a supported format. Thus, most Java and Java bytecode analyzers, which could have been leveraged, are actually useless in the Android ecosystem. As an example, the mature FindBugs[3] tool, which has demonstrated its capabilities to discover bugs in Java bytecode, can not readily be exploited for Android programs.

Program entry point. Unlike programs in most general programming languages such as Java and C, Android apps do not have a `main` method. Instead, each app program contains several entry points which are implicitly called by the Android framework at runtime. Consequently, it is tedious for a static analyzer to build a global call graph of the app. Instead, the analyzer must first search for all entry-points and build several call graphs with no assurance on how these graphs connect to each other, if ever.

Component Lifecycle. In Android, unlike in Java or C, different components of an application, have their own lifecycle. Each component indeed implements its lifecyle methods which

[3]http://findbugs.sourceforge.net

are called by the Android system to start/stop/resume the component following environment needs. For example, an application in background (i.e., invisible lifetime), can first be stopped, when the system is under memory pressure, and later be restarted when the user attempts to put it in foreground. Unfortunately, because these lifecycle methods are not directly connected to the execution flow, they hinder the soundness of some analysis scenarios.

Inter-Component Communication (ICC). Android has introduced a special mechanism for allowing an application's components to exchange messages through the system to components of the same application or of other applications. This communication is usually triggered by specific methods, hereafter referred to as ICC methods. ICC methods use a special parameter, containing all necessary information, to specify their target components and the action requested. Similarly to the lifecycle methods, ICC methods are actually processed by the system who is in charge of resolving and brokering it at runtime. Consequently, static analyzer will find it hazardous to hypothesize on how components connect to one another unless using advanced heuristics. As an example, FlowDroid, one of the most-advanced static analyzers for Android, fails to take into account ICCs in its analysis.

Libraries. An android apk is a standalone package containing a Dalvik bytecode consisting of the actual app code and all library suites, such as advertisement libraries and burdensome frameworks. These libraries may represent thousands of lines of code, leading to the size of actual app to be significantly smaller than the included libraries. This situation causes two major difficulties: (1) the analysis of an app may spend more time vetting library code than the real code; (2) the analysis results may comprise two many false positives due to the analysis of library "dead code". As an example, analyzing all method calls in an apk to discover the set of permissions required may lead to listing permissions which are not actually necessary for the actual app code.

2.2.2. Java-inherited Challenges

Since Android apps are mainly written in Java, developers of static analyzers for such apps are faced with the same challenges as with Java programs, including the issues of handling dynamic code loading, reflection, native code integration, multithreading and the support of polymorphism.

Reflection. In the case of dynamic code loading and reflective calls, it is currently difficult to statically handle them. The classes that are loaded at runtime are often practically impossible to analyze since they often sit in remote locations, or may be generated on the fly.

Native Code. Addressing the issue of native code is a different research adventure. Most of the time, such code comes in a compiled binary format, making it difficult to analyze.

Multi-threading. Analyzing multithreaded programs is challenging as it is complicated to characterize the effect of the interactions between threads. Besides, to analyze all interleavings of statements from parallel threads usually result in an exponential analysis times.

Polymorphism. Finally, polymorphic features also add extra difficulties for static analysis. As an example, let us assume that method m_1 of class A has been overridden in class B (B extends A). For statement $a.m_1()$, where a is an instance of A, a static analyzer in default will consider the body of $m_1()$ in A instead of the actual body of $m_1()$ in B, even a was instantiated from B (e.g., with $A\ a\ =\ new\ B()$). This obvious situation is however tedious to resolve in practice by most static analyzers and thus leads to unsound results.

3. Methodology for the SLR

The methodology that we followed for this SLR is based on the guidelines provided by Kitchenham [17]. Figure 4 illustrates the protocol that we have designed to conduct the SLR:

- In a first step we define the research questions motivating this SLR, and subsequently identify the relevant information to collect from the publications in the literature. (cf. Section 3.1)
- Then, we enumerate the different search keywords that will allow us to crawl the largest possible set of relevant publications within the scope of this SLR.
- The search process itself is conducted following two scenarios: the first one considers the well-known publication repositories, while the second one focuses on the lists of publications from top venues, including both conferences and journals. (cf. Section 3.2)
- To limit our study to very relevant papers, we apply exclusion criteria on the search results, thus filtering out papers of likely limited interest. (cf. Section 3.3)
- Then we merge the sets of results from both search scenarios to produce the overall list of publications to review.
- Finally, we further consolidate this list by applying another set of exclusion criteria based on the content of the papers' abstracts. The final list of papers is hereafter referred to as *primary publications/studies*. (cf. Section 3.4)

Given the high number of publications relevant to the systematic literature review that we undertake to conduct, we must devise a strategy of review which guarantees that each publication is investigated thoroughly and that the extracted information is reliable. To that end, we further proceed with the following steps:

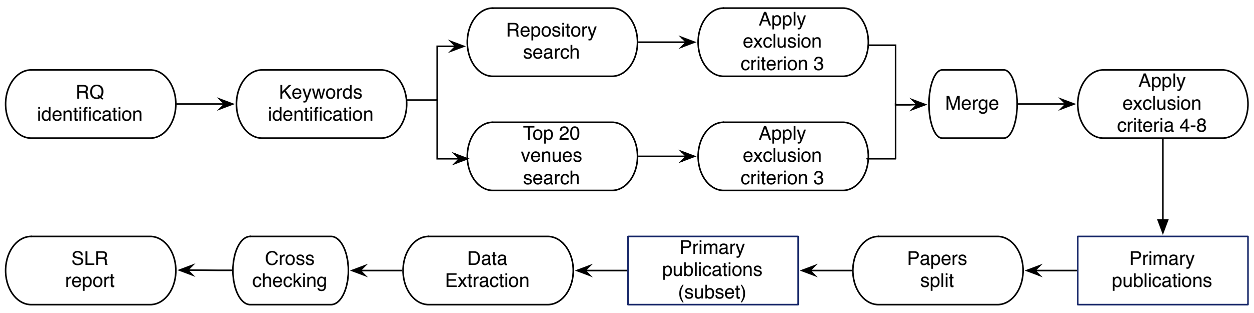


Figure 4: Overview of our SLR process.

- First, we assign the primary publications to the authors of this SLR who will share the heavy workload of paper examinations.
- Then, each primary publication is fully reviewed by the SLR author to whom it was attributed. Based on their reviews, each SLR author must fill a common spreadsheet with relevant information in categories that were previously enumerated.
- To ensure that the review information for a given paper is reliable, we first cross-check these reviews among reviewers. Then, once all information is collected, we engage in a *self-checking process* where we forward our findings to the authors of the reviewed papers. These authors then confirm our investigation results or demonstrate any inaccuracies in the classifications.
- Eventually, we report on the findings to the research community.

3.1. Research Questions

This SLR aims to address the following research questions:

RQ1: What are the purposes of the Analyses? With this research question, we will survey the various issues targeted by static analysis, i.e., the concerns in the Android ecosystem that researchers attempt to resolve by statically analyzing app code.

RQ2: How are the analyses designed and implemented? In this second research question, we study in detail the in-depth analysis that are developed by researchers. To that end, we investigate the following sub-questions:

RQ 2.1: What code representations are used in the analysis process? To facilitate analysis with existing frameworks, analyses often require that the app byte code is translated back to Java or other intermediate representations.

RQ 2.2: What fundamental techniques are used by the community of static analysis of Android apps?

RQ 2.3: What sensitivity-related metrics are applied?

RQ 2.4: What Android-specific characteristics are taken into account?

RQ3: Are the research outputs available and usable? With this research question, we are interested in investigating whether the developed tools are readily available to practitioners and/or the reported experiments can be reproduced by other researchers. For each technical contribution, we check that the data sets used in the validation of approaches are available, and that the experimental protocol is described in detail.

RQ4: What challenges remain to be addressed? Finally, with this fourth research question we survey the issues that have not yet benefited from a significant research effort. To that end, we investigate the following questions:

RQ 4.1: To what extent are the enumerated analysis challenges covered? We survey the proportion of research approaches that account for reflective calls, native code, multithreading, etc.

RQ 4.2: What are the trends in the analyses? We study how the focus of researchers evolved over time and whether this correlates with the needs of practitioners.

3.2. Search Strategy

We now detail the search keywords and the datasets that we leveraged for the search of our relevant publications.

3.2.1. Search keywords

Thanks to the research questions outlined in Section 3.1, we can summarize our search terms with keywords that are (1) related to analysis activities, or (2) related to key aspects of static analysis, and (3) to the target programs. Table 1 depicts the actual keywords that we used based on a manual investigation of some relevant publications.

Table 1: Search keywords.

Line	Keywords
1	Analysis; Analyz*; Analys*;
2	Data-Flow; ”Data Flow*”; Control-Flow; ”Control Flow*”; ”Information-Flow*”; ”Information Flow*”; Static*; Taint;
3	Android; Mobile; Smartphone*; ”Smart Phone*”;

Our search string s is formed as a conjunction of the three lines of keywords, i.e., $s =: l_1$ **AND** l_2 **AND** l_3, where each line is represented as a disjunction of its keywords, e.g., $l_1 =:$ {*Analysis* **OR** *Analyz** **OR** *Analys**}.

3.2.2. Search datasets

As shown in Fig. 4, our data search is conducted in two scenarios: repository search and top venue search. We now detail them separately.

Repository Search. To find datasets of publications we first leverage five well-known electronic repositories, namely ACM Digital Library[4], IEEE Xplore Digital Library[5], SpringerLink[6], Web of Knowledge[7] and ScienceDirect[8]. Because in some cases the repository search engine imposes a limit to the amount of search result meta-data that can be downloaded, we consider, for such cases, splitting the search string and iterating until we collect all relevant meta-data of publications. For example, SpringerLink only allows to collect information on the first 1,000 items from its search results. Unfortunately, by applying our predefined search string, we get more than 10,000 results on this repository. Consequently, we must split our search string to narrow down the findings and afterwords combine all the findings into a final set. In other cases, such as with ACM Digital Library, where the repository search engine does not provide a way to download a batch of search results (meta-data), we resort to python scripts to scale up the collection of relevant publications.

Top Venue Search. A few conference and journal venues, such as the Network and Distributed System Security Symposium, have policies (e.g., open proceedings) that make their publications unavailable in the previously listed electronic repositories. Thus, to ensure that our search results are, to some extent, exhaustive, we consider all publications from well-known venues. For this SLR we have considered the top[9] 20 venues: 10 venues are from the field of software engineering and programming languages while the other 10 venues are from the security and privacy field. Table 2 lists these venues considered at the time of review, where some venues on cryptography fields (including EUROCRYPT, CRYPTO, TCC, CHES and PKC), parallel programming (PPoPP), magazines (such as IEEE Software) and non-official proceedings (e.g., arXiv Cryptography and Security) are excluded. Because those venues are mainly not the focuses of static analysis of Android apps. The *H5-index* in Table 2 is defined by Google Scholar as a special h-index where only those of its articles published in the last 5 complete calendar years (in our case is from 2010 to 2014) are considered. The h-index of a publication is the largest number h such that at least h articles in that publication were cited at least h times each [18]. Intuitively, the higher *H5-index*, the better the venue.

Our top 20 venues search is performed on DBLP[10]. We only use such keywords that are listed in line 3 in Table 1 for this search, as DBLP provides papers' title only, it is not necessary for us to use the same keywords that we use in the repository search step. Ideally, all the papers that are related to smartphones (including Android, Windows, iOS and so on) are taken into account. As a result, this coarse-granularity strategy has introduced some irrelevant papers (e.g., papers that analyze iOS apps). Fortunately, because of the small number of venues, we are able to manually exclude those irrelevant papers from our interesting set, more details are given in the next section.

