Front. Comput. Sci., 2019, 13(6): 1210–1227
https://doi.org/10.1007/s11704-018-7008-1

REVIEW ARTICLE

Analyses for specific defects in Android applications: a survey

Tianyong WU^{1,2}, Xi DENG (✉)^{1,2,3}, Jun YAN (✉)^{1,2,3}, Jian ZHANG^{1,2}

1 State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
2 University of Chinese Academy of Sciences, Beijing 100049, China
3 Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China

Abstract Android applications (APPS) are in widespread use and have enriched our life. To ensure the quality and security of the apps, many approaches have been proposed in recent years for detecting bugs and defects in the apps, of which program analysis is a major one. This paper mainly makes an investigation of existing works on the analysis of Android apps. We summarize the purposes and proposed techniques of existing approaches, and make a taxonomy of these works, based on which we point out the trends and challenges of research in this field. From our survey, we sum up four main findings: (1) program analysis in Android security field has gained particular attention in the past years, the fields of functionality and performance should also gain proper attention; the infrastructure that supports detection of various defects should be enriched to meet the industry's need; (2) many kinds of defects result from developers' misunderstanding or misuse of the characteristics and mechanisms in Android system, thus the works that can systematically collect and formalize Android recommendations are in demand; (3) various program analysis approaches with techniques in other fields are applied in analyzing Android apps; however, they can be improved with more precise techniques to be more applicable; (4) The fragmentation and evolution of Android system blocks the usability of existing tools, which should be taken into consideration when developing new approaches.

Keywords Android apps, program analysis, security, functionality, performance

Received January 9, 2017; accepted May 7, 2018

E-mail: dengxi@ios.ac.cn; yanjun@ios.ac.cn

1 Introduction

In recent years, Android has become the most popular OS for the mobile devices (e.g., smartphone, tablets). At the same time, Android markets are also expanding rapidly due to the continuous development of Android apps. It is estimated that more than two million Android apps are available in Google Play (see Statistic website). Such large-scale Android apps enrich our life, but also bring new challenges and risks, e.g., the energy consumption problem and rampant malicious apps.

Although Android has been proposed for only a few years, there have been a large number of researchers focusing on various issues about Android apps. Their different objectives, such as detecting the potential bugs in the apps, improving the performance of the apps, and protecting the privacy data in the smartphone from the malicious apps, drive them to propose numerous approaches and techniques and implement various prototypes.

This paper aims to explore and survey the existing works of program analysis for Android apps. To this end, the published articles from 2010 to 2016 are collected and reviewed, which are mainly chosen from the well-known conferences/journals about software engineering and security. Papers that do not aim at detecting or fixing specific defects in Android apps, or that propose abstract methodology without prototype tools, are filtered out. Then we summarize the purposes and key techniques of the rest of works. Finally, we form a taxonomy to systematize these works and figure out the contributions of relevant works with similar topics or using similar techniques. In addition, we also conclude several findings about the trends and challenges in the field of the analysis of Android apps.

The main contributions of this paper are as follows.

- We collect thousands of published papers and select 77 from them to form a taxonomy of existing research works on the analysis of Android apps;
- We point out the unique contribution of each work among the related ones through comparison;
- We summarize the trends of this research field and provide insights for potential research opportunities.

The rest of this survey is organized as follows. In Section 2, we will give a brief introduction to the characteristics of Android and the program analysis techniques. In Section 3, we introduce the methodology of this survey. In Section 4, we present our taxonomy of existing researches on Android apps. Sections 5–7 survey the prior works, which are classified into three categories of the taxonomy. Section 8 shows the statistic information and findings summarized from the existing works. Section 9 introduces some related works and in the last section, we give the conclusion and discuss promising directions for future research.

2 Background

Before presenting our taxonomy, we provide a brief introduction to the Android system and program analysis techniques. These fundamental information will help the readers to understand the challenges and risks brought by the new features of Android system, which are the root of the problems addressed by the newly proposed program analysis approaches and techniques for Android apps.

2.1 Android characteristics

Android is an open-source operating system based on Linux kernel for portable devices. Android apps are GUI programs that are written in Java and compiled into Dalvik byte-code, and run on the Dalvik virtual machine (DVM). Besides the Java code, an app may contain some native libraries (binary library), which are provided by Android system or implemented by developers. The Android app can invoke the native libraries through java native interface (JNI). There are several new features in Android system, devices, apps and ecosystems that we list as follows.

Intermediate representation An Android app is compiled into Dalvik byte-code (instead of Java byte-code) and packed in a Dalvik executable (DEX) file to be deployed. Similar to Java byte-code, Dalvik byte-code is also a platform-independent language and can be easily handled by reverse engineers. Hence, the Android apps suffer from app repackaging for malicious intents, such as injecting ads, malware, and pirating. In addition, the new intermediate representation makes the existing tools or techniques for Java programs fail to analyze Android apps directly.

Event-driven mechanism Android apps are driven by events and callbacks to perform the interaction with users. Besides the callbacks, Android also offers a message mechanism for the communication between the components in the apps. The inter-component communication (ICC) is performed by using *Intent* that is a class defined in Android framework. Although the callbacks and message mechanism make the development more flexible, they also bring about problems and challenges for program analysis, since the calling of these callbacks can not be obtained from the code of the app, and an app often has complex execution paths due to the large number of potential combinations of the callbacks.

Resources and performance Android devices often possess relatively limited resources (e.g., CPU, memory) compared with PCs, while the smartphones are frequently used by many people every day. Except for the traditional resources, Android provides a number of APIs for the developers to access exotic resources (such as GPS, camera and different kinds of sensors) that enrich the apps' functionalities and user experience. However, the inappropriate usage of these exotic resources may also decrease the performance of the app. In addition, battery energy is one of the most important things the users care about.

Privacy and security Android devices contain lots of private data, such as device ID, contacts, messages, and bank account information. These sensitive data have become the target of attackers in recent years [1]. In addition, the centralized way (app store) of the deployment also increases the security risks of the apps. Android system designs the permission mechanism to limit app's access of the sensitive operations, however, it is not always an easy task to properly use this mechanism.

Continuous upgrading of APIs and fragmentation The Android APIs are often upgraded for adding new functionalities or fixing bugs. However, McDonnell et al. [2] point out that the time for the developers to eventually use newer API versions is much longer than the average API release interval. In addition, it is a trend that many manufacturers reshape the Android system when developing their own Android devices.

Furthermore, there are various Android devices with vastly different shapes, performance levels and screen sizes. In this situation, the application behaviors may vary in different devices, and it is called "fragmentation".

Although Android reference gives several criteria to guide developers in implementing high-quality apps (see "Core app quality") from functionality aspect to performance and stability aspects, there are many challenges for developers to implement a defect-free Android app, such as the mutability of APIs, the complexity of event sequences in an app, the fragmentation of the devices and the different versions of frameworks. They all make the development process error-prone. On the other hand, the popularity of Android system and the centralized release of Android apps make them the target of attackers; more and more malware are deployed in the app markets and lead to economic loss. A lot of researchers proposed many approaches based on program analysis techniques to enhance the quality and security of Android apps, which we aim to survey in this paper.

2.2 Program analysis

Program analysis is a commonly used technique to ensure the quality of software systems, which automatically analyzes the behaviors of the given software regarding some properties (e.g., correctness, safety and liveness). According to whether the analyzed program is executed, the program analysis technique can be categorized into two kinds, i.e., static analysis and dynamic analysis.

Static analysis Static analysis scans the program code without executing it, and performs a comprehensive analysis regarding the given properties to figure out potential bugs and defects. Two commonly used techniques arecontrol flow analysis and data flow analysis. The former one converts the program into a control flow graph (CFG) where each node represents a basic block (a sequence of statements) and each edge indicates the possible transition between nodes. Then the static analysis approach traverses the program paths based on CFG and checks whether any paths violate the given properties. The latter one focuses on gathering information about the possible values of the variables at some specific program points. For instance, a typical data flow analysis problem, reaching definition, aims to determine which definitions for the values of some variables may reach a given point. A number of methods based on these two techniques have been proposed for statically analyzing the programs, such as symbolic execution [3], abstract interpretation [4], static taint analysis [5], and static program slicing [6]. Table 1 gives a brief introduction of these techniques.

Table 1 Techniques for static analysis

Techniques	Description
Symbolic execution	Assume symbolic values for inputs for determining what inputs drive the program to execute along some specific paths
Abstract interpretation	A theory to approximate the semantics of the programs and simplify the analysis on partial program execution
Static taint analysis	Track the propagation of some specific (sensitive) data and check whether it can reach some program point by statically analyzing the code
Static program slicing	Extract all the program statements that may affect the values of some variables of interest by statically analyzing the code

Dynamic analysis In contrast, dynamic analysis is performed by executing the original or instrumented program under test cases as well as recording the execution traces and outputs during runtime. Through further analysis of the recorded information, it checks whether the program satisfies or violates the given specification and properties. To obtain various runtime information, the dynamic analysis usually leverages the code instrumentation technique [7] to insert probes (extra statements or instructions) for monitoring the program execution. The inserted probes depend on the kinds of the problems to be solved by the dynamic analysis. An essential issue of the dynamic analysis is the overhead introduced by the inserted probes, and some approaches have been proposed to address this issue, like path profiling [8], dynamic taint analysis [7] and dynamic program slicing [9]. Table 2 gives a brief introduction of these techniques. Another factor affecting the effects of the dynamic analysis is the quality of test cases, which also attracts researchers' attention [3,10,11]. Besides, the Android app runs on the DVM so that the code instrumentation can be performed on either the app or the DVM.

