Android Repository Mining for Detecting Publicly Accessible Functions Missing Permission Checks

***Abstract*—Android has become the most popular mobile operating system. Millions of applications, including many malware, haven been developed for it. Even though its overall system architecture and many APIs are documented, many other methods and implementation details are not, not to mention potential bugs and vulnerabilities that may be exploited. Manual documentation may also be easily outdated as Android evolves constantly with changing features and higher complexities. Techniques and tool supports are thus needed to automatically extract information from different versions of Android to facilitate whole-system analysis of undocumented code. This paper presents an approach for alleviating the challenges associated with whole-system analysis. It performs usual program analysis for different versions of Android by control-flow and data-flow analyses. More importantly, it integrates information retrieval and query heuristics to customize the graphs for purposes related to the queries and make whole-system analyses more efficient. In particular, we use the approach to curate functions in Android that can be invoked by applications in either benign or malicious way, which are referred to as *publicly accessible functions* in this paper, and with the queries we provided, identify functions that may access sensitive system and/or user data and should be protected by certain permission checks. Based on such information, we can detect some publicly accessible functions in the system that may miss sufficient permission checks. As a proof of concept, this paper has analyzed six Android versions and shows basic statistics about the publicly accessible functions in the Android versions, and detects and verifies several system functions that miss permission checks and may have security implications.**

***Keywords*-android; program comprehension; program analysis; information retrieval; call graph; dependency;**

I. Introduction

Android now accounts for more than 80% of the global smartphone operating system (OS) market [1]. The system architecture of Android and many of its APIs have evolved much across different versions of the Android system from API level 1 to API level 25 for Nougat 7.1, exhibiting inconsistent behaviors and leading to incompatibility across versions. Applications running on top of the Android system may invoke the APIs available from the system, but exhibiting different behaviors depending on the implementation of the APIs and their versions. Malicious applications may even invoke both documented and undocumented methods in the system, and exploit potential bugs and vulnerabilities in the system, causing harms to smartphone users.

Understanding Android system behaviors and APIs becomes even more important and challenging when Android evolves with much diversity across its ecosystem and faces the fragmentation problem [2]. The size of the Android system has also grown into tens of million lines of code including code generated during build, imposing significant technical challenges when performing whole-system program analysis of all code involved.

In the area of static analysis of Android systems and applications, much work [3]–[5] that focuses on analyzing Android applications require expert knowledge and models of the Android systems, such as the lifecycle callbacks, the back stack, etc. Such expert and manual modelling of the systems cannot keep up with the evolution and fragmentation of the systems. There are 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. Whole-system analysis on the Android system and its benefits for automated modeling of the system and facilitating programmers to understand system behaviors and APIs have been considered in the literature, but it comes with many challenges (e.g., [6]).

This paper takes a step to circumvent the challenges by combining program analysis techniques with information retrieval techniques to analyze the whole Android system across different versions. The paper aims to build a framework that can automatically curate the methods in the Android system that can be accessible by applications, providing a "navigation map" of the methods for different versions of Android systems for programmers to understand. The framework has two major interleaved components: one is a program analysis component built with the Soot analysis framework [7] that constructs basic Java class hierarchies, call graphs, and control-flow graphs, and performs data flow analysis along the graphs for different versions of Android systems; the other is a text-based information retrieval and graph customization component that can take in user-provided query words to tailor the graphs for the users' needs. In particular, as an application of the framework, we consider the methods in the system that access sensitive system-level and/or user data and thus often require certain permission protections. With the query words we provided (such as "permission", "allow" for permission checking, and "telephony", "location" for particular sensitive data), our information retrieval and graph customization component can slice the outputs from the program analysis component in a way that is more compact, facilitating more efficient identification and analysis the relevant methods, and detecting methods that may miss permission checks but are *publicly accessible*.

We have analyzed six versions of Android (4.1.1, 4.2.2, 4.3, 4.4.4, 5.1.0, 6.0.1) with our framework, and curated publicly accessible functions from them. The numbers of publicly accessible functions generally increase with increasing version numbers and Android code sizes, from more than 50K to more

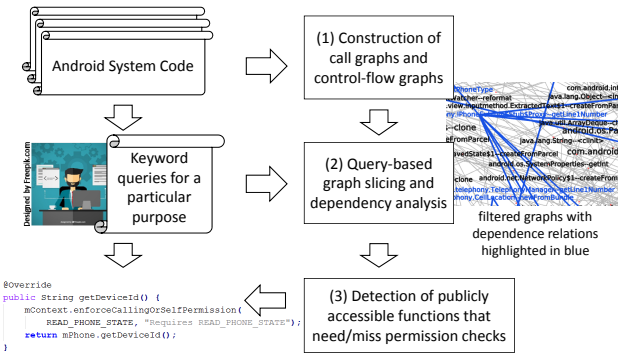

Fig. 1. Approach Overview

than 70K. The large numbers of publicly accessible functions have more potential venues for unauthorized access to sensitive information in the system. As a proof of concept, with the query words we used together with framework, we show that in the versions of 4.1.1 and 4.2.2, there are quite a number of methods (e.g. the system class android.media.AudioManager has method isSpeechRecognitionActive) that are not sufficiently protected by permissions and may be invoked by malicious applications to obtain sensitive information about the system and users.