[4]`http://dl.acm.org`

[5]`http://ieeexplore.ieee.org`

[6]`http://link.springer.com`

[7]`http://apps.webofknowledge.com`

[8]`http://www.sciencedirect.com`

[9]Following Google Scholar Metrics: `https://scholar.google.lu/citations?view_op=top_venues&hl=en`

[10]http://dblp.uni-trier.de

Table 2: The top 20 venues including both conference proceedings and journals in SE/PL and S&P fields.

Acronym	Full Name	H5-index
	Software Engineering and Programming Languages (SE/PL)	
ICSE	International Conference on Software Engineering	57
TSE	IEEE Transactions on Software Engineering	47
PLDI	SIGPLAN Conference on Programming Language Design and Implementation	46
IST	Information and Software Technology	45
POPL	ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages	45
JSS	Journal of Systems and Software	41
FSE	Foundations of Software Engineering	38
OOPSLA	Conference on Object-Oriented Programming Systems, Languages, and Applications	34
ISSTA	International Symposium on Software Testing and Analysis	31
TACAS	International Conference on Tools and Algorithms for the Construction and Analysis of Systems	31
	Security and Privacy (S&P)	
CCS	ACM Conference on Computer and Communications Security	65
S&P	IEEE Symposium on Security and Privacy	53
SEC	USENIX Security Symposium	51
TISSEC	IEEE Transactions on Information Forensics and Security	47
NDSS	Network and Distributed System Security Symposium	39
TDSC	IEEE Transactions on Dependable and Secure Computing	39
ASIACRYPT	International Conference on The Theory and Application of Cryptology and Information Security	34
COMPSEC	Computers & Security	34
ACSAC	Annual Computer Security Applications Conference	29
SOUPS	Symposium On Usable Privacy and Security	29

3.3. *Exclusion Criteria*

The search terms provided above are purposely broad to allow the collection of a near exhaustive list of publications. However, this broadness also suggests that many of the search results may actually be irrelevant and focus on the primary publications. For our SLR we use the following exclusion criteria:

1. First to account for the common language spoken by the reviewers, and the fact that most of today's scientific works are published in English, we filter out *non-English* written publications.
2. Second, we want to focus on extensive works with detailed publications. Thus, we exclude *short papers*, i.e., heuristically, papers with less than 7 pages in LNCS single-column format or with less than 5 pages in IEEE/ACM-like double-column format. Further, it should be noted that such papers are often preliminary work that are later published in full format and are thus likely to be included in the final set.
3. Third, related to the second exclusion criteria, we attempt to identify *duplicate papers* for exclusion. Usually, those are papers published in the context of a conference venue and extended to a journal venue. We look for such papers by first comparing systematically the lists of authors, paper titles and abstract texts. Then we manually check that suspicious pairs of identified papers share a lot of content or not. When duplication is confirmed we filter out the less extensive publication.
4. Fourth, because our search terms include "mobile" to collect most papers, the collected set includes papers about "mobile networking" or iOS/Windows platforms. We exclude such *non Android-related* papers. This exclusion

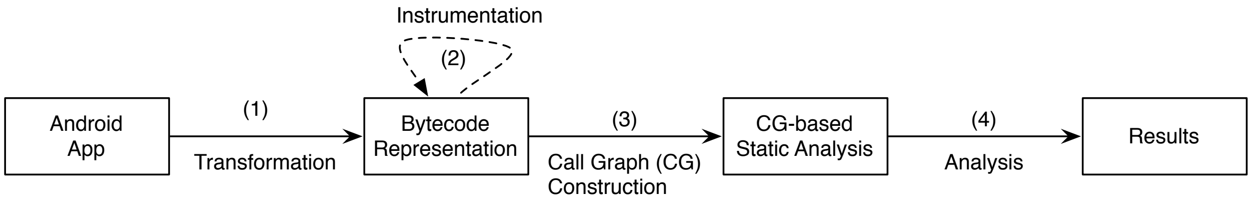


Figure 5: Inclusion Criteria. Papers that fall into paths (1) → [(2) →] (3) → (4) are selected. Besides, we also take into account such papers that focus themselves on step (1) or (2). Because these approaches tackle the fundamental part of static analysis, meaning they are essential and reusable for other static analysis approaches.

Table 3: Summary of the selection of primary publications. The total number of searched publications are given only after the merge step.

Steps	IEEE	ACM	Springer	Elsevier	Web of Knowledge	Top-20-venues	Total
Search results (up to May. 2015)	961	327	10,788	2,225	584	137	-
Scripts verification (with same keywords)	328	271	58	11	365	137	-
Scripts exclusion (criterion 3)	238	267	51	11	365	137	-
Merge							997
After reviewing title/abstract (criteria 4 → 6)							240
After skimming full paper (criteria 7 and 8)							114
After final discussion							88
Author recommendation							+4
Total							92

criterion allows to remove over half of the collected papers, now creating the opportunity for an acceptable manual effort of assessing the relevancy of remaining papers.

5. Fifth, we quickly skim through the remaining papers and exclude those that target Android but *do not deal with static analysis techniques*. For example, publications about dynamic analysis/testing of Android apps are excluded.

Since Android has been a hot topic in the latest years, the set of relevant papers constituted after having applied the above exclusion criteria is still large. These are all papers that propose approaches relevant to Android, based on static analysis of the apps. Figure 5 illustrates the typical, and relevant, workflow used by different approaches in the literature. We have noticed that some of the collected papers do not present this workflow. Instead, the static operations that they perform do not often require the construction of a call graph: e.g., some approaches simply read the manifest file to list permissions requested, or simply attempt to match specific API names in function calls. Thus, we devise three more exclusion criteria to filter out such publications:

1. We exclude papers that statically analyze Android Operating System (OS) rather than Android apps. Because our main focus in this survey is to survey works related to static analysis of Android apps. As examples, we have dismissed PSCout [19] and EdgeMiner [20] in this paper because they are dedicated to analyze the Android framework.

2. We dismiss papers that do not actually parse the app program code and build a call-graph.

3. We also dismiss papers that simply build on static analysis results to perform empirical studies or to perform machine learning-based malware detection. Such papers are indeed irrelevant since they do not contribute to the research on static analysis of Android apps.

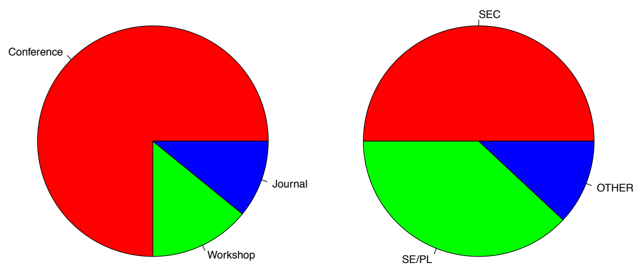
3.4. Primary publications selection

In this section, we give details on our final selection results of primary publications, which are summarized in Table 3.

The first two lines (search results and scripts verification) provide statistics on papers found through the keywords defined previously. In the first line, we focus on the output from the repositories search (with full paper search, whenever possible, because we want to collect as many relevant papers as possible in this phase). Through this repositories search, we collect data such as *paper title* or *paper abstract*. The second line shows the results of an additional verification step on the collected data. More specifically, we perform automated keywords search on the data (with exactly the same keywords as the previous step). We adopt this second step because of the flaws in "advanced" search functionality of the five repositories, where the search results are often inaccurate, resulting in a set noised by some irrelevant papers. After performing the second step (line 2), the number of potential relevant papers is significantly reduced.

The third line shows the results of applying our exclusion criterion 3 (exclude short papers) for the results of line 2. The only big difference happens in IEEE repository. We further look into the publications of IEEE found that those short papers are mostly 4 page conference papers with insufficient description of their proposed approaches/tools. Therefore it makes sense for us to remove those short papers from our interesting set.

Line 4 merges the results of line 3 to one set in order to redundant workload (otherwise, a same paper in two repositories would be reviewed twice). We have noticed that the redundancy occurs in three cases:



(a) Type. (b) Domain.

Figure 6: Statistics of examined publications.

1. The ACM repository may contain papers that originally published in IEEE or Springer and
2. The Web of Knowledge repository may contain papers that published in Elsevier.
3. The five repositories may contain papers that appear in the top-20-venues set.

For the self-checking process, we have collected 343 distinct email addresses of 419 authors for the 88 primary publications selected in our search. We then sent the SLR to these authors and request their help in checking that their publications were correctly classified. Within a week, we have received 25 feedback messages which we took into account. Authors further recommended a total of 19 papers to include in the SLR. 4 of those recommended papers were found to be borderline with regards to our exclusion criteria (e.g., dynamic approaches which resort to some limited static analyses). We review them against our inclusion criteria and decide to include them (Table 3, line 8). The 15 remaining papers are outside the window of this SLR which ends in May 2015. Many of the papers have even just been accepted but are still unpublished in proceedings.

In total, our SLR checks 92 publications. Fig. 6 illustrates the distributions of these publications by types(Fig 6a) and publication domains (Fig. 6b). Over 70% of our collected papers are published in conferences. Workshops, generally co-located with top conferences, have also seen a fair share of contributions on Android static analysis. These findings are not surprising, since the high competition in Android research forces authors to aim for targets where publication process is fastest. Presentations in conferences can also stimulate more collaborations as can be seen in most recent papers on Android analysis. We further find that half of the publications were made in Security venues while another larger proportion was published in Software Engineering venues. Very few contributions were published in other types of venues. MIGDroid [21], published in the *Network* domain and CloneCloud [22], published in the *Systems* domain, are the rare exceptions. This finding conforts our initial heuristics of considering top venues in SE/PL and SEC to verify that our repository search was exhaustive.

Fig. 7 shows a word cloud of the conference names where our primary publications were presented. Most of the reviewed publications are from top conference venues (e.g., ICSE, NDSS and CCS), suggesting that the state-of-the-art research has been included at a substantial scale.

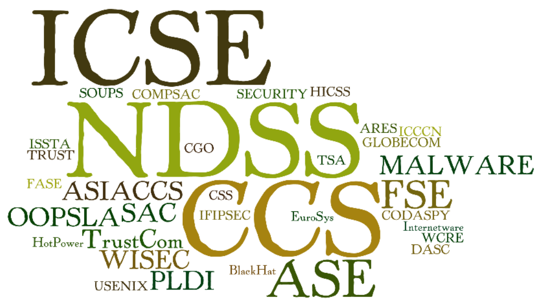


Figure 7: Word cloud of all the conference names of examined publications.

4. Data Extraction

Once relevant papers have been collected, we build a taxonomy of the information that must extracted from each paper in order (1) to cover the research questions enumerated above, (2) to be systematic in the assessment of each paper, and (3) to provide a baseline for classifying and comparing the different approaches. Fig. 8 overviews the metrics extracted from the selected publications.

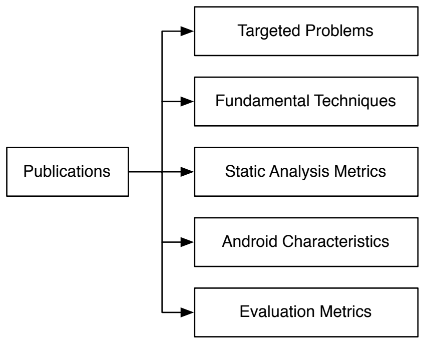


Figure 8: Overview of metrics extracted from a given primary publication.

Targeted Problems. Approaches are also classified on the targeted problems. Examples of problems include privacy leakage, permission management, energy optimization, etc.

Fundamental Techniques. This dimension focuses on the fundamental techniques adopted by examined primary publications. The fundamental techniques in this work include not only such techniques like *taint analysis* and *program slicing* that solve problems through different means but also the existing tools such as Soot or WALA that are leveraged.

Static Analysis Metrics. This dimension includes static analysis related metrics. Given a primary publications, we would like to check whether it is context-sensitive, flow-sensitive, path-sensitive, object-sensitive, field-sensitive, static-aware, implicit-flow-aware, alias-aware. Besides, in this dimension, the dynamic code loading, reflection supporting, native code supporting are also inspected.

