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[a,2], 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, Darmstadt, Germany*
[c] *University of Wisconsin and Pennsylvania State University*

Abstract

Context: Static analysis exploits techniques that parse program source code or bytecode, often traversing program paths to check some program properties. Static analysis approaches have been proposed for different tasks, including for assessing the security of Android apps, detecting app clones, automating test cases generation, or for uncovering non-functional issues related to performance or energy. 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 analysers face when dealing with Android apps.

Objective: We aim to provide a clear view of the state-of-the-art works that statically analyse 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 (SLR) which involves studying 124 research papers published in software engineering, programming languages and security venues in the last 5 years (January 2011 - December 2015). 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 has 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 artefacts 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 other mobile systems, since Android accounts for 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, 5], to discover sensitive data leaks [6, 7], to uncover vulnerabilities [8, 9], 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 implemen-

Email address: li.li@uni.lu (Li Li)
[1] Corresponding author.
[2] the author was employed at the Technical University of Darmstadt, Germany, when he first worked on this paper

tation 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 works. Besides these specific challenges, Android carries a number of challenges for the analysis of Java programs, such as how to resolve the targets of Java reflection statements 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 [6] 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 [10] 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 (SLR) 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[3] 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.

[3]Repository available at: `http://lilicoding.github.io/SA3Repo`

The paper continues as follows. Section 2 explains the necessary background on static analysis and on the Android system. Section 3 describes the methodology we followed for the literature review. Section 4 presents the data we extract from the primarily selected papers and Section 5 leverages the data that we extract to answer the proposed research questions. In Section 6 and Section 7, we discuss our findings and the potential threats to the validity of this study, respectively. Section 8 discusses related work and Section 9 concludes this paper.

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 all the code is analyzed. This differs from dynamic analysis where portions of code could only be executed under some specific conditions that could never be met during the analysis phase. A typical static analysis process starts by representing the analyzed app code to some abstract models (e.g., call graph, control-flow graph, or UML class/sequence diagram) based on the purpose of analysis. Those abstract models actually provide a simplified interface for supporting upper-level client analyses such as taint analysis. Other information, such as the values of variables (e.g., propagated from constant values) at different statements of the CFG can also be collected to allow the static analysis to support more in-depth verification, e.g., through data-flow analysis.

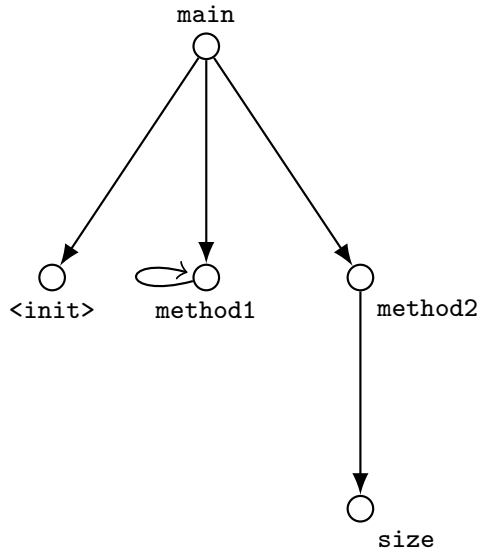
We now briefly summarize the key concepts of static analysis, including the main analysis techniques (in Section 2.1.1), the construction of call graphs (in Section 2.1.2) and the call graph enrichment related techniques (in Section 2.1.3). For more details, we encourage interested readers to refer to the doctoral dissertation of Alexandre Bartel [11], a co-author of this literature review.

Note that in this section, we will mainly focus on detailing call graphs for static analysis, instead of other representations such as UML class/sequence diagram. The main reason for this emphasis on call graphs is that, to the best of our knowledge, most relevant research works that perform inter-procedural analysis on object-oriented programs (e.g., Android apps) have somehow leveraged call graphs in their analyses. This observation is also confirmed by the primary publications selected in this SLR. Although other representations (e.g., UML-based instead of call graph) are also possible for facilitating

```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 1: Source Code of a two-classes Java program and its Call Graph Generated from the main Method

the process of static analysis, call graph is more widely used in the community. A possible reason for this trend could be that many static approaches, which statically analyze Android apps, are implemented on top of well-known frameworks such as Soot [12] and WALA [13] that provide, by default, off-the-shelf call graph construction facilities, making call graph construction a common step for inter-procedural static analysis.

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 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 [14] 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 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 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 1), 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 [15], RTA [16], VTA [17], Andersen [18], Steensguard [19], 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 2 with example code snippets and the corresponding call graphs extracted in both cases where the property holds and where it does not.

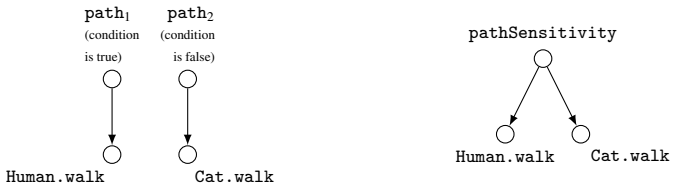
Flow Sensitivity. A flow-sensitive CG is a CG that is aware of the order of program statements. In the illustrative example of Figure 2a, a *Human* instance is first created and referred to by

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

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, 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`. While in some cases a flow-insensitive CG may be sufficient (e.g., to count the existence of certain APIs), in other cases, it brings imprecision which will necessarily lead to false positives (i.e., incorrect results) in the analysis.

Path Sensitivity. A path-sensitive CG takes the execution path into account. In the illustrative example of Figure 2b, 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 number of execution paths.

Field Sensitivity. A field-sensitive approach models each field of each object. Take the code snippet of Figure 2c 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[4]. 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*).

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 2d, 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 2e as an example. At lines 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 mainly built around one or several of the four (4) types of components whose possible interactions are illustrated in Figure 3 :

1. an *Activity* represents the visible part of Android apps, i.e., the user interfaces;
2. a *Service*, which is dedicated to executing (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] 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.

4. a *Content Provider* acts as a standard interface for other components/apps to access structured data.

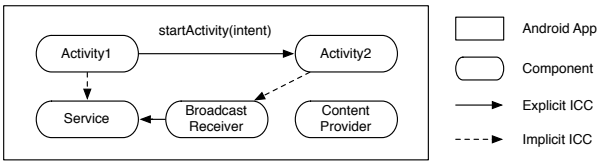


Figure 3: Overview of basic concepts of Android apps.

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 (except Content Provider-related ICC methods) 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. Android apps are primarily developed in Java, but are compiled into Dex bytecode that runs in Dalvik virtual machine (now ART), which features a register-based instruction model. Although, there existing open-source repositories where many apps source code is shared, developers use official/commercial markets to distribute their APKs. Thus, in practice, 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 source code and bytecode analyzers, which could have been leveraged, are actually useless in the Android ecosystem. As an example, the mature FindBugs[5] tool, which has demonstrated its capabilities to discover bugs in Java bytecode, can not readily be exploited for Android programs, since an additional step is required to transforms Android APKs into Java Jars.

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 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 may connect to each other.

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 are called by the Android system to start/stop/resume the component following environment needs. For example, an application in the 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 the foreground. Unfortunately, because these lifecycle methods are not directly connected to the execution flow, they hinder the soundness of some analysis scenarios.

User/System Events. Besides lifecycle methods, methods for handling user events (e.g., UI actions) and system events (e.g., low memory event, GPS location changes event [20]) constitute a challenge for static analysis of Android apps. Indeed, as such events can be fired at any time, static analysers cannot build and maintain a reliable model of the events [21]. It is further expensive to consider all the possible variants of a given model, due to limited resources [22, 23]. Nevertheless, not taking into account paths that include event-related methods may render some analysis scenarios unsound.

Inter-Component Communication (ICC). Android has put in place a specific 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 too 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

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

challenges as with Java programs, including the issues of handling dynamic code loading, reflection, native code integration, multi-threading 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 multi-threaded 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 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 if 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 [24] and which have already been used by other SLRs [25]. 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. We further consolidate this list by applying another set of exclusion criteria based on the content of the papers' abstracts (cf. Section 3.3).
- Finally, we perform a lightweight backward-snowballing on the selected publications, 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:

- 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 depth of 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 be 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 features are applied?

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

RQ3: Are the research outputs publicly available? 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

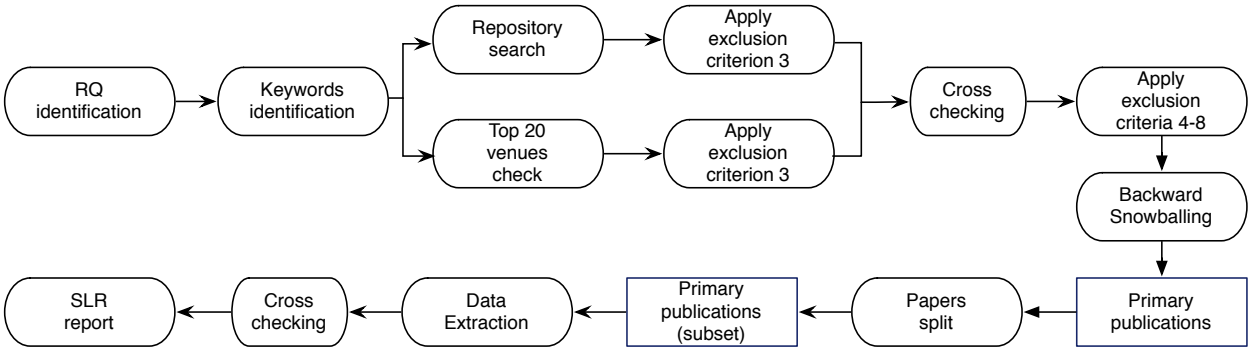


Figure 4: Overview of our SLR process.

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, multi-threading, 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 based on repositories and is complemented by a check against top venues in software engineering and security. Repository search is intended for finding the relevant publications, while the top venue check is used only as an additional checking process, to ensure that the repository search did not miss any major publication. In the following, we give more details of the two steps:

Repository Search. To find datasets of publications we first leverage five well-known electronic repositories, namely ACM Digital Library[6], IEEE Xplore Digital Library[7], SpringerLink[8], Web of Knowledge[9] and ScienceDirect[10]. 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 afterwards 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 Check. 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 repository search results are, to some extent, exhaustive, we consider all publications from well-known venues. For this SLR we have considered the top[11] 20 venues: 10 venues are from the field of software engineering and programming languages while the other 10 venues are

[6] `http://dl.acm.org`
[7] `http://ieeexplore.ieee.org`
[8] `http://link.springer.com`
[9] `http://apps.webofknowledge.com`
[10] `http://www.sciencedirect.com`
[11] Following Google Scholar Metrics: `https://scholar.google.lu/citations?view_op=top_venues&hl=en`

from the security and privacy field. Table 2 lists these venues considered at the time of review (cf. Oct. 2015), where some venues dealing with fundamental cryptography (including EUROCRYPT, CRYPTO, TCC, CHES and PKC), parallel programming (PPoPP), as well as magazines (such as IEEE Software) and non-official proceedings (e.g., arXiv Cryptography and Security) are excluded. Indeed, such venues are not the main focus 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 [26]. Intuitively, the higher *H5-index*, the better the venue.

Our top 20 venues check is performed on DBLP[12]. 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.

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 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. We have noticed that some of the collected papers 1) do not contribute to the research on static analysis of Android apps, or 2) simply grep or match API names. For example, 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 four more exclusion criteria to filter out such publications:

6. 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 [27] and EdgeMiner [28] in this paper because they are dedicated to analysing the Android framework.