Our novel technical contribution lies in the integration of information retrieval techniques with program analysis techniques to improve the scalability of the analysis and allow customized heuristics based on user-provided query words for tailored whole-system analysis.

II. 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. For example, FlowDroid [4] detects private data leaks. It performs static analysis on Android app based on interprocedural control flows and data flows. IccTA [3] performs inter-component taint analysis. It uses a highly precise control-flow graph through instrumentation of the code of applications to detect inter-component flows. CHEX [8] detects possible hijack-enabling flows through low-overhead reachability tests on customized system dependence graphs. Yang et al. [9] present a control-flow representation and analysis of user-driven callback behavior based on context-sensitive analysis of event handlers. Rasthofer et al. [5] describe various static and dynamic code analysis techniques of Android malware and emphasize the challenges that hinder automated malware analysis, such as lack of complete and accurate models of the systems and scalability difficulties in performing whole-system analysis.

This paper tries to work towards (semi-)automated whole-system modeling and analysis of Android systems and applications. This objective on whole-program analysis has been considered in the literature. For example, Yan et al. [6] propose to extend Soot with summary-based analysis so as to scale up to whole-program analysis. StubDroid [10] generates summaries for system/library APIs before performing data-flow analysis for applications. To facilitate whole-system taint analysis, SuSi [11] uses machine learning techniques to discover and categorize data sources and sinks in the Android framework automatically. Our approach in this paper is also for whole-system modeling and analysis, but utilizes information retrieval techniques and user-provided keyword queries to make analysis much efficient and targeted for a specific purpose related to the queries. For example, our work may be used for detecting functions in the Android system that may be invoked by applications and access sensitive data but miss sufficient permission checks.

III. Approach

A. Overview

As shown in Figure 1, our approach consists 3 main steps.

(1) For a version of Android, we collect all of Java classes involved, construct call graphs for the whole Android system, and the control flow graphs for individual methods.

(2) Then, based on query keywords from users (e.g., "phone device id"), we identify potentially relevant methods using information retrieval techniques, and using NetworkX to slice the graphs to keep only the relevant methods while maintaining their connectivity in the sliced graphs. Then, further data-flow and control-flow analysis of the methods based on the sliced graphs are performed to produce likely dependence relations among the objects and function calls needed for invoking each of the methods, which are represented as a dependency graph.

(3) Also based on queries related to permission checking, we identify potentially publicly accessible functions, and check whether there are permission controls for the objects and function calls needed to invoke a publicly accessible function. If there is a possible permission check in the paths in the publicly accessible function's dependence graph, we assume the function is sufficiently protected by permissions in Android. For example, the sample code fragment in Figure 1 shows that `getDeviceId` is protected by the `READ_PHONE_STATE` permission.

Steps (2) and (3) are both based on information retrieval techniques and customization of the graphs. Combining them together, we may detect system functions that may access sensitive data but without sufficient permission protections.

B. Implementation Details

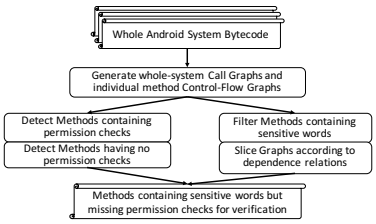

Fig. 2. Detecting Sensitive Methods Missing Permission Checks

Figure 2 illustrates the technical steps in more details.