Android Characteristics. This dimension includes such metrics that are closely related to Android such as ICC, IAC (Inter-App Communication), Framework and so on. Questions like "Do they take care of the lifecycle or callback methods?" or "Are the studied approaches support ICC or IAC?" are belonging to this dimension.

Evaluation Metrics. This dimension focuses on the evaluation methods of primary publications, intending to answer the question how their approaches are evaluated. To this end, this dimension will count whether their approaches are evaluated through in-the-lab apps (i.e., artificial apps with knowing the ground truth in advance) or in-the-wild apps (i.e., the real-world marketing apps). Questions like how many in-the-wild apps are evaluated are also addressed in this dimension.

5. Summary of Findings

We now report on the findings of this SLR in light of the research questions enumerated in Section 3.1.

5.1. Purposes of the Analyses

In the literature of Android, static analysis has been applied to highlight various security issues (such as private data leaks or permission management concerns), to verify code, or to assess code efficiency in terms of energy consumption for embedded systems). We have identified six recurring purposes of Android-targeted static analysis approaches in the literature. We detail these purposes and provide statistiques of approaches which target them.

Private Data Leaks. Recently, concerns on privacy with Android apps have led researchers to focus on the private data leaks. FlowDroid [5], introduced in 2014, is probably the most advanced approach addressing this issue. It performs static taint analysis on Android apps with a flow-, context-, field-, object-sensitive and implicit flow-, lifecycle-, static-, alias-aware analysis, resulting in a highly precise approach. The associated tool has been open-sourced and many other approaches [23, 24, 25, 26, 6] have leveraged it to perform more extensive analysis.

Vulnerabilities. Security vulnerabilities are another concern for app users who must be protected against malware exploiting the data and privileges of benign apps. Many of the vulnerabilities addressed in the literature are related to the ICC mechanism and its potential misuses such as for component hijacking (i.e., gain unauthorised access to protected or private resources through exported components in vulnerable apps) or intent injection (i.e., manipulate user input data to execute code through it). For example, CHEX [7] detects potential component hijacking-based flows through reachability analysis on customized system dependence graphs. Epicc [8] and IC3 [27] are tools that propose static analysis techniques for implementing detection scenarios of inter-component vulnerabilities. Based on this, PCLeaks [26] goes one step further by performing sensitive data-flow analysis on top of component vulnerabilities, enabling it to not only know what is the issue but also to know what sensitive data will leak through that issue. Similarly to PCLeaks, ContentScope [28] detects sensitive data leaks focusing on Content Provider-based vulnerabilities in Android apps.

Permission Misuse. Permission checking is a pillar in the security architecture of Android. The Android permission-based security model associates sensitive resources with a set of permissions that must be granted before access. However, as shown by Bartel et al. [29, 30], this permission model is an intrinsic risk, since apps can be granted more permissions than they actually need. Malicious may indeed leverage permissions (which are unnecessary to the core app functionality) to achieve their malicious goals. PSCout [19] is currently the most extensive work that dissects the Android permission specification from Android OS source code using static analysis.

Energy Consumption. Battery stand-by time has been a problem for mobile devices for a long time. Larger screens found in modern smartphones constitute the most energy consuming components. As shown by Li et al. [31], modern smart phones use OLED, which consumes more energy when displaying light colors than dark colors. In their investigation, the energy consumption could be reduced by 40% if more efficient web pages are built for mobile systems (e.g., in dark background color). To reach this conclusion, they performed extensive program analysis on the structure of web apps, more specifically, through automatically rewriting web apps so as to generate more efficient web pages. Li et al. [32] present a tool to calculate source line level energy consumption through combining program analysis and statistical modeling.

Other Purposes. Besides the four aforementioned concerns, state-of-the-art works have also targeted less hot topics, often about highly specific issues. More representatively, works such as CryptoLint [33] and CMA [34] have leveraged static analysis to identify cryptography implementation problems in Android apps. Researchers have also extended the Julia [4] static analyzer to perform formal analyses of Android programs.

Table 4 enumerates approaches from our primary publications which fall into the 6 purposes described above. The summary statistics in Fig. 9 show that Security concerns are the focus of most static analysis approaches for Android. Energy efficiency is also a popular concern ahead of program correctness.

***RQ 1**: Static analysis is largely performed on Android programs to uncover security and privacy issues.*

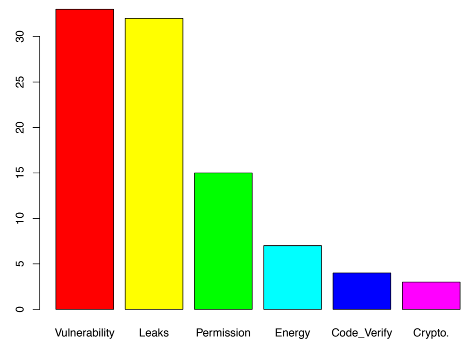


Figure 9: Statistics of main concerns addressed by the publications.

Table 4: Recurrent analysis Purposes and related Publications.

Tool	Leaks	Cryptography	Permission	Vulnerability	Code verification	Energy
DroidChecker [35]	✓	✗	✗	✗	✗	✗
vLens [32]	✗	✗	✗	✗	✗	✓
Nyx [31]	✗	✗	✗	✗	✗	✓
Anadroid [36]	✓	✗	✓	✓	✗	✗
SADroid [37]	✗	✗	✓	✓	✗	✗
eLens [38]	✗	✗	✗	✗	✗	✓
SAAF [39]	✗	✗	✓	✓	✗	✗
MIGDroid [21]	✗	✗	✗	✓	✗	✗
Redexer [40]	✗	✗	✓	✗	✗	✗
Scandal [41]	✓	✗	✗	✗	✗	✗
DidFail [24]	✓	✗	✗	✗	✗	✗
MalloDroid [42]	✗	✗	✗	✓	✗	✗
Adagio [43]	✗	✗	✗	✓	✗	✗
android-app-analysis-tool [44]	✗	✗	✓	✗	✗	✗
AndroidLeaks [45]	✓	✗	✓	✗	✗	✗
Gible et al. [46]	✗	✗	✗	✓	✗	✗
Graa et al. [47]	✗	✗	✗	✓	✗	✗
AdRisk [48]	✗	✗	✗	✓	✗	✗
Woodpecker [49]	✓	✗	✗	✗	✗	✗
Relda [50]	✓	✗	✗	✗	✗	✗
CHEX [7]	✓	✗	✗	✓	✗	✗
Lu et al. [51]	✗	✗	✗	✗	✓	✗
Brox [52]	✓	✗	✗	✗	✗	✗
Mann et al. [53]	✓	✗	✗	✗	✗	✗
Van et al. [54]	✗	✗	✗	✗	✓	✗
Poeplau et al. [55]	✗	✗	✗	✓	✗	✗
FUSE [56]	✓	✗	✓	✓	✗	✗
PermissionFlow [57]	✓	✗	✓	✓	✗	✗
BlueSeal [58]	✓	✗	✓	✗	✗	✗
CMA [34]	✗	✓	✗	✗	✗	✗
SMV-Hunter [59]	✗	✗	✗	✓	✗	✗
Vekris et al. [60]	✗	✗	✗	✗	✗	✓
A5 [61]	✗	✗	✗	✓	✗	✗
CloneCloud [22]	✗	✗	✗	✗	✗	✓
Androguard [62]	✗	✗	✗	✓	✗	✗
Androguard [63]	✗	✗	✗	✓	✗	✗
CryptoLint [33]	✗	✓	✗	✗	✗	✗
Bartsch et al. [64]	✗	✗	✗	✓	✗	✗
Androlizer [65]	✗	✗	✗	✓	✗	✗
Chen et al. [66]	✗	✗	✗	✓	✗	✗
Pegasus [67]	✗	✗	✗	✓	✗	✗
MobSafe [68]	✗	✗	✗	✓	✗	✗
A3 [69]	✗	✗	✗	✓	✗	✗
DroidSIFT [70]	✗	✗	✗	✓	✗	✗
DPartner [71]	✗	✗	✗	✗	✗	✓
TrustDroid [72]	✓	✗	✗	✗	✗	✗
SmartDroid [73]	✓	✗	✗	✗	✗	✗
DroidAlarm [74]	✓	✗	✗	✓	✗	✗
Covert [75]	✓	✗	✗	✓	✗	✗
IFT [76]	✓	✗	✗	✗	✗	✗
Julia [4]	✗	✗	✗	✗	✓	✗
Capper [77]	✓	✗	✗	✗	✗	✗
AppSealer [78]	✓	✗	✗	✗	✗	✗
AppCaulk [79]	✓	✗	✗	✗	✗	✗
DroidSafe [80]	✓	✗	✓	✓	✗	✗
PaddyFrog [81]	✗	✗	✗	✓	✗	✗
Apposcopy [82]	✓	✗	✗	✗	✗	✗
AppContext [83]	✗	✗	✓	✗	✗	✗
SEFA [84]	✓	✗	✓	✓	✗	✗
Amandroid [85]	✓	✓	✗	✓	✗	✗
ContentScope [86]	✓	✗	✗	✓	✗	✗
Wognsen et al. [87]	✗	✗	✗	✓	✗	✗
Wang et al. [88]	✗	✗	✗	✗	✗	✓
AsDroid [89]	✓	✗	✗	✗	✗	✗
AppIntent [90]	✓	✗	✗	✗	✗	✗
FlowDroid [5]	✓	✗	✗	✗	✗	✗
Bartel et al. [30]	✗	✗	✓	✗	✗	✗
COPES [29]	✗	✗	✓	✗	✗	✗
IccTA [6]	✓	✗	✗	✗	✗	✗
Epicc [8]	✗	✗	✗	✓	✗	✗
Lin et al. [91]	✗	✗	✓	✗	✗	✗
Cassandra [92]	✓	✗	✗	✗	✓	✗
Apparecium [93]	✓	✗	✗	✗	✗	✗
Total	**32**	**3**	**15**	**33**	**4**	**7**

5.2. *Form and Extent of Analysis*

We now investigate how the analyses described in the literature are implemented. In particular, we study the support tools that they leverage (Section 5.2.1), the fundamental analysis methods that they apply (Section 5.2.2), the sensitivities supported by their analysis (Section 5.2.3) as well as the Android peculiarities that they deal with (Section 5.2.4).

5.2.1. *Code Representations and Support Tools*

Table 5 enumerates the recurrent tools that are used by approaches in the literature to support their analyses. Such tools often come as off-the-shelf components that implements common analysis processes (e.g., for the translation between bytecode forms or for the automatic construction of call-graphs). The table also provides for each tool information on the intermediate representation (IR) that it deals with. The IR is a simplified code format to represent the original Dalvik bytecode and facilitate processing since Android Dalvik itself is known to be complex and challenging to manipulate. Finally, the table highlights the usage of such tools in specific reviewed approaches.

Table 6 goes into more details into the adoption of the different code representations by the examined approaches. Fig. 10 summarizes the frequency of usages, where *Jimple*, which is used by the popular Soot tool, appears as the most used IR followed by Java bytecode, then the *Smali* intermediate representation, which is used by Apktool.

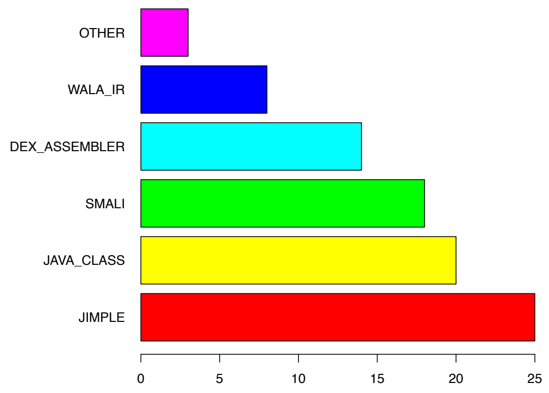


Figure 10: Distribution of code representations used by examined publications.

