WHOLE-SYSTEM ANALYSIS FOR UNDERSTANDING PUBLICLY ACCESSIBLE FUNCTIONS IN ANDROID

Nguyen Huu Hoang [(1)] [(2)] *, Lingxiao Jiang [(2)], Quan Thanh Tho [(1)]
[(1)] Ho Chi Minh City University of Technology, Viet Nam
[(2)] School of Information Systems, Singapore Management University, Singapore
Email: hhnguyen@smu.edu.sg

ABSTRACT

Android has become the most popular mobile operating system. Millions of applications, including many malwares, haven been developed for it. Android itself evolves constantly with changing features and higher complexities. It is challenging for application developers to keep up with the changes and maintain the compatibility of their apps across Android versions. Therefore, there are many challenges for application analysis tools to accurately model and analyze app behaviors across Android versions. Even though the overall system architecture of Android and many APIs are documented, many other APIs and implementation details are not, not to mention potential bugs and vulnerabilities. Techniques and tool supports are thus needed to automatically extract information from different versions of Android to help programmers understand system behaviors and APIs across different versions. This paper aims to address the need. It performs whole-system analysis for different versions of Android by using both backward and forward static analysis of intra-procedural and inter-procedural control-flow and data-flow graphs. It can collect information about functions in Android that can be invoked by applications, which are referred to as *publicly accessible functions* in this paper. Such information can help programmers better understand the ways in which their applications utilize system functions. We have analyzed Android versions 4.1.1, 4.2.2, 4.3, 4.4.4, 5.1.0, 6.0.1, and show basic statistics about the publicly accessible functions in different Android versions. We also use an example to illustrate that the information about publicly accessible functions can be useful in identifying unprotected system functions whose invocations may not be protected by proper permissions and may lead to security and privacy violations.

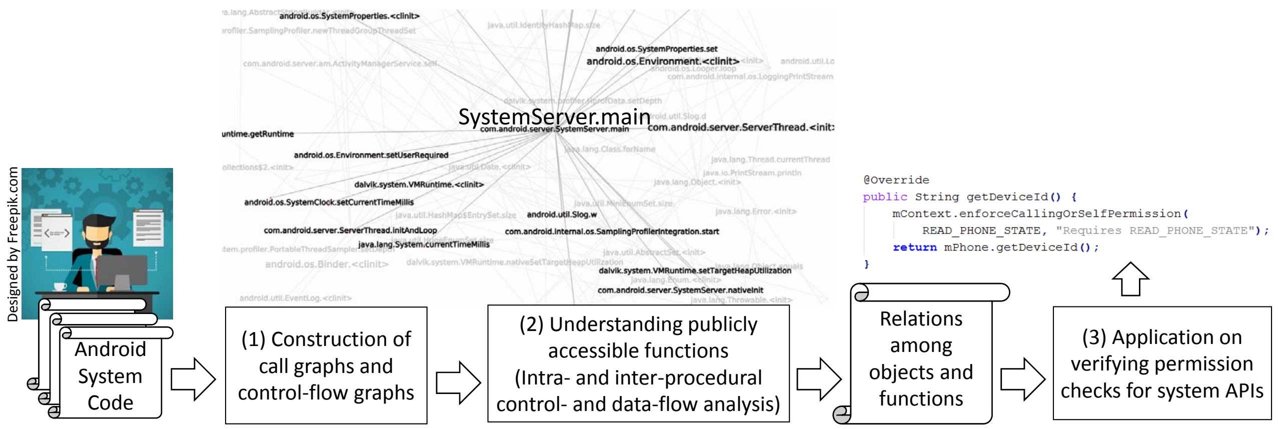
* This work is done when the first author is a visiting student in the School of Information Systems at Singapore Management University.

KEYWORDS: *android, call graph, control flow analysis, data flow analysis, program comprehension, permission check*

1. INTRODUCTION

Android now accounts for more than 80% of the global smartphone operating system (OS) market [8]. There have been more than 8 million mobile application programmers worldwide in 2014 [7]. To help programmers, especially novice ones, to develop applications for a mobile OS quickly, often a first step is to get them become familiar with the architecture of the mobile OS and all APIs available in the OS. Thus, programmers can structure their apps according the ways in which the OS manages apps and able to invoke needed OS functionalities via the APIs. Even though the overall system architecture of Android and many of its APIs are documented, many APIs have evolved much across different versions of the Android system from API level 1 to API level 24 for Nougat 7.0, exhibiting different behaviors and causing incompatibility across versions and inconsistencies between actual code behaviors and the documentations. There are also many undocumented APIs in the system that could be invoked by apps, not to mention many potential bugs and vulnerabilities that can be exploited by apps too.