7. We dismiss papers that do not actually parse the app program code, e.g., research papers that only perform static analysis on the meta-data (i.e., descriptions, configuration files, etc.) of apps are excluded.

8. We filter out technical reports, such as SCanDroid [29]. Such non-peer-reviewed papers are often re-published in a conference and journal venue, and are thus likely to be included in our search set with a different title and author list. For example, the technical report paper on IccTA [30] has eventually appeared in a conference proceeding [7], which was also collected.

9. We also dismiss papers that simply leverage the statement sequences, or grep API names from the source/app code. As an example, we exclude Juxtapp [31], a clone detection approach, from consideration since it simply takes opcode sequences for its static analysis.

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

Table 2: The top 20 venues including both conference proceedings and journals in SE/PL and S&P fields (Collected in Oct. 2015).

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
TIFS	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	Computer Security Applications Conference	29
SOUPS	USENIX Symposium On Usable Privacy and Security	29

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	1,048	387	11,826	2,416	655	155	-
Scripts verification (with same keywords)	369	302	70	17	453	155	-
Scripts exclusion (criterion 3)	264	289	57	16	453	155	-
Merge							1123
After reviewing title/abstract (criteria 4 → 5)							302
After skimming full paper (criteria 6, 7 and 8)							142
After final discussion							118
Author recommendation							+4
Backward Snowballing							+2
Total							124

3.4. Backward Snowballing

As recommended in the guidelines of Kitchenham and Charters [32], we perform a lightweight[13] backward snowballing from reference lists of the articles identified via repository search. The objective is to find additional relevant papers in the literature that may have not been matched via our search keywords.

It is tedious and time-consuming to perform this step manually. Thus, we leverage python scripts to automatically extract references from our selected primary publications. In particular, we first leverage *pdfx*[14] to transfer a given paper from pdf to text format. Then, we retrieve all the references from the previously generated text files. To further mitigate manual efforts, we again use scripts to filter out references whose publication date fall outside of our SLR timeline and whose titles appear without keywords that are defined in the scope of this SLR. After these steps, we manually read and check the remaining references. If a given reference (paper) is in line with this work but is not yet available in our primary publication set, we then include it into our final paper set.

3.5. 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.

[13] We only perform backward-snowballing once, meaning that the newly found publications are not considered for snowballing.

[14] https://github.com/metachris/pdfx

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 list of papers to be analyzed.

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

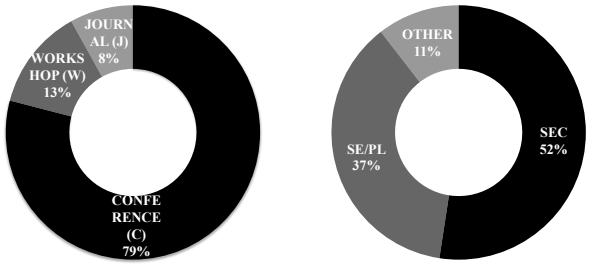
1. The ACM repository may contain papers that were originally published in IEEE or Springer and
2. The Web of Knowledge repository may contain papers that are published in Elsevier.
3. The five repositories may contain papers that appear in the top-20-venues set.

After the merge step, we split the searched papers to all the co-authors of this paper. We manually review the title/abstract and examine the full content of the selected papers, with applying the exclusion criteria 4-8. With final discussion between authors, we finally select 118 papers as primary publications.

For the self-checking process, we have collected 343 distinct email addresses of 419 authors for the 88 primary publications selected in our search ((up to May 2015)). 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. 15 of them are already considered by our SLR (from Jul. 2015 to Dec. 2015). The remaining 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).

Regarding to the backward-snowballing, overall, we found 1815 referenced articles whose publication date fall within our SLR timeline. Only 53 of these articles had titles without keywords that could be matched by search. We reviewed these papers and found that only 2 fit our criteria (i.e., deal with static analysis in the context of Android apps).

In total, our SLR examines 124 publications, which are listed in Table A.12 and Table A.13. Fig. 5 illustrates the distributions of these publications by types(Fig 5a) and publication domains (Fig. 5b). 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



(a) Type. (b) Domain.

Figure 5: Statistics of examined publications. SE/PL stands for Software Engineering/Programming Language, SEC stands for Security.

Figure 6: Word cloud of all the conference names of examined publications. Journal and workshop publications are not considered.

while another larger proportion was published in Software Engineering venues. Very few contributions were published in other types of venues. MIGDroid [33], published in the *Network* domain and CloneCloud [34], published in the *Systems* domain, are the rare exceptions. This finding comforts our initial heuristics of considering top venues in SE/PL and SEC to verify that our repository search was exhaustive.

Fig. 6 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 our study has at least considered the important relevant works.

4. Data Extraction

Once relevant papers have been collected, we build a taxonomy of the information that must be 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. 7 overviews the information extracted from the selected publications.

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 publi-

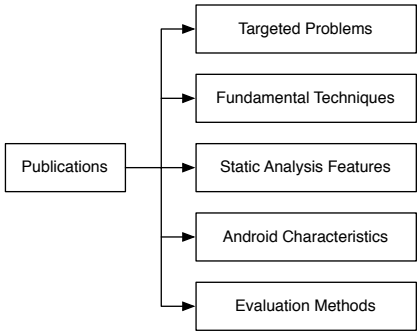


Figure 7: Overview of information extracted from a given primary publication.

cations. 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 [12] or WALA [13] that are leveraged.

Static Analysis features. This dimension includes static analysis related features. Given a primary publication, 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 characteristics 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 "Do the studied approaches support ICC or IAC?" belong to this dimension.

Evaluation methods. 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 apps). Questions like how many in-the-wild apps are evaluated are also addressed in this dimension.

5. Summary of Findings

In this section, we 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 for achieving various tasks. Among others, such analyses are implemented to highlight various security issues (such as private data leaks or permission management concerns), to verify code, to compare apps for detecting clones, to automate the generation of test cases, or to assess code efficiency in terms of performance and energy consumption. We have identified 8 recurring purposes of Android-targeted static analysis approaches in the literature. We detail these purposes and provide statistics 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 [6], 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 [35, 36, 37, 38, 7] 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 [8] detects potential component hijacking-based flows through reachability analysis on customized system dependence graphs. Epicc [9] and IC3 [39] are tools that implement static analysis techniques for implementing detection scenarios of inter-component vulnerabilities. Based on these studies, PCLeaks [38] 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 [40] 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. [41, 42], this permission model is an intrinsic risk, since apps can be granted more permissions than they actually need. Malware may indeed leverage permissions (which are unnecessary to the core app functionality) to achieve their malicious goals. PSCout [27] is currently the most extensive work that dissects the Android permission specification from Android OS source code using static analysis. However, we do not take PSCout into consideration in this SLR as our focus is static analysis of Android apps rather than Android OS.

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. [43], 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. [44] present a tool to calculate source line level energy consumption through combining program analysis and statistical modeling. The output of these analyses can then be leveraged to perform quantitative

and qualitative empirical investigations into the categories of API calls and usage patterns that exhibit high energy consumption profiles [45].

Clone Detection. Researchers have also leveraged static analysis to perform clone detection of Android apps. Indeed, as presented by Ruiz et al. [46, 47], who have conducted a large scale empirical study, app clone is very common in mobile apps. Towards taming app clones, several works such as DNADroid [48] and AnDarwin [49] have been provided by the community to detect cloned apps. Recently, studies have further shown that it is also necessary for clone detection approaches to consider the context of code obfuscation and library usages [50, 51], so as to improve their accuracy.

Test Case Generation. The pervasiveness of Android apps have underlined the need for applicable automated testing techniques. Test case generation aims to provide a set of executable test cases, which can be leveraged to support automatic and repeatable testings. A common means to generate test cases is to conduct symbolic execution on source code with the guide of some pre-extracted models, so as to ensure the reachability of certain branches. For example, SIG-Droid [52], a framework for system testing of Android apps, automatically generates test cases through symbolic execution with two models: *Interface Model*, which is leveraged to find values that a given app can receive and *Behavior Model*, which is used to generate the sequences of events in order to drive the symbolic execution.

Code Verification. Code verification intends to ensure the correctness of a given app. For instance, Cassandra [53] is proposed to check whether Android apps comply with their personal privacy requirements before installing an app. As another example, researchers have also extended the Julia [4] static analyzer to perform code verification through formal analyses of Android programs.

Cryptography Implementation Issues. In addition to the aforementioned concerns, state-of-the-art works have also targeted the cryptography implementation issues. As an example, CryptoLint [54] leverages program analysis techniques to automatically check apps hosted on the official Google Play store. It finds that 10,327 out of 11,748 apps (nearly 88%) that use cryptographic APIs have made at least one mistake, demonstrating that cryptographic APIs are not used in a fashion way that maximize the overall security of apps.

Table 4 enumerates approaches from our primary publications which fall into the 8 purposes described above. The summary statistics in Fig. 8 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.*

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).

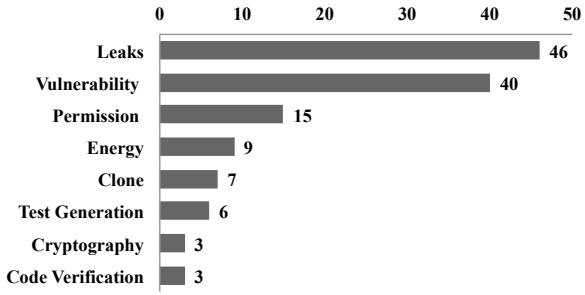


Figure 8: Statistics of main concerns addressed by the publications.

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 implement 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. 9 summarizes the frequency of usages, where *Jimple*, which is used by the popular Soot tool, appears as the most used IR followed by the *Smali* intermediate representation, which is used by Apktool.

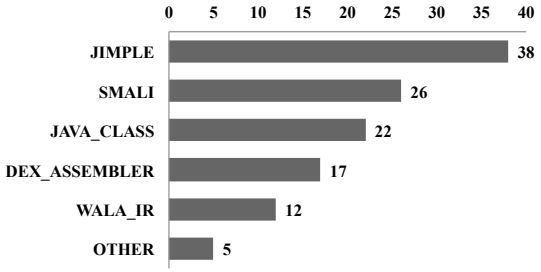


Figure 9: Distribution of code representations used by examined publications.

> ***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

Table 4: Recurrent analysis Purposes and related Publications.