RQ 2.1: *The Soot framework and the Jimple intermediate representation are the most adopted basic support tool and format for static analysis of Android apps.*

5.2.2. *Fundamental Analysis Methods*

While all reviewed approaches build on control-flow or data-flow analyses, specific techniques are employed to enhance the results and reached the target purposes. In our study, we have identified four fundamental techniques which are used, often in conjunction, in the literature.

Abstract Interpretation. Abstract interpretation is a theory of approximating the semantics of programs, where soundness of the analysis can be guaranteed and thereby to avoid

Table 5: List of recurrent support tools for static analysis of Android apps.

TOOL	Brief Description	IR	Example Usages
Soot [94]	A Java/Android static analysis and optimization framework	Jimple, Jasmin	FlowDroid [5], IccTA [6], AppIntent [90]
WALA[a]	A Java/Javascript static analysis framework	WALA-IR (SSA-based)	AsDroid [89], Asynchronizer [95], ORBIT [96]
Chord [97]	A Java program analysis platform	Chord-IR (SSA-based)	CloneCloud [22]
Androguard [63, 62]	Reverse engineering, malware/goodware analysis of Android apps	Androguard-IR	MalloDroid [42], Relda [50]
Ded [98]	A DEX to Java bytecode translator	Class	Enck et al. [99]
Dare [100]	A DEX to Java bytecode translator	Class	Epicc [8], IC3 [27]
Dexpler [101]	A DEX to Jimple translator	Jimple	BlueSeal [58]
Smali/Baksmali[b]	A DEX to Smali translator (and verse visa)	Smali	Woodpecker [49], SEFA [84]
Apktool[c]	A tool for reverse engineering Android apps	Smali	PaddyFrog [81], Androlizer [65]
dex2jar[d]	A DEX to Java bytecode translator	Class	DroidChecker [35], Vekris et al. [60]
dedexer[e]	A disassembler for DEX files	DEX-assembler	Brox [52], AQUA [102]
dexdump	A disassembler for DEX files	DEX-assembler	ScanDal [41]
dx	A Java bytecode to DEX translator	DEX-assembler	EdgeMiner [20]
jd-gui[f]	A Java bytecode to source code translator (and also an IDE)	Java	Wang et al. [88]
ASM [103, 104]	A Java manipulation and analysis framework	Class	COPES [29]
BCEL[g]	A library for analyzing and instrumenting Java bytecode	Class	vLens [32], Julia [4], Nyx [31]
Redexer	A reengineering tool that manipulates Android app binaries	DEX-assembler	Brahmastra [105]

[a]http://wala.sourceforge.net
[b]http://baksmali.com
[c]http://ibotpeaches.github.io/Apktool/
[d]https://github.com/pxb1988/dex2jar
[e]http://dedexer.sourceforge.net
[f]https://github.com/java-decompiler/jd-gui
[g]https://commons.apache.org/bcel/

Table 6: A Summary of examined approaches through the code representations that they use.

Code Representation	Publications
WALA_IR	AndroidLeaks [45], AsDroid [89], ORBIT [96], THRESHER [106], Asynchronizer [95], A3E [107], CHEX [7], Poeplau et al. [55]
JIMPLE	A5 [61], ApkCombiner [108], COPES [29], AppContext [83], Vekris et al. [60], Epicc [8], FlowDroid [5], AppIntent [90], BlueSeal [58], Capper [77], AppSealer [78], Gator2 [109], Covert [75], Apposcopy [82], DidFail [24], Lotrack [110], IccTA [6], DroidSafe [80], IC3 [27], Van et al. [54], PerfChecker [111], android-app-analysis-tool [44], Gator [112], ACTEve [113], Bartel et al. [30]
DEX_ASSEMBLER	Lin et al. [91], StaDynA [114], MalloDroid [42], Relda [50], Redexer [40], CryptoLint [33], Androguard [62], Adagio [43], Brahmastra [105], Scandal [41], Mann et al. [53], AQUA [102], Androguard [63], Brox [52]
SMALI	SMV-Hunter [59], Woodpecker [49], CMA [34], A3 [69], MIGDroid [21], SmartDroid [73], PaddyFrog [81], AppCaulk [79], Anadroid [36], MobSafe [68], Apparecium [93], SAAF [39], Androlizer [65], Wognsen et al. [87], SADroid [37], AdRisk [48], ContentScope [86], SEFA [84]
OTHER	Amandroid [85], FUSE [56], Nyx [31]
JAVA_CLASS	Pegasus [67], Bartsch et al. [64], vLens [32], Julia [4], Chen et al. [66], SIF [115], DroidChecker [35], EvoDroid [116], IFT [76], CloneCloud [22], DroidAlarm [74], Mirzaei et al. [117], TrustDroid [72], eLens [38], Wang et al. [88], Lu et al. [51], Choi et al. [118], DroidSIFT [70], DPartner [71], PermissionFlow [57]

for yielding false negative results. A concrete implementation of abstract interpretation is through formal program analysis. As an example, Julia [4] is a tool that uses abstract interpretation to automatically static analyze Java and Android apps for the development of high-quality, formally verified products. SCanDal [41], another sound and automatic static analyzer, also leverages abstract interpretation to detect privacy leaks in Android apps.

Taint Analysis. Taint analysis is implemented in at first tainting a data if it is defined to be sensitive, and then tracking it through data-flow analysis. If a tainted data flows to a point where it should not be, then specific instructions can be applied, e.g., to stop this behavior and report it to the administrators. As an example, AppSealer [78] leverages taint analysis to automatically generate patches for Android component hijacking attacks. When a tainted data is going to violate the predefined polices, AppSealer injects a patch before the violation to alert the app user though a pop-up dialog box. FlowDroid [5], as another example, performs static taint analysis to detect sensitive data leaks. Based on a predefined set of *source* and *sink* methods, which are automatically extracted from the Android SDK (cf. SUSI [119]), sensitive data leaks are reported if and only if the data are obtained from *source* methods (i.e., these data are tainted) and eventually flow to *sink* methods (i.e., violate security polices).

Symbolic Execution. Symbolic execution is a promising approach to generate possible program inputs, which is different from abstract interpretation. Because it assumes symbolic

Table 7: Summary through the adoption of different fundamental techniques.

Techniques	Publications	Percentage[a]
Abstract Interpretation	Julia [4], Scandal [41], Rocha et al. [120], Mann et al. [53], Lu et al. [51], Anadroid [36]	6.6%
Taint Analysis	AppContext [83], FUSE [56], FlowDroid [5], AppSealer [78], Capper [77], AppCaulk [79], Apposcopy [82], Brox [52], Anadroid [36], DidFail [24], Amandroid [85], MobSafe [68], DroidChecker [35], Lotrack [110], IccTA [6], DroidSafe [80], AndroidLeaks [45], Apparecium [93], Mann et al. [53], TrustDroid [72], SEFA [84], CHEX [7], PermissionFlow [57]	25.3%
Symbolic Execution	Mirzaei et al. [117], AppIntent [90], ACTEve [113]	3.3%
Program Slicing	CryptoLint [33], Capper [77], AppSealer [78], AppCaulk [79], Brox [52], Poeplau et al. [55], MobSafe [68], AndroidLeaks [45], Apparecium [93], SAAF [39], eLens [38], Rocha et al. [120], AQUA [102]	14.3%
Code Instrumentation	vLens [32], Nyx [31], Brahmastra [105], CMA [34], SmartDroid [73], Capper [77], AppSealer [78], SIF [115], AppCaulk [79], DidFail [24], IccTA [6], DroidSafe [80], Androguard [62], ORBIT [96], android-app-analysis-tool [44], Rocha et al. [120], Androguard [63], Cassandra [92], ACTEve [113]	20.9%
Type/Model Checking	IFT [76], DroidAlarm [74], Mann et al. [53], Choi et al. [118], Lu et al. [51], SADroid [37], Covert [75]	7.7%

[a]Some primary papers leverage basic data-flow analysis only and thus are not categorized, making the sum of percentages less than 100%.

values for inputs rather than obtains actual inputs as abstract interpretation does.

As an example, AppIntent [90] uses symbolic execution to generate a sequence of GUI manipulations that lead to data transmission. As the basic straightforward symbolic execution is too time-consuming for Android apps, AppIntent thus leverages the unique Android execution model to reduce the search space without sacrificing code coverage.

Program Slicing. Program slicing has been used as a common means in program analysis field to reduce the set of program behaviors, meanwhile keep the interested program behavior unchanged. Given a variable v in program p that we are interested in, a possible slice would consists of all statements in p that may affect the value of v. As an example, Hoffmann et al. [39] present a framework called SAAF to create program slices so as to perform backward data-flow analysis to track parameter values for a given Android method. CryptoLint [33] computes static program slices that terminate in calls to cryptographic API methods, and then extract the necessary information from these slices.

Code Instrumentation. In static analysis, code instrumentation is usually used to tackle some complicated problems (e.g., inter-component communication, reflection, etc.). In Android community, Arzt et al. [121] have introduced several means to instrument Android apps, in which they have illustrated Soot is a good tool to support the instrumentation of Android apps. As an example, IccTA [122] instruments Android apps to reduce an inter-component taint propagation problem to an intra-component problem. Nyx [31] instruments android web app to modify the background of web pages, so as to reduce the display power consumption and thereby letting web app become more energy efficient. Except Soot, other tools/frameworks such as WALA and ASM are also capable to support the instrumentation of Android apps.

Type/Model Checking. Type and model checking are two prevalent approaches to program verification. The main difference between them is that type checking is usually based on syntactic and modular style whereas model checking is usually defined in a semantic and whole-program style. Actually, this difference makes these two approaches complementary to one another: type checking is good at explaining why a program was accepted while model checking is good at explaining why a program was rejected [123]. As an example, COVERT [75] first extracts relevant security specifications from a given app and then applies a formal model checking engine to verify whether the analyzed app is safe or not. For type checking, Cassandra [92] is presented to enable users of mobile devices to check whether Android apps comply with their personal privacy requirements even before installing these apps. Ernst et al. [76] also present a type checking system for Android apps, which checks the information flow type qualifiers and ensures that only such flows defined beforehand can occur at run time.

Table 7 provides information on the works that use the different techniques. The summary statistiques show that taint analysis, which is used for tracking data, is the most applied technique (25.3% of primary publications), while 20.9% primary publications involve code instrumentation and 14.3% primary publications have applied program slicing technique in their approaches. 7.7% approaches are dealing with Type/-Model checking and 6.6% of primary publications use abstract interpretation to perform their static analysis. Symbolic execution however is applied in only 3 (3.3%) primary publications.

RQ 2.2*: Taint analysis remains the most applied technique in static analysis of Android apps. This is inline with the finding in RQ1 which shows that the most recurrent purpose of state-of-the-art approaches is on security and privacy.*

5.2.3. Static Analysis Sensitivities

We now investigate the depth of the analyses presented in the primary publications. To that end we assess the sensitivities (cf. Sections 2.1.2 and 2.1.3). Table 8 classifies the different approaches according to the the sensitivities that their analyses take into account. *Field-sensitivity* appears to be the most considered with 36 primary publications taking it into account. This finding is understandable since Android apps are generally written in Java, an Object-Oriented language where object fields are pervasively used to hold data. *Context-sensitivity* and *Flow-sensitivity* are also largely taken into account (with 30 and 31 publications respectively). The least considered sensitivity is *Path-sensitivity* (only 5 publications), probability due to the scalability issues that it raises.

Table 8: Classification of Approaches according to the Sensitivities considered in Call-Graph Construction.