Table 2 Techniques for dynamic analysis

Techniques	Description
Path profiling	Insert minimal probes to record the executed program paths during runtime
Dynamic taint analysis	Track the propagation of some specific (sensitive) data by code instrumentation
Dynamic program slicing	Extract the program statements of a given execution path that may affect the values of some variables of interest

3 Method for the survey

In preparation for this survey, we first selected 23 well-known journals and conferences about software engineering (including software testing and analysis), programming language and systems, security, and mobile computing and systems, which are shown in Table 3.

Table 3 Selected journals and conferences

Acronym	Full name
ICSE	International conference on software engineering
FSE	International symposium on the foundations of software engineering
ASE	International conference on automated software engineering
ISSTA	International symposium on software testing and analysis
ISSRE	International symposium on software reliability engineering
ICST	International conference on software testing, verification and validation
TOSEM	ACM transactions on software engineering and methodology
TSE	IEEE transactions on software engineering
JSS	Journal of systems and software
STVR	Software testing, verification & reliability
IST	Information and software technology
OOPSLA	International conference on object-oriented programming, systems, languages, and applications
PLDI	ACM SIGPLAN conference on programming language design and implementation
POPL	ACM SIGPLAN-SIGACT symposium on principles of programming languages
OSDI	USENIX symposium on operating systems design and implementation
TOPLAS	ACM transactions on programming languages & systems
CCS	ACM conference on computer and communications security
S&P	IEEE symposium on security and privacy
SEC	USENIX security symposium
NDSS	Network and distributed system security symposium
TDSC	IEEE transactions on dependable and security computing
MobiCom	ACM/IEEE international conference on mobile computing and metworking
MobiSys	International conference on mobile systems, applications, and services

Then according to the topics that this survey focuses on, we defined a number of keywords for searching papers, which are given in Table 4. Besides, we extended them with regular expressions and part-of-speech conversion to obtain more keywords. For instance, the keyword "Analysis" can be extended to the verbs "Analyze" or "Analyse", and "App" can be extended to "Applications".

Table 4 Keywords

Android	Mobile	Smartphone
App	Portable	Program analysis
Dynamic analysis	Static analysis	Control flow
Data flow	Taint	Privacy
Fragmentation	Permission	Compatibility
Performance	Security	Energy
Dalvik	Resource	Test

Based on these keywords, we searched the related literature published in the selected journals/conferences from 2010 to 2016 in the repository DBLP. Specifically, for each conference of each year, we first got the content table (paper lists) on DBLP, performed the search with the keywords, and totally collected 1967 research papers. Then we downloaded these papers from the corresponding sources, like IEEE Xplore Digital Library, ACM Digital Library, and ScienceDirect. Through further review of the contents of these papers, we finally selected 77 regular papers that target at program analysis for Android apps after filtering the short/tool papers and discarding the irrelevant papers. It is noted that we only focus on the works that employ program analysis techniques to detect some kinds of bugs or defects in Android apps. There are many approaches proposed for the infrastructure to assist program analysis on Android apps, such as Dalvik byte-code transformation to java byte-code [12], GUI modeling [13], precise inter-procedural control flow graph construction [14–19], execution profiling [20], loop analysis [21], string analysis [22], and privacy data identification [23–25], which are excluded from this survey. The papers that perform abstract researches and yield no prototype tools for their presented approaches are also excluded.

4 Taxonomy

In this section, we present our taxonomy of existing works on analyzing Android apps. The selected papers are categorized into different groups in terms of the defect types in the apps that the works aim to detect, including security bugs, functionality bugs and performance bugs. We further refine the classifications in the same category according to either their proposed techniques or the more fine-grained bug types of the works. Figure 1 shows the taxonomy tree that contains the three main categories and eight sub-categories as follows.

- **Security** As mentioned before, the popularity of Android system makes Android apps the target of attackers and suffer from increasingly serious malware threats. And in response, a lot of research works that aim to detect security bugs via program analysis techniques come into sight, which mainly focus on privacy leak, permission abuse, and app vulnerability.
- **Functionality** The functionality of apps, which determines the competitiveness and popularity of the apps, is one of the most important things that the developers and users care about. Testing (one of the dynamic analysis techniques) is a generic technique to find general func-

tionality bugs, like functionality inconsistency and app crash. Besides the general bugs, there are also several approaches proposed for detecting specific functionality bugs related to the Android specification.

- **Performance** Besides the functionalities of the apps, the performance is another significant factor that the developers and users concern about, which affects the user-friendliness of the apps. The responsiveness, resource usage and energy consumption are three major concerns for users and developers, and there have been several research works to improve the performance and detect energy bugs.

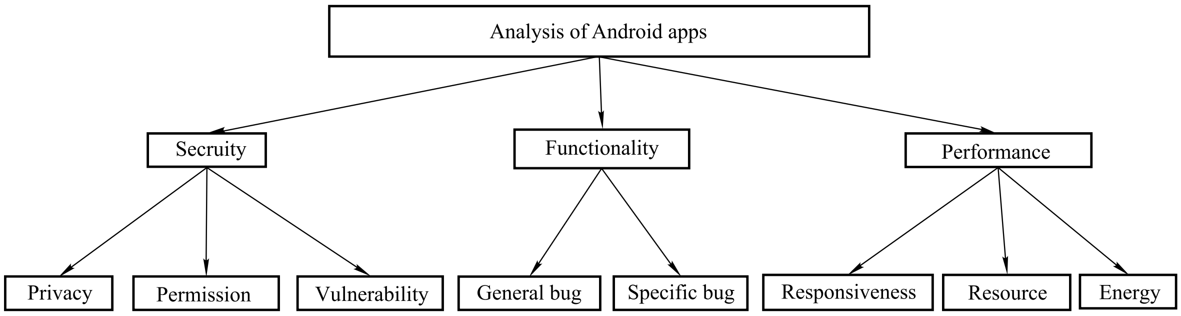


Fig. 1 Taxonomy of program analysis for Android apps

In the next three sections, we will give the details of the existing works about these categories, respectively.

5 Security

A lot of research works have come into sight recently, which aim to detect the security bugs, including privacy leak, permission misuse, and app vulnerability, in Android apps via program analysis techniques.

5.1 Privacy leak

Android devices contain lots of private data, such as device ID, contacts, and accounts, which are usually targeted by the attackers. Since the attackers often attempt to steal these privacy data by injecting malicious code into the apps, a number of researchers focus on detecting such kind of privacy leak bugs in Android apps, and propose several approaches which are mainly based on the taint analysis technique, which can be either static or dynamic.

Static taint analysis is a classic data-flow analysis method to track the data in sensitive program entities (so-called source points) and determine whether the data can be propagated to other specific program points (so-called sink points). In privacy leak detection, the source points are the privacy data obtaining statements (or APIs) and the sink points are the data transmission statements (or APIs). However, directly applying the static taint analysis technique to detect privacy leaks in Android apps may meet some challenges introduced by Android language characteristics, which are: (1) the implicit control flow transition introduced by callback and ICC mechanisms that makes the generation of accurate control flow graph very difficult; (2) the characteristics inherited from Java language like polymorphism, alias and reflection that make the data-flow analysis time-consuming; (3) and the hybrid of Java code and JavaScript code which are enabled in Android apps. There are several works [5, 26–29] trying to overcome one or more of those challenges when applying static taint analysis technique to detect privacy leaks. For instance, Arzt et al. [5] and Li et al. [27] present the tool FlowDroid which overcomes the first challenge while modeling the Android life-cycle and performing an on-demand alias analysis technique to improve the data-flow analysis efficiency. Lee et al. [30] perform string analysis on the inter-operation APIs for Java and JavaScript code to expose the inter-communication between them in the app. However, currently there is no available tool that can support a complete set of Android characteristics.

Applying dynamic taint analysis technique to detect privacy leak does not need to handle the above challenges. In turn, it usually demands the execution and requires to monitor the data propagation in runtime. In general, the work-flow might be as follows. First it identifies the source and sink APIs just as the static one does. Second it instruments in the interpreter of Dalvik VM or the app to obtain runtime information. Third it executes the app and monitors the propagation of sensitive data in the smartphones. There are several works [7,31–33] that follow such work-flow. The majority are based on instrumenting the Dalvik VM, but it fails in Android 5.0 or later versions because the Android run time (ART) architecture, which adopts the ahead-of-time compilation technique, is introduced. Therefore, Sun et al. [34] instrument the ART compiler and Runtime to insert the tracing instruction into the apps, according to the taint propagation logic obtained by the static analysis of the code. The modification on the system may be unacceptable in some cases, thus You et al. [35] propose the reference hijacking technique, of which the main idea is to reset the execution environment to redirect the invocation of system libraries to their own libraries.

