Reevaluating Android Permission Gaps with Static and Dynamic Analysis

Haoyu Wang*, Yao Guo*, Zihao Tang*, Guangdong Bai†, Xiangqun Chen*
*Key Laboratory of High-Confidence Software Technologies (Ministry of Education),
School of Electronics Engineering and Computer Science, Peking University, Beijing, China
†National University of Singapore, Singapore
{howiepku, yaoguo, tangzihao, cherry}@pku.edu.cn, baiguangdong@nus.edu.sg

***Abstract*—Recent studies on the Android permission system have found that there exists a permission gap between the requested permissions and permissions actually used in an Android app. However, current approaches face some challenges when detecting such permission gaps in Android apps due to the limitation of static analysis techniques. This paper proposes a novel approach to detect permission gaps in Android apps and determine the precise set of permissions that an app needs to run correctly. Our approach includes a static analysis technique to extract permission usage information from API invocations, and a dynamic testing technique to test and monitor the runtime permission usage behaviors of apps. By combining static analysis and dynamic testing, our approach can detect significantly more permission usage information compared to static analysis, indicating that our approach could improve the detection accuracy and reduce the false positives in permission gap detection. We have implemented a prototype to study more than 1,000 popular apps from Google Play. The results show that our approach could detect on average 30% more permissions that are used in apps, while more than 8% of the overprivileged apps detected by previous approaches are false positives.**

***Keywords*-Android; Permission; Overprivilege; Dynamic Analysis; Static Analysis**

I. Introduction

Android has dominated the smartphone market, with more than 1.5 million devices activated daily [1]. Meanwhile, a wide variety of feature-rich mobile apps have been developed. Currently, the number of Android apps in the Google Play Market has surpassed the 1.6 million mark [2].

Android uses install-time permissions to control accesses to system resources. Each app declares what permissions it requires, and users will be notified during installation. Due to the large number of permissions and undocumented APIs in the Android system, the permission specification for Android is very complicated, leading to many cases of improper permission requests for some developers.

The attack surface of apps can be reduced if the developers follow *the principle of least privilege* (POLP) in app development [3]. Previous studies [4, 5, 3, 6] have analyzed the requested permissions and permissions actually used in Android apps. They found that there exist *permission gaps* in many Android apps.

A typical permission gap is called *permission overprivilege*, where an app requests more permissions than it actually uses. One most recent work [6] shows that more than 85% apps in vendor customized phones suffer from permission overprivilege. Intuitively, this gap can be identified by building a permission map that identifies what permissions are needed for each API call, and then static analysis can be used to determine what API invocations an app makes.

However, based on pure static analysis, existing techniques face challenges to achieve high coverage in detecting used permissions, thus leading to false positives in detecting permission gaps. First, it is a known challenge for static analysis to handle dynamic features such as Java reflection calls and dynamic loading, which are widely used in Android apps, such that it may miss the invocations of some APIs. Second, the permission map is incomplete and none of the existing approaches could obtain a complete coverage of Android APIs. Third, the permissions for some API invocations are relevant to their parameters and contexts, which are difficult to determine using static analysis.

In this paper, we propose a novel approach to detect permission gap in Android apps and attempt to determine the precise set of permissions that an app needs to run properly. Our approach uses dynamic testing together with static analysis technique to improve coverage and accuracy.

In particular, we use static analysis to calculate the permission usage of Android API invocation, Intent usage and Content Provider usage. Meanwhile, we introduce a new dynamic testing process, in which we instrument the permission checking method in Android framework and monitor the dynamic permission usage information at runtime. Comparisons between the permission usage sets detected statically and dynamically show that they can complement each other to achieve significant high coverage of used permissions and low false positive rate when detecting permission gaps.

To the best of our knowledge, we are the first to use both static and dynamic techniques to detect the permission gap in Android apps. We have implemented a prototype system to study more than 1,000 popular apps from Google Play. The results show that our approach could detect more than 30% used permissions on average compared to previous approaches, and we find that more than 8% overprivileged apps detected by previous approaches are false positives.

II. Background and Challenges

In this section, we first introduce the permission gap problem, define the terms used in this paper, and then discuss the challenges in permission gap detection.