Understanding system behaviors and APIs becomes even more challenging when Android faces the fragmentation problem: there are over 1000 brands and more than 24000 models of Android phones in 2015 [15]; the manufacturers of the brands and models often customize the Android system for their phones in different ways without sufficient documentations for programmers. Manually going through documents and source code to track and understand the relations among different system APIs and behaviors is very time-consuming and error-prone; such tasks should be facilitated by tools (e.g., code navigation via Android X Ref [6]) that can work across different versions and fragments of the system.


Designed by Freepik.com
SystemServer.main
Android System Code
(1) Construction of call graphs and control-flow graphs
(2) Understanding publicly accessible functions (Intra- and inter-procedural control- and data-flow analysis)
Relations among objects and functions
(3) Application on verifying permission checks for system APIs

```
@Override
public String getDeviceId() {
    mContext.enforceCallingOrSelfPermission(
        READ_PHONE_STATE, "Requires READ_PHONE_STATE");
    return mPhone.getDeviceId();
}
```

Fig 1. Approach Overview

In the area of static analysis of Android systems and applications, much work [4,12,16,19] that focuses on analyzing Android applications require expert knowledge and models of the Android systems, such as the lifecycle callbacks used by the system to manage individual app activities [1], the back stack used in the system to manage sets of running activities [2], etc. Such expert (and manual) modelling of the systems cannot keep up with the evolution and fragmentation of the systems, leading to inaccurate models and analysis results. So, there are also needs to automate the modeling of various versions of the Android system, and more generally, to automate the modeling of large-scale frameworks and libraries used for application development, so that analysis of apps can be more accurate.

The ultimate goal of this work is to achieve automated modeling of Android systems and facilitate programmers to understand system behaviors and APIs. One way towards this goal is to perform whole-program analysis of a version of Android together with an application, so as to avoid the need to manually construct a behavior model of the system before analyzing the app. This way has been considered in the literature and comes with many challenges (e.g., [18]).

This paper takes a first step to analyze whole Android systems in different versions, to automatically curate Android system functions that are accessible by applications. It provides a "navigation map" for different versions of Android systems for programmers to understand what functionalities in different versions of Android systems can be invoked by which APIs. Technically, we construct Java class hierarchies, call graphs and control-flow graphs for different versions of Android systems using the Soot analysis framework [17], and then identify public functions in Android that can be invoked by applications with suitable parameters that can be obtained by the applications too. Such curated functions can reveal to programmers how system APIs, documented or undocumented, can be invoked in applications, helping them to understand the capabilities of the APIs. We have analyzed six versions of Android: 4.1.1, 4.2.2, 4.3, 4.4.4, 5.1.0, 6.0.1, and curated publicly accessible functions from them. The numbers of publicly accessible functions generally increase with increasing version numbers and Android code sizes, from about 53K to 74K. Together with more analyses (e.g., checking whether permissions are added for publicly accessible functions to prevent unauthorized accesses), we aim to show that curated information about such functions can be useful in helping programmers understand system functions and revealing unprotected system APIs that may lead to security and privacy violations in Android.

The rest of the paper is organized as follows: Section 2 briefly introduces related work; Section 3 describes our approach; Section 4 presents our actual setups and evaluations, and discusses our limitations and possible future work; Section 5 concludes.

2. RELATED WORK

The work in this paper is closely related to much work in static analysis of Android systems and applications. Much work that focuses on analyzing Android applications require expert knowledge and models of the Android systems.

FlowDroid [4], was introduced in 2014, focuses on the private data leaks. It performs static analysis on Android app based on inter-procedural control-flow and data-flow. As a part of this work, they modeled the effects of callbacks by generating a dummy main method.

As shown by Li et al. [12], IccTA performs inter-component communication based taint analysis (ICC). They used a highly precise control-flow graph through instrumentation of the code of applications to detect ICC based privacy leaks.

CHEX [13] detects possible hijack-enabling flows through conducting low-overhead reachability tests on customized system dependence graphs. They model the vulnerabilities from a data-flow analysis perspective.