Besides the Android language characteristic challenges or instrumentation challenge, another difficulty in privacy leak detection is to distinguish the benign and malicious behaviors. Based on the fact that benign and malicious apps may use the same APIs to send private data, the inability to handle such difficulty may cause false positives. To address this issue, some external information comes in handy, such as statistical signatures of malicious apps, the user intention extracted from GUI and app specification (like the app privacy policy).

The malware signature based technique is a widely adopted and efficient approach to detect the malicious software in both academic and industrial communities. Useful signature information might be obtained and manipulated into some kind of form for identifying the malicious behaviors. Feng et al. [36] make use of the control-flow and data-flow semantic information of apps from known Android malware families of privacy leak, and propose a high-level specification language to describe them. Point-to analysis and data flow analysis are then used to determine whether a new app matches the given malware families. Obviously they can not handle the unknown malicious behaviors. Other works make use of machine learning techniques [37–40], such as support vector machine (SVM) and K-nearest neighbour (KNN), to automatically obtain the malicious signatures from the training Android apps. The features they select to represent the privacy leak might be pairs of sensitive data sources and sinks [37], suspicious sensitive APIs' program contexts (the pre-condition of the data transmission statements, like a certain system time) [39], or the dynamic information (i.e., the program execution trace) [41]. Each of the feature selection is based on some observations, and is applied with proper machine learning algorithm to reach a higher accuracy in privacy leak detection.

Besides, the content in the GUI window of apps (e.g., the text in a button) is used as a good indicator of the user intention for distinguishing the benign and malicious. For example, if the user clicks a button with the text "OK", it often means the user agrees to perform the following program behavior. Huang et al. [42] target at detecting mismatches between user's expectation and real program behaviors. They first perform static analysis to find the paths with potential privacy leaks, then check the text of the corresponding user interface artifacts with keyword matching. If the contents in the UI components are not text but images or are synonym words outside their keyword repository, this approach may be evaded.

Privacy policy, a kind of app specifications, declares what private information is collected by the app in natural language and is attached to the app to be released in app stores. Slavin et al. [43] aim to detect the malicious behaviors through figuring out the mismatch between the code of Android apps and their privacy policy. They manually map the Android APIs and privacy policies by their terminologies with ontology technique, and perform information flow analysis to check whether the app code violates its declared privacy policy. However, some Android apps often do not have privacy policy declaration, especially the apps in non-official markets.

5.2 Permission misuse

Android system provides a permission-based mechanism to protect the use of some APIs that access sensitive data or involve sensitive operations. The app needs to declare the required permissions in the manifest file, and Android system will ensure that the unrequested ones are not used by the app. However, there are several issues with this mechanism, among which permission over-privilege and permission re-delegation are two representative ones.

Permission over-privilege Permission over-privilege means that the app routinely requests more permissions than what are required in the code or declared in its specification. How to figure out the permissions requested in the code or specification is the essential task to detect the permission over-privilege defects.

For checking the consistency of permissions declared in manifest file and used in the code, an essential step is to map between system APIs and permission set, since the Android document does not provide such one. With the help of this mapping, it is easy to obtain the permissions that are indeed required in the code of the analyzed app by just collecting the set of APIs it invokes. Finally, we can compare the real required permissions with the declared permissions in the manifest file, and find the permission over-privilege. To collect such mapping, Felt et al. [44] employ a testing technique to execute the Android APIs and log the program execution information that can be used to infer the permissions required by the Android APIs, and implement a static analysis tool that can collect the set of APIs invoked in the given app. Bartel et al. [45] make use of similar technical route with the difference that they calculate the permission map of Android APIs through static analysis of Android framework code rather than testing. They first generate the call graph of the framework code, then for each API extract the permission enforcement points that call permission checking methods (e.g., *checkPermission*), and finally add the checked permission in such methods into the required permission set of the API.

Developers often release the app description attached to the app to illustrate the functionalities of the app, which is also used to detect the permission over-privilege defects. For checking the consistency of the code and app description, the mapping between the natural language entities (like words, phrases, and sentences) and the permission set is indispensable. With this mapping, it is easy to compare the permission required in the code and app description to identify the inconsistency. Pandita et al. [46] present a natural language processing (NLP) based framework called WHYPER to identify the sentences that describe the need for a given permission in the description of an app. They employ a fixed vocabulary to describe the mapping relation between permission and the noun phrase. With the same goal, Qu et al. [47] designed a learning-based algorithm to analyze how close a given noun phrase is related to a permission and construct description-to-permission association model.

The above approaches may be easily bypassed by malicious apps. For instance, the attackers can insert unnecessary APIs that require a sensitive permission into their apps or add some fake functional descriptions in the app store. Xu et al. [48] propose a dynamic analysis based approach to figure out the uses of the requested permissions in Android apps. They generate the user events to execute the app and record all the function invocations by profiling functionality of the activity manager. They implemented a tool called Permlyzer, which can identify the location, the cause, and the purpose of permission use, and assist analyzers to determine whether the permission use is abnormal.

Permission re-delegation Permission re-delegation occurs when an app without permissions accesses the sensitive data/APIs through another app with permissions. A work by Felt et al. [49] shows that permission re-delegation is a widespread threat in Android apps. They find more than a third of Android apps in their experiment request permissions for sensitive resources and also expose public interfaces that may bring the risk of permission re-delegation.

Bagheri et al. [50] implement a tool COVERT that can detect permission leakages in the combination of Android apps, rather than a single app. They first use the static analysis techniques to analyze the app's configuration file and the source code to obtain the model of program behavior, including inter-procedure communication (IPC) messages, permission information and vulnerable paths without permission check. Then they use the Alloy language (an object modelling notation) to describe the combination of models of all analyzed apps, and perform the model finding technique to verify the whole model.

Permission re-delegation may cause the component hijacking attack in Android apps, which means the improper access control of the privacy data for the external requests. Several approaches have been proposed to detect and prevent the component hijacking attack [51–53]. Their main idea is to track the propagation of private information, which is similar to privacy leak detection.

5.3 Vulnerability

Android system and devices provide various flexible manners for the apps to communicate with other apps, external programs and devices, like network, bluetooth, Wifi, and ICC mechanism. Besides, Android apps can be easily handled by reverse engineers due to the open-source nature of Android system. The above two factors result in the vulnerability in more aspects of Android apps. The existing works about the Android apps' vulnerability can be categorized into four types according to the attack channels or methods, including network and web attacks, ICC attacks, app clone, and external code.

Network and web attacks Currently a state-of-the-art Android app is inevitable to access the network for communicating with external services so that it suffers from much network vulnerability. To build a well-developed and security-aware app, data coming through the network needs to be recognized and their harmfulness should be analyzed. Jin et al. [6] focus on the cross-site scripting (XSS) attack in HTML5-based Android apps, to prevent code injection from increasing data channels in Android apps, such as SMS, Contact, and barcode. They use the program slicing technique to obtain the points that read the data from untrusted channels and make use of the static taint analysis to detect whether the untrusted data is further processed. Shao et al. [54] aim to figure out the misuse of Unix domain sockets in Android apps, which may make the apps vulnerable to the attacks (e.g., DoS). They first collect all the Unix domain socket addresses used in the app and discard those addresses that are under protection and not vulnerable. Then they examine whether the app uses the authentication mechanisms. If the app does not adopt any checks or only adopts weak checks, then they employ the static analysis techniques to determine the reachability of the vulnerable points.

ICC attacks ICC mechanism provides a convenient way for the communication among components. However, the misuse of the ICC mechanism also brings potential security defects. Chin et al. [55] implement a tool ComDroid to detect the two kinds of vulnerabilities (unauthorized intent receipt and intent spoofing) in the ICC of Android apps. It performs a flow-sensitive static analysis to capture the intent sending behavior. Hay et al. [56] propose an iterative test generation approach to detect the vulnerabilities (XSS, SQL injection, etc.) in the inter-application communication of Android apps. In each iteration, they recover the custom fields (variables) of Intent by instrumenting the APIs (e.g., `getStringExtra`) that are used to read such fields and monitor the app execution. When a new field is captured, they generate various random values to cover potential paths related to this field, and monitor the app log and HTTP traffic to detect the vulnerabilities.

App clone Repackaged Android apps, also called app clones [57], have been found in many third-party markets. Chen et al. [58] introduce the geometry characteristic centroid, to measure the similarity of two methods' CFGs in two apps, and further implement an app clone detection system based on method level similarity. WuKong, a tool proposed by Wang et al. [59] to detect Android app clone, makes use of the number of API calls and manhattan distance to measure the similarity of the apps. Then, it employs an existing counting-based code clone detection approach [60] to refine the similarity detection of the selected potentially cloned apps by using more information of code structure.