Tool	*Leaks*	*Cryptography*	*Permission*	*Vulnerability*	*Code verification*	*Energy*	*Clone*	*Test Case Generation*
A3 [55]				✓				
A3E [57]								✓
A5 [59]				✓				
AAPL [61]	✓			✓				
ACTEve [63]								✓
Adagio [65]				✓				
AdRisk [67]				✓				
Amandroid [68]	✓	✓		✓				
Anadroid [69]	✓		✓	✓				
AnDarwin [49]							✓	
AndRadar [72]							✓	
Androguard2 [74]				✓				
Androguard [76]				✓			✓	
android-app-analysis-tool [78]			✓					
AndroidLeaks [80]	✓		✓					
Androlizer [81]				✓				
Apparecium [83]	✓							
AppAudit [85]	✓							
AppCaulk [86]	✓							
AppContext [88]			✓					
AppIntent [90]	✓							✓
Apposcopy [92]	✓							
AppSealer [94]	✓							
AsDroid [95]	✓							
Bartel et al. [42]			✓					
Bartsch et al. [97]				✓				
Bastani et al. [99]	✓							
BlueSeal [101]	✓		✓					
Brox [103]	✓							
Capper [105]	✓							
Cassandra [53]	✓				✓			
Chen et al. [108]							✓	
Chen et al. [110]				✓				
CHEX [8]	✓			✓				
ClickRelease [113]				✓				
CloneCloud [34]						✓		
CMA [116]		✓						
ComDroid [117]				✓				
ContentScope [119]	✓			✓				
COPES [41]			✓					
Cortesi et al. [122]	✓							
Covert [124]	✓			✓				
CredMiner [126]	✓							
CryptoLint [54]		✓						
DescribeMe [129]	✓							
DEvA [130]				✓				
Dflow+DroidInfer [132]	✓							
DidFail [36]	✓							
DNADroid [48]							✓	
DPartner [136]						✓		
DroidAlarm [138]	✓			✓				
DroidChecker [140]	✓							
Total								

Tool	*Leaks*	*Cryptography*	*Permission*	*Vulnerability*	*Code verification*	*Energy*	*Clone*	*Test Case Generation*
DroidJust [56]	✓							
DroidSafe [58]	✓		✓	✓				
DroidSIFT [60]				✓				
DroidSim [62]							✓	
EcoDroid [64]						✓		
eLens [66]						✓		
Epicc [9]				✓				
FlowDroid [6]	✓							
FUSE [70]	✓		✓	✓				
Gallingani et al. [71]				✓				
Gible et al. [73]				✓				
Graa et al. [75]				✓				
HelDroid [77]	✓							
Hopper [79]				✓				
IccTA [7]	✓							
IFT [82]	✓							
Jensen et al. [84]								✓
Julia [4]					✓			
Lin et al. [87]			✓					
Lu et al. [89]					✓			
MalloDroid [91]				✓				
Mann et al. [93]	✓							
MIGDroid [33]				✓			✓	
MobSafe [96]				✓				
Nyx [43]						✓		
PaddyFrog [98]				✓				
Pathak et al. [100]						✓		
Pegasus [102]				✓				
PermissionFlow [104]	✓		✓	✓				
Poeplau et al. [106]				✓				
Redexer [107]			✓					
Relda [109]	✓							
SAAF [111]			✓	✓				
SADroid [112]			✓	✓				
Scandal [114]	✓							
SEFA [115]	✓		✓	✓				
SIG-Droid [52]								✓
SmartDroid [118]	✓							
SMV-Hunter [120]				✓				
Sufatrio et al. [121]	✓							
TASMAN [123]	✓							
TrustDroid [125]	✓							
Uranine [127]	✓							
Vekris et al. [128]						✓		
vLens [44]						✓		
W2AIScanner [131]	✓							✓
Wang et al. [133]						✓		
WeChecker [134]	✓							
Wognsen et al. [135]				✓				
Woodpecker [137]	✓							
Zuo et al. [139]				✓				
	46	**3**	**15**	**40**	**3**	**9**	**7**	**6**

Others (Publications Without ✓)

ApkCombiner [141], AQUA [142], AsyncDroid [143], Asynchronizer [144], AutoPPG [145], Brahmastra [146], Choi et al. [147], EvoDroid [148], Gator2 [149], Gator3 [150], Gator [151], I-ARM-Droid [152], IC3 [39], Lotrack [153], ORBIT [154], PerfChecker [155], Rocha et al. [156], SIF [157], StaDynA [158], THRESHER [159], Violist [160]

results and thus to reach the target purposes. In our study, we have identified six 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 yielding false negative results. An abstract interpretation typically involves three artefacts: an abstract value, a flow function and an initial state. An abstract value is a set of concrete values. For example, T could stand for the set of all integer values in a standard integer constant propagation analysis. A flow function defines the abstract semantics of every statement type, which usually takes as input a statement with an abstract state and the output is the abstract state after executing the statement. An abstract interpretation needs to maintain an interpreter state and an initial state is needed to specify the point when interpretation starts (a typical initial state from constant propagation is to set everything as T). 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 and statically analyze Java and Android apps for the development of high-quality, formally verified products. SCanDal [114], another sound and automatic static analyzer, also leverages abstract interpretation to detect privacy leaks in

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

TOOL	Brief Description	IR	Example Usages
Soot [12]	A Java/Android static analysis and optimization framework	JIMPLE, JASMIN	FlowDroid [6], IccTA [7], AppIntent [90]
WALA[a]	A Java/Javascript static analysis framework	WALA-IR (SSA-based)	AsDroid [95], Asynchronizer [144], ORBIT [154]
Chord [161]	A Java program analysis platform	CHORD-IR (SSA-based)	CloneCloud [34]
Androguard [74, 76]	Reverse engineering, malware/goodware analysis of Android apps	DEX_ASSEMBLER	MalloDroid [91], Relda [109]
Ded [162]	A DEX to Java bytecode translator	JAVA_CLASS	Enck et al. [163]
Dare [164]	A DEX to Java bytecode translator	JAVA_CLASS	Epicc [9], IC3 [39]
Dexpler [165]	A DEX to Jimple translator	JIMPLE	BlueSeal [101]
Smali/Baksmali[b]	A DEX to Smali translator (and verse visa)	SMALI	Woodpecker [137], SEFA [115]
Apktool[c]	A tool for reverse engineering Android apps	SMALI	PaddyFrog [98], Androlizer [81]
dex2jar[d]	A DEX to Java bytecode translator	JAVA_CLASS	DroidChecker [140], Vekris et al. [128]
dedexer[e]	A disassembler for DEX files	DEX_ASSEMBLER	Brox [103], AQUA [142]
dexdump	A disassembler for DEX files	DEX_ASSEMBLER	ScanDal [114]
dx	A Java bytecode to DEX translator	DEX_ASSEMBLER	EdgeMiner [28]
jd-gui[f]	A Java bytecode to source code translator (and also an IDE)	JAVA_CLASS	Wang et al. [133]
ASM [166, 167]	A Java manipulation and analysis framework	JAVA_CLASS	COPES [41]
BCEL[g]	A library for analyzing and instrumenting Java bytecode	JAVA_CLASS	vLens [44], Julia [4], Nyx [43]
Redexer	A reengineering tool that manipulates Android app binaries	DEX_ASSEMBLER	Brahmastra [146]

[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	A3E [57], AAPL [61], AnDarwin [49], AndroidLeaks [80], AsDroid [95], Asynchronizer [144], CHEX [8], DNADroid [48], Hopper [79], ORBIT [154], Poeplau et al. [106], THRESHER [159]
JIMPLE	A5 [59], ACTEve [63], android-app-analysis-tool [78], ApkCombiner [141], AppContext [88], AppIntent [90], Apposcopy [92], AppSealer [94], AutoPPG [145], Bartel et al. [42], Bastani et al. [99], BlueSeal [101], Capper [105], COPES [41], Covert [124], DescribeMe [129], DEvA [130], DidFail [36], DroidJust [56], DroidSafe [58], EcoDroid [64], Epicc [9], FlowDroid [6], Gallingani et al. [71], Gator2 [149], Gator3 [150], Gator [151], HelDroid [77], IC3 [39], IccTA [7], Lotrack [153], PerfChecker [155], Sufatrio et al. [121], TASMAN [123], Vekris et al. [128], Violist [160], W2AIScanner [131], WeChecker [134]
DEX_ASSEMBLER	Adagio [65], AndRadar [72], Androguard2 [74], Androguard [76], AQUA [142], Brahmastra [146], Brox [103], ComDroid [117], CryptoLint [54], Dflow+DroidInfer [132], Lin et al. [87], MalloDroid [91], Mann et al. [93], Redexer [107], Relda [109], Scandal [114], StaDynA [158]
SMALI	A3 [55], AdRisk [67], Anadroid [69], Androlizer [81], Apparecium [83], AppCaulk [86], Chen et al. [108], ClickRelease [113], CMA [116], ContentScope [119], CredMiner [126], DroidSim [62], I-ARM-Droid [152], Jensen et al. [84], MIGDroid [33], MobSafe [96], PaddyFrog [98], SAAF [111], SADroid [112], SEFA [115], SmartDroid [118], SMV-Hunter [120], Uranine [127], Wognsen et al. [135], Woodpecker [137], Zuo et al. [139]
OTHER	Amandroid [68], Cortesi et al. [122], FUSE [70], Nyx [43], SIG-Droid [52]
JAVA_CLASS	AppAudit [85], AsyncDroid [143], Bartsch et al. [97], Chen et al. [110], Choi et al. [147], CloneCloud [34], DPartner [136], DroidAlarm [138], DroidChecker [140], DroidSIFT [60], eLens [66], EvoDroid [148], IFT [82], Julia [4], Lu et al. [89], Pathak et al. [100], Pegasus [102], PermissionFlow [104], SIF [157], TrustDroid [125], vLens [44], Wang et al. [133]

Android apps.

Taint Analysis. A taint analysis is a kind of information flow analysis where objects are tainted and tracked using a data-flow analysis. If a tainted object flows to a point where it should not, i.e. a sink, then an alarm is raised. FlowDroid [6], for 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 [168]), 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). As another example, AppSealer [94] 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.

Symbolic Execution. Symbolic execution is useful for generating possible program inputs, detecting infeasible paths, etc. Typically, symbolic values are considered for inputs to propagate the execution. Those symbolic values will be used to generate expressions and constraints that could be further leveraged

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

Techniques	Publications	Percentage[a]
Abstract Interpretation	Anadroid [69] Cortesi et al. [122], Hopper [79], Julia [4], Lu et al. [89], Mann et al. [93], Rocha et al. [156], Scandal [114],	6.5%
Taint Analysis	AAPL [61], Amandroid [68], Anadroid [69], AndroidLeaks [80], Apparecium [83], AppAudit [85], AppCaulk [86], AppContext [88], Apposcopy [92], AppSealer [94], Bastani et al. [99], Brox [103], Capper [105], CHEX [8], Cortesi et al. [122], CredMiner [126], DescribeMe [129], Dflow+DroidInfer [132], DidFail [36], DroidChecker [140], DroidJust [56], DroidSafe [58], FlowDroid [6], FUSE [70], Gallingani et al. [71], HelDroid [77], IccTA [7], Lotrack [153], Mann et al. [93], MobSafe [96], PermissionFlow [104], SEFA [115], Sufatrio et al. [121], TASMAN [123], TrustDroid [125], Uranine [127], W2AIScanner [131], WeChecker [134]	30.6%
Symbolic Execution	ACTEve [63], AppIntent [90], ClickRelease [113], Gallingani et al. [71], Jensen et al. [84], SIG-Droid [52], TASMAN [123], W2AIScanner [131]	6.5%
Program Slicing	AndroidLeaks [80], Apparecium [83], AppCaulk [86], AppSealer [94], AQUA [142] Brox [103], Capper [105], CredMiner [126], CryptoLint [54], eLens [66], Hopper [79], MobSafe [96], Poeplau et al. [106], Rocha et al. [156], SAAF [111],	12.1%
Code Instrumentation	ACTEve [63] Androguard2 [74], android-app-analysis-tool [78], AppCaulk [86], AppSealer [94], AsyncDroid [143], Bastani et al. [99], Brahmastra [146], Capper [105], Cassandra [53], CMA [116], DidFail [36], DroidSafe [58], I-ARM-Droid [152], IccTA [7], Nyx [43], ORBIT [154], Rocha et al. [156], SIF [157], SmartDroid [118], Sufatrio et al. [121], Uranine [127], vLens [44],	18.5%
Type/Model Checking	Choi et al. [147], Covert [124] Dflow+DroidInfer [132], DroidAlarm [138], IFT [82], Lu et al. [89], Mann et al. [93], SADroid [112],	6.5%

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

(e.g., by a constraint solver) to produce possible inputs fulfilling all the conditional branches inside the given path. Those inputs can then be taken as test cases to explore the given path for repeatable dynamic analysis. If no input is produced, the given path is thus confirmed to be infeasible. 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 the field of program analysis to reduce the set of program behaviors while keeping the interesting program behavior unchanged. Given a variable v in program p that we are interested in, a possible slice would consist of all statements in p that may affect the value of v. As an example, Hoffmann et al. [111] 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 [54] computes static program slices that terminate in calls to cryptographic API methods, and then extract the necessary information from these slices.