A. The Permission Gap Problem

The Android framework exports a large number of APIs to app developers. To make the problem even worse, many of these APIs are undocumented. As it is shown by a previous study [5], there are more than 16,000 undocumented APIs that require permissions for different versions of Android. Consequently, it is difficult for developers to declare a precise set of permissions that they use in their apps, even for experienced developers.

In one of the first studies of permission gaps in Android apps, Felt *et al.* [4] find that many apps request more permissions than they need, which they call "*permission overprivilege*". The potential cause of permission overprivilege is that a large number of APIs are undocumented and even some APIs are wrongly documented, thus developers tend to declare more permissions than those actually needed by the app to prevent app crash.

Declaring unused permissions would introduce security risks. For example, previous study [3] suggests that the unused permissions can be leveraged by malware to achieve malicious goals in several ways, such as code injection and return-oriented programming (ROP) attack [7]. Identifying permission gaps not only could help make apps more robust, but also reduce the attack surface [8].

B. Definitions

We define the following terms related to Android permissions, which will be used throughout this paper:

- ***Permissions Requested (*PR*)***. For Android apps, each app must declare which permissions it requires in its manifest, and users will be notified during installation. We use *permissions requested* to define the explicit permissions declared in the corresponding manifest file of each app.
- ***Permissions Used (*PU*)***. Although each app declares a list of permissions, there is no guarantee that the app will use exactly this set of permissions when they are executed. We defined *permissions used* as the permissions actually used by an app, which reflect the actual functionalities of an app. The list of permissions used can be identified through static analysis of the bytecode of an app, or through dynamic analysis to track all permission requests at runtime.
- ***Permission Gaps (*PG*)***. When permissions requested are not exactly the same as permissions used for a given app, it indicates that there exists a *permission gap* between the requested permissions and the permission actually used (either in source code, or during execution).
- ***Permission Overprivilege***. Permission Overprivilege occurs when an app declares more permissions than it uses. Google requires app developers to follow the principle of least privilege in app development. According to previous work [3], requesting redundant permissions will increase the attack surface of an app. The overprivilege scenario have also been studied in other previous work [4, 5, 6].

C. Challenges in Permission Gap Detection

Recent studies [3, 5, 4] have proposed approaches to detect permission gaps in Android apps. The typical process involves building a permission map that identifies what permissions are needed for each API call, and then performing static analysis to identify permission-related API calls.

Although existing approaches have shown many interesting results, they usually apply only static analysis and rely heavily on the permission map. This brings several challenges when we attempt to obtain a complete and accurate list of used permissions, which makes permission gaps detection more difficult and inaccurate.

1) Java Reflection: Reflection is an important feature of Java, because it provides a program the ability to inspect classes, methods, interfaces and fields at runtime, without knowing the names of the classes and methods in prior. SlicingDroids [9] finds that more than 57% of apps make use of reflection APIs. Methods can be implicitly invoked with $java.lang.reflect.Method.invoke()$ or $java.lang.reflect.Constructor.newInstance()$. However, it is also a challenging problem for static analysis techniques. Because static analysis cannot handle Java Reflection well, some permissions actually used by apps cannot be detected merely by static analysis.

2) Dynamic Loading: Dynamic loading is widely used in Android apps, which provides apps with the ability to install software components at runtime. Specifically, Android apps could leverage the *DexClassLoader* feature to load classes from embedded .jar and .apk files. This feature makes static analysis impossible since the loaded classes are only known at runtime. Because static analysis techniques could not handle dynamic loading, it may lead to false positives when detecting permission gaps.

3) API coverage: None of the existing approaches could obtain a complete coverage of Android APIs, especially with the rapid change of the Android framework. STOWAWAY [4] could achieve 85% coverage of Android APIs, while COPES [3] could only detect 71 of 115 permissions in the Android 2.2 framework.