1) We generate the whole-system call graphs and the control flow graphs for individual methods via Soot [7]. This paper does not consider many challenges (e.g., modeling event handlers, back stack of activities, intent broadcasting, XML configurations, etc.) in constructing accurate call graphs for Android. The graphs are exported to NetworkX's pickle format [12] for more efficient analysis.

```
Data: listOfPermissionWords = {"permission", "check", "mode", "allow"};
Data: callGraph = the call graphs of the whole Android system;
Result: Set of methods in call graphs containing permission query words.
Set allMethods = callGraph.getAllMethods();
Set methodsContainPermissionChecks = ∅;
// Identify methods directly containing permission checks:
foreach method in allMethods do
    Body bodyMethod = method.getBody();
    if bodyMethod CONTAIN a word in listOfPermissionCheckWords then
        methodsContainPermissionChecks.add(method);
    end
end
Set methodsContainPermisisonCheckClosure = ∅;
Set checkedMethods = methodsContainPermissionChecks;
// Identify all callers that are indirectly guarded by their callees:
while checkedMethods IS NOT empty do
    currentMethod = checkedMethods.pop();
    methodsContainPermisisonCheckClosure.add(currentMethod);
    Set callerMethods = callGraph.getCallerMethods(currentMethod);
    foreach caller in callerMethods do
        if caller is NOT in methodsContainPermisisonCheckClosure then
            checkedMethods.add(caller);
        end
    end
end
return methodsContainPermisisonCheckClosure;
```

Algorithm 1: Detecting methods containing permission checks.

2) We use query words ("permission", "check", "mode", "allow") as a heuristic to identify permission checks in Android code in a path- and context-insensitive way. If a method body has a function call or a conditional expression that involves identifier names containing a query word, we assume it has a permission check; and the caller of the method would be assumed to be guarded by the permission check too. Algorithm 1 is applied to detect methods contain permission check in whole-system call graph. Then, we get the difference between the set of all methods and the set of methods containing permission checks, and treat the methods in the difference set as the methods missing permission check.
3) We also use query words provided by users (e.g., "location") to identify methods that may be relevant for sensitive data. The matching technique is based on information retrieval used in the literature ([13]–[16], e.g., similarity measurement based on term-frequency/inverse-document-frequency). Algorithm 2 filters methods based on whether they can match a sensitive query word. [1]
4) Finally, we take the intersection of the results from (2) and (3) to verify whether the methods in the intersection may indicate methods that may access sensitive data and be invoked by applications without sufficient permissions.

IV. Evaluation

A. Setups

For Soot to construct call graphs for a version of Android, we feed it with `jar` files in the Android SDK. For versions from 4.1.1 to 4.4.4, we used all of `jar` files in /system/framework converted from the `dex` files in the Genymotion Nexus 4 emulator [17]. Since version 5.0, Android changed its architecture, and the `dex` files were combined to Linux binary (`ELF`) files.

[1]Methods that are in the dependence slice of a sensitive method can also be included as matching methods. Our prototype implementation does not include the dependencies, which will affect the later permission checking results; we leave the evaluation of various query matching heuristics for future work.

```
Data: sensitiveWords = {"account", "bluetooth", "contact", "database", "camera",
      "fingerprint", "usb", "location", "media", "audio", "volume", "wifi", "http",
      "nfc", "storage", "notification", "voice", "keystore", "telecom",
      "telephony", "radio", "gsm", "cdma", "sms"};
Data: callGraph = the call graphs of the whole Android system;
Result: Set of methods containing sensitive words
Set methodsContainSensitiveWords = ∅;
foreach method in allMethods do
    if method.getBody() CONTAIN a word in sensitiveWords then
        methodsContainSensitiveWords.add(method);
    end
end
return methodsContainSensitiveWords;
```

Algorithm 2: Detecting methods containing sensitive words

Since Soot did not support `ELF`, we used the Android packages from Grepcode [18] for versions 5.1.0 and 6.0.1.

B. Result Summary

Our evaluation provides preliminary answers to (1) basic statistics about methods in different versions of Android, and (2) more importantly, whether we can find publicly accessible functions that access sensitive system/user data without permission checks.

1) Method Statistics in the Android Versions: The total numbers of methods (the blue line with squares in Figure 3) increase along versions, ranging from about 110K to more than 160K. More than 50% of the methods are public methods, much more than what are documented in Android API/SDK documents, implying that there may be much more ways, either benign or malicious, to invoke system functionality.

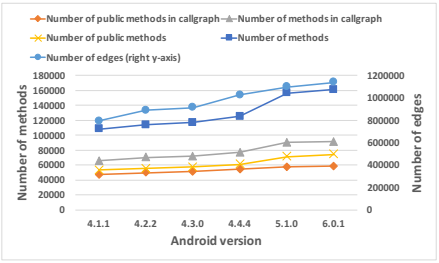

Fig. 3. Numbers of Methods and Edges

We use the interaction degrees among methods in call graphs to illustrate the complexity of Android functions. Figure 4 shows the in-degree and out-degree distributions for Android 4.1.1. The distributions exhibit power-law like relations, especially for degrees in the range of [1, 100]. Other versions exhibit similar distributions. Sample methods of high in-degrees are `toString`, `append` in the general purpose `StringBuilder`, `Object` classes. Interestingly, methods of high out-degree may not be of high complexity. 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 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.