Tool	*Context-Sensitive*	*Flow-Sensitive*	*Field-Sensitive*	*Object-Sensitive*	*Path-Sensitive*
Anadroid [36]	✓	✗	✓	✓	✓
Lotrack [110]	✓	✓	✓	✓	✗
AQUA [102]	✗	✓	✗	✗	✗
Scandal [41]	✓	✓	✗	✗	✗
DidFail [24]	✓	✓	✓	✓	✗
AndroidLeaks [45]	✓	✗	✗	✗	✗
Woodpecker [49]	✗	✗	✓	✗	✓
CHEX [7]	✓	✓	✓	✓	✗
Brox [52]	✓	✓	✗	✗	✗
Mann et al. [53]	✗	✗	✓	✗	✗
Van et al. [54]	✗	✗	✓	✗	✗
FUSE [56]	✓	✗	✓	✗	✗
Rocha et al. [120]	✗	✓	✗	✗	✗
PermissionFlow [57]	✓	✓	✓	✗	✗
Vekris et al. [60]	✓	✓	✗	✗	✗
Choi et al. [118]	✗	✗	✗	✓	✗
CloneCloud [22]	✗	✗	✓	✗	✗
CryptoLint [33]	✗	✗	✓	✗	✗
A3E [107]	✗	✓	✗	✗	✗
Bartsch et al. [64]	✓	✓	✗	✗	✗
THRESHER [106]	✓	✓	✓	✓	✓
Chen et al. [66]	✗	✓	✗	✗	✗
Pegasus [67]	✓	✓	✓	✗	✗
DroidSIFT [70]	✓	✓	✗	✗	✗
DPartner [71]	✗	✗	✓	✗	✗
TrustDroid [72]	✗	✗	✓	✗	✗
Covert [75]	✓	✗	✓	✗	✗
IFT [76]	✓	✓	✓	✗	✗
Julia [4]	✗	✓	✗	✗	✗
Capper [77]	✓	✓	✓	✗	✗
AppSealer [78]	✓	✓	✓	✗	✗
AppCaulk [79]	✓	✗	✓	✗	✗
Gator [112]	✓	✗	✗	✓	✗
Gator2 [109]	✓	✗	✗	✓	✗
IC3 [27]	✓	✓	✓	✗	✗
DroidSafe [80]	✓	✓	✓	✓	✗
Apposcopy [82]	✓	✗	✓	✓	✗
AppContext [83]	✓	✓	✓	✓	✗
SEFA [84]	✗	✗	✓	✗	✗
Amandroid [85]	✓	✓	✓	✓	✗
ContentScope [86]	✗	✗	✗	✗	✓
Wognsen et al. [87]	✗	✓	✓	✗	✗
AsDroid [89]	✗	✓	✓	✓	✗
FlowDroid [5]	✓	✓	✓	✓	✗
Bartel et al. [30]	✗	✓	✓	✗	✗
COPES [29]	✗	✓	✓	✗	✗
IccTA [6]	✓	✓	✓	✓	✗
Epicc [8]	✓	✓	✓	✗	✗
Asynchronizer [95]	✓	✗	✓	✗	✗
Cassandra [92]	✗	✗	✓	✗	✗
Apparecium [93]	✗	✓	✓	✗	✓
Total	**30**	**31**	**36**	**15**	**5**

In theory, the more sensitivities considered, the more precise the analysis is. It is thus reasonable to state that only one approach, namely TRESHER [106], achieves high precision by taking into account all sensitivities. However, given the relatively high performance of existing state-of-the-art works, it seems unnecessary to support all sensitivities to be useful in practice.

***RQ 2.3**: Most approaches support up to 3 of the 5 sensitivities in Call-Graph construction for static analysis. Path-sensitivity is the least taken into account by the Android research community.*

Table 9: Classification of Approaches according to their support for Android specificities.

Tool	*Lifecycle*	*Callback-Methods*	*Entry-Points*	*ICC*	*IAC*	*XML Layout*
DroidChecker [35]	✓	✓	✓	✗	✗	✗
Anadroid [36]	✓	✓	✓	✗	✗	✓
Lotrack [110]	✓	✓	✓	✗	✗	✓
Scandal [41]	✗	✓	✓	✗	✗	✗
DidFail [24]	✓	✓	✓	✓	✓	✓
AndroidLeaks [45]	✗	✓	✓	✗	✗	✗
AdRisk [48]	✗	✗	✓	✗	✗	✗
Woodpecker [49]	✓	✓	✓	✗	✗	✗
Relda [50]	✓	✓	✓	✗	✗	✗
CHEX [7]	✓	✓	✓	✗	✗	✗
Brox [52]	✗	✗	✓	✗	✗	✗
EvoDroid [116]	✓	✓	✓	✗	✗	✓
Mann et al. [53]	✓	✓	✓	✗	✗	✗
Van et al. [54]	✓	✓	✓	✗	✗	✗
Mirzaei et al. [117]	✓	✓	✓	✗	✗	✓
Poeplau et al. [55]	✓	✗	✓	✗	✗	✗
FUSE [56]	✓	✓	✓	✓	✓	✓
PermissionFlow [57]	✓	✓	✓	✗	✗	✗
BlueSeal [58]	✗	✓	✓	✓	✓	✓
CMA [34]	✗	✗	✓	✗	✗	✗
SMV-Hunter [59]	✗	✗	✓	✗	✗	✓
Vekris et al. [60]	✓	✓	✓	✓	✗	✗
A5 [61]	✗	✓	✓	✗	✗	✗
Choi et al. [118]	✓	✓	✗	✓	✗	✗
CloneCloud [22]	✗	✗	✓	✗	✗	✗
CryptoLint [33]	✗	✗	✓	✗	✗	✗
A3E [107]	✓	✓	✓	✗	✗	✓
Bartsch et al. [64]	✓	✓	✓	✓	✗	✗
THRESHER [106]	✗	✗	✓	✗	✗	✗
Pegasus [67]	✓	✓	✓	✓	✗	✗
ORBIT [96]	✗	✗	✓	✗	✗	✗
DroidSIFT [70]	✗	✓	✓	✗	✗	✗
DPartner [71]	✗	✓	✓	✗	✗	✗
SmartDroid [73]	✗	✗	✓	✗	✗	✓
DroidAlarm [74]	✗	✗	✓	✓	✗	✗
Covert [75]	✓	✓	✓	✓	✓	✗
IFT [76]	✗	✗	✗	✓	✗	✗
Julia [4]	✗	✗	✓	✗	✗	✓
Capper [77]	✓	✗	✓	✗	✗	✗
AppSealer [78]	✓	✗	✓	✗	✗	✗
Gator [112]	✓	✓	✓	✗	✗	✓
Gator2 [109]	✓	✓	✓	✓	✗	✓
IC3 [27]	✓	✓	✓	✗	✗	✗
DroidSafe [80]	✓	✓	✓	✓	✓	✓
PaddyFrog [81]	✗	✗	✓	✓	✗	✗
Apposcopy [82]	✓	✗	✓	✓	✗	✗
AppContext [83]	✓	✓	✓	✓	✗	✗
SEFA [84]	✗	✗	✓	✓	✓	✗
Amandroid [85]	✓	✓	✓	✓	✗	✓
ContentScope [86]	✗	✓	✓	✗	✗	✗
Wognsen et al. [87]	✓	✓	✓	✗	✗	✗
AsDroid [89]	✓	✓	✓	✓	✗	✓
AppIntent [90]	✓	✓	✓	✓	✗	✗
FlowDroid [5]	✓	✓	✓	✗	✗	✓
IccTA [6]	✓	✓	✓	✓	✗	✓
Epicc [8]	✓	✓	✓	✗	✗	✗
Asynchronizer [95]	✓	✓	✓	✗	✗	✗
PerfChecker [111]	✓	✓	✓	✗	✗	✓
Cassandra [92]	✗	✗	✓	✗	✗	✗
Apparecium [93]	✓	✓	✗	✗	✗	✗
ACTEve [113]	✓	✓	✗	✗	✗	✓
Brahmastra [105]	✓	✓	✗	✓	✗	✓
Total	**40**	**43**	**57**	**21**	**6**	**21**

Table 10: List of approaches with publicly available tool support, and information on evaluation settings from the publications. With *lab* $\Longleftrightarrow$ *in-the-lab experiments; wild* $\Longleftrightarrow$ *in-the-wild experiments; and # of apps* $\Longleftrightarrow$ *the number of apps that are evaluated in their in-the-wild experiments.* Note that in the last column, ”-” means that the number of apps is not mentioned in the studied paper.

Approach	Open-source tool-support	Evaluation		
		lab	wild	# of apps
Anadroid [36]	✓	✗	✗	0
Lotrack [110]	✓	✗	✓	100
SAAF [39]	✓	✗	✓	142100
Redexer [40]	✓	✗	✓	14
DidFail [24]	✓	✓	✗	0
MalloDroid [42]	✓	✗	✓	13500
Adagio [43]	✓	✗	✓	147950
android-app-analysis-tool [44]	✓	✗	✓	265
Poeplau et al. [55]	✓	✗	✓	1632
FUSE [56]	✗	✓	✓	2573
BlueSeal [58]	✓	✓	✗	0
A5 [61]	✓	✗	✓	0
Choi et al. [118]	✓	✓	✗	0
Androguard [62]	✓	✓	✗	0
Androguard [63]	✓	✓	✗	0
A3E [107]	✓	✗	✓	25
THRESHER [106]	✓	✗	✓	7
Covert [75]	✗	✗	✓	200
IFT [76]	✗	✗	✓	72
Gator [112]	✓	✗	✓	20
Gator2 [109]	✓	✗	✓	20
IC3 [27]	✓	✗	✓	460
DroidSafe [80]	✓	✓	✓	24
StaDynA [114]	✓	✗	✓	10
Amandroid [85]	✓	✓	✓	853
Wognsen et al. [87]	✓	✗	✓	1700
FlowDroid [5]	✓	✓	✓	1500
ApkCombiner [108]	✗	✓	✓	0
IccTA [6]	✓	✓	✓	15000
Epicc [8]	✗	✗	✓	1200
Asynchronizer [95]	✗	✗	✓	13
PerfChecker [111]	✓	✗	✓	29
Cassandra [92]	✓	✗	✗	0
Apparecium [93]	✓	✗	✓	100

5.2.4. Android Specificities

Although Android apps are written in Java, they present specific characteristics in their functioning. Typically, they extensively make use of a set of lifecycle event-based methods that the system requires to interact with apps, and rely on the inter-component communication mechanism to make application parts interact. These characteristics however may constitute challenges for a static analysis approach.

Component Lifecycle. Because lifecycle callback methods (i.e., *onStop()*, *onStart()*, *onRestart()*, *onPause()* and *onResume()*) have neither connection among them nor directly with app code, static analysis are challenged by completing control-flow graphs (e.g., to continuously keep track of sensitive data flows). We found that 40 of the reviewed publications propose static analysis approaches that take into account component lifecyle.

UI Callbacks. Besides lifecycle methods, a number of callbacks are used in Android to handle various events. In particular, UI events are detected by the system and notified to developer apps through callback methods (e.g., to react when a user clicks on a button). There are several such callbacks defined by in various Android classes. Similarly to lifecycle methods, taking into account such callback methods leads to a more complete control-flow graph. Our review reveals that 43 of publications are considering specific analysis processes that take into account callback methods.

EntryPoint. Most static approaches for Android apps must build an entry point, in the form of a dummy main, to allow the construction of call-graph by state-of-the-art tools such as Soot and WALA. 57 publications from our set explicitely discussed their handling of the single entry-point issue.

ICC. The inter-component communication (ICC) is well-know to challenge static analysis of Android programs. Recently, several works have focused on its inner-working to highlight vulnerabilities and malicious activities in Android apps. Among the set of collected primary publications, 21 research papers explicitely deal with ICC. As examples, Epicc [8] and IC3 [27] attempts to extract the necessary information of ICC in Android apps which can support other approaches, including IccTA [6], and DidFail [24] , in performing ICC-aware analyses across components. AmanDroid [85] also resolves ICC information for supporting inter-component data-flow analysis, for the purpose of vetting the security of Android apps.