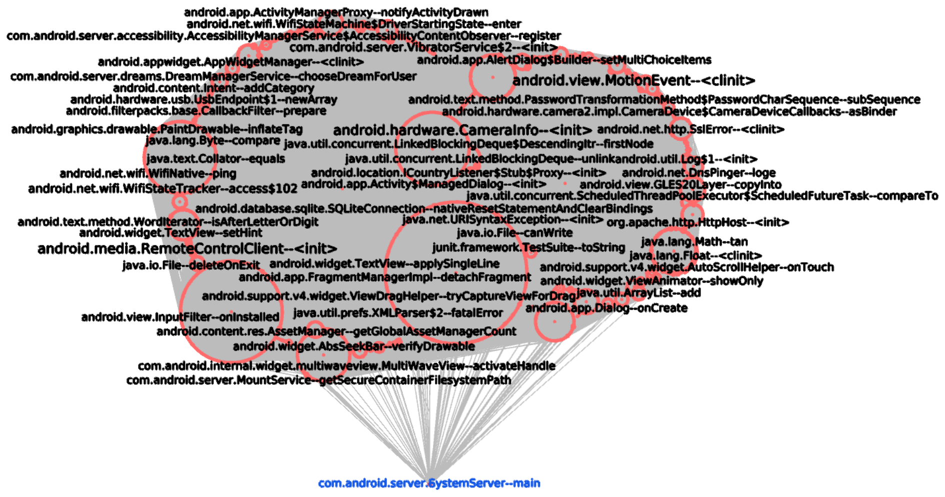


Fig 2. The whole-system call graph of Android version 4.4.4

In the study by Yang et al. [19], they presented a control-flow representation or user-driven callback behavior based on context-sensitive analysis of event handlers. They traverse context-compatible inter-procedural control-flow paths and identify trigger callbacks of the statements to perform graph reachability.

In 2015, Rasthofer et al. [16] described in detail the components of current state-of-the-art static and dynamic code analysis techniques of Android malware development. They emphasize the challenges for automatic malware analysis frameworks. These problems hinder the fully automatic detection of Android malicious components significantly.

Generally, the effectiveness of such analysis is prone to the inaccuracies of the manually constructed models of the systems. This paper tries to work towards automated modeling of Android systems.

Thus, a major objective of the work is performed whole-program analysis of any version of Android system, so as to reduce the need for manually produced behavior models before analyzing an application, and to make the analysis of the application more accurate. This objective on whole-program analysis has been considered in the literature. For example, Yan et al. [18] propose to extend Soot with summary-based analysis so as to scale up to whole-program analysis. StubDroid [3] generates summaries for system/library APIs before performing data-flow analysis for applications.

Those studies focus on building summaries and models for automated application analyses, not application development; their outputs are not intended for programmers to read, and may not be useful for programmers who are developing applications and want to understand system APIs and behavior models better.

The work on this paper focuses more on the understandability of the whole systems, although techniques used in other work can be adapted for our purposes, and the outputs from our work may be used for both application analysis and system analysis, e.g., checking proper permission controls in Android systems and applications [5,11].

3. APPROACH

3.1 Overview

At a high level, our technical idea is straightforward as illustrated in Figure 1 mainly consisting 3 steps.

(1) For a version of Android, we collect all of its code, identify all Java classes involved, construct call graphs for the whole Android system from the system's main entry point (`com.android.server.SystemServer`).

(2) Then, we identify potentially publicly accessible functions, and use data-flow and control-flow analysis based on the call graphs to identify objects and other functions that are needed to invoke each of the public functions. Such analysis will produce information on the dependencies among objects and functions and indicate possible paths in the code that link the dependent elements together.

(3) To illustrate the usefulness of such information, we follow the dependencies and paths to check whether publicly accessible functions are protected with certain permissions in the Android system (e.g., the sample code fragment in Figure 1 shows that `getDeviceId` is protected by the `READ_PHONE_STATE` permission), and detect possible unprotected system APIs that access system or private resources without proper permission checking. Such unprotected system APIs could be potentially invoked by applications without permissions, causing security and privacy issues.