Code Instrumentation. Static analysis is often necessary to find sweet spots where to insert code for collecting runtime behaviour data (cf. SIF framework [157]). In recent works, code instrumentation has also been employed to address challenges for static analysis in Android apps, including for artificially linking components for inter-component communication detection [7], or replacing reflection calls with standard java calls so as to reduce incomplete analyses due to broken control-flows [169]. In Android community, Arzt et al. [170] have introduced several means to instrument Android apps based on Soot. As an example, IccTA [7] instruments Android apps to reduce an inter-component taint propagation problem to an intra-component problem. Nyx [43] instruments an 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. Besides Soot [12], other tools/frameworks such as WALA [13] and ASM [166] are also able to support instrumentation of Android apps.

Type/Model Checking. Type and model checking are two prevalent approaches to program verification. Type checking, which can occur at compile/execution time (i.e., static/dynamic type checking), is the process of verifying type constraints of a program. The goal of type checking is to ensure that a given program is type-safe where the possibility of type errors (e.g., a float operation is performed on an integer or an integer operator is applied to strings) is kept to a minimum [171]. Model checking is the process of verifying whether a finite-state system has met a given specification [172]. The main difference between type and model checking 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 [173]. As an example, COVERT [124] 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 [53] 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. [82] 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 different techniques. The summary statistics show that taint analysis, which is used for tracking data, is the most applied technique (30.6% of primary publications), while 18.5% primary publications involve code instrumentation and 12.1% primary publications have applied program slicing technique in their approaches. Type/Model checking, abstract interpretation, and symbolic execution account each for 6.5% of the primary publications.

> ***RQ 2.2****: Taint analysis remains the most applied technique in static analysis of Android apps. This is in line 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 precision 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 sensitivities that their analyses take into account. *Field-sensitivity* appears to be the most considered with 48 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 42 and 40 publications respectively). The least considered sensitivity is *Path-sensitivity* (only 6 publications), probably due to the scalability issues that it raises.

In theory, the more sensitivities considered, the more precise the analysis is. Indeed, as shown by Arzt et al. [6], the authors of FlowDroid, who claimed that the design of being context-, flow-, field-, and object-sensitive maximizes precision and recall, i.e., aims at minimizing the number of missed leaks and false warnings. Livshits et al. [174, 175] also stated that both context-sensitivity and path-sensitivity are necessary to achieve a low false positive rate (i.e., higher precision). It is thus reasonable to state that only two approaches, namely TRESHER [159] and Hopper [79], achieves high precision by taking into account all sensitivities. However, given the relatively high precision of other 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 for static analysis. Path-sensitivity is the least taken into account by the Android research community.*

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, it is challenging for static analysis approaches to build control-flow graphs (e.g., to continuously keep track of sensitive data flows). We found that 57 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 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 64 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. 74 publications from our set explicitly 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, 30 research papers explicitly deal with ICC. As examples, Epicc [9] and IC3 [39] attempt to extract the necessary information of ICC in Android apps which can support other approaches, including IccTA [7], and DidFail [36], in performing ICC-aware analyses across components. AmanDroid [68] 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 30 publications, from our set, take into account XML layouts to support more complete analysis scenarios. Mostly, those analyses consist in tracking callback methods indicated in XML files but which are not explicitly registered in the app code. Consider for example the code snippets in Listing 1 where we provide two distinct, but equivalent, implementations for registering a listener to a layout Button.

Table 8: Classification of Approaches according to the Sensitivities considered.

Tool	Context-Sensitive	Flow-Sensitive	Field-Sensitive	Object-Sensitive	Path-Sensitive
A3E [57]		✓			
AAPL [61]	✓	✓	✓	✓	
Amandroid [68]	✓	✓	✓	✓	
Anadroid [69]	✓		✓	✓	✓
AndroidLeaks [80]	✓				
Apparecium [83]		✓	✓		✓
AppAudit [85]			✓		
AppCaulk [86]	✓		✓		
AppContext [88]	✓	✓	✓	✓	
Apposcopy [92]	✓		✓	✓	
AppSealer [94]	✓	✓	✓		
AQUA [142]		✓			
AsDroid [95]		✓	✓	✓	
Asynchronizer [144]	✓		✓		
Bartel et al. [42]		✓	✓		
Bartsch et al. [97]	✓	✓			
Bastani et al. [99]			✓		
Brox [103]	✓	✓			
Capper [105]	✓	✓	✓		
Cassandra [53]			✓		
Chen et al. [110]		✓			
CHEX [8]	✓	✓	✓	✓	
Choi et al. [147]				✓	
CloneCloud [34]			✓		
ComDroid [117]		✓			
ContentScope [119]					✓
COPES [41]		✓	✓		
Covert [124]	✓		✓		
CredMiner [126]			✓		
CryptoLint [54]			✓		
DescribeMe [129]	✓	✓	✓	✓	
Dflow+DroidInfer [132]	✓		✓	✓	
DidFail [36]	✓	✓	✓	✓	
DPartner [136]			✓		
DroidJust [56]	✓	✓	✓	✓	
DroidSafe [58]	✓	✓	✓	✓	
DroidSIFT [60]	✓	✓			
Epicc [9]	✓	✓	✓		
FlowDroid [6]	✓	✓	✓	✓	
FUSE [70]	✓		✓		
Gator2 [149]	✓			✓	
Gator3 [150]	✓			✓	
Gator [151]	✓			✓	
HelDroid [77]	✓	✓	✓	✓	
Hopper [79]	✓	✓	✓	✓	✓
IC3 [39]	✓	✓	✓		
IccTA [7]	✓	✓	✓	✓	
IFT [82]	✓	✓	✓		
Julia [4]		✓			
Lotrack [153]	✓	✓	✓	✓	
Mann et al. [93]			✓		
Pegasus [102]	✓	✓	✓		
PermissionFlow [104]	✓	✓	✓		
Rocha et al. [156]		✓			
Scandal [114]	✓	✓			
SEFA [115]			✓		
Sufatrio et al. [121]	✓	✓	✓	✓	
TASMAN [123]	✓	✓	✓	✓	
THRESHER [159]	✓	✓	✓	✓	✓
TrustDroid [125]			✓		
Vekris et al. [128]	✓	✓			
Violist [160]	✓			✓	
W2AIScanner [131]	✓		✓	✓	
WeChecker [134]	✓	✓	✓	✓	
Wognsen et al. [135]		✓	✓		
Woodpecker [137]			✓		✓
Total	**42**	**40**	**48**	**27**	**6**

Others (Publications Without ✓)

A3 [55], A5 [59], ACTEve [63], Adagio [65], AdRisk [67], AnDarwin [49], AndRadar [72], Androguard2 [74], Androguard [76], android-app-analysis-tool [78], Androlizer [81], ApkCombiner [141], AppIntent [90], AsyncDroid [143], AutoPPG [145], BlueSeal [101], Brahmastra [146], Chen et al. [108], ClickRelease [113], CMA [116], Cortesi et al. [122], DEvA [130], DNADroid [48], DroidAlarm [138], DroidChecker [140], DroidSim [62], EcoDroid [64], eLens [66], EvoDroid [148], Gallingani et al. [71], Gible et al. [73], Graa et al. [75], I-ARM-Droid [152], Jensen et al. [84], Lin et al. [87], Lu et al. [89], MalloDroid [91], MIGDroid [33], MobSafe [96], Nyx [43], ORBIT [154], PaddyFrog [98], Pathak et al. [100], PerfChecker [155], Poeplau et al. [106], Redexer [107], Relda [109], SAAF [111], SADroid [112], SIF [157], SIG-Droid [52], SmartDroid [118], SMV-Hunter [120], StaDynA [158], Uranine [127], vLens [44], Wang et al. [133], Zuo et al. [139]

If the XML implementation is not retained, function `myFancyMethod` would be considered as dead code for any analyzer not taking into account XML information.

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***: *Only few works 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, which are further delivered directly within their implementation (as a whole), leaving little opportunity for reuse by other approaches.*

5.3. Availability 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 124 reviewed papers, 41 (i.e., only 33%) have provided a publicly available tool. This finding suggests that,

```
1  //Registration of onClickListener in app code
2  Button btn = (Button)
       findViewById(R.id.mybutton);
3  btn.setOnClickListener(new
       View.OnClickListener() {
4   @Override
5   public void onClick(View v) {
6    myFancyMethod(v);
7   }
8  });
9  }
10
11 //Registration of onClickListener through XML
12 <Button android:id="@+id/mybutton"
13   android:text="Click Me!"