IAC. The inter-app communication (IAC) mechanism extends the ICC mechanism for components across different apps. Because, most approaches focus on analysing single apps, IAC-supported analyses are scarce in the literature. We found only 6 publications that deal with such scenarios of interactions. A main challenge of tackling IAC-aware analyses is the support of scalability for market-scale analyses.

XML-Layout. The structure of user interfaces of Android apps are defined by layouts, which can be declared in either XML configurations or Java code. The XML layout mechanism provides a well-defined vocabulary corresponding to the View classes, sub-classes and also their possible event handlers. We found that 21 publications, from our set, take into account XML layouts to support more complete analysis scenarios.

Overall, Table 9 summarizes the support for addressing the enumerated challenges by approaches from the literature. We list in this table only those publications from our collected set that are explicitly addressing at least one of the challenges.

RQ 2.4*: No single work in the literature has proposed to tackle at once all challenges due to Android specificities. Instead, most approaches select to deal partially with those challenges directly within their implementation with little opportunity for reuse by other approaches.*

5.3. Usability of Research Output

We now investigate whether the works behind our primary publications have produced usable tools and whether their evaluations are extensive enough to make their conclusions reliable or meaningful.

Among the 92 reviewed papers, 33 (i.e., only 36%) have provided a publicly available tool support. This finding suggests that, currently, the majority of researchers in the field of static analysis of Android apps, do not share their research efforts. Table 10 summarizes the publicly available approaches, among which 27 of them are further open-sourced.

We now consider how researchers in the field of static analysis of Android apps evaluate their approaches. We differentiate *in-the-lab experiments*, which are mainly performed with a few hand-crafted test cases to highlight efficacy and/or correctness, from *in-the-wild experiments*, which consider a large number of real-world apps to demonstrate efficiency and/or scalability. In-the-lab experiments help to quantify an approach through standard metrics (e.g., precision and recall), which is very difficult to obtain through in-the-wild experiments, because of missing of ground truth. In-the-wild experiments are however also essential for static approaches. They are dedicated for finding real and possibly solve problems of real-word apps, which may have already been used by thousands of users. In our review, 26 out of the 33 approaches in Table 10 have evaluated their approaches through in-the-wild experiments. We also found that on average, the number of apps that those in-the-wild experiments consider are 362.5 (median) and 13,480 (mean). The maximum number of evaluated apps is 147,950, which is considered for Adagio [43].

Unfortunately, as shown in Table 10, only 7 approaches have taken into account in-the-lab and in-the-wild experiments at the same time.

> ***RQ3***: *Only a small portion of state-of-the-art works that perform static analysis on Android apps have made their contributions available in public (e.g., tool support). Among those approaches, only a few have fully evaluated their approaches.*

5.4. Trends and Overlooked Challenges

Although Android is a recent ecosystem, research on analysing its programs have flourished. We investigate the general trends in the research and make an overview of the challenges that are/are not dealt with.

5.4.1. Trend Analysis

Fig. 11 shows the distribution of publications from our set in according to their year of publication. Research papers appear to have started in 2011, about two years and a half after its commercial release in September 2008. Then, a rush of papers ensued with a peak in 2014 for both Security and Software engineering communities.

Fig. 12 shows that, as time goes by, research works are considering more sensitivities and addressing more challenges to produce precise analyzers which are aware of more and more analysis specificities. We further look into the ICC challenge for static analyzers to show the rapid increase of publications which deal with it.

> ***RQ 4.1***: *Research on static analysis for Android is maturing, yielding more analysis approaches which consider more analysis sensitivities and are aware of more specificities of Android.*

5.4.2. Dealing with Analysis Challenges

We now discuss our findings on the different challenges addressed in analyses to make them static-, implicit-flow, alias-, dynamic-code-loading-, reflection-, native-code-, and multi-threading-aware.

```
SmsManager sms = SmsManager.getDefault();
//sms.sendTextMessage("+49 1234", null,
    "123", null, null);
for (int i = 0; i < 123; i++)
 sms.sendTextMessage("+49 1234", null,
    "count", null, null);
```

Listing 1: Example of an implicit flow.

We consider an approach to be static-aware when it takes into account *static* object values in Java program to improve analysis precision. 24 approaches explicitly taking this into account. 18 primary publications consider aliases. Both challenges are the most considered in approaches from the literature as they are essential for performing precise static analysis.

We found 18 primary publications which take into account multi-threading. We further investigate these supports since multi-threading is well-known to be challenging even in Java ecosystem. We note that those approaches partially solve multi-threading issues in very simply manner, based on a predefined whitelist of multi-threading mechanisms. For example, when *Thread.start()* is launched, they simply bridge the gap between method *start()* and *run()* through an explicit call graph edge. However, other complex multi-threading processes (e.g., those unknown in advance) or the synchronization among different threads are not yet addressed by the community of static analysis researchers for Android apps.

Another challenge is on considering implicit flows, i.e., flow information inferred from control-flow dependencies. Let us take Listing 1 as an example, if an Android app does not send out message 123 directly, but sends 123 times the word “count”, the attacker can actually gains the same information as if the app had directly sent the 123 value directly.

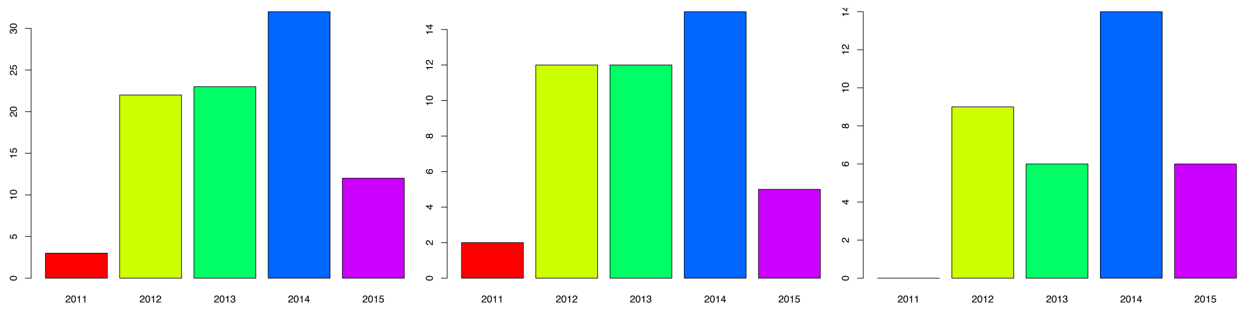
The remaining challenges include reflection, native code and dynamic code loading (DCL) which are taken into account by 10, 3 and 3 publications respectively.

Table 11 provides information on which challenges are addressed by the studied papers.

> ***RQ 4.2***: *There are a number of analysis challenges that remain to be addressed more largely by the community to build approaches that are aware of implicit-Flows, dynamic code loading features, reflective calls, native code and multi-threading, so as to implement sound and highly precise static analyzers.*

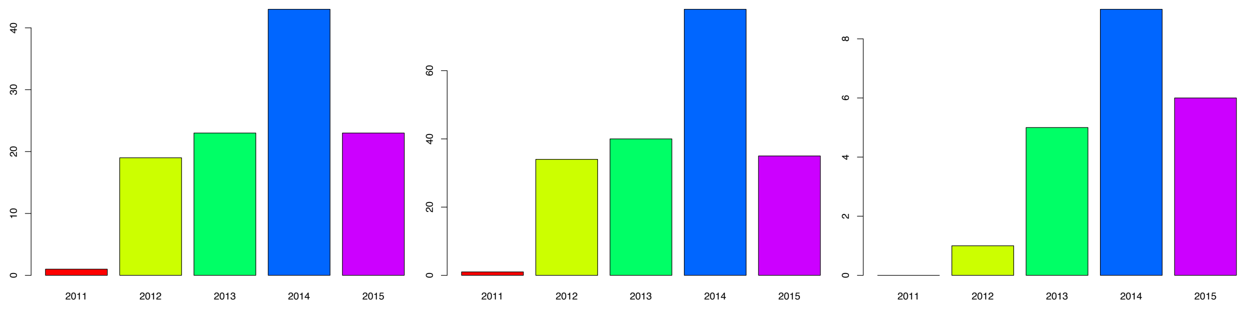
6. Threats To Validity

Although we have attempted to collect relevant papers as much as possible by combining both repository search and top-venue search, our results may have still missed some relevant publications. In particular, we have observed that currently the state-of-the-art repository search engine (e.g., the one provided by Springer) are not so accurate. Besides, we have only checked



(a) Published years (total). (b) Published years (SEC). (c) Published years (SE/PL).

Figure 11: Distribution of examined publications through published year.



(a) Trend of sensitivity. (b) Trend of awareness. (c) Trend of ICC.

Figure 12: Trend analysis.

the 20 top venues for potential missed publications (i.e., the top-venue search), which may not be enough. However, the attempt of searching on top ranked venues has guaranteed that the influential[11] papers have been taken into account. As future work, we plan to mitigate this by performing a snowballing based on the current primary publications.

Given our interest in systematic literature reviews, we are likely to have made some errors on the side of including or excluding primary publications, although each ”borderline” publication has been cross-checked by the authors of this SLR.

In order to share the heavy workload of data extraction, we have split the collected primary publications into different authors to perform the detailed examination. As suggested by Brereton et al. [124], we have applied a cross-checking mechanism: for the data extracted by a researcher, we have assigned it to another researcher to validate. However, some of the data we extracted may be erroneous as well, as founded by Turner et al. [125], the extractor/checker mode of working can lead to data extraction and aggregation problems when there are a large number of primary studies or the data is complex. To further mitigate the inevitable erroneous, we plan to validate our extracted data through their original authors.

The rank of the top 20 venues we select are based on their h5-index, which may change from time to time. Besides, we have heuristically removed some potentially irrelevant venues (e.g., cryptography-related venues), even it is rare, it is still possible that we may miss publications from those venues.

[11]Although it may not be always the case, we still believe that papers published in better venues can consequently acquire more impacts.

7. Related Work

To the best of our knowledge, there is no systematic literature review in the research area of static analysis of Android apps. There is also no surveys that specifically focus on this research area. However, several Android security related surveys have been proposed in the literature. Unlike our approach, these approaches are actually not done systematically (not SLRs). As a result, there are always some well-known approaches missing. Indeed, our review in this report has shown better coverage than those surveys in terms of publications in the area of static analysis of Android apps.

Tan et al. [9] presents a survey on general Android security, including both static and dynamic approaches. This survey first introduces a taxonomy with five main categories based on the existing security solutions on Android. Then, it classifies existing works into those five categories and thereby comparatively examines them. In the end, this survey has highlighted the limitation of existing works and also discussed potential future research directions.

Faruki et al. [126] present another survey mainly focusing on Android malware, e.g., the growth of malware, the existence of anti-analysis techniques. This survey has revealed that the traditional signature-based and static analysis-based approaches are potentially vulnerable. Remarkably, this survey has proposed a platform, to the researchers and practitioners,

Table 11: Summary through different aspects of static analysis.