3.2 Call Graph Construction

Statically constructing call graphs of the Android system faces many technical challenges, some of which are common for all Java programs, other are unique to Android. For example, Java reflection is often used to invoke dynamically loaded classes and methods whose signatures may be difficult to determine without accurate string analysis of possible method names. This is a common problem handled by the Soot static analysis framework [17] to some extends. Android also extensively uses event-driven programming and registered listeners to invoke various kinds of call backs to process messages. And, broadcasts (or, intents) can be sent by both the system and applications to each other to invoke different functionalities based on intent matching. Android also supports defining call backs and intents via XML configuration files, which incurs additional challenges for accurate static call graph construction. We do not address these challenges; instead, we reply on the capabilities of Soot to construct the call graphs starting from `SystemServer.main()`.

Android is a large system was built in many years, there is other challenge as the call graph is a huge graph (about one million edges and fifty thousand vertices with Android version 4.4.4). Figure 2 describes the complexity of whole-system call graph Android. It is difficult to load the data from call graph to memory and apply the pathfinding algorithms. We resolve this problem via NetworkX [14] as an intermediate library. It dumps the call graph memory to the special format is called **Pickle** to improve the access performance.

3.3 Publicly Accessible Android Functions

A basic condition for a function in the Android system to be publicly accessible is that the function should be a public method in Java. Also, the second more complex condition is that the objects needed for invoking the function should be either publicly accessible too or could be constructed easily. To check whether a function satisfies the two conditions, we need to recursively check whether all involved objects and functions are publicly accessible, which is essentially a backward data-flow analysis with heuristic filters based on coding and naming conventions.

The following is essentially a backward data-flow analysis:

(1) For objects of primary types, they can be easily constructed with random values, and we consider them publicly accessible.

(2) For an object that is a **public** field in a class, it is considered publicly accessible if one of the class's constructor and the parameters for the constructor are publicly accessible.
If the object is a **private** or **protected** field, it can still be publicly accessible when the class has a getter function whose name is the same as the field name.

(3) For a **public static** method that takes in primary types as parameters, the method can be easily invoked from outside of the Android system with randomly generated parameter values.

(4) Otherwise, we need to check whether it is easy to obtain all the objects needed for invoking the public method.

For example, if an object of type **T** is needed to invoke a public function **f**, then we need to check whether one of **T**'s constructor is publicly accessible, or whether an instance of **T** can be returned via some **`getter`** functions.[1] If either the constructor and the **`getter`** function requires additional parameters to be invoked, recursive checks are performed.

To simplify the analysis and get preliminary results, this paper considers a function to be publicly accessible as long as it is **public**. In addition, there are also many functions in the Android system (about 38-43%) that are not reachable from the `SystemServer.main` entry. We consider such functions are not intended to be invoked by the system itself, and are more likely designed for applications to use. So, we consider such functions publicly accessible as long as they are **public**.

3.4 Understanding of Android System Functions and Checking Permissions

We consider three kinds of information that may be useful in understanding a publicly accessible function if a programmer wants to invoke it properly:

(1) the information about all the dependencies, including objects and other functions needed, *before* invoking the function.

(2) the information about what happens, including objects affected and functions invoked, *after* the function is called.

(3) the information about what happens, including objects affected and functions invoked, *within* the function.

Item (1) can be collected using a backward flow analysis that identifies objects and functions that programmers need to construct or invoke before invoking the function.

Item (2) can be collected by an impact analysis that traverses control and data flows in a forward mode, together with an escape analysis and heuristics based on.

Item (3) can be collected by an impact analysis too, but focuses on inner workings of a function and may only

[1] E.g., a function named `getSomething` returns type **T**. These heuristics are neither sound nor complete; they are to simplify the analysis.

be needed if programmers want to understand the function's internals. Section 4 presents some summarized statistics about the collected information.

As a particular application of such information, we can apply it to address a security and privacy question: are all the publicly accessible functions that access system and/or private resources protected by proper permissions? Given a publicly accessible function **f**:

(1) if the objects and functions needed before invoking **f** do not have any permission protection.
(2) if the objects and functions within the invocation of **f** do not have any permission protection.
(3) if the invocation of **f** can change some external objects, then it is very likely **f** is not properly protected by a permission.