External code It is a trend that most apps are embedded with some external code to conveniently integrate specific functionalities. However, this mechanism is not safe enough for Android apps. For instance, the mobile ad is a classic third-party program, which provides a way for developers to generate revenue and incidentally introduces the vulnerability [61]. Crussell et al. [62] identify two patterns of fraudulent ad behaviors, including requesting ads in background and clicking on ads without user interaction. They first run the target apps in both the foreground and background, and record the HTTP requests. Then they make use of the machine learning technique to classify the ad requests and detect the fraudulent ad behaviors. Besides, the dynamic code loading mechanism is often used to execute the external code in runtime. Poeplau et al. [63] discuss the vulnerability of Android apps introduced by this mechanism. They systematically summarize five ways to dynamically load code, and propose an approach based on the static analysis techniques (e.g., backward program slicing) to detect all the uses of the dynamic loading techniques in the analyzed Android app.

6 Functionality

There needs a comprehensive test and analysis on Android apps that provide more and more interesting and diverse functionalities to attract users' attention, thus to improve the software's quality and gain confidence for users. Functionality bugs are categorized into general bugs and specific ones.

6.1 General bugs

By "general bugs", we mean functionality inconsistency and app crash, which are often detected by the testing techniques. In this section, we categorize the works about this topic according to their used techniques.

Random testing Random testing is a generic black-box approach for testing Android apps. Dynodroid, an input generation system constructed by Machiry et al. [64], takes the random exploration strategy to generate event sequences according to the variation of widgets' layout. It can generate the system events as well as the user events, and determine the layout of widgets on the current screen by instrumenting the Android framework to monitor the reaction of an app to each event, then it applies a modified random exploration strategy with the consideration of the visited state of the candidate events. PUMA, proposed by Hao et al. [65], is an automated and programmable framework for exploring app execution and collecting runtime information. It only provides random exploration of the app and uses the finite state machine as the model of the app. However, it can be easily extended with different exploration strategies.

Model-based testing Model-based testing is a popular and important testing approach for GUI programs, like Android apps. Several approaches [10, 66, 67] based on this technique have been proposed to generate the test cases. These methods usually first construct the GUI model by either static analysis of the layout files or dynamic GUI exploration, and then design the exploration strategies to emulate the generated event sequences. These works do not systematically discuss the abstraction levels of the states in the model, which Baek and Bae [68] aim to characterize. They propose five GUI comparison criteria (GUICC) to provide different abstraction levels of the states. In addition, they also propose a generic framework based on the GUI ripping technique to construct the model according to the given GUICC and generate test suites.

Symbolic/concolic execution Symbolic/concolic execution is a commonly used approach in white-box testing to systematically and precisely explore the code of the software under test (SUT). In general, it first traverses the SUT to get the program path, then assumes symbolic values for inputs and finally determines what inputs drive the program to execute along the path with the help of constraint solving technique. Different with traditional console programs, the input of an Android app consists of two parts, the user events and the content that can be typed through the GUI window. In addition, a program path is composed of all callbacks and methods triggered by the given user event sequence.

To capture the two kinds of inputs, Mirzaei et al. [3] propose the interface model and behavior model. The former is used to find the inputs that an app can receive through its user interfaces and the latter drives the apps by symbolic execution and generates sequences of events. With these models, they employ symbolic path finder (SPF) [69], a symbolic execution engine built on top of java path finder (JPF) [70] to generate test inputs, by creating symbolic variables corresponding to the interface model and profiling the program paths from the behavior model. Jensen et al. [71] focus on covering the specific code (called target) that requires complex event sequences in an Android app, rather than systematically exploring the program paths according to some coverage criteria. They first make use of concolic execution to get the function summaries of each event handler. Based on the UI model and function summaries, they perform a breadth-first backward traversal in the UI model and identify the anchor events by checking whether the event handler summaries affect the variables of targets. Then they complement the anchor events to complete event sequences that cover the target.

Search-based testing Search-based testing (SBT) make use of evolutionary algorithms to generate test cases, which is guided by an objective function that is defined according to a desirable test target. Mahmood et al. [72] apply the SBT technique for test generation of Android apps, with the help of call graph models obtained from static analysis to avoid lots of invalid test cases in the crossover step. Mao et al. [73] also employ the search-based technique to test Android apps. Different with Mahmood et al.'s [72] work, they consider not only the coverage of the generated test cases, but also the sequence length and fault detection capability of the test cases. They employ a Pareto-optimal Search Based Software Engineering (SBSE) approach [74] to optimize these three objectives at the same time.

Others Mirzaei et al. [11] leverage the *t*-way combinatorial testing (CT) technique to emulate the possible interactions among GUI components for testing Android apps. Their main contribution lies in the CT model construction process, where they reduce the model scale by pruning the combinations of the GUI widgets that do not interact with each other. To do that, they make use of control flow and data flow analysis techniques to determine whether the two widgets are dependable. Liang et al. [75] implement a cloud service named Caiipa for testing Android apps. The difference between their work and the above-mentioned approaches is that they take the app context into account in the testing process, like CPU performance levels, amount of available memory, and different network environment. Payet and Spoto [76] focus on detecting several kinds of bugs, like dead-code and program termination, in Android apps. Their work is built on top of Julia [77], an abstract interpretation based static analysis framework for Java byte-code. They extend it to handle a number of Android characteristics, like XML manifest file, the life-cycle mechanism, and the callbacks.

6.2 Specific bugs

By "specific bugs", we mean the bugs involving some Android characteristics, such as concurrency task, Activity life-cycle of the app, and compatibility.

Concurrency task There are several works focusing on detecting the concurrency bugs in Android apps, like event-driven race [78–81] that is caused by the absence of the necessary synchronization among user events. These detection approaches first extract a number of happen-before (HB) rules to describe a part of the concurrency semantics of Android apps. Then they instrument the apps to trace the event and memory usage when the app is running, and finally detect and report the races when the traces violate the HB rules.

Activity life-cycle Shan et al. [82] reveal a new kind of bugs in Android apps, which is caused by the absence of data save and restore operations when the app exits. It may lead to data loss or failure to restart the app. They propose an approach based on control flow and data flow analysis techniques to automatically detect this kind of bugs in Android apps. They also leverage the guided testing technique to reproduce and verify the reported bugs.

Compatibility Compatibility bugs in Android apps are caused by the fragmentation of Android system and devices and may lead to different program behaviors of the same app in various systems and devices, as characterized by Wei et al. [83] in an empirical study on dozens of open-source apps. From the study, they observe the common patterns that many compatibility issues exhibit, and define an API-context pair model to formalize them. Based on the model, they propose an approach using the data flow analysis and program slicing techniques to detect the compatibility bugs in Android apps.

7 Performance

The portable devices with Android system often have limited resources (CPU, memory etc.) compared with PCs, so the performance of the apps is an essential issue the developers need to consider. Among performance defects, the responsiveness, resource usage, and battery energy efficiency are three prominent ones.

7.1 Responsiveness

Responsiveness is critical for user experience, and is shown to be in poor state in Android apps [84]. Approaches have been proposed to detect the responsiveness bugs or improve the responsiveness of the app.

Responsiveness bug detection A typical responsiveness bug is GUI lagging and it is caused by executing long-running operations in the UI event thread, which is found in an empirical study conducted by Liu et al. [84] on 70 real-world Android apps to characterize the performance bugs in the apps. They further summarize the code pattern, and implement a static code analyzer called PerfChecker to automatically detect the bugs. Kang et al. [85] also focus on detecting GUI lagging defects in Android apps. They first conduct an empirical study to figure out a threshold of UI delay for users' patience. Based on this threshold, they propose a dynamic analysis approach to detect operations with poor responsiveness by runtime monitoring. Kang et al. [86] focus on the unexpected asynchronous task execution that leads to the responsiveness decrease. They first classify the tremendous ways to start asynchronous tasks into five categories, and then employ static analysis techniques to obtain the necessary information for profiling the asynchronous task execution and the dependencies between these tasks. Finally, they make use of a plugin testing method to exercise the app and record the instrumentation information and performance metrics to detect performance issues.

Responsiveness improvement To eliminate the GUI lagging caused by executing long-running operations, Lin et al. [87, 88] implement a code transformation tool that can enable developers to move the code from the main thread into AsyncTask to improve the app responsiveness. They employ a static analysis technique, dominator analysis, to determine which part of the code should be put into the AsyncTask. Besides, the computation offloading technique, which executes the time-consuming code in a remote server, is also a generic approach to improve the software responsiveness. Based on this technique, Zhang et al. [89] first design a pattern to enable all the Java classes in the app to interact with each other locally or remotely, and then develop a refactoring tool called DPartner, which can automatically rewrite the byte-code in an Android app to implement the pattern with the help of the static analysis techniques.

7.2 Resource usage

As mentioned before, the computational resources are relatively limited, thus the usage effectiveness of these resources is important. Besides, Android provides a number of exotic resources, such as GPS and Sensors, the inappropriate usage of which may also decrease the performance of the app or even lead to app crash.

For the computational resources (e.g., memory), Yan et al. [90] propose a testing approach to detect the misuse of these resources in some specific scenarios, which are summarized from their observations. Specifically, they propose a definition called "neutral cycles" to represent such user event sequences that should not lead to increase in resource usage. For example, the resource usage must not increase in such user operations as rotating the screen from vertical to horizontal and then rotating back. They traverse the paths to generate the test cases that can cover the "neutral cycles", and detect the resource misuse by monitoring the runtime information and capturing the resource usage increase.