4) Context-related Permission Mapping: The permissions for some APIs are related with their parameters and contexts, which are difficult to determine using static analysis. For example, the Android API *"android.app.NotificationManager.notify"* requires the *"VIBRATE"* permission only if *"Notification.vibrate"* is set for

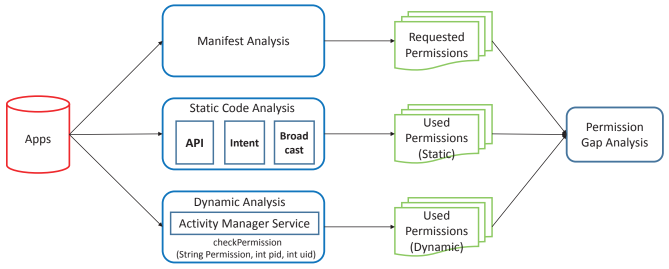

Figure 1. Permission usage analysis framework for Android apps

the Notification parameter, otherwise it does not need this permission.

In order to address these challenges, we introduce a hybrid analysis technique which combines dynamic analysis with static analysis. On one hand, similar to existing approaches [4, 3, 5], static permission-related invocation analysis is used to identify permission-related API calls, Intents and Content Providers. On the other hand, we introduce dynamic techniques, including dynamic fuzzy testing and logging analysis, to activate possible execution paths and monitor dynamic permission requests at execution time. Dynamic analysis could potentially address the challenges mentioned above, such as identifying the permissions used with dynamic behaviors (Java Reflection and Dynamic Loading), and determining the context-related permission mapping.

III. Permission Analysis Method

Figure 1 shows the key components of our permission analysis method. We perform analysis on Android apps in three different level: (1) *Manifest analysis* is used to retrieve the requested permissions (*PR*) in the analyzed app; (2) *Static code analysis* is used to generate the permissions used (*PUs*) by each app through inspecting the permission related APIs, and (3) *Dynamic analysis* is performed to collect more permissions used at runtime (*PUd*). We can then calculate the total permission used (*PU*) by merging *PUs* and *PUd*. Finally, we conduct a *permission gap analysis* by comparing the difference of PR and PU.

A. Manifest Analysis

In order to access specific resources, app developers must declare permissions in the manifest file. We inspect the $<uses-permission>$ elements in the manifest file to obtain *PR*. Note that we only consider the usage of permissions defined in the standard AOSP[1] framework, while the usage of user-specific permissions is not considered in this paper.

B. Static Analysis

For Android apps, three types of operations are permission-related: (1) explicit calls to standard Android APIs that lead to the $checkPermission$ method, (2) methods involving sending/receiving Intents, and (3) methods involving management of Content Providers.

We implement a similar method introduced by STOWAWAY [4] and leverage the permission map it provided to determine which permissions are actually used. We implement a lightweight disassembler based on DEX bytecode that can extract all API invocations, Intents, and Content Providers information from an app. To extract API invocations, we parse the disassembled DEX files and identify all invocations to Android APIs. We also track classes that inherit methods from Android classes to differentiate between invocations of app-defined methods and Android-defined methods. To identify sensitive Intents, we detect Intents that require permissions. As to Content Providers, we extract strings that can be used as Content Provider URIs.

After we extract all these permission-related information, we have got the permission usage of APIs ($PUs-a$), Intents ($PUs-i$) and Content Providers ($PUs-c$) based on the permission map, respectively. We calculate statically identified used permissions (PUs) as follows:

$$PU_s = PU_{s-a} \bigcup PU_{s-i} \bigcup PU_{s-c}$$

C. Dynamic Analysis

We introduce a dynamic analysis phase to deal with the challenges that cannot be handled by static analysis, such as Java Reflection and Dynamic Loading.

The Android permission system is enforced by both the Android framework and Linux kernel. Permissions checked by the Android framework are finally handled by the *checkPermission* method in *ActivityManagerService*. The permissions to protect file system and network are enforced by the Linux kernel, while most other permissions are enforced by the Android framework. According to [3], 107 of 115 permissions in Android 2.2 are enforced by the Android framework. Yang *et al.* [10] show that kernel-enforced permissions are also passed to *ActivityManagerService* after Android 4.2. Thus, it is adequate to capture the permissions used by the analyzed apps from the Android framework only.

We instrument the method *checkPermission* to monitor the checked permissions at runtime and the package names of apps that use the permissions. We leverage an automated fuzzer Monkey [11] to dynamically test the apps. Meanwhile, we collect the dynamic permission usage statistics of the tested apps from the *checkPermission* method. Monkey is a command-line tool that runs on an emulator or device, which can generate pseudo-random sequences of user events such as clicks, touches, and gestures, as well as a number of system-level events.