Based on the question mentioned above, we propose Algorithm 1 to detect the permission checking issues. This algorithm is a backward-flow analysis through the call graph paths to get all methods that have permission checking. Section 4.2.3 illustrates how such permission checking can be done with an example.

```
Result: Set methods has permission checking
Set setMethods = null;
Stack stackPaths = getCallGraphPaths();
while stackPaths NOT empty do
    Path currentPath = stackPaths.pop();
    while currentPath NOT empty do
        Method startingMethod = currentPath.pop();
        if startingMethod HAS permission checking and NOT in setMethods then
            setMethods.add(startingMethod);
        end
        StackStmt stackCallerStmts = locateCallerStatements(startingMethod);
        while stackCallerStmts NOT empty do
            Stmt callerStmt = stackCallerStmts.pop();
            if callerStmt HAS permission checking then
                Method callerMethod = callerStmt.getMethod();
                setMethods.add(callerMethod);
            else
                Stack stackRelatedVars = locateRelatedVars(callerStmt);
                Stack stackRelatedStmts =
                 locateStatementsOfRelatedVars(stackRelatedVars);
                while stackRelatedStmts NOT empty do
                    Stmt currentStmt = stackRelatedStmts.pop();
                    if currentStmt HAS permission checking then
                        Method currentMethod = currentStmt.getMethod();
                        setMethods.add(currentMethod);
                    end
                end
            end
        end
    end
end
```

Algorithm 1. Detect permission checking

4. EVALUATION

4.1 Setups

Our approach uses Soot to generate call graphs of Android systems and output them in the **Dot** file format. Then we use NetworkX library [14] to analyze the graphs, to traverse call graphs to find paths, analyze in-degrees, out-degrees and output graphs into **Pickle** file.

For Soot to construct call graphs for a version of Android, we feed it with all system jar files (including `android.jar`) in the Android SDK as distributed by Google. To analyze the Android system from version 4.1 to 4.4.4, we used all of **jar** files in `/system/framework` that extracted from Google Nexus 4 emulator of Genymotion tool [9]; these file included **Dex** file of Dalvik Virtual Machine. From version 5.0.0 and later, Android changed its architecture, replacing Dalvik Virtual Machine (VM) with Android Runtime (ART) and the **Dex** files were combined to Linux binary files (**ELF** file). Since Soot framework did not support Linux binary files, we used the Java bytecode of Android from Grepcode [10] for Android version 5.1.0 and 6.0.1.

Table 1. The generation time for each version

Version	Generated Duration (seconds)
4.1.1	31988
4.2.2	35790
4.3	41288
4.4.4	43133
5.1.0	80463
6.0.1	86521

The duration to generate and read the call graph files is a challenge issue. We generated the call graphs on our server with the specifications as Windows Server 2008 R2 64-bit, Intel(R) Xeon(R) E5540 2.53Ghz, RAM 64 GB. Table 1 presents the generated duration for each Android version. The generation time in the **Dot** format is approximate 9 hours for version 4.1.1 and up to 24 hours for version 6.0.1. Then, it takes about 10 minutes to read and convert the **Dot** file to **Pickle** via NetworkX, and the following graph traversals based on the **Pickle** files can be performed quickly.

4.2 Result Summary

Our evaluation addresses the following research questions:

(1) How the numbers of total and public methods changes for each Android version?
(2) How are the degree distributions of the methods?
(3) Could we find publicly accessible functions that are not properly protected by permission checks?

The next sections address each research question in detail.

4.2.1 Methods in Each Android Version

Comparing Android 4.4.4 with 5.1.0 (the gray line in Figure 3), the numbers of call graph methods increase 14.7%. The change rate is comparable between Android 4.1.1 and 4.4.4. The large change rates indicate the needs of tools to help programmers to understand the Android system functions. The numbers of edges also change significantly (e.g., Android 4.2.2 increased 10.8% and 4.4.4 increased 11.2%). These changes also contribute to the complexity of the Android system after each new version. The orange line in Figure 3 presents the number of public methods for each Android version. The public methods occupy about 51% of all system methods from Android 4.1.1 to 4.4.4. After Dalvik VM was replaced by ART from Android version 5.0, this ratio changed to 54% as our results on Android 5.1.0 and 6.0.1. These ratios demonstrate the difference between Dalvik VM architecture and ART architecture.