14   android:onClick="myFancyMethod" />
```

Listing 1: Registering a callback function for a button

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

Tool	Lifecycle	Callback-Methods	Entry-Points	ICC	IAC	XML Layout
A3E [57]	✓	✓	✓			✓
A5 [59]		✓	✓			
AAPL [61]	✓	✓	✓			
ACTEve [63]	✓	✓				✓
AdRisk [67]			✓			
Amandroid [68]	✓	✓	✓	✓		✓
Anadroid [69]	✓	✓	✓			✓
AndroidLeaks [80]		✓	✓			
Apparecium [83]	✓	✓				
AppAudit [85]	✓	✓	✓	✓		
AppContext [88]	✓	✓	✓	✓		
AppIntent [90]	✓	✓	✓	✓		
Apposcopy [92]	✓		✓	✓		
AppSealer [94]	✓		✓			
AsDroid [95]	✓	✓	✓	✓		✓
Asynchronizer [144]	✓	✓	✓			
AutoPPG [145]		✓		✓		
Bartsch et al. [97]	✓	✓	✓	✓		
Bastani et al. [99]	✓	✓	✓			
BlueSeal [101]		✓	✓	✓	✓	✓
Brahmastra [146]	✓	✓		✓		✓
Brox [103]			✓			
Capper [105]	✓		✓			
Cassandra [53]			✓			
CHEX [8]	✓	✓	✓			
Choi et al. [147]	✓	✓		✓		
ClickRelease [113]	✓	✓	✓			✓
CloneCloud [34]			✓			
CMA [116]			✓			
ComDroid [117]		✓				
ContentScope [119]		✓	✓			
Covert [124]	✓	✓	✓	✓	✓	
CredMiner [126]		✓				
CryptoLint [54]			✓			
DescribeMe [129]	✓	✓	✓			✓
DEvA [130]	✓	✓	✓			
Dflow+DroidInfer [132]	✓	✓	✓	✓		
DidFail [36]	✓	✓	✓	✓	✓	✓
DPartner [136]		✓	✓			
DroidAlarm [138]			✓	✓		
DroidChecker [140]	✓	✓	✓			
DroidJust [56]	✓	✓	✓			✓
Total						

Tool	Lifecycle	Callback-Methods	Entry-Points	ICC	IAC	XML Layout
DroidSafe [58]	✓	✓	✓	✓	✓	✓
DroidSIFT [60]		✓	✓			
DroidSim [62]		✓				
EcoDroid [64]		✓				
Epicc [9]	✓	✓	✓			
EvoDroid [148]	✓	✓	✓			✓
FlowDroid [6]	✓	✓	✓			✓
FUSE [70]	✓	✓	✓	✓	✓	✓
Gator2 [149]	✓	✓	✓	✓		✓
Gator3 [150]	✓	✓	✓	✓		✓
Gator [151]	✓	✓	✓			✓
HelDroid [77]	✓	✓	✓			✓
Hopper [79]	✓	✓	✓			
IC3 [39]	✓	✓	✓			
IccTA [7]	✓	✓	✓	✓		✓
IFT [82]				✓		
Julia [4]			✓			✓
Lotrack [153]	✓	✓	✓			✓
Mann et al. [93]	✓	✓	✓			
ORBIT [154]			✓			
PaddyFrog [98]			✓	✓		
Pathak et al. [100]	✓		✓	✓		
Pegasus [102]	✓	✓	✓	✓		
PerfChecker [155]	✓	✓	✓			✓
PermissionFlow [104]	✓	✓	✓			
Poeplau et al. [106]	✓		✓			
Relda [109]	✓	✓	✓			
Scandal [114]		✓	✓			
SEFA [115]			✓	✓	✓	
SIG-Droid [52]	✓	✓	✓	✓		✓
SmartDroid [118]			✓			✓
SMV-Hunter [120]			✓			✓
Sufatrio et al. [121]	✓	✓	✓	✓		✓
TASMAN [123]	✓	✓	✓			✓
THRESHER [159]			✓			
Uranine [127]	✓	✓	✓			
Vekris et al. [128]	✓	✓	✓	✓		
W2AIScanner [131]	✓	✓	✓			✓
WeChecker [134]	✓	✓	✓	✓		✓
Wognsen et al. [135]	✓	✓	✓			
Woodpecker [137]	✓	✓	✓			
Zuo et al. [139]	✓	✓	✓	✓		
Total	**57**	**64**	**74**	**30**	**6**	**30**

Others (Publications Without ✓)

A3 [55], Adagio [65], AnDarwin [49], AndRadar [72], Androguard2 [74], Androguard [76], android-app-analysis-tool [78], Androlizer [81], ApkCombiner [141], AppCaulk [86], AQUA [142], AsyncDroid [143], Bartel et al. [42], Chen et al. [108], Chen et al. [110], COPES [41], Cortesi et al. [122], DNADroid [48], eLens [66], Gallingani et al. [71], Gible et al. [73], Graa et al. [75], I-ARM-Droid [152], Jensen et al. [84], Lin et al. [87], Lu et al. [89], MalloDroid [91], MIGDroid [33], MobSafe [96], Nyx [43], Redexer [107], Rocha et al. [156], SAAF [111], SADroid [112], SIF [157], StaDynA [158], TrustDroid [125], Violist [160], vLens [44], Wang et al. [133]

currently, despite the increasing size of the community working on static analysis of Android apps, sharing of research efforts is still limited. Table 10 summarizes the publicly available approaches, among which 34 are 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 to finding real and possibly solve problems of real-word apps, which may have already been used by thousands of users. As shown in Table 10, only 7 approaches have taken into account in-the-lab and in-the-wild experiments at the same time.

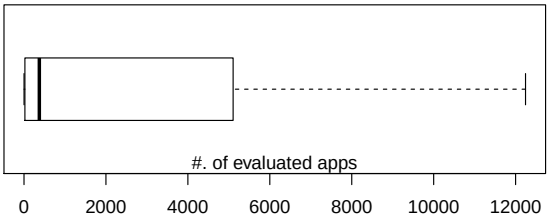


Figure 10: The distribution of the number of evaluated apps (for all the approaches reviewed in this work).

Figure 10 represents the distribution of the number of apps evaluated through in-the-wild experiments by the approaches reviewed in this work. The median number of apps that those in-the-wild experiments consider are 374. The maximum number of evaluated apps is 318,515, which is considered by An-

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
A3E [57]	✓		✓	25
A5 [59]	✓		✓	1260
Adagio [65]	✓		✓	147950
Amandroid [68]	✓	✓	✓	853
Anadroid [69]	✓			0
Androguard2 [74]	✓	✓		0
Androguard [76]	✓	✓		0
android-app-analysis-tool [78]	✓		✓	265
ApkCombiner [141]		✓	✓	3000
Apparecium [83]	✓		✓	100
AsyncDroid [143]			✓	611
Asynchronizer [144]			✓	13
BlueSeal [101]	✓	✓		4039
Cassandra [53]	✓			0
Choi et al. [147]	✓	✓		0
ComDroid [117]	✓		✓	20
Covert [124]			✓	200
DEvA [130]	✓		✓	12
DidFail [36]	✓	✓		0
DroidSafe [58]	✓	✓	✓	24
Epicc [9]			✓	1200
FlowDroid [6]	✓	✓	✓	1500
FUSE [70]		✓	✓	2573
Gator2 [149]	✓		✓	20
Gator3 [150]	✓		✓	20
Gator [151]	✓		✓	20
Hopper [79]	✓		✓	10
IC3 [39]	✓		✓	460
IccTA [7]	✓	✓	✓	15000
IFT [82]			✓	72
Lotrack [153]	✓		✓	100
MalloDroid [91]	✓		✓	13500
PerfChecker [155]	✓		✓	29
Poeplau et al. [106]	✓		✓	1632
Redexer [107]	✓		✓	14
SAAF [111]	✓		✓	142100
StaDynA [158]	✓		✓	10
THRESHER [159]	✓		✓	7
Violist [160]	✓		✓	0
WeChecker [134]	✓	✓	✓	1137
Wognsen et al. [135]	✓		✓	1700

dRadar [72].

> ***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. Among those approaches, only a few have evaluated their approaches in both in-the-lab and in-the-wild experiments.*

5.4. Trends and Overlooked Challenges

Although Android has already been released in 2008, research on analysing its programs has flourished in the last years. We investigate the general trends in the research and make an overview of the challenges that are (or are not) dealt with.

5.4.1. Trend Analysis

Fig. 11 shows the distribution of publications from our set according to their year of publication. Research papers meeting our criteria 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 for both Security and Software engineering communities.

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

Listing 2: Example of an implicit flow.

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, where the number of sensitivities (cf. Figure 12a) and awareness (cf. Figure 12b) are calculated by adding all the checks (✓) appearing in Table 8 and Table 11 respectively for a given year. 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

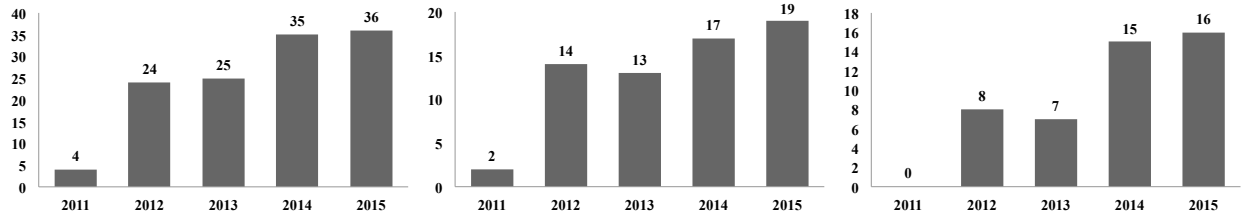
We now discuss our findings on the different challenges addressed in analyses to make them static-, implicit-flow, alias-, dynamic-code-loading-, reflection-, native-code-, and multi-threading-aware.

We consider an approach to be static-aware when it takes into account *static* object values in Java program to improve an analysis' precision. 32 approaches explicitly take this into account. 30 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 23 primary publications which take into account multi-threading. We further investigate these supports since multi-threading is well-known to be challenging even in the Java ecosystem. We note that those approaches partially solve multi-threading issues in a very simple 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.

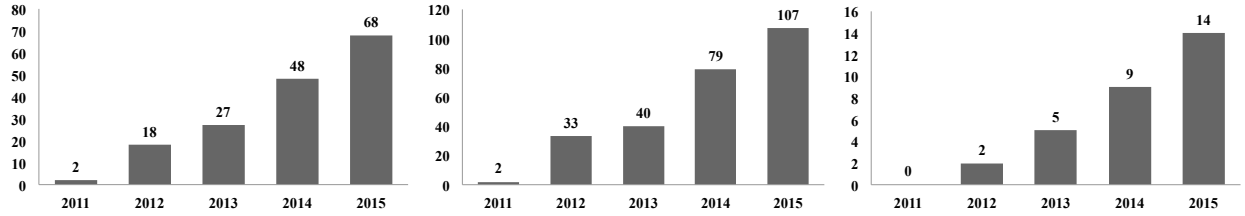
Another challenge is on considering implicit flows, i.e., flow information inferred from control-flow dependencies. Let us take Listing 2 as an example, if an Android app does not send out message 123 directly, but sends 123 times the word "count", the attacker can actually gain the same information 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 15, 3 and 3 publications respectively.



(a) Publications per year (total). (b) Publications per year (SEC). (c) Publications per year (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.

Table 11: Summary through different aspects of static analysis.

Tool	*Static-Aware*	*Implicit-Flow*	*Alias analysis*	*Dynamic Code Loading*	*Reflection*	*Native*	*Multi-Threading*
Amandroid [68]	✓		✓				
Anadroid [69]							✓
Apparecium [83]	✓						
AppAudit [85]	✓						✓
AppCaulk [86]	✓					✓	
AppContext [88]	✓	✓	✓				
Apposcopy [92]	✓		✓				
AppSealer [94]	✓						✓
AsDroid [95]							✓
Asynchronizer [144]							✓
Bastani et al. [99]			✓				
BlueSeal [101]					✓		✓
Brox [103]							✓
Capper [105]	✓						✓
Cassandra [53]		✓				✓	
ClickRelease [113]					✓		
CloneCloud [34]	✓						
ComDroid [117]			✓				
ContentScope [119]							✓
Covert [124]			✓				
CredMiner [126]							✓
CryptoLint [54]	✓						
DEvA [130]			✓				
Dflow+DroidInfer [132]			✓				
DidFail [36]	✓	✓	✓				
DPartner [136]					✓		
DroidJust [56]	✓		✓		✓		
DroidSafe [58]	✓	✓	✓		✓	✓	✓
DroidSIFT [60]					✓		✓
DroidSim [62]					✓		✓
eLens [66]							✓
Epicc [9]	✓		✓				
FlowDroid [6]	✓	✓	✓				
Total							

Tool	*Static-Aware*	*Implicit-Flow*	*Alias analysis*	*Dynamic Code Loading*	*Reflection*	*Native*	*Multi-Threading*
FUSE [70]	✓		✓		✓		✓
Gible et al. [73]		✓					
Graa et al. [75]		✓					
HelDroid [77]	✓		✓		✓		
Hopper [79]	✓		✓				
IC3 [39]	✓		✓				
IccTA [7]	✓	✓	✓				
IFT [82]		✓			✓		
Lotrack [153]	✓	✓	✓				
Mann et al. [93]	✓						
MobSafe [96]			✓				
Pathak et al. [100]							✓
Pegasus [102]				✓	✓		
PerfChecker [155]							✓
PermissionFlow [104]	✓		✓				
Poeplau et al. [106]				✓			
Relda [109]							✓
Rocha et al. [156]	✓	✓					
SAAF [111]			✓				
Scandal [114]					✓		
SIF [157]	✓						✓
SMV-Hunter [120]		✓					
StaDynA [158]				✓	✓		✓
Sufatrio et al. [121]	✓	✓	✓				
TASMAN [123]	✓	✓	✓				
THRESHER [159]	✓		✓				
TrustDroid [125]	✓						✓
Vekris et al. [128]			✓				
Violist [160]			✓				
W2AIScanner [131]	✓		✓		✓		✓
WeChecker [134]	✓		✓				
Wognsen et al. [135]	✓		✓		✓		
Woodpecker [137]							✓
Total	**32**	**14**	**30**	**3**	**15**	**3**	**23**

Others (Publications Without ✓)

A3 [55], A3E [57], A5 [59], AAPL [61], ACTEve [63], Adagio [65], AdRisk [67], AnDarwin [49], AndRadar [72], Androguard2 [74], Androguard [76], android-app-analysis-tool [78], AndroidLeaks [80], Androlizer [81], ApkCombiner [141], AppIntent [90], AQUA [142], AsyncDroid [143], AutoPPG [145], Bartel et al. [42], Bartsch et al. [97], Brahmastra [146], Chen et al. [108], Chen et al. [110], CHEX [8], Choi et al. [147], CMA [116], COPES [41], Cortesi et al. [122], DescribeMe [129], DNADroid [48], DroidAlarm [138], DroidChecker [140], EcoDroid [64], EvoDroid [148], Gallingani et al. [71], Gator2 [149], Gator3 [150], Gator [151], I-ARM-Droid [152], Jensen et al. [84], Julia [4], Lin et al. [87], Lu et al. [89], Mallo-Droid [91], MIGDroid [33], Nyx [43], ORBIT [154], PaddyFrog [98], Redexer [107], SADroid [112], SEFA [115], SIG-Droid [52], SmartDroid [118], Uranine [127], vLens [44], Wang et al. [133], Zuo et al. [139]

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. Discussions

The findings yielded by investigating the research questions of this SLR constitute many discussion points around the research and practice of Android.