Note that although dynamic analysis cannot catch all the permissions due to the difficulty in achieving full coverage of the whole execution path, it works as a useful complement to static analysis because it can find more permissions related to dynamic behaviors.

[1]Android Open Source Project, *https://source.android.com/*

Table I
THE RESULTS OF PERMISSION ANALYSIS

Category	# of Apps	Average # of PR	Average # of PUs	Average # of PUd	Average # of PU	# of Over-Priv(s)	% Over-Priv(s)	# of Over-Priv (s+d)	% OverPriv (s+d)
WALLPAPER	107	5.70	5.33	2.34	6.05	84	78.5%	77	72.0%
WIDGETS	103	10.76	9.25	4.21	10.67	99	96.1%	90	87.4%
BOOKS AND REFERENCE	105	6.8	5.62	3.17	6.70	104	99.0%	100	95.2%
BUSINESS	105	7.46	6.34	3.10	7.34	100	95.2%	95	90.5%
COMICS	108	5.17	4.63	2.58	5.69	94	87.0%	87	81.0%
EDUCATION	105	6.50	8.08	3.14	9.25	94	90.0%	89	85.0%
ENTERTAINMENT	107	7.02	7.11	3.93	8.57	102	95.3%	81	75.7%
FINANCE	105	7.61	6.04	2.85	7.14	95	90.5%	88	83.8%
LIFESTYLE	105	8.14	7.78	4.03	9.19	102	97.1%	90	85.7%
TOOLS	105	9.89	8.14	4.26	9.88	91	86.7%	83	79.0%
TOTAL	1055	7.50	6.83	3.36	8.04	965	91.5%	880	83.4%

IV. EVALUATION

We perform experiments on 1,055 popular apps of 10 categories from Google Play. For each app, we conduct static analysis and dynamic analysis separately as described in the previous section. We instrument the Android framework, and write scripts to dynamically test these apps in a Nexus 5 phone with instrumented Android 4.4.2 .

In particular, we analyze the permission usage of these apps to investigate the following research questions:

- *RQ1: Can our approach detect more permission usage information than state-of-the-art approaches based on static analysis?* For this purpose, we calculate the number of new permissions that can be detected by dynamic analysis but cannot be identified by static analysis. If we can identify more permissions than previous static approaches, we could reduce the false positives in permission gaps detected in previous work[2].
- *RQ2: How many apps are permission overprivileged?* For this purpose, we compare the results of PR and PU captured for each app. If PR-PU[3] is not empty, the app is overprivileged.
- *RQ3: What are the reasons that lead to permission overprivilege?* It is an important question that has not been adequately investigated in previous studies. We believe it is beneficial to help app developers to precisely request permissions, and help mobile users understand and manage the permissions requested by installed apps.

A. Overall Results

The permission usage analysis results are shown in Table I. An app declares roughly 7.5 permissions on average in their manifest files. Apps in the *WIDGETS* category request more than 10 permissions on average.

[2]False negatives in used permissions detection will lead to false positives in permission gap detection.

[3]It denotes a set minus operation here.

As shown in Table I, static analysis is able to identify an average of 6.83 permissions used by each app, while dynamic analysis detects an average of 3.36 permission for each app. After we combine the permissions found in PUs and PUd[4], an app uses roughly 8 permissions on average, which is slightly higher than the number of permissions requested.

B. Dynamic analysis capabilities

In order to check the capabilities of dynamic analysis, we calculate $PU_d - PU_s$ for each app, which refers to the permissions detected in dynamic analysis while not detected in static analysis. The result is shown in Table II.

For more than 60% of the apps, dynamic analysis could detect an average of two more permissions than static analysis, which represents almost 30% increase in detected permissions compared to static analysis only approaches. Furthermore, in contrast to static analysis that identifies permissions used in the code (may not be reachable at all), the permissions detected by dynamic analysis are all actually executed at runtime.

1) Top permissions detected by dynamic analysis: In Table III, we list the top permissions detected by dynamic analysis that cannot be detected by static analysis. Network related permissions, such as *ACCESS_NETWORK_STATE*, *ACCESS_WIFI_STATE*, and *CHANGE_WIFI_STATE*, account for most of the differences.