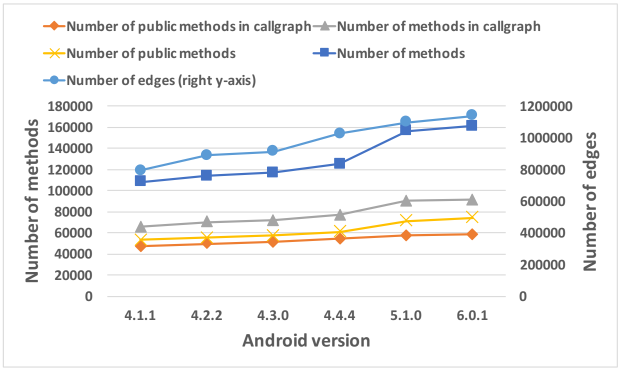
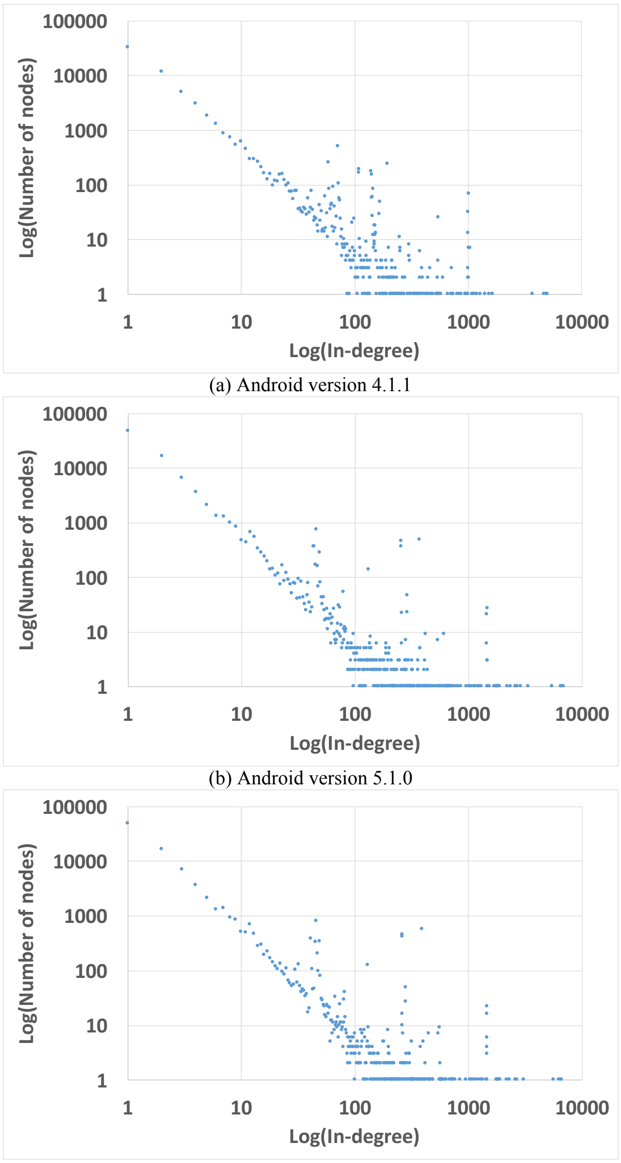


Fig 3. Numbers of Methods and Edges

4.2.2 Degrees of Methods

We use the interaction degrees among methods in call graphs to illustrate the complexity of Android functions. Figures 4 and 5 show the in-degree and out-degree distributions, respectively, for Android. The distributions exhibit power-law like relations, especially for degrees in the range of [1, 100]. The degrees in the range of [100, 10000] are distributed in a more arbitrary fashion. Samples of nodes of high in-degrees are `toString`, `append` methods in the general purpose Java `StringBuilder`, `Object` classes. More interestingly, nodes of high out-degree may not indicate high complexities of the involved methods. E.g., `equals`, `toString` methods in the Android `core.KeyValueMap`, `text.SpannableStringBuilder` classes have almost the highest out-degree, reflecting the fact that it deals with generic object types, but the code of these functions are often short and easy to understand. More complex methods are in fact those having tens of out-degrees and dealing with specific object types with rich contents, e.g., `parseIntent`, `getLastLocation` in the Android `Intent`, `LocationManagerService` classes.