6.1. Security will remain a strong focus of Android research

Static analysis of Android apps, as shown in the investigation of RQ1, is largely focused on uncovering security and privacy issues. This suggests that security and privacy are a big concern for both users and researchers nowadays. Several studies have already shown how the permission system can be misused [42] and more recently how app uninstallation can leave residual data which can be exploited in attacks [176].

On the one hand, the open source nature of Android development code base is central in the interest that it generates in the research community. While it is easy for malware writers to find exploitable security holes, it is also easy for researchers to collect data, perform experiments and test solutions on this platform.

On the other hand, time-to-market pressure is substantially higher in the mobile ecosystem than in traditional desktop computing, making testing a neglected concern by developers and users.

Unfortunately, as revealed by Goseva-Popstojanova et al. [177], which has been also confirmed by this SLR, the state-of-the-art static analysis tools are not very effective in detecting security vulnerabilities, showing that further advanced improvements to techniques and tools for static code analysis are still needed.

6.2. Considering external code is essential for a sound analysis

There are two types of external code available in Android apps outside the main *classes.dex*: Dalvik bytecode (often hidden via the extension of another file format such as xml) and binary code. Dalvik bytecode can be accessed through reflection and dynamic code loading (DCL) while binary code can be leveraged via the Java native interface (JNI) APIs. Unfortunately, as shown in Table 11, both DCL and native code are rarely considered by the state-of-the-art static analyzers. As a result, current analyses, which do not consider DCL and native code in their implementation, miss the opportunity to discover problems hidden inside external code, leading to incomplete results.

6.3. Improving basic support tools such as Soot is a priority

The majority of research works reviewed in this SLR build their analyses based on support tools such as Soot and WALA. Unfortunately, these tools present various limitations. For example, the transformation performed by Soot to translate Dalvik bytecode into Jimple and back to bytecode is still without guarantees that the rebuilt application is runnable (i.e., will not crash during execution). Several approaches and tools that instrument apps, such as AppSealer, are impacted by this limitation. It is thus essential that researchers focus on contributing in improving the static analysis and transformations supported by these support tools. It is noteworthy that a small improvement in the performance of these tools (e.g., providing more precise call graph) will benefit many more research approaches (e.g., all the approaches relying on call graph construction).

6.4. Sensitivities are key in the trade-off between precision and performance

The more sensitivities a static analysis takes into account, the more precise its results will be. However, as showcased in FlowDroid [6], this precision will come at the cost of performance: execution then becomes time consuming, and may even fail on more corner case apps. Limiting the sensitivities may yield some false positives and false negatives, but will produce an approach that could be successfully applied at the scale of markets. For instance, an imprecise but fast approach could be used to quickly filter a huge set of applications. The remaining application set, which should be much smaller than the initial set, is then analyzed using a more precise, but much slower, analysis.

6.5. Android specificities can help improve code coverage and limit over-approximations

Android specificities such as lifecycle awareness are a must to consider in order to not miss any code of the application related to individual Android components. For an ICC-aware static analysis, handling the inter-component communication precisely would increase the connectivity of the call-graph and would thus reduce the overall analysis time. Indeed, some connection between components would now be recognized as over-approximations and will no longer be taken into account by the analysis.

The analysis for Android specificities are often intertwined with the main static analysis itself. The precision of one analysis has a direct impact on the other. For instance, the lifecycle awareness requires to cover the whole call graph. Having a precise call graph, by handling many sensitivities, may increase the precision in considering component's lifecycle.

6.6. Researchers must strive to provide clean and reusable artifacts

Our SLR showed that most approaches select a subset of challenges that they address directly within their implementation, while providing little opportunity for reuse in other approaches. Furthermore, only a few have made their tools publicly available. This leads to a situation where redundant con-

tributions are made in the community without any comparison among state-of-the-art to further advance the research. Researchers should thus be strongly encouraged to make available at least the datasets used in their assessment experiments.

6.7. The booming Android ecosystem is appealing for holistic analysis

As shown in Table 10, the number of apps considered by the state-of-the-art works is much less than the total number of existing Android apps (e.g., over 2.4 million apps available on Google Play)[15]. Thus, there is a need to develop static analyzers that are capable of market-scale analysis. Furthermore, since attackers can orchestrate several apps to perform advanced attacks in order to bypass analyzers that only target single applications [178], there is also a need to perform compositional analysis among multiple Android apps. Unfortunately, inter-app analysis is not yet well investigated by the community. As shown in Table 9, less than 5% of examined approaches have taken into account inter-app analysis. More urgently, researchers should at least consider all the apps installed in a device as a whole to perform holistic analysis and contribute to minimize end-user exposure to security threats.

6.8. Combining multiple approaches could be the key towards highly precise analysis

Static analysis leads to over-approximations for multiple reasons, one being that it analyzes all code, including dead code. As a result, static analyses may generate false positives. On the contrary, dynamic analyses under-approximates, as it is challenging to cover all code, and thus tend to produce false negatives. Both approaches are thus complementary and can be combined to perform practical analyses. Typically, dynamic analysis can focus on checking if the reported results of static analysis are false positives, thus reducing the number of false alarms.

Besides combining static and dynamic analysis approaches, the community should consider also combining several static approaches to conduct highly precise analysis. Recently, TASMAN [123] has proposed to perform targeted symbolic execution to remove false data leaks reported by FlowDroid, where FlowDroid leverages static taint analysis to detect data leaks. TASMAN's promising results are encouraging for the research direction on using multiple static approaches to ensure a minimum false positives in analysis approaches.

7. 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 engines (e.g., the one provided by Springer) are not so accurate. Besides, we have only checked 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[16] papers have been taken into account. To further mitigate this, we have also performed a backward-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 between different authors to perform the detailed examination. As suggested by Brereton et al. [179], 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. [180], 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 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 if it is rare, it is still possible that we may miss publications from those venues.

8. 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 survey that specifically focuses 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 publications 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.

Sufatrio et al. [10] present 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. The survey shows in particular, that most research solutions addressing security issues in Android are leveraging static analysis. This relates to the finding in our SLR that most static analysis works for Android target security concerns. In their discussion of static analysis for Android,

[15] https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

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

they also enumerate the different challenges of Android programming (e.g., multiple entry points, GUI, call-backs in event-based system, etc.) that analyzers must account for. In our SLR, we detail those challenges and further clarify which works deal with which challenge.

Faruki et al. [181] present another survey mainly focusing on a set of known Android malware detection approaches/tools, e.g., the growth of malware, the existence of anti-analysis techniques (i.e., being able to detect and thus evade analyzing systems [182]) . This survey highlights the need to take into account the existence of anti-analysis techniques (i.e., techniques that allow a malicious sample to detect and evade analysing systems [182]). For example, they discuss how traditional signature-based and static analysis-based approaches can be vulnerable to stealthy techniques such as encryption. Eventually, it offers an overall overview for the directions that must be taken to tackle the remaining issues in statically analyzing Android apps.

Rashidi et al. [183] present another survey on existing Android security threats and security enforcement solutions. This survey classifies Android security mechanisms into four dimensions: Information Leaks, Vulnerabilities (Privilege Escalation and Colluding), Denial of Service (DoS) attacks, and App Clones. Our SLR finds that security papers using static analysis indeed target issues mainly in three of the dimensions they consider. We did not find any work that statically detects or hints on DoS attacks, which is reasonable. In contrast with their approach, the systematic nature of the SLR, allowed us to find papers in another security-related dimension, namely Cryptography misuses.

Haris et al. [184] present a survey 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, [185] and [186] present state-of-the-art reviews with considering multiple mobile platforms.

More recently, Martin et al. [187] produced a technical report that reviews the state-of-the-art works on app store analysis in the software engineering field. In their survey, they have reported both non-technical and technical information to learn behaviors and trends of software repositories. More specifically, they have reviewed the literature works in 7 dimensions: API Analysis, Feature Analysis, Review Analysis, Security, Store Ecosystem, Size and Effort Prediction, and Others, including 127 non-technical and 60 technical papers. In their findings, they report that security is a pervasive concern in reviewed papers. This finding is inline with one of our SLR finding stating that static analysis is mainly used for the purpose of assessing app security. The whole focus of their work is however different from ours.

Sadeghi et al. [188] also present a technical report that studies the taxonomy and qualitative comparison of program analysis techniques, with a special focus on Android security. They have examined 100 research papers, including both static analysis and dynamic analysis approaches. Comparing to this SLR, we focus on static analysis of Android apps only. Among the findings, Sadeghi et al. show that the most used intermediate representation (IR) for their examined approaches are Jimple, accounting for 29%, which is in line with our findings. Besides, they also show that it is difficult to perform replication study on security-based researches. Indeed, as what we have shown in this SLR, most research tools and their evaluated artifacts are not publicly available.

All in all, our SLR differs from all the aforementioned surveys in a way that we exclusively focus on static analysis of Android apps. We believe those surveys can compliment ours and thus to provide a better view on the landscape of Android-based research.

9. 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-of-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 124 research papers published in Software engineering, programming languages and security conference and journal venues.

Our review has consisted in 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 artifacts, such as tools and datasets, are often unpublished.

10. Acknowledgment

The authors would like to thank the anonymous reviewers for their helpful comments and suggestions, as well as all the authors of static Android 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 projects AndroMap C13/IS/5921289 and Recommend C15/IS/10449467.

Appendix A. The Full List of Examined Publications

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] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Auto-completing bug reports for android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 673–686. ACM, 2015.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] Stephen Fink and Julian Dolby. Wala–the tj watson libraries for analysis, 2012.

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

[15] 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.

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

[17] 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.

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

[19] 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.

[20] Chieh-Jan Mike Liang, Nicholas D Lane, Niels Brouwers, Li Zhang, Börje F Karlsson, Hao Liu, Yan Liu, Jun Tang, Xiang Shan, Ranveer Chandra, et al. Caiipa: automated large-scale mobile app testing through contextual fuzzing. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 519–530. ACM, 2014.

[21] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. Systematic execution of android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 83–93. ACM, 2015.

[22] Razieh Nokhbeh Zaeem, Mukul R Prasad, and Sarfraz Khurshid. Automated generation of oracles for testing user-interaction features of mobile apps. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 183–192. IEEE, 2014.