Tool	Static-Aware	Implicit-Flow	Alias analysis	Dynamic Code Loading	Reflection	Native	Multi-Threading
Anadroid [36]	✗	✗	✗	✗	✗	✗	✓
Lotrack [110]	✓	✓	✓	✗	✗	✗	✗
eLens [38]	✗	✗	✗	✗	✗	✗	✓
SAAF [39]	✗	✗	✓	✗	✗	✗	✗
Scandal [41]	✗	✗	✗	✗	✓	✗	✗
DidFail [24]	✓	✓	✓	✗	✗	✗	✗
Gible et al. [46]	✗	✓	✗	✗	✗	✗	✗
Graa et al. [47]	✗	✓	✗	✗	✗	✗	✗
Woodpecker [49]	✗	✗	✗	✗	✗	✗	✓
Relda [50]	✗	✗	✗	✗	✗	✗	✓
Brox [52]	✗	✗	✗	✗	✗	✗	✓
Mann et al. [53]	✓	✗	✗	✗	✗	✗	✗
Poeplau et al. [55]	✗	✗	✗	✓	✗	✗	✗
FUSE [56]	✓	✗	✓	✗	✓	✗	✓
Rocha et al. [120]	✓	✓	✗	✗	✗	✗	✗
PermissionFlow [57]	✓	✗	✓	✗	✗	✗	✗
BlueSeal [58]	✗	✗	✗	✗	✓	✗	✓
SMV-Hunter [59]	✗	✓	✗	✗	✗	✗	✗
Vekris et al. [60]	✗	✗	✓	✗	✗	✗	✗
CloneCloud [22]	✓	✗	✗	✗	✗	✗	✗
CryptoLint [33]	✓	✗	✗	✗	✗	✗	✗
THRESHER [106]	✓	✗	✓	✗	✗	✗	✗
Pegasus [67]	✗	✗	✗	✓	✓	✗	✗
MobSafe [68]	✗	✗	✓	✗	✗	✗	✗
DroidSIFT [70]	✗	✗	✗	✗	✓	✗	✓
DPartner [71]	✗	✗	✗	✗	✓	✗	✗
TrustDroid [72]	✓	✗	✗	✗	✗	✗	✓
Covert [75]	✗	✗	✓	✗	✗	✗	✗
IFT [76]	✗	✓	✗	✗	✓	✗	✗
Capper [77]	✓	✗	✗	✗	✗	✗	✓
AppSealer [78]	✓	✗	✗	✗	✗	✗	✓
AppCaulk [79]	✓	✗	✗	✗	✗	✓	✗
IC3 [27]	✓	✗	✓	✗	✗	✗	✗
DroidSafe [80]	✓	✓	✓	✗	✓	✓	✓
Apposcopy [82]	✓	✗	✓	✗	✗	✗	✗
StaDynA [114]	✗	✗	✗	✓	✓	✗	✓
AppContext [83]	✓	✓	✓	✗	✗	✗	✗
Amandroid [85]	✓	✗	✓	✗	✗	✗	✗
ContentScope [86]	✗	✗	✗	✗	✗	✗	✓
Wognsen et al. [87]	✓	✗	✓	✗	✓	✗	✗
AsDroid [89]	✗	✗	✗	✗	✗	✗	✓
FlowDroid [5]	✓	✓	✓	✗	✗	✗	✗
IccTA [6]	✓	✓	✓	✗	✗	✗	✗
Epicc [8]	✓	✗	✓	✗	✗	✗	✗
Asynchronizer [95]	✗	✗	✗	✗	✗	✗	✓
PerfChecker [111]	✗	✗	✗	✗	✗	✗	✓
Cassandra [92]	✗	✓	✗	✗	✗	✓	✗
Apparecium [93]	✓	✗	✗	✗	✗	✗	✗
SIF [115]	✓	✗	✗	✗	✗	✗	✓
Total	**24**	**12**	**18**	**3**	**10**	**3**	**18**

for further techniques that focus on Android malware analysis and detection.

Rashidi et al. [127] present a survey on existing Android security threats and security enforcement solutions. This survey classifies Android security mechanism into three aspects: Android permission, app sandbox and inter-component communication (ICC) and categorizes existing solutions into three folds: Prevention-based, Analysis-based and Runtime monitoring-based approaches.

Haris et al. [128] present a survey especially focusing on privacy leaks and their associated risks in mobile computing. This survey has studied privacy in the area of mobile connectivity (e.g., cellular and surveillance technology) and in the area of mobile sensing (e.g., users prospects on sensor data). Besides, the authors have studied not only Android-specific leakages but also other mobile platforms including iOS and Windows. Similarly, [129] and [130] present state-of-the-art reviews with considering multiple mobile platforms.

8. Conclusions

Research on static analysis of Android apps is quickly maturing, producing more and more advanced approaches for statically uncovering security issues in app code. To summarize the state-f-the-art and enumerate the challenges to be addressed by the research community we have conducted a systematic literature review of publications on approaches involving the use of static analysis on Android apps. In the process of this review, we have collected around 90 research papers published in Software engineering, programming languages and security conference and journal venues.

Our review has consisted of investigating the categories of issues targeted by static analysis, the fundamental techniques leveraged in the approaches, the implementation of the analysis itself (i.e., which analysis sensitivities are considered, and what android characteristics are taken into account?), how the evaluation was performed, and whether the research output is available for use by the community.

We have found that, (1) most analyses are performed to uncover security flaws in Android apps; (2) many approaches are built on top of a single analysis framework, namely Soot; (3) taint analysis is the most applied fundamental analysis technique in the publications; (4) although most approaches support multiple sensitivities, path sensitivity appears overlooked; (5) all approaches are missing to consider at least 1 characteristic of Android programming in their analysis; (6), finally, research contributions artefacts, such as tools and datasets, are often unpublished.

Acknowledgment

We thank all authors of Android static analysis who have provided useful feedback to the initial draft of this SLR, during the self-checking process. This work was supported by the Fonds National de la Recherche (FNR), Luxembourg, under the project AndroMap C13/IS/5921289.

References

[1] Gartner, gartner says sales of smartphones grew 20 percent in third quarter of 2014. `https://www.gartner.com/newsroom/id/2944819/`. Accessed: 2015-08-22.

[2] Developer economics q1 2015: State of the developer nation. `https://www.developereconomics.com/reports/developer-economics-q1-2015/`. Accessed: 2015-08-22.

[3] G data: Mobile malware report. `https://public.gdatasoftware.com/Presse/Publikationen/Malware_Reports/G_DATA_MobileMWR_Q2_2015_EN.pdf`. Accessed: 2015-08-22.

[4] Étienne Payet and Fausto Spoto. Static analysis of android programs. *Information and Software Technology*, 54(11):1192–1201, 2012.

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 2014)*, 2014.

[6] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*, 2015.

[7] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240. ACM, 2012.

[8] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[9] Darell JJ Tan, Tong-Wei Chua, Vrizlynn LL Thing, et al. Securing android: A survey, taxonomy, and challenges. *ACM Computing Surveys (CSUR)*, 47(4):58, 2015.

[10] Alexandre Bartel. *Security Analysis of Permission-Based Systems using Static Analysis: An Application to the Android Stack*. PhD thesis, University of Luxembourg, 2014.

[11] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, Principles, Techniques*. Addison wesley, 1986.

[12] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP95Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, pages 77–101. Springer, 1995.

[13] David F Bacon and Peter F Sweeney. Fast static analysis of c++ virtual function calls. *ACM Sigplan Notices*, 31(10):324–341, 1996.

[14] Vijay Sundaresan, Laurie J. Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pages 264–280, 2000.

[15] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[16] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41. ACM, 1996.

[17] Barbara Kitchenham. Procedures for performing systematic reviews. 2004.

[18] Google scholar metrics: Available metrics. `https://scholar.google.com.sg/intl/en/scholar/metrics.html#metrics`. Accessed: 2015-08-22.

[19] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[20] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2015.

[21] Wenjun Hu, Jing Tao, Xiaobo Ma, Wenyu Zhou, Shuang Zhao, and Ting Han. Migdroid: Detecting app-repackaging android malware via method invocation graph. In *Computer Communication and Networks (ICCCN), 2014 23rd International Conference on*, pages 1–7. IEEE, 2014.

[22] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[23] Siegfried Rasthofer, Steven Arzt, Enrico Lovat, and Eric Bodden. Droidforce: enforcing complex, data-centric, system-wide policies in android. In *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on*, pages 40–49. IEEE, 2014.

[24] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6. ACM, 2014.

[25] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data.

[26] Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Automatically exploiting potential component leaks in android applications. In *Proceedings of the 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2014)*, 2014.

[27] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015.

[28] Yajin Zhou and Xuxian Jiang. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*, 2013.

[29] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Automatically securing permission-based software by reducing the attack surface: An application to android. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 274–277. ACM, 2012.

[30] Alexandre Bartel, John Klein, Martin Monperrus, and Yves Le Traon. Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android. *Software Engineering, IEEE Transactions on*, 40(6):617–632, 2014.

[31] Ding Li, Angelica Huyen Tran, and William GJ Halfond. Making web applications more energy efficient for oled smartphones. In *Proceedings of the 36th International Conference on Software Engineering*, pages 527–538. ACM, 2014.

[32] Ding Li, Shuai Hao, William GJ Halfond, and Ramesh Govindan. Calculating source line level energy information for android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 78–89. ACM, 2013.

[33] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 73–84. ACM, 2013.

[34] Shao Shuai, Dong Guowei, Guo Tao, Yang Tianchang, and Shi Chenjie. Modelling analysis and auto-detection of cryptographic misuse in android applications. In *Dependable, Autonomic and Secure Computing (DASC), 2014 IEEE 12th International Conference on*, pages 75–80. IEEE, 2014.

[35] Patrick PF Chan, Lucas CK Hui, and Siu-Ming Yiu. Droidchecker: analyzing android applications for capability leak. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 125–136. ACM, 2012.

[36] Shuying Liang, Andrew W Keep, Matthew Might, Steven Lyde, Thomas Gilray, Petey Aldous, and David Van Horn. Sound and precise malware analysis for android via pushdown reachability and entry-point saturation. In *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices*, pages 21–32. ACM, 2013.

[37] Zhihui Han, Liang Cheng, Yang Zhang, Shuke Zeng, Yi Deng, and Xiaoshan Sun. Systematic analysis and detection of misconfiguration vulnerabilities in android smartphones. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on*, pages 432–439. IEEE, 2014.

[38] Shuai Hao, Ding Li, William GJ Halfond, and Ramesh Govindan. Estimating mobile application energy consumption using program analysis. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 92–101. IEEE, 2013.

[39] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. Slicing droids: program slicing for smali code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1844–1851. ACM, 2013.

[40] Jinseong Jeon, Kristopher K Micinski, Jeffrey A Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S Foster, and Todd Millstein. Dr. android and mr. hide: fine-grained permissions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2012.

[41] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, Junbum Shin, and SWRD Center. Scandal: Static analyzer for detecting privacy leaks in android applications. *MoST*, 2012.

[42] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: An analysis of android ssl (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61. ACM, 2012.

[43] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54. ACM, 2013.

[44] Dimitris Geneiatakis, Igor Nai Fovino, Ioannis Kounelis, and Paquale Stirparo. A permission verification approach for android mobile applications. *Computers & Security*, 49:192–205, 2015.

[45] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. *AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale*. Springer, 2012.

[46] Mariem Graa, Nora Cuppens Boulahia, Frédéric Cuppens, and Ana Cavalliy. Protection against code obfuscation attacks based on control dependencies in android systems. In *Software Security and Reliability-Companion (SERE-C), 2014 IEEE Eighth International Conference on*, pages 149–157. IEEE, 2014.

[47] Mariem Graa, Nora Cuppens-Boulahia, Frédéric Cuppens, and Ana Cavalli. Detecting control flow in smarphones: Combining static and dynamic analyses. In *Cyberspace Safety and Security*, pages 33–47. Springer, 2012.

[48] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 101–112. ACM, 2012.

[49] Michael C Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS*, 2012.

[50] Chaorong Guo, Jian Zhang, Jun Yan, Zhiqiang Zhang, and Yanli Zhang. Characterizing and detecting resource leaks in android applications. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 389–398. IEEE, 2013.

[51] Zheng Lu and Supratik Mukhopadhyay. Model-based static source code analysis of java programs with applications to android security. In *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*, pages 322–327. IEEE, 2012.

[52] Siyuan Ma, Zhushou Tang, Qiuyu Xiao, Jiafa Liu, Tran Triet Duong, Xiaodong Lin, and Haojin Zhu. Detecting gps information leakage in android applications. In *Global Communications Conference (GLOBECOM), 2013 IEEE*, pages 826–831. IEEE, 2013.