(a) Android version 4.1.1

(b) Android version 5.1.0

(c) Android version 6.0.1

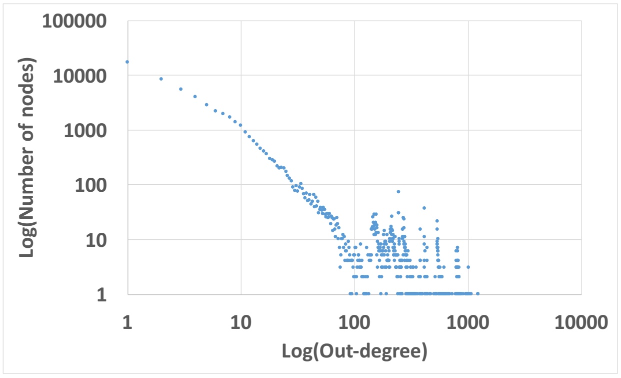
Fig 4. In-degree distributions of Android

4.2.3 Accessing system resources without permissions

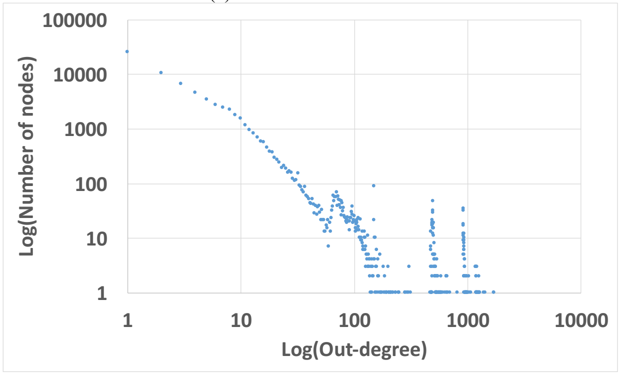
Information curated by our analysis could be applied to find potential security and privacy violations.

For example, consider the public method `setStreamVolume()` of system class `com.android.server.audio.AudioService`. Based on the call graph of Android version 4.3, we realize this public method invokes many other methods in the system class `AudioService` in a certain order, such as `ensureValidStreamType()`, `getDeviceForStream()`, `rescaleIndex()` and finally `sendVolumeUpdate()`. The method `sendVolumeUpdate()` could update the device volume without permission. So, it means the applications could access its device volume without any permission. This bug was fixed from Android 4.4, Google used method

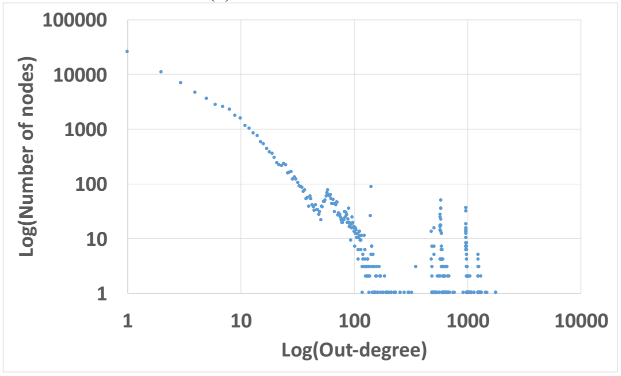
`AppOpsManager.noteOp()!=AppOpsManager.MODE_ALLOWED` to check permission before calling `rescaleIndex()`.



(a) Android version 4.1.1



(b) Android version 5.1.0



(c) Android version 6.0.1

Fig 5. Out-degree distributions of Android

4.3 Discussion

As the example in Section 4.2.3, our technique could be used to detect the vulnerabilities of the operating system Android. It's also more potential for detecting data leaks or support the developer understands Android system better.

Note that directly invoking publicly accessible functions in the system is not the only way for Android applications to invoke functionalities of the system. There are other ways, such as broadcasting an intent that will be matched and processed by systems, or using Java reflection to invoke functions dynamically, etc. These ways of invoking system functions are not yet detected as publicly accessible functions in this paper and are left for future work.

5. CONCLUSION

This paper takes a first step to analyze the whole Android system across various versions, with the intention to automate the modeling of system behaviors and facilitate more accurate and scalable whole-program analysis of Android applications. The work so far focuses on providing better understandability of system functions that are publicly accessible. Based on limited studies of six versions of the Android system from 4.1.1 to 6.0.1, we show that system functions change a lot across different versions; the numbers of publicly accessible APIs change from about 53K to about 74K. Also, the complexity involved in many APIs can be high based on their in-degrees and out-degrees. We also illustrate the potential usefulness of the studies by showing missed permission checks for a publicly accessible API that may violate security and privacy. In the near future, we will integrate the API dependencies and control/data-flow information into modern IDEs (e.g., Android Studio, Eclipse, IntelliJ IDEA) in a visual way to help programmers navigate through and understand the complex system APIs, and will automate the permission checks for discovering unprotected APIs that may lead to security and privacy violations.