[23] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. Automatically discovering, reporting and reproducing android application crashes. *ICST'16*, 2016.

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

[25] Phu H Nguyen, Max Kramer, Jacques Klein, and Yves Le Traon. An extensive systematic review on the model-driven development of secure systems. *Information and Software Technology*, 68:62–81, 2015.

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

[27] 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.

[28] 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.

[29] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. Scandroid: Automated security certification of android. 2009.

[30] 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. *arXiv preprint arXiv:1404.7431*, 2014.

[31] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: A scalable system for detecting code reuse among android applications. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 62–81. Springer, 2012.

[32] B. Kitchenham and S Charters. Guidelines for performing systematic literature reviews in software engineering, 2007.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy api usage patterns in android apps: an empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 2–11. ACM, 2014.

[46] Israel J Mojica Ruiz, Meiyappan Nagappan, Bram Adams, and Ahmed E Hassan. Understanding reuse in the android market. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 113–122. IEEE, 2012.

[47] Israel J Mojica, Bram Adams, Meiyappan Nagappan, Steffen Dienst, Thorsten Berger, and Ahmed E Hassan. A large-scale empirical study on software reuse in mobile apps. *IEEE software*, 31(2):78–86, 2014.

[48] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: Detecting cloned applications on android markets. In *Computer Security–ESORICS 2012*, pages 37–54. Springer, 2012.

[49] Jonathan Crussell, Clint Gibler, and Hao Chen. Andarwin: Scalable detection of semantically similar android applications. In *Computer Security–ESORICS 2013*, pages 182–199. Springer, 2013.

[50] Mario Linares-Vásquez, Andrew Holtzhauer, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Revisiting android reuse studies in the context of code obfuscation and library usages. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 242–251. ACM, 2014.

[51] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *The 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*, 2016.

[52] Nariman Mirzaei, Hamid Bagheri, Riyadh Mahmood, and Sam Malek. Sig-droid: Automated system input generation for android applications. In *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pages 461–471. IEEE, 2015.

[53] 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.

[54] 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.

[55] 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.

[56] Xin Chen and Sencun Zhu. Droidjust: automated functionality-aware privacy leakage analysis for android applications. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, page 5. ACM, 2015.

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

[58] 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.

[59] 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.

[60] 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.

[61] Kangjie Lu, Zhichun Li, Vasileios P Kemerlis, Zhenyu Wu, Long Lu, Cong Zheng, Zhiyun Qian, Wenke Lee, and Guofei Jiang. Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In *NDSS*, 2015.

[62] Xin Sun, Yibing Zhongyang, Zhi Xin, Bing Mao, and Li Xie. Detecting code reuse in android applications using component-based control flow graph. In *ICT Systems Security and Privacy Protection*, pages 142–155. Springer, 2014.

[63] 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.

[64] Reyhaneh Jabbarvand, Alireza Sadeghi, Joshua Garcia, Sam Malek, and Paul Ammann. Ecodroid: an approach for energy-based ranking of android apps. In *Proceedings of the Fourth International Workshop on Green and Sustainable Software*, pages 8–14. IEEE Press, 2015.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] Daniele Gallingani, Rigel Gjomemo, VN Venkatakrishnan, and Stefano Zanero. Static detection and automatic exploitation of intent message vulnerabilities in android applications. 2015.

[72] Martina Lindorfer, Stamatis Volanis, Alessandro Sisto, Matthias Neugschwandtner, Elias Athanasopoulos, Federico Maggi, Christian Platzer, Stefano Zanero, and Sotiris Ioannidis. Andradar: fast discovery of android applications in alternative markets. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 51–71. Springer, 2014.

[73] 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.

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

[75] 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.

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

[77] Nicoló Andronio, Stefano Zanero, and Federico Maggi. Heldroid: Dissecting and detecting mobile ransomware. In *Research in Attacks, Intrusions, and Defenses*, pages 382–404. Springer, 2015.

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

[79] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Selective control-flow abstraction via jumping. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 163–182. ACM, 2015.

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

[81] 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.

[82] Michael D Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Xu. 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.

[83] 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.

[84] Casper S Jensen, Mukul R Prasad, and Anders Møller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 67–77. ACM, 2013.

[85] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. Effective real-time android application auditing. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 899–914. IEEE, 2015.

[86] 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.

[87] 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.

[88] 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.

[89] 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.

[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] 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.

[92] 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.

[93] 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.

[94] 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.

[95] 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.

[96] 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.

[97] 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.

[98] 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.

[99] Osbert Bastani, Saswat Anand, and Alex Aiken. Interactively verifying absence of explicit information flows in android apps. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 299–315. ACM, 2015.

[100] Abhinav Pathak, Abhilash Jindal, Y Charlie Hu, and Samuel P Midkiff. What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 267–280. ACM, 2012.

[101] 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.

[102] 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.

[103] 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.

[104] 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.

[105] 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.

[106] 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.

[107] 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.

[108] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 175–186. ACM, 2014.

[109] 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.

[110] 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.

[111] 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.

[112] 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.

[113] Kristopher Micinski, Jonathan Fetter-Degges, Jinseong Jeon, Jeffrey S Foster, and Michael R Clarkson. Checking interaction-based declassification policies for android using symbolic execution. In *Computer Security–ESORICS 2015*, pages 520–538. Springer, 2015.

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

[115] 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.

[116] 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.

[117] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252. ACM, 2011.

[118] 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.

[119] 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.

[120] 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.

[121] Sufatrio, Tong-Wei Chua, Darell JJ Tan, and Vrizlynn LL Thing. Accurate specification for robust detection of malicious behavior in mobile environments. In *Computer Security–ESORICS 2015*, pages 355–375. Springer, 2015.

[122] Agostino Cortesi, Pietro Ferrara, Marco Pistoia, and Omer Tripp. Datacentric semantics for verification of privacy policy compliance by mobile applications. In *Verification, Model Checking, and Abstract Interpretation*, pages 61–79. Springer, 2015.

[123] Steven Arzt, Siegfried Rasthofer, Robert Hahn, and Eric Bodden. Using targeted symbolic execution for reducing false-positives in dataflow analysis. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 1–6. ACM, 2015.

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

[125] 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.

[126] Yajin Zhou, Lei Wu, Zhi Wang, and Xuxian Jiang. Harvesting developer credentials in android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, page 23. ACM, 2015.

[127] Vaibhav Rastogi, Zhengyang Qu, Jedidiah McClurg, Yinzhi Cao, and Yan Chen. Uranine: Real-time privacy leakage monitoring without system modification for android. In *Security and Privacy in Communication Networks*, pages 256–276. Springer, 2015.

[128] 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.

[129] Mu Zhang, Yue Duan, Qian Feng, and Heng Yin. Towards automatic generation of security-centric descriptions for android apps. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 518–529. ACM, 2015.

[130] Gholamreza Safi, Arman Shahbazian, WG Halfond, and Nenad Medvidovic. Detecting event anomalies in event-based systems. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2015.

[131] Behnaz Hassanshahi, Yaoqi Jia, Roland HC Yap, Prateek Saxena, and Zhenkai Liang. Web-to-application injection attacks on android: Characterization and detection. In *Computer Security–ESORICS 2015*, pages 577–598. Springer, 2015.

[132] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. Scalable and precise taint analysis for android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 106–117. ACM, 2015.

[133] Jingtian Wang, Guoquan Wu, Xiaoquan Wu, and Jun Wei. 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.

[134] Xingmin Cui, Jingxuan Wang, Lucas CK Hui, Zhongwei Xie, Tian Zeng, and SM Yiu. Wechecker: efficient and precise detection of privilege escalation vulnerabilities in android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, page 25. ACM, 2015.

[135] 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.

[136] 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.

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

[138] 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.

[139] Chaoshun Zuo, Jianliang Wu, and Shanqing Guo. Automatically detecting ssl error-handling vulnerabilities in hybrid mobile web apps. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 591–596. ACM, 2015.

[140] 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.

[141] 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*.

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

[143] Yu Lin, Semih Okur, and Danny Dig. Study and refactoring of android asynchronous programming. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 224–235. IEEE, 2015.

[144] 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.

[145] Le Yu, Tao Zhang, Xiapu Luo, and Lei Xue. Autoppg: Towards automatic generation of privacy policy for android applications. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 39–50. ACM, 2015.

[146] 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.

[147] 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.

[148] 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.

[149] 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.

[150] Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. Static window transition graphs for android. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 658–668. IEEE, 2015.

[151] 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.

[152] Benjamin Davis, Ben Sanders, Armen Khodaverdian, and Hao Chen. I-arm-droid: A rewriting framework for in-app reference monitors for android applications. *Mobile Security Technologies*, 2012, 2012.

[153] 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.

[154] 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.

[155] 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.

[156] 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.

[157] 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.

[158] 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.

[159] 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.

[160] Ding Li, Yingjun Lyu, Mian Wan, and William GJ Halfond. String analysis for java and android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 661–672. ACM, 2015.

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

[162] 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.

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

[164] 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.

[165] 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.

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

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

[168] 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.

[169] Li Li, Tegawend F. Bissyand, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole program analysis of android apps. In *Proceedings of the 2016 International Symposium on Software Testing and Analysis*, ISSTA, 2016.

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

[171] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985.

[172] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and software verification: model-checking techniques and tools*. Springer Science & Business Media, 2013.

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

[174] V Benjamin Livshits and Monica S Lam. Tracking pointers with path and context sensitivity for bug detection in c programs. *ACM SIGSOFT Software Engineering Notes*, 28(5):317–326, 2003.

[175] V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in java applications with static analysis. In *Usenix Security*, 2005.

[176] Xiao Zhang, Kailiang Ying, Yousra Aafer, Zhenshen Qiu, and Wenliang Du. Life after app uninstallation: Are the data still alive? data residue attacks on android. NDSS '16.

[177] Katerina Goseva-Popstojanova and Andrei Perhinschi. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*, 68:18–33, 2015.

[178] Karim O Elish, Danfeng Yao, and Barbara G Ryder. On the need of precise inter-app icc classification for detecting android malware collusions. In *Most@S&P*, 2015.

[179] 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.

[180] 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.

[181] 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.

[182] Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 447–458. ACM, 2014.

[183] 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.

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

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

[186] 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.

[187] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. A survey of app store analysis for software engineering. *RN*, 16:02, 2016.

[188] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, and Sam Malek. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android apps. 2016.

Table A.12: The Full List of Examined Publications (Part I).