Besides, Guo et al. [91] and Wu et al. [92] focus on the misuse of the exotic resources, like Camera and GPS, which require the developers to request and release by themselves. The absence of releasing operations of these resources will cause resource leaks, which may lead to huge memory and energy consumption or even app crash. They design a flow-insensitive and a flow-sensitive static analysis algorithm based on the function summary technique to find the mismatch of the resource operations. In their further work, Liu et al. [93] aim to automatically and safely fix the resource leaks by static analysis and code instrumentation techniques. They use the control flow analysis method to decide the locations where the missing resource release operations need to be inserted.

Android devices are embedded with various hardwares, which often consume much battery energy. Banerjee et al. [94] focus on detecting energy hot-spots and energy bugs by monitoring the system calls in Android apps, which are related to major power-consuming hardware, such as screen and WIFI. Their main idea based on dynamic testing technique is to profile the energy consumption related system calls under given test cases by code instrumentation on the apps.

WakeLock mechanism is provided by Android system to protect critical operations in the app from being disrupted by the device sleeping mechanism for optimizing the energy efficiency when the device is idle. However, inappropriate request and release operations of resource *WakeLock* may also lead to energy bugs, which is targeted by several works [95, 96]. For instance, A. Pathak et al. [95] present a solution to detect these energy bugs by converting the non-sleep behavior to reaching definition (RD) dataflow problem.

A number of sensors are used in Android apps to obtain external information, like GPS to get location information, of which the misuse is also a cause of energy bugs. Liu et al. [97] focus on two kinds of misuse of sensors that may lead to energy inefficiency, including unregistered sensor listener and sensory data under-utilization. They use the dynamic analysis technique to monitor the usage of sensor listeners and the sensory data propagation when the app is running.

7.3 Energy efficiency

The battery energy efficiency is one of the most important things the users care about, since it is not feasible to charge the portable devices most of the time. In this subsection we present approaches that are proposed to estimate and optimize the energy consumption.

Energy consumption estimation The estimation of energy consumption for Android apps can make the developers aware of the energy cost of each code module in their apps and further give a chance to improve the energy efficiency. The estimation can be performed in different code levels, such as instruction and source code line levels.

Hao et al. [98] estimate the energy consumption of Android apps on the instruction level. They assume there is a per-instruction energy model, and then instrument the app to trace the executed instructions during runtime, and finally calculate the energy cost of a given test event sequence. In contrast, Li et al. [8] aim to estimate source line level energy consumption. They apply a combination of program analysis, code instrumentation and statistical modeling based on hardware power measurements. To be specific, they run the instrumented app under manually designed test cases on a power measurement platform to obtain the energy consumption corresponding to each path, and then make use of linear regression analysis to calculate the energy consumption of each line of code.

Energy optimization Li et al. [99] aim to optimize the energy consumption of the HTTP requests in Android apps. Their main idea is to merge multiple small HTTP requests into a large HTTP request. To do that, they apply the dominator analysis technique to determine which HTTP requests can be bundled together, and rewrite the code by introducing a proxy mechanism for redirecting the HTTP requests and receiving the responses in runtime.

8 Statistics and findings

We also perform statistic analyses on these papers to summarize findings about the trends of the analysis of Android apps.

Distribution of selected papers in venue and year We perform some statistic analysis on these papers to figure out the distributions of them in the publication venues and the prevailing trend of each year. Figure 2 gives the distribution of selected papers in each journal/conference, which indicates that about 30% selected papers were published in the top-tier conferences ICSE and CCS. Figure 3 shows the number of selected papers in each year from 2010 to 2016. From this figure, we can see that the number of papers about the analysis of Android apps has been increasing in recent years.

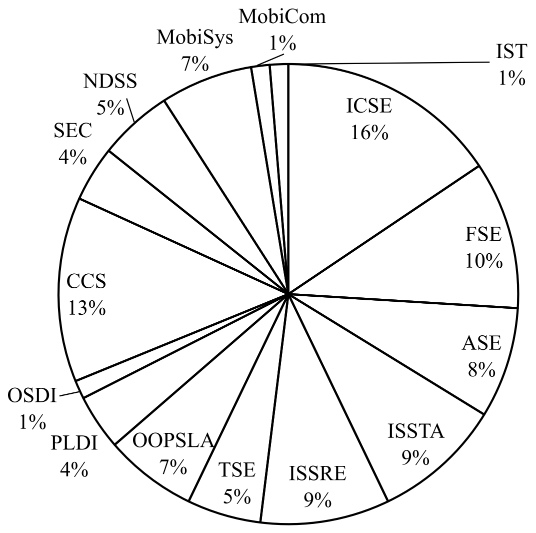


Fig. 2 Distribution of selected papers in each journal/conference

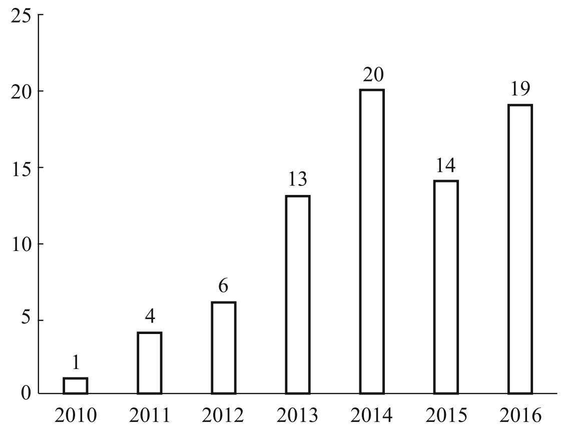


Fig. 3 Numbers of selected papers in each year

Hot topic We also count the numbers of selected papers about our categories. Table 5 shows the details. The third column is the number of papers in each sub-category and the last column gives the number and percentage of each category. The numbers of papers about three topics are 41, 19 and 17 respectively. Figure 4 shows the word cloud from the titles of selected papers. We can see that Android security, especially privacy leak detection, is a hot topic in the field of the analysis of Android apps.

Table 5 The numbers of selected papers of each topic

Category	Sub-category	#N	#TN/P
Security	Privacy	21	41/53%
	Permission	10	
	Vulnerability	10	
Functionality	General bug	13	19/25%
	Specific bug	6	
Performance	Responsiveness	6	17/22%
	Resource	8	
	Energy	3	

Fig. 4 Word cloud of the titles of the selected papers

Distribution of program analysis techniques Table 6 shows the numbers of selected papers with different program analysis techniques, including static analysis, dynamic analysis, and the hybrid of the two techniques. We can see that about half of the works in our survey leverage the static analysis approach to detect the defects in Android apps. In further investigation we find that the majority of static analysis works make use of data-flow analysis, while seven works [40, 50, 51, 58, 63, 76, 88] are based on control flow analysis and nine works [43, 46, 47, 55, 57, 59, 84, 87, 89] take other representation forms like information flow or function call graph into consideration. For dynamic analysis technique, the application usually lies in the functionality testing of Android apps. And there is no trend for the detailed techniques applied in dynamic analysis works, except for that many works make use of taint analysis and path profiling techniques. As mentioned before, both static and dynamic techniques have some shortcomings, i.e., static analysis suffers from false positives while dynamic analysis depends on the quality of test cases. As a result, a number of works (18%) combining static analysis and dynamic analysis techniques have been proposed in recent years to detect and reproduce the defects.

Table 6 The numbers of selected papers of each technique

Technique	Paper	P
Static analysis	Arzt [5], Gordon [28], Huang [29], Li [27], Wei [26], Lee [30], Feng [36], Fan [40] Avdiienko [37], Wu [38], Yang [39], Huang [42], Slavin [43], Bartel [45], Pandita [46], Qu [47], Bagheri [50], Grace [51], Lu [52], Zhang [53], Jin [6], Shao [54], Chin [55], Gibler [57], Chen [58], Wang [59], Poeplau [63], Payet [76], Wei [83], Liu [84], Lin [87, 88], Zhang [89], Guo [91], Wu [92], Liu [93], Liu [96], Pathak [95]	49%
Dynamic analysis	Enck [7], Yan [33], Zhang [32], Sun [34], You [35], Xu [41], Felt [49], Hay [56] Hornyack [31], Crussell [62], Machiry [64], Hao [65], Baek [68], Jensen [71], Mahmood [72], Liang [75], Bielik [79], Hsiao [80], Hu [81], Maiya [78], Kang [85], Yan [90], Li [8], Banerjee [94], Liu [97]	33%
Hybrid	Yang [100], Felt [44], Xu [48], Gui [61], Anand [67], Azim [10], Choi [66], Mirzaei [3], Mao [73], Mirzaei [11], Shan [82], Kang [86], Hao [98], Li [99]	18%

Supplementary techniques from other areas From the previous sections, we see that several other techniques, like machine learning and natural language processing, are introduced into program analysis for detecting some defects, which are related to not only the code but also some extra information (e.g., user intention, app description) and can not be described as a fixed property. For instance, in privacy leak detection, researchers [37–41] leverage machine learning to distinguish between benign and malicious private data sending behavior. In permission misuse detection, several works [46, 47] are based on natural language processing techniques to generate the mapping relations between the specifications of Android apps and the permissions.