REFERENCES

[1] Android Open Source Project. Android API guides: Activities. https://developer.android.com/guide/components/activities.html.

[2] Android Open Source Project. Android API guides: Tasks and back stack. https://developer.android.com/guide/components/tasks-and-back-stack.html.

[3] S. Arzt and E. Bodden. StubDroid: automatic inference of precise data-flow summaries for the android framework. In Proceedings of the 38th International Conference on Software Engineering, ICSE, pages 725–735, Austin, TX, USA, May 14–22 2016.

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, page 29, Edinburgh, United Kingdom, June 9–11 2014.

[5] A. Bartel, J. Klein, M. Monperrus, and Y. L. Traon. Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android. IEEE Trans. Software Eng., 40(6):617–632, 2014.

[6] R. Chiossi. Android x ref. http://androidxref.com/, 2016.

[7] Evans Data Corporation. Mobile developers now number 8.7 million worldwide. http://www.fiercewireless.com/developer/ evans-data-mobile-developers-now-number-8-7-million-worldwide, June 2014.

[8] Gartner. Worldwide smartphone sales grew 9.7 percent in fourth quarter of 2015. http://www.gartner.com/newsroom/id/3215217, February 2016.

[9] Genymotion. Fast and easy android emulator. https://www.genymotion.com/.

[10] Grepcode. Java source code search 2.0. http://repository.grepcode.com/java/ext/com/google/ android/android/.

[11] S. M. Kywe, Y. Li, K. Petal, and M. Grace. Attacking android smartphone systems without permissions. In 14th Annual Conference on Privacy, Security and Trust, 2016. To appear.

[12] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. IccTA: Detecting inter-component privacy leaks in android apps. In 37th IEEE/ACM International Conference on Software Engineering, ICSE, volume 1, pages 280–291, Florence, Italy, May 16–24 2015.

[13] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In Proceedings of the 2012 ACM conference on Computer and communications security, pages 229–240. ACM, 2012.

[14] NetworkX. High-productivity software for complex networks. http://networkx.github.io/, 2016.

[15] OpenSignal. Android fragmentation visualized. http://opensignal.com/reports/2015/08/android-fragmentation/, August 2015.

[16] S. Rasthofer, I. Asrar, S. Huber, and E. Bodden. How current android malware seeks to evade automated code analysis. In Information Security Theory and Practice - 9th IFIP WG 11.2 International Conference, WISTP, pages 187–202, 2015.

[17] Sable Research Group. Soot: A framework for analyzing and transforming java and android applications. https://sable.github.io/soot/, 2016.

[18] D. Yan, G. Xu, and A. Rountev. Rethinking Soot for summary-based whole-program analysis. In ACM SIGPLAN International Workshop on the State Of the Art in Java Program Analysis @ PLDI, pages 9–13, 2012.

[19] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in Android applications. In International Conference on Software Engineering, pages 89–99, 2015.

PHOTOS AND INFORMATION

	Nguyen Huu Hoang received his Bachelor's degree in Electronics and Telecommunications, Ho Chi Minh city University of Science (HCMUS), Vietnam in 2013. He became Master student in Computer Science, Ho Chi Minh City University of Technology (HCMUT), Vietnam in 2014. He had one year in School of Information Systems, Singapore Management University as a visiting student. His current research interests include programming languages, program analysis and mobile security. He also has experience in mobile application development.
	Jiang Lingxiao received his Bachelor's degree in Information Science and Master's degree in Applied Mathematics from the School of Mathematical Sciences at Peking University (1996-2003). He completed his Ph.D. with Prof. Zhendong Su in the Department of Computer Science at University of California, Davis (2003-2009). He joined the faculty of School of Information Systems at Singapore Management University in November 2009 as a Professor. He interested in programming languages and data mining, looking for research ideas and techniques that can help solve software engineering and security problems.
	Quan Thanh Tho is an Associate Professor in the Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology (HCMUT), Vietnam. He received his B.Eng. degree in Information Technology from HCMUT in 1998 and received Ph.D. degree in 2006 from Nanyang Technological University, Singapore. His current research interests include formal methods, program analysis/verification, the Semantic Web, machine learning/data mining and intelligent systems. Currently, he heads the Department of Software Engineering of the Faculty. He is also serving as the Chair of Computer Science Program (undergraduate level).