[53] Christopher Mann and Artem Starostin. A framework for static detection of privacy leaks in android applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1457–1462. ACM, 2012.

[54] Heila van der Merwe, Oksana Tkachuk, Brink van der Merwe, and Willem Visser. Generation of library models for verification of android applications. *ACM SIGSOFT Software Engineering Notes*, 40(1):1–5, 2015.

[55] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *Proceedings of the 20th Annual Network & Distributed System Security Symposium (NDSS)*, 2014.

[56] Tristan Ravitch, E Rogan Creswick, Aaron Tomb, Adam Foltzer, Trevor Elliott, and Ledah Casburn. Multi-app security analysis with fuse: Statically detecting android app collusion. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, page 4. ACM, 2014.

[57] Dragos Sbîrlea, Michael G Burke, Salvatore Guarnieri, Marco Pistoia, and Vivek Sarkar. Automatic detection of inter-application permission leaks in android applications. *IBM Journal of Research and Development*, 57(6):10–1, 2013.

[58] Feng Shen, Namita Vishnubhotla, Chirag Todarka, Mohit Arora, Babu Dhandapani, Eric John Lehner, Steven Y Ko, and Lukasz Ziarek. Information flows as a permission mechanism. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 515–526. ACM, 2014.

[59] David Sounthiraraj, Justin Sahs, Garrett Greenwood, Zhiqiang Lin, and Latifur Khan. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. In *Proceedings of the 19th Network and Distributed System Security Symposium*, 2014.

[60] Panagiotis Vekris, Ranjit Jhala, Sorin Lerner, and Yuvraj Agarwal. Towards verifying android apps for the absence of no-sleep energy bugs. In *Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems*, pages 3–3. USENIX Association, 2012.

[61] Timothy Vidas, Jiaqi Tan, Jay Nahata, Chaur Lih Tan, Nicolas Christin, and Patrick Tague. A5: Automated analysis of adversarial android applications. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, pages 39–50. ACM, 2014.

[62] Anthony Desnos. Android: Static analysis using similarity distance. In *System Science (HICSS), 2012 45th Hawaii International Conference on*, pages 5394–5403. IEEE, 2012.

[63] Anthony Desnos and Geoffroy Gueguen. Android: From reversing to decompilation. *Proc. of Black Hat Abu Dhabi*, pages 77–101, 2011.

[64] Steffen Bartsch, Bernhard Berger, Michaela Bunke, and Karsten Sohr. The transitivity-of-trust problem in android application interaction. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 291–296. IEEE, 2013.

[65] Leonid Batyuk, Markus Herpich, Seyit Ahmet Camtepe, Karsten Raddatz, Aubrey-Derrick Schmidt, and Sahin Albayrak. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, pages 66–72. IEEE, 2011.

[66] Chia-Mei Chen, Je-Ming Lin, and Gu-Hsin Lai. Detecting mobile application malicious behaviors based on data flow of source code. In *Trustworthy Systems and their Applications (TSA), 2014 International Conference on*, pages 1–6. IEEE, 2014.

[67] Kevin Zhijie Chen, Noah M Johnson, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Thomas R Magrino, Edward XueJun Wu, Martin Rinard, and Dawn Xiaodong Song. Contextual policy enforcement in android applications with permission event graphs. In *NDSS*, 2013.

[68] Jianlin Xu, Yifan Yu, Zhen Chen, Bin Cao, Wenyu Dong, Yu Guo, and Junwei Cao. Mobsafe: cloud computing based forensic analysis for massive mobile applications using data mining. *Tsinghua Science and Technology*, 18(4), 2013.

[69] Zhang Luoshi, Niu Yan, Wu Xiao, Wang Zhaoguo, and Xue Yibo. A3: Automatic analysis of android malware. In *1st International Workshop on Cloud Computing and Information Security*. Atlantis Press, 2013.

[70] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1105–1116. ACM, 2014.

[71] Ying Zhang, Gang Huang, Xuanzhe Liu, Wei Zhang, Hong Mei, and Shunxiang Yang. Refactoring android java code for on-demand computation offloading. In *ACM SIGPLAN Notices*, volume 47, pages 233–248. ACM, 2012.

[72] Zhibo Zhao and Fernando C Colon Osono. trustdroid¢: Preventing the use of smartphones for information leaking in corporate networks through the used of static analysis taint tracking. In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*, pages 135–143. IEEE, 2012.

[73] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 93–104. ACM, 2012.

[74] Yibing Zhongyang, Zhi Xin, Bing Mao, and Li Xie. Droidalarm: an all-sided static analysis tool for android privilege-escalation malware. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 353–358. ACM, 2013.

[75] Hamid Bagheri, Alireza Sadeghi, Joshua Garcia, and Sam Malek. Covert: Compositional analysis of android inter-app permission leakage. 2015.

[76] Michael D Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros Barros, Ravi Bhoraskar, Seungyeop Han, et al. Collaborative verification of information flow for a high-assurance app store. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1092–1104. ACM, 2014.

[77] Mu Zhang and Heng Yin. Efficient, context-aware privacy leakage confinement for android applications without firmware modding. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 259–270. ACM, 2014.

[78] Mu Zhang and Heng Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS 2014)*, 2014.

[79] Julian Schutte, Dennis Titze, and JM De Fuentes. Appcaulk: Data leak prevention by injecting targeted taint tracking into android apps. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on*, pages 370–379. IEEE, 2014.

[80] Michael I Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information-flow analysis of android applications in droidsafe. In *Proc. of the Network and Distributed System Security Symposium (NDSS). The Internet Society*, 2015.

[81] Jianliang Wu, Tingting Cui, Tao Ban, Shanqing Guo, and Lizhen Cui. Paddyfrog: systematically detecting confused deputy vulnerability in android applications. *Security and Communication Networks*, 2015.

[82] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 576–587. ACM, 2014.

[83] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Proc. of the International Conference on Software Engineering (ICSE)*, 2015.

[84] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. The impact of vendor customizations on android security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 623–634. ACM, 2013.

[85] Fengguo Wei, Sankardas Roy, Xinming 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*, pages 1329–1341. ACM, 2014.

[86] Zhou Yajin and Jiang Xuxian. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*, 2013.

[87] Erik Ramsgaard Wognsen, Henrik Søndberg Karlsen, Mads Chr Olesen, and René Rydhof Hansen. Formalisation and analysis of dalvik bytecode. *Science of Computer Programming*, 92:25–55, 2014.

[88] Jingtian Wang, Xiaoquan Wu, Jun Wei, et al. Detect and optimize the energy consumption of mobile app through static analysis: an initial research. In *Proceedings of the Fourth Asia-Pacific Symposium on Internetware*, page 22. ACM, 2012.

[89] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1036–1046. ACM, 2014.

[90] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054. ACM, 2013.

[91] Jialiu Lin, Bin Lin, Norman Sadeh, and Jason Hong. Modeling users mobile app privacy preferences: Restoring usability in a sea of permission settings. In *Symposium on Usable Privacy and Security (SOUPS)*, 2014.

[92] Steffen Lortz, Heiko Mantel, Artem Starostin, Timo Bähr, David Schneider, and Alexandra 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.

[93] Dennis Titze and Julian Schütte. Apparecium: Revealing data flows in android applications. In *Proceedings of the 29th International Conference on Advanced Information Networking and Applications (AINA)*, 2015.

[94] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[95] Yu Lin, Cosmin Radoi, and Danny Dig. Retrofitting concurrency for android applications through refactoring. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 341–352. ACM, 2014.

[96] Wei Yang, Mukul R Prasad, and Tao Xie. A grey-box approach for automated gui-model generation of mobile applications. In *Fundamental Approaches to Software Engineering*, pages 250–265. Springer, 2013.

[97] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications.

[98] Damien Octeau, William Enck, and Patrick McDaniel. The ded decompiler. *Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, Tech. Rep. NAS-TR-0140-2010*, 2010.

[99] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *USENIX security symposium*, volume 2, page 2, 2011.

[100] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting android applications to java bytecode. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 6. ACM, 2012.

[101] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot. In *ACM Sigplan International Workshop on the State Of The Art in Java Program Analysis*, 2012.

[102] Chon Ju Kim and Phyllis Frankl. Aqua: Android query analyzer. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 395–404. IEEE, 2012.

[103] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30, 2002.

[104] Eugene Kuleshov. Using the asm framework to implement common java bytecode transformation patterns. *Aspect-Oriented Software Development*, 2007.

[105] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. Brahmastra: Driving apps to test the security of third-party components. In *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[106] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Thresher: Precise refutations for heap reachability. In *ACM SIGPLAN Notices*, volume 48, pages 275–286. ACM, 2013.

[107] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. *ACM SIGPLAN Notices*, 48(10):641–660, 2013.

[108] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. ApkCombiner: Combining Multiple Android Apps to Support Inter-App Analysis. In *Proceedings of the 30th IFIP International Conference on ICT Systems Security and Privacy Protection (SEC 2015), year=2015*.

[109] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *International Conference on Software Engineering (ICSE)*, 2015.

[110] Max Lillack, Christian Kästner, and Eric Bodden. Tracking load-time configuration options. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 445–456. ACM, 2014.

[111] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1013–1024. ACM, 2014.

[112] Atanas Rountev and Dacong Yan. Static reference analysis for gui objects in android software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 143. ACM, 2014.

[113] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE)*, page 59. ACM, 2012.

[114] Yury Zhauniarovich, Maqsood Ahmad, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. Stadyna: Addressing the problem of dynamic code updates in the security analysis of android applications. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 37–48. ACM, 2015.

[115] Shuai Hao, Ding Li, William GJ Halfond, and Ramesh Govindan. Sif: a selective instrumentation framework for mobile applications. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services (MobiSys)*, pages 167–180. ACM, 2013.

[116] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. Evodroid: segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 599–609. ACM, 2014.

[117] Nariman Mirzaei, Sam Malek, Corina S Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

[118] Kwanghoon Choi and Byeong-Mo Chang. A type and effect system for activation flow of components in android programs. *Information Processing Letters*, 114(11):620–627, 2014.

[119] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2014.

[120] Bruno PS Rocha, Marco Conti, Sandro Etalle, and Bruno Crispo. Hybrid static-runtime information flow and declassification enforcement. *Information Forensics and Security, IEEE Transactions on*, 8(8):1294–1305, 2013.

[121] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Instrumenting android and java applications as easy as abc. In *Runtime Verification*, pages 364–381. Springer, 2013.

[122] Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis. Technical Report 978-2-87971-129-4_TR-SNT-2014-9, April 2014.

[123] Mayur Naik and Jens Palsberg. A type system equivalent to a model checker. In *Programming Languages and Systems*, pages 374–388. Springer, 2005.

[124] Pearl Brereton, Barbara A Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of systems and software*, 80(4):571–583, 2007.

[125] Mark Turner, Barbara Kitchenham, David Budgen, and OP Brereton. Lessons learnt undertaking a large-scale systematic literature review. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2008.

[126] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Gaur, Marco Conti, and Raj Muttukrishnan. Android security: A survey of issues, malware penetration and defenses. *IEEE Communications Surveys & Tutorials*, 17:998–1022, 2015.

[127] Bahman Rashidi and Carol Fung. A survey of android security threats and defenses. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, 6, 2015.

[128] Muhammad Haris, Hamed Haddadi, and Pan Hui. Privacy leakage in mobile computing: Tools, methods, and characteristics. *arXiv preprint arXiv:1410.4978*, 2014.

[129] Mariantonietta La Polla, Fabio Martinelli, and Daniele Sgandurra. A survey on security for mobile devices. *Communications Surveys & Tutorials, IEEE*, 15(1):446–471, 2013.

[130] Guillermo Suarez-Tangil, Juan E Tapiador, Pedro Peris-Lopez, and Arturo Ribagorda. Evolution, detection and analysis of malware for smart devices. *Communications Surveys & Tutorials, IEEE*, 16(2):961–987, 2014.