Misuse of Android APIs and mechanisms Android system provides various APIs, components and mechanisms for developers to simplify the developing process and enrich user experience. However, they also increase the difficulty for developers to control over their systems. Besides, the unclear illustrations of some mechanisms often confuse the developers (see "Android lifecycle issue" and "Examples for Android Launch modes" in Stackoverflow). From our survey, we find that a number of defects are caused by developers' misunderstanding or misuse of the APIs and mechanisms in Android system, and several works try to statically analyze Android apps for revealing defects, taking the Android specification into consideration. For example, the life-cycle of Activity components is a complex mechanism, which may be misunderstood by developers. Several works [82, 92] show that such misunderstanding of life-cycle may lead to defects such as resource leak and data loss.

9 Other surveys

There are several prior works that survey proposed approaches and techniques for improving the quality and security of Android apps. In this section, we give the overviews of several surveys and highlight the differences between these works and ours.

Haris et al. [101] give a survey of various research works on privacy leakage in mobile computing. They first give the definition of privacy by several characteristics, e.g., individual consent and privacy data. They also summarize the privacy risks into four categories, including application privacy risk, connectivity privacy risk, sensing privacy risk and user related privacy risk, and discuss the related works on detecting the privacy leakages and protecting the privacy data of these categories.

Choudhary et al. [102] conduct an empirical study on available automatic test generation tools for Android apps to evaluate their effectiveness. They evaluate from four aspects, including code coverage, fault detection, ability to work on multiple platforms and ease of use. An interesting finding of their work is that Monkey and Dynodroid, which employ the simplest exploration strategy (random), outperform other tools that make use of more complex exploration strategies on code coverage and fault detection capability.

Martin et al. [103] perform a broad study on the analysis of the apps mined from App Stores. Based on the objectives, they categorize the existing works into seven parts, including API analysis, feature analysis, releasing engineering, review analysis, security analysis, store ecosystem comparison and size and effort prediction. The authors focus on the scale of app samples used in the existing works, and they find that the scale has been increasing in these years. Specifically, the number of sampled apps used in the studies reached 10,000 to 100,000 in 2015.

Sufatrio et al. [104] thoroughly investigate the existing research works about security enhancements to Android and analysis techniques to detect malicious Android apps. They give a clear and precise taxonomy to classify the prior works according to the five app deployment stages in the Android ecosystem, and also a detailed comparison about pros and cons of similar works in the same categories. Their survey shows there are two areas in Android security that draw most attention from current researchers, which are app analysis and runtime-based platform protection.

Although the above previous works involve the analysis or testing of Android apps, they focus on either one specific problem of Android (e.g., privacy leak, security) or comparison of existing tools. In this paper, we want to investigate how the program analysis techniques, the commonly adopted approaches in ensuring software quality, are applied to Android apps, what challenges are introduced by Android characteristics for program analysis techniques, and what kind of solutions are proposed.

Li et al. [105] perform a systematic literature review (SLR) on the research works about static analysis for Android apps. From their survey, they summarize several key findings about the application scenarios and challenges of the static analysis technique adopted in Android apps. However, they only survey the papers published from 2010 to 2014, while we collect more than 30 papers published in the recent two years. Besides, their work does not involve the dynamic analysis techniques. From our survey, we observe that it is a trend to combine the dynamic analysis and static analysis techniques to detect the defects in Android apps.

Sadeghi et al. [106] also conduct an SLR, but it is on the existing works of the security of Android apps. They extend the scope to the program analysis techniques and form several taxonomies of the existing works by categorizing them into different groups based on dimensions, which include the approaches' objectives and intents, the used techniques, and the assessment and evaluation methods. The difference between Sadeghi's work and our survey lies on the emphasis when introducing the existing works. In our work, we mainly expect to illustrate the motivation of the works (i.e., the problems that threat app quality or existing works) and the proposed approaches. For instance, the works about privacy leak detection are mostly based on taint analysis techniques, while there are a number of extra features making this technique not directly adaptable, like user intent. Naturally, the supplementary techniques are introduced into the program analysis, which are what we intend to demonstrate in this paper.

To sum up, we select a larger scope of topics compared with the existing surveys. The topics contain security, which is a relatively mature domain and has been surveyed with detailed taxonomies in several works, and functionality and performance, which have long been of much importance from the perspective of software engineering but have not been summarized by the existing surveys. According to our views, to form the industrialized infrastructure, developer should support the systematic and comprehensive analysis to detect the defects and improve quality from all the above aspects. In addition, since program analysis is a commonly used technique to ensure software quality, we surveyed in terms of program analysis in a wide scope to give out a high-level overview. Our survey eliminates the need for domain knowledge, meanwhile introduces interesting works and finally gives advice from many fields. We expect the brilliant ideas can be connected to inspire following studies.

10 Conclusion and future direction

Program analysis techniques have been widely adopted in ensuring the quality and security of Android apps. In this paper, we conduct a systematic survey on published literature to summarize the state-of-the-art research works on this issue. We categorize the existing works into different groups according to the objectives and proposed techniques.

In the end, we propose some recommendations to improve the program analysis for Android apps: (1) Currently, most of works are built on top of Java program analysis frameworks (Soot, WALA, JPF) and handle only a part of Android characteristics respectively. In addition, many works make duplicate efforts to deal with the same language characteristics. A generic analysis framework or tool that can support a complete set of Android characteristics is essential to reduce duplicate efforts and to improve the analysis precision for future works; (2) There is no sound framework for static analysis on Android apps that can perform path-sensitive analysis, especially systematic white-box testing. Current testing tools for Android apps are mostly based-on GUI exploring, a black-box testing technique, which may not be able to effectively traverse many potential program paths in Android apps and cannot achieve high code coverage for testing. Since Android apps are event-driven, the program paths will be more unpredictable and more complex. Therefore, new techniques are also supposed to address the inherent challenges of path-sensitive analysis, like path explosion; (3) Android developer reference suggests how to use specific APIs or mechanisms. However, there still exist a number of mechanisms that are not illustrated clearly. In addition, the Android APIs are upgraded frequently so that developers may not keep pace with such changes in time [2]. These factors bring challenges for developers to use the APIs correctly, which might lead to ignoration or misunderstanding of these suggestions and result in various defects in apps. It is a remarkable research direction to systematically collect, characterize, formalize those recommendations, and finally detect the violation of them by program analysis; (4) Android systems are continuously being upgraded and introduce significant changes in the architecture, like hybrid of Java code and JavaScript code. These features may disable the existing approaches, like the dynamic analysis approaches with the instrumentation on DVM. So it is necessary for the researchers to consider the capability of their works to be compatible with the changes in Android system.

Acknowledgements The authors would like to thank the anonymous reviewers for their helpful comments and suggestions. This work was supported by the National Natural Science Foundation of China (Grant No. 61672505), the National Key Basic Research (973) Program of China (2014CB340701), and Key Research Program of Frontier Sciences, Chinese of Academy Sciences (QYZDJ-SSW-JSC036).

References

1. Zhou Y, Jiang X. Dissecting Android malware: characterization and evolution. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy. 2012, 95–109
2. McDonnell T, Ray B, Kim M. An empirical study of API stability and adoption in the Android ecosystem. In: Proceedings of the 2013 IEEE International Conference on Software Maintenance. 2013, 70–79
3. Mirzaei N, Bagheri H, Mahmood R, Malek S. SIG-Droid: automated system input generation for Android applications. In: Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering. 2015, 461–471
4. Kim J, Yoon Y, Yi K, Shin J. SCANDAL: static analyzer for detecting

privacy leaks in Android applications. Mobile Security Technologies, 2012, 12: 110

5. Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, Traon Y L, Octeau D, McDaniel P. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation. 2014, 259–269
6. Jin X, Hu X, Ying K, Du W, Yin H, Peri G N. Code injection attacks on HTML5-based mobile apps: characterization, detection and mitigation. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. 2014, 66–77
7. Enck W, Gilbert P, Chun B, Cox LP, Jung J, McDaniel P, Sheth A. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation. 2010, 393–407
8. Li D, Hao S, Halfond W G J, Govindan R. Calculating source line level energy information for Android applications. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis. 2013, 78–89
9. Xu G H, Mitchell N, Arnold M, Rountev A, Schonberg E, Sevitsky G. Scalable runtime bloat detection using abstract dynamic slicing. ACM Transactions on Software Engineering Methodology, 2014, 23(3): 23
10. Azim T, Neamtiu I. Targeted and depth-first exploration for systematic testing of Android apps. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming, Systems, Languages, and Applications. 2013, 641–660
11. Mirzaei N, Garcia J, Bagheri H, Sadeghi A, Malek S. Reducing combinatorics in GUI testing of Android applications. In: Proceedings of the 38th International Conference on Software Engineering. 2016, 559–570
12. Octeau D, Jha S, McDaniel P. Retargeting Android applications to Java bytecode. In: Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering. 2012, 6
13. Yang S, Zhang H, Wu H, Wang Y, Yan D, Rountev A. Static window transition graphs for Android. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering. 2015, 658–668
14. Yang S, Yan D, Wu H, Wang Y, Rountev A. Static control-flow analysis of user-driven callbacks in Android applications. In: Proceedings of the 37th IEEE/ACM International Conference on Software Engineering. 2015, 89–99
15. Cao Y, Fratantonio Y, Bianchi A, Egele M, Kruegel C, Vigna G, Chen Y. EdgeMiner: automatically detecting implicit control flow transitions through the Android framework. In: Proceedings of the 22nd Annual Network and Distributed System Security Symposium. 2015
16. Octeau D, McDaniel P, Jha S, Bartel A, Bodden E, Klein J, Traon Y L. Effective inter-component communication mapping in Android: an essential step towards holistic security analysis. In: Proceedings of the 22nd USENIX Security Symposium. 2013, 543–558
17. Octeau D, Luchaup D, Dering M, Jha S, McDaniel P. Composite constant propagation: application to Android inter-component communication analysis. In: Proceedings of the 37th IEEE/ACM International Conference on Software Engineering. 2015, 77–88
18. Octeau D, Luchaup D, Jha S, McDaniel P D. Composite constant propagation and its application to android program analysis. IEEE Transactions on Software Engineering, 2016, 42(11): 999–1014
19. Octeau D, Jha S, Dering M, McDaniel P D, Bartel A, Li L, Klein J, Traon Y L. Combining static analysis with probabilistic models to enable market-scale Android inter-component analysis. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 2016, 469–484
20. Wei X, Gomez L, Neamtiu I, Faloutsos M. ProfileDroid: multi-layer profiling of Android applications. In: Proceedings of the 18th Annual International Conference on Mobile Computing and Networking. 2012, 137–148
21. Fratantonio Y, Machiry A, Bianchi A, Kruegel C, Vigna G. CLAPP: characterizing loops in Android applications. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering. 2015, 687–697
22. Li D, Lyu Y, Wan M, Halfond W G J. String analysis for Java and Android applications. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering. 2015, 661–672
23. Huang J, Li Z, Xiao X, Wu Z, Lu K, Zhang X, Jiang G. SUPOR: precise and scalable sensitive user input detection for Android apps. In: Proceedings of the 24th USENIX Security Symposium. 2015, 977–992
24. Nan Y, Yang M, Yang Z, Zhou S, Gu G, Wang X. UIPicker: user-input privacy identification in mobile applications. In: Proceedings of the 24th USENIX Security Symposium. 2015, 993–1008
25. Rasthofer S, Arzt S, Bodden E. A machine-learning approach for classifying and categorizing Android sources and sinks. In: Proceedings of the 21st Annual Network and Distributed System Security Symposium. 2014
26. Wei F, Roy S, Ou X, 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. 2014, 1329–1341
27. Li L, Bartel A, Bissyandé T F, Klein J, Traon Y L, Arzt S, Rasthofer S, Bodden E, Octeau D, McDaniel P. IccTA: detecting inter-component privacy leaks in Android apps. In: Proceedings of the 37th IEEE/ACM International Conference on Software Engineering. 2015, 280–291
28. Gordon M I, Kim D, Perkins J H, Gilham L, Nguyen N, Rinard M C. Information flow analysis of Android applications in droidsafe. In: Proceedings of the 22nd Annual Network and Distributed System Security Symposium. 2015
29. Huang W, Dong Y, Milanova A, Dolby J. Scalable and precise taint analysis for Android. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. 2015, 106–117
30. Lee S, Dolby J, Ryu S. HybriDroid: static analysis framework for Android hybrid applications. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. 2016, 250–261
31. Hornyack P, Han S, Jung J, Schechter S E, Wetherall D. These aren't the droids you're looking for: retrofitting Android to protect data from imperious applications. In: Proceedings of the 18th ACM Conference on Computer and Communications Security. 2011, 639–652
32. Zhang Y, Yang M, Xu B, Yang Z, Gu G, Ning P, Wang X S, Zang B. Vetting undesirable behaviors in Android apps with permission

use analysis. In: Proceedings of 2013 ACM SIGSAC Conference on Computer and Communications Security. 2013, 611–622

33. Yan L, Yin H. DroidScope: seamlessly reconstructing the OS and dalvik semantic views for dynamic Android malware analysis. In: Proceedings of the 21st USENIX Security Symposium. 2012, 569–584

34. Sun M, Wei T, Lui J. TaintART: a practical multi-level information-flow tracking system for Android runtime. In: Proceedings of 2016 ACM SIGSAC Conference on Computer and Communications Security. 2016, 331–342

35. You W, Liang B, Shi W, Zhu S, Wang P, Xie S, Zhang X. Reference hijacking: patching, protecting and analyzing on unmodified and non-rooted Android devices. In: Proceedings of the 38th International Conference on Software Engineering. 2016, 959–970

36. Feng Y, Anand S, Dillig I, Aiken A. Apposcopy: semantics-based detection of Android malware through static analysis. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2014, 576–587

37. Avdiienko V, Kuznetsov K, Gorla A, Zeller A, Arzt S, Rasthofer S, Bodden E. Mining apps for abnormal usage of sensitive data. In: Proceedings of the 37th IEEE/ACM International Conference on Software Engineering. 2015, 426–436

38. Wu S, Wang P, Li X, Zhang Y. Effective detection of Android malware based on the usage of data flow APIs and machine learning. Information and Software Technology, 2016, 75: 17–25

39. Yang W, Xiao X, Andow B, Li S, Xie T, Enck W. AppContext: differentiating malicious and benign mobile app behaviors using context. In: Proceedings of the 37th IEEE/ACM International Conference on Software Engineering. 2015, 303–313

40. Fan M, Liu J, Luo X, Chen K, Chen T, Tian Z, Zhang X, Zheng Q, Liu T. Frequent subgraph based familial classification of android malware. In: Proceedings of the IEEE International Symposium on Software Reliability Engineering. 2016, 24–35

41. Xu H, Zhou Y, Gao C, Kang Y, Lyu M R. SpyAware: investigating the privacy leakage signatures in app execution traces. In: Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering. 2015, 348–358

42. Huang J, Zhang X, Tan L, Wang P, Liang B. AsDroid: detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. In: Proceedings of the 36th International Conference on Software Engineering. 2014, 1036–1046

43. Slavin R, Wang X, Hosseini M B, Hester J, Krishnan R, Bhatia J, Breaux T D, Niu J. Toward a framework for detecting privacy policy violations in Android application code. In: Proceedings of the 38th International Conference on Software Engineering. 2016, 25–36

44. Felt A P, Chin E, Hanna S, Song D, Wagner D. Android permissions demystified. In: Proceedings of the 18th ACM Conference on Computer and Communications Security. 2011, 627–638

45. Bartel A, Klein J, Monperrus M, Traon Y L. Static analysis for extracting permission checks of a large scale framework: the challenges and solutions for analyzing Android. IEEE Transactions on Software Engineering, 2014, 40(6): 617–632

46. Pandita R, Xiao X, Yang W, Enck W, Xie T. WHYPER: towards automating risk assessment of mobile applications. In: Proceedings of the 22nd USENIX Security Symposium. 2013, 527–542

47. Qu Z, Rastogi V, Zhang X, Chen Y, Zhu T, Chen Z. AutoCog: measuring the description-to-permission fidelity in Android applications. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. 2014, 1354–1365

48. Xu W, Zhang F, Zhu S. Permlyzer: analyzing permission usage in Android applications. In: Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering. 2013, 400–410

49. Felt A P, Wang H J, Moshchuk A, Hanna S, Chin E. Permission re-delegation: attacks and defenses. In: Proceedings of the 20th USENIX Security Symposium. 2011

50. Bagheri H, Sadeghi A, Garcia J, Malek S. COVERT: compositional analysis of Android inter-app permission leakage. IEEE Transactions on Software Engineering, 2015, 41(9): 866–886

51. Grace M C, Zhou Y, Wang Z, Jiang X. Systematic detection of capability leaks in stock Android smartphones. In: Proceedings of the 19th Annual Network and Distributed System Security Symposium. 2012

52. Lu L, Li Z, Wu Z, Lee W, Jiang G. CHEX: statically vetting Android apps for component hijacking vulnerabilities. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. 2012, 229–240

53. Zhang M, Yin H. AppSealer: automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications. In: Proceedings of the 21st Annual Network and Distributed System Security Symposium. 2014

54. Shao Y, Ott J, Jia Y J, Qian Z, Mao Z M. The misuse of Android unix domain sockets and security implications. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. 2016, 80–91

55. Chin E, Felt A P, Greenwood K, Wagner D. Analyzing inter-application communication in Android. In: Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services. 2011, 239–252

56. Hay R, Tripp O, Pistoia M. Dynamic detection of inter-application communication vulnerabilities in Android. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. 2015, 118–128

57. Gibler C, Stevens R, Crussell J, Chen H, Zang H, Choi H. AdRob: examining the landscape and impact of Android application plagiarism. In: Proceedings of the 11th Annual International Conference on Mobile Systems, Applications, and Services. 2013, 431–444

58. Chen K, Liu P, Zhang Y. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In: Proceedings of the 36th International Conference on Software Engineering. 2014, 175–186

59. Wang H, Guo Y, Ma Z, Chen X. WuKong: a scalable and accurate two-phase approach to Android app clone detection. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. 2015, 71–82