Year	VenueType	VenueName	Title
2015	ACM	SPSM (W)	AutoPPG: Towards Automatic Generation of Privacy Policy for Android Applications [145]
2015	Other	NDSS	Checking More and Alerting Less: Detecting Privacy Leakages via Enhanced Data-flow Analysis and Peer Voting [61]
2015	IEEE	AINA	Apparecium: Revealing Data Flows in Android Applications [83]
2015	IEEE/ACM	ICSE	IccTA : Detecting Inter-Component Privacy Leaks in Android Apps [7]
2015	Springer	IFIP SEC	ApkCombiner : Combining Multiple Android Apps to Support Inter-App Analysis [141]
2015	ACM	WiSec	DroidJust: automated functionality-aware privacy leakage analysis for Android applications [56]
2015	WoK	SCN (J)	PaddyFrog: systematically detecting confused deputy vulnerability in Android applications [98]
2015	ACM	CODASPY	StaDynA : Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications [158]
2015	Springer	ESORICS	Accurate Specification for Robust Detection of Malicious Behavior in Mobile Environments [121]
2015	ACM	FSE	String Analysis for Java and Android Applications [160]
2015	IEEE/ACM	ICSE	Static Control-Flow Analysis of User-Driven Callbacks in Android Applications [149]
2015	Springer	RAID	HelDroid: Dissecting and Detecting Mobile Ransomware [77]
2015	IEEE/ACM	ASE	Study and Refactoring of Android Asynchronous Programming [143]
2015	ACM	ISSTA	Scalable and precise taint analysis for Android [132]
2015	IEEE	S&P	Effective Real-Time Android Application Auditing [85]
2015	Other	NDSS	Information-Flow Analysis of Android Applications in DroidSafe [58]
2015	ACM	AsiaCCS	Automatically Detecting SSL Error-Handling Vulnerabilities in Hybrid Mobile Web Apps [139]
2015	IEEE/ACM	ICSE	AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context [88]
2015	Springer	ESORICS	Web-to-Application Injection Attacks on Android: Characterization and Detection [131]
2015	ACM	WiSec	WeChecker: efficient and precise detection of privilege escalation vulnerabilities in Android apps [134]
2015	ACM	WiSec	Harvesting developer credentials in Android apps [126]
2015	IEEE	GREENS (W)	EcoDroid: an approach for energy-based ranking of Android apps [64]
2015	Springer	SecureComm	Uranine: Real-time Privacy Leakage Monitoring without System Modification for Android [127]
2015	IEEE/ACM	ICSE	Composite Constant Propagation: Application to Android Inter-Component Communication Analysis [39]
2015	ACM	CCS	Towards Automatic Generation of Security-Centric Descriptions for Android Apps [129]
2015	Springer	VMCAI	Datacentric Semantics for Verification of Privacy Policy Compliance by Mobile Applications [122]
2015	IEEE	ISSRE	SIG-Droid: Automated system input generation for Android applications [52]
2015	ACM	OOPSLA	Selective Control-Flow Abstraction via Jumping [79]
2015	Springer	ESORICS	Checking Interaction-Based Declassification Policies for Android Using Symbolic Execution [113]
2015	IEEE/ACM	ASE	Static Window Transition Graphs for Android [150]
2015	ACM	FSE	Detecting Event Anomalies in Event-Based Systems [130]
2015	ACM	SOAP (W)	Using targeted symbolic execution for reducing false-positives in dataflow analysis [123]
2015	ACM	OOPSLA	Interactively verifying absence of explicit information flows in Android apps [99]
2015	IEEE	MoST (W)	Static Detection and Automatic Exploitation of Intent Message Vulnerabilities in Android Applications [71]
2015	IEEE	TSE	COVERT: Compositional Analysis of Android Inter-App Permission Leakage [124]
2015	Elsevier	CompSec (J)	A Permission verification approach for android mobile applications [78]
2014	Other	NDSS	AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications [94]
2014	USENIX	SOUPS	Modeling Users' Mobile App Privacy Preferences : Restoring Usability in a Sea of Permission Settings [87]
2014	IEEE	TSA	Detecting Mobile Application Malicious Behaviors Based on Data Flow of Source Code [110]
2014	ACM	FSE	Apposcopy : Semantics-Based Detection of Android Malware Through Static Analysis [92]
2014	ACM	SPSM (W)	A5 : Automated Analysis of Adversarial Android Applications [59]
2014	Elsevier	IPL (J)	A type and effect system for activation flow of components in Android programs [147]
2014	ACM	FSE	EvoDroid: segmented evolutionary testing of Android apps [148]
2014	USENIX	Security	Brahmastra: Driving Apps to Test the Security of Third-Party Components [146]
2014	ACM	AsiaCCS	Efficient, Context-Aware Privacy Leakage Confinement for Android Applications without Firmware Modding [105]
2014	ACM	SOAP (W)	Android taint flow analysis for app sets [36]
2014	Springer	DIMVA	AndRadar: Fast Discovery of Android Applications in Alternative Markets [72]
2014	ACM	CCS	Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs [60]
2014	IEEE	TSE (J)	Static Analysis for Extracting Permission Checks of a Large Scale Framework: The Challenges And Solutions for Analyzing Android [42]
2014	IEEE/ACM	ASE	Information flows as a permission mechanism [101]
2014	Other	NDSS	Execute this! analyzing unsafe and malicious dynamic code loading in android applications [106]
2014	IEEE/ACM	ICSE	Achieving accuracy and scalability simultaneously in detecting application clones on Android markets [108]
2014	IEEE	DASC	Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications [116]
2014	Elsevier	SCP (J)	Formalisation and analysis of Dalvik bytecode [135]
2014	ACM	SPSM (W)	Cassandra: Towards a Certifying App Store for Android [53]
2014	ACM	CCS	Amandroid : A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps [68]
2014	IEEE/ACM	ICSE	Making web applications more energy efficient for OLED smartphones [43]
2014	IEEE/ACM	ICSE	Characterizing and detecting performance bugs for smartphone applications [155]
2014	IEEE	ICCCN	MIGDroid: Detecting APP-Repackaging Android malware via method invocation graph [33]
2014	ACM	CCS	Collaborative Verification of Information Flow for a High-Assurance App Store [82]
2014	IEEE	TrustCom	Systematic Analysis and Detection of Misconfiguration Vulnerabilities in Android Smartphones [112]
2014	IEEE/ACM	ICSE	AsDroid : Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction [95]

(W) stands for workshp; (J) stands for journal; WoK stands for Web of Knowledge.

Table A.13: The Full List of Examined Publications (Part II).

Year	VenueType	VenueName	Title
2014	Other	NDSS	SMV-HUNTER: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps [120]
2014	ACM	PPREW (W)	Multi-App Security Analysis with FUSE : Statically Detecting Android App Collusion [70]
2014	IEEE	SERE	Protection against Code Obfuscation Attacks based on control dependencies in Android Systems [73]
2014	Springer	IFIP SEC	Detecting Code Reuse in Android Applications Using Component-Based Control Flow Graph [62]
2014	IEEE	TrustCom	AppCaulk: Data Leak Prevention by Injecting Targeted Taint Tracking Into Android Apps [86]
2014	ACM	CGO	Static Reference Analysis for GUI Objects in Android Software [151]
2014	ACM	FSE	Retrofitting concurrency for Android applications through refactoring [144]
2014	ACM	PLDI	FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps [6]
2014	IEEE/ACM	ASE	Tracking Load-time Configuration Options [153]
2013	IEEE	GLOBECOM	Detecting GPS information leakage in Android applications [103]
2013	ACM	SPSM (W)	Sound and precise malware analysis for android via pushdown reachability and entry-point saturation [69]
2013	Other	NDSS	Contextual Policy Enforcement in Android Applications with Permission Event Graphs [102]
2013	ACM	AISec (W)	Structural Detection of Android Malware using Embedded Call Graphs [65]
2013	ACM	ISSTA	Calculating source line level energy information for Android applications [44]
2013	IEEE	TIFS (J)	Hybrid static-runtime information flow and declassification enforcement [156]
2013	ACM	AsiaCCS	DroidAlarm: an all-sided static analysis tool for android privilege-escalation malware [138]
2013	ACM	SAC	Slicing droids: program slicing for smali code [111]
2013	IEEE	ARES	The Transitivity-of-Trust Problem in Android Application Interaction [97]
2013	WoK	IBM R&D (J)	Automatic Detection of Inter-application Permission Leaks in Android Applications [104]
2013	ACM	ISSTA	automated testing with targeted event sequence generation [84]
2013	ACM	MobiSys	SIF: A Selective Instrumentation Framework for Mobile Applications [157]
2013	IEEE/ACM	ASE	Characterizing and detecting resource leaks in Android applications [109]
2013	USENIX	Security	Effective Inter-Component Communication Mapping in Android with Epicc : An Essential Step Towards Holistic Security Analysis [9]
2013	Springer	ESORICS	AnDarwin: Scalable Detection of Semantically Similar Android Applications [49]
2013	Other	CCIS (W)	A3: Automatic Analysis of Android Malware [55]
2013	WoK	Tsinghua S&T (J)	MobSafe: cloud computing based forensic analysis for massive mobile applications using data mining [96]
2013	ACM	PLDI	Thresher: precise refutations for heap reachability [159]
2013	ACM	CCS	The impact of vendor customizations on android security [115]
2013	Springer	FASE	A grey-box approach for automated GUI-model generation of mobile applications [154]
2013	IEEE/ACM	ICSE	Estimating mobile application energy consumption using program analysis [66]
2013	ACM	CCS	An empirical study of cryptographic misuse in android applications [54]
2013	ACM	CCS	AppIntent : Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection [90]
2013	Other	NDSS	Detecting passive content leaks and pollution in android applications [119]
2013	ACM	OOPSLA	Targeted and depth-first exploration for systematic testing of android apps [57]
2012	ACM	WiSec	DroidChecker : Analyzing Android Applications for Capability Leak [140]
2012	Other	NDSS	Systematic Detection of Capability Leaks in Stock Android Smartphones. [137]
2012	IEEE	WCRE	AQUA: Android QUery Analyzer [142]
2012	IEEE	MoST (W)	Scandal: Static Analyzer for Detecting Privacy Leaks in Android Applications [114]
2012	IEEE	COMPSAC	Model-based static source code analysis of java programs with applications to android security [89]
2012	ACM	CCS	Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security [91]
2012	ACM	SAC	A framework for static detection of privacy leaks in android applications [93]
2012	ACM	SPSM (W)	Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications [118]
2012	IEEE	HICSS	Android: Static analysis using similarity distance [76]
2012	ACM	Internetware	Detect and optimize the energy consumption of mobile app through static analysis: an initial research [133]
2012	USENIX	HotPower	Towards verifying android apps for the absence of no-sleep energy bugs [128]
2012	Springer	TRUST	AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale [80]
2012	ACM	OOPSLA	Refactoring android Java code for on-demand computation offloading [136]
2012	ACM	WiSec	Unsafe exposure analysis of mobile in-app advertisements [67]
2012	ACM	MobiSys	what is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone apps [100]
2012	ACM	FSE	Automated Concolic Testing of Smartphone Apps [63]
2012	Springer	ESORICS	Attack of the clones: detecting cloned applications on Android markets [48]
2012	ACM	CCS	CHEX : Statically Vetting Android Apps for Component Hijacking Vulnerabilities [8]
2012	IEEE	MoST (W)	I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications [152]
2012	Springer	CSS	Detecting control flow in smarphones: Combining static and dynamic analyses [75]
2012	IEEE/ACM	ASE	Automatically securing permission-based software by reducing the attack surface: An application to android [41]
2012	IEEE	MALWARE	TrustDroid: Preventing the use of SmartPhones for information leaking in corporate networks through the used of static analysis taint tracking [125]
2012	ACM	SPSM (W)	Dr . Android and Mr . Hide : Fine-grained Permissions in Android Applications [107]
2012	Elsevier	IST (J)	Static analysis of Android programs [4]
2011	ACM	EuroSys	Clonecloud: elastic execution between mobile device and cloud [34]
2011	IEEE	MALWARE	Using Static Analysis for Automatic Assessment and Mitigation of Unwanted and Malicious Activities Within Android Applications [81]
2011	ACM	MobiSys	Analyzing Inter-Application Communication in Android [117]
2011	Other	BlackHat	Android : From Reversing to Decompilation [74]

(W) stands for workshp; (J) stands for journal; WoK stands for Web of Knowledge.