2) Accessing sensitive data without permissions: Information curated by our prototype tool with the specific query words could be applied to find potential security and privacy violations. Based on preliminary analysis we performed for

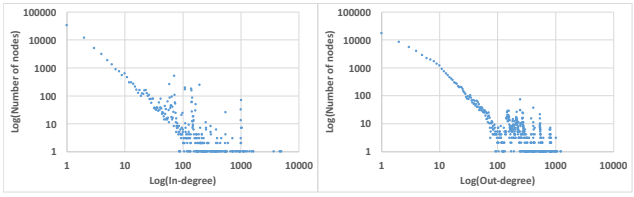

Fig. 4. Method call degree distributions of Android version 4.1.1

Android 4.1.1 and 4.2.2, we provide three examples here to illustrate the capability of our approach for detecting sensitive methods missing permission checks.

The first example is the public method `setStreamVolume()` of a system class `com.android.server.audio.AudioService`. This method invokes many other methods in AudioService in a certain order, such as ensureValidStreamType(), getDeviceForStream(), rescaleIndex(), and sendVolumeUpdate(). Along the call chain, there is no any permission check. So, an applications could change a device volume without any permission. This potential security bug was fixed since Android 4.4.

The second example involves four publicly accessible methods missing permission checks in a system class android.location.Country: getCountryIso, getSource, describeContents, and hashCode. Especially, an application could read the system country code (sensitive although not critical information) without any permission check via getCountryIso. In latest Android version 7.1.1, these problems are still there.

The third example, the method isSpeechRecognitionActive of a system class android.media.AudioManager, provides the speech recognition status without permission checks. The method isSpeechRecognitionActive was removed since Android version 4.4.4.

C. Discussion

Our approach for detecting methods missing permission checks is based on heuristic queries; it is neither sound or complete. When it reports a method missing permission checks, we verify it by manually constructing a simple application without any permission to invoke the method. The application prepares the necessary objects for invoking the method, by normal method calls and/or Java reflection. There are many more ways for Android applications to invoke system functionalities, such as broadcasting an intent that will be matched and processed by the system. We leave studies on automated construction of method calls to test app behavior for future work.

More broadly, our approach, combining program analysis with information retrieval, may be used to customize graph constructions and whole-system analysis, to make them more efficient and adaptable for queries beyond permission checking. It may also be useful for both application and system developers to navigate through the API jungle in the Android system better.

V. Conclusion

This paper proposes an approach that utilizes information retrieval queries to customize program analysis for the whole Android system across various versions, with the intention to enable more flexible and scalable whole-system analysis that can be tailored for a specific need.

Based on limited studies of six Android versions, we show that the numbers of system functions change a lot and there are more than 50% methods that may be publicly accessible, leaving many venues for potential security and privacy violations. We then illustrate the potential usefulness of our approach by using queries related to permission controls on sensitive system/user data to reveal publicly accessible functions that may access sensitive data without permission checks.

In the near future, we will improve the algorithms and implementation to provide more flexible configurations to customize graphs and program analysis for various kinds of queries to help developers understand the complex Android system functions better.

References

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

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

[3] 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*, vol. 1, Florence, Italy, May 16–24 2015, pp. 280–291.

[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)*, June 9–11 2014, p. 29.

[5] 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*, 2015, pp. 187–202.

[6] 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*, 2012, pp. 9–13.

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

[8] 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*. ACM, 2012, pp. 229–240.

[9] 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*, 2015, pp. 89–99.

[10] 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*, Austin, TX, USA, May 14–22 2016, pp. 725–735.

[11] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," in *21st Annual Network and Distributed System Security Symposium, NDSS*, 2014.

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

[13] S. Wang, D. Lo, Z. Xing, and L. Jiang, "Concern localization using information retrieval: An empirical study on linux kernel," in *18th Working Conference on Reverse Engineering*, Oct 2011, pp. 92–96.

[14] A. Marcus and S. Haiduc, *Text Retrieval Approaches for Concept Location in Source Code*. Springer, 2013, pp. 126–158.

[15] D. Poshyvanyk and A. Marcus, "Combining formal concept analysis with information retrieval for concept location in source code," in *15th International Conference on Program Comprehension (ICPC)*, Banff, Alberta, Canada, June 26-29 2007, pp. 37–48.

[16] B. Dit, M. Revelle, and D. Poshyvanyk, "Integrating information retrieval, execution and link analysis algorithms to improve feature location in software," *Empirical Software Engineering*, vol. 18, no. 2, pp. 277–309, 2013.

[17] Genymotion, "Fast and easy android emulator," https://www.genymotion.com/.

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