60. Yuan Y, Guo Y. Boreas: an accurate and scalable token-based approach to code clone detection. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. 2012, 286–289

61. Gui J, McIlroy S, Nagappan M, Halfond W G J. Truth in advertising: the hidden cost of mobile ads for software developers. In: Proceed-

ings of the 37th IEEE/ACM International Conference on Software Engineering. 2015, 100–110

62. Crussell J, Stevens R, Chen H. Madfraud: investigating ad fraud in Android applications. In: Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services. 2014, 123–134

63. Poeplau S, Fratantonio Y, Bianchi A, Kruegel C, Vigna G. Execute this! analyzing unsafe and malicious dynamic code loading in Android applications. In: Proceedings of the 21st Annual Network and Distributed System Security Symposium. 2014

64. Machiry A, Tahiliani R, Naik M. Dynodroid: an input generation system for Android apps. In: Proceedings of the 9th Joint Meeting on Foundations of Software Engineering. 2013, 224–234

65. Hao S, Liu B, Nath S, Halfond W G J, Govindan R. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In: Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services. 2014, 204–217

66. Choi W, Necula G C, Sen K. Guided GUI testing of Android apps with minimal restart and approximate learning. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications. 2013, 623–640

67. Anand S, Naik M, Harrold M J, Yang H. Automated concolic testing of smartphone apps. In: Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering. 2012, 59

68. Baek Y M, Bae D. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. 2016, 238–249

69. Pasareanu C S, Visser W, Bushnell D H, Geldenhuys J, Mehlitz P C, Rungta N. Symbolic pathfinder: integrating symbolic execution with model checking for Java bytecode analysis. Automated Software Engineering, 2013, 20(3): 391–425

70. Visser W, Pasareanu C S, Khurshid S. Test input generation with Java PathFinder. In: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis. 2004, 97–107

71. Jensen C S, Prasad M R, Møller A. Automated testing with targeted event sequence generation. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis. 2013, 67–77

72. Mahmood R, Mirzaei N, Malek S. EvoDroid: segmented evolutionary testing of Android apps. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2014, 599–609

73. Mao K, Harman M, Jia Y. Sapienz: multi-objective automated testing for Android applications. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. 2016, 94–105

74. Harman M, Mansouri A, Zhang Y. Search based software engineering: trends, techniques and applications. ACM Computing Surveys, 2012, 45(1): 11

75. Liang C M, Lane N D, Brouwers N, Zhang L, Karlsson B, Liu H, Liu Y, Tang J, Shan X, Chandra R, Zhao F. Caiipa: automated large-scale mobile app testing through contextual fuzzing. In: Proceedings of the 20th Annual International Conference on Mobile Computing and Networking. 2014, 519–530

76. Payet É, Spoto F. Static analysis of Android programs. Information and Software Technology, 2012, 54(11): 1192–1201

77. Cousot P, Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1977, 238–252

78. Maiya P, Kanade A, Majumdar R. Race detection for Android applications. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. 2014, 316–325

79. Bielik P, Raychev V, Vechev M T. Scalable race detection for Android applications. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. 2015, 332–348

80. Hsiao C, Pereira C, Yu J, Pokam G, Narayanasamy S, Chen P M, Kong Z, Flinn J. Race detection for event-driven mobile applications. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. 2014, 326–336

81. Hu Y, Neamtiu I, Alavi A. Automatically verifying and reproducing event-based races in Android apps. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. 2016, 377–388

82. Shan Z, Azim T, Neamtiu I. Finding resume and restart errors in Android applications. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. 2016, 864–880

83. Wei L, Liu Y, Cheung S. Taming Android fragmentation: characterizing and detecting compatibility issues for Android apps. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. 2016, 226–237

84. Liu Y, Xu C, Cheung S. Characterizing and detecting performance bugs for smartphone applications. In: Proceedings of the 36th International Conference on Software Engineering. 2014, 1013–1024

85. Kang Y, Zhou Y, Gao M, Sun Y, Lyu M R. Experience report: detecting poor-responsive UI in android applications. In: Proceedings of the IEEE International Symposium on Software Reliability Engineering. 2016, 490–501

86. Kang Y, Zhou Y, Xu H, Lyu M R. DiagDroid: Android performance diagnosis via anatomizing asynchronous executions. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2016, 410–421

87. Lin Y, Okur S, Dig D. Study and refactoring of Android asynchronous programming. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering. 2015, 224–235

88. Lin Y, Radoi C, Dig D. Retrofitting concurrency for Android applications through refactoring. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2014, 341–352

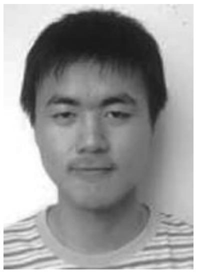
89. Zhang Y, Huang G, Liu X, Zhang W, Mei H, Yang S. Refactoring Android Java code for on-demand computation offloading. In: Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. 2012, 233–248

90. Yan D, Yang S, Rountev A. Systematic testing for resource leaks in Android applications. In: Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering. 2013, 411–420

91. Guo C, Zhang J, Yan J, Zhang Z, Zhang Y. Characterizing and detect-

ing resource leaks in Android applications. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering. 2013, 389–398

92. Wu T, Liu J, Xu Z, Guo C, Zhang Y, Yan J, Zhang J. Light-weight, inter-procedural and callback-aware resource leak detection for Android apps. IEEE Transactions on Software Engineering, 2016, 42(11): 1054–1076
93. Liu J, Wu T, Yan J, Zhang J. Fixing resource leaks in Android apps with light-weight static analysis and low-overhead instrumentation. In: Proceedings of the 27th IEEE International Symposium on Software Reliability Engineering. 2016, 342–352
94. Banerjee A, Chong L K, Chattopadhyay S, Roychoudhury A. Detecting energy bugs and hotspots in mobile apps. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2014, 588–598
95. Pathak A, Jindal A, Hu Y C, Midkiff S P. 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. 2012, 267–280
96. Liu Y, Xu C, Cheung S, Terragni V. Understanding and detecting wake lock misuses for Android applications. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2016, 396–409
97. Liu Y, Xu C, Cheung S, Lu J. Greendroid: automated diagnosis of energy inefficiency for smartphone applications. IEEE Transactions on Software Engineering, 2014, 40(9): 911–940
98. Hao S, Li D, Halfond W G J, Govindan R. Estimating mobile application energy consumption using program analysis. In: Proceedings of the 35th International Conference on Software Engineering. 2013, 92–101
99. Li D, Lyu Y, Gui J, Halfond W G J. Automated energy optimization of HTTP requests for mobile applications. In: Proceedings of the 38th International Conference on Software Engineering. 2016, 249–260
100. Yang Z, Yang M, Zhang Y, Gu G, Ning P, Wang X S. AppIntent: analyzing sensitive data transmission in Android for privacy leakage detection. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security. 2013, 1043–1054
101. Haris M, Haddadi H, Hui P. Privacy leakage in mobile computing: tools, methods, and characteristics. 2014, arXiv preprint arXiv:1410.4978
102. Choudhary S R, Gorla A, Orso A. Automated test input generation for Android: are we there yet? In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering. 2015, 429–440
103. Martin W, Sarro F, Jia Y, Zhang Y, Harman M. A survey of app store analysis for software engineering. IEEE Transactions on Software Engineering, 2017, 43(9): 817–847
104. Sufatrio, Tan D J J, Chua T, Thing V L L. Securing Android: a survey, taxonomy, and challenges. ACM Computing Surveys, 2015, 47(4): 58
105. Li L, Bissyandé T F, Papadakis M, Rasthofer S, Bartel A, Octeau D, Klein J, Traon Y L. Static analysis of Android apps: a systematic literature review. Information and Software Technology, 2017, 88: 67–95
106. Sadeghi A, Bagheri H, Garcia J, Malek S. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. IEEE Transactions on Software Engineering, 2017, 43(6): 492–530



Tianyong Wu received his BS and PhD degrees from Xiamen University (XMU), China in 2011 and University of Chinese Academy of Sciences (UCAS), China in 2017, respectively, China. His research interests include software test generation and static analysis.

Xi Deng received the BS degree from Wuhan University, China in 2015. She is currently working toward the PhD degree at the University of Chinese Academy of Sciences, China. Her research interests include software testing and static analysis.

Jun Yan received his BS degree from University of Science and Technology of China (USTC), China in 2001 and his PhD degree from University of Chinese Academy of Science (UCAS), China in 2007, respectively. He is now an associate research professor at Institute of Software, Chinese Academy of Sciences (ISCAS), China. His research interests include program analysis and software testing. He is a senior member of China Computer Federation (CCF).

Jian Zhang is a research professor at the Institute of Software, Chinese Academy of Sciences (ISCAS), China. His main research interests include automated reasoning, constraint satisfaction, program analysis and software testing. He has served on the program committee of about 70 international conferences. He also serves on the editorial boards of several journals including Frontiers of Computer Science, Journal of Computer Science and Technology, IEEE Transactions on Reliability. He is a senior member of ACM, the IEEE, and a distinguished member of China Computer Federation (CCF).