2) Why these permissions cannot be detected by static analysis?: We explore this question from two angles.

First, we log the call stack to track the API calls leading to the check point of these permissions. We find that some APIs are permission-related, but they are not listed in the permission map of STOWAWAY. After we read the documentation of these APIs and check them with the permission map, we identify two main reasons. One reason is that the permission map is outdated. The permission map of STOWAWAY is

[4]The numbers do not add up because there exist some duplicated permission in PUs and PUd. Note that we use set union to calculate PU.

Table II
NUMBER OF APPS WHERE DYNAMIC ANALYSIS IS ABLE TO FIND MORE PERMISSIONS THAN STATIC ANALYSIS

Category	# of apps with new permissions	% of apps	avg # of new permissions
WALLPAPER	38	35.5%	2.03
WIDGETS	61	59.2%	2.39
BOOKS AND REFERENCE	57	54.3%	1.98
BUSINESS	56	53.3%	1.93
COMICS	57	53.0%	2.04
EDUCATION	77	73.3%	1.60
ENTERTAINMENT	75	70.1%	2.09
FINANCE	64	61.0%	1.86
LIFESTYLE	74	70.5%	2.01
TOOLS	76	72.4%	2.43
TOTAL	635	60.2%	2.04

Table III
THE TOP 10 PERMISSIONS DETECTED BY DYNAMIC ANALYSIS (THAT ARE NOT DETECTED IN STATIC ANALYSIS).

Permission	Count	Percentage
ACCESS_NETWORK_STATE	243	38.3%
ACCESS_WIFI_STATE	202	31.8%
CHANGE_WIFI_STATE	199	31.3%
VIBRATE	172	27.1 %
READ_PHONE_STATE	167	26.3%
INTERNET	42	6.6%
WRITE_EXTERNAL_STORAGE	40	6.3%
CAMERA	37	5.8%
GET_TASKS	36	5.7%
WAKE_LOCK	25	3.9%

only based on API level 8, while we find some APIs added in higher API levels. For example, the API *isActiveNetworkMetered* is added in API level 16, which requires the app to hold the permission *ACCESS_NETWORK_STATE*. As another example, the API *startAudio* is introduced in API level 9, which requires *RECORD_AUDIO*, *ACCESS_WIFI_STATE* and *WAKE_LOCK* permissions. The other reason is that the permission map is also incomplete even for API level 8. For example, we find that the API *updateNetwork* requires permission *CHANGE_WIFI_STATE*, but this API-Permission mapping is not listed in the permission map of STOWAWAY.

Second, we compare the dynamic behaviors of the apps with permissions that can only be detected by dynamic analysis to the other apps. First, we search for *java.lang.reflect.Method.invoke()* or *java.lang.reflect.Constructor.newInstance()* in the decompiled code to identify whether the apps use Java reflection features. Then, we search for *DexClassLoader* to identify whether the apps load classes from embedded .jar and .apk files. We find that the percentage of these apps using dynamic behaviors is higher than the other apps, which indicates that dynamic analysis is more effective in handling dynamic behaviors (such as Java Reflection and Dynamic Loading) than static analysis.

Table IV
THE TOP 10 OVERDECLARED PERMISSIONS

Permission	Count	Percentage
WRITE_EXTERNAL_STORAGE	672	76.4%
ACCESS_WIFI_STATE	239	27.2%
RECEIVE_BOOT_COMPLETED	153	17.4%
WAKE_LOCK	142	16.1%
MOUNT_UNMOUNT_FILESYSTEMS	135	15.3%
ACCESS_NETWORK_STATE	121	13.8%
INTERNET	100	11.4%
GET_TASKS	98	11.1%
READ_PHONE_STATE	96	10.9%
RESTART_PACKAGES	95	10.8%

C. Permission Overprivilege

1) The Distribution of Overprivileged Apps: Column 9 in Table I lists the number of overprivileged apps. Over 91% of the apps declare some permissions that they might not use when we only consider static analysis results, while the overprivileged ratio decreases to 83% when we apply both dynamic and static analysis.

The results also indicate that if we use only static analysis to detect permission overprivilege, we might not be able to identify all the permissions used, which might result in high false positives in permission overprivilege detection. If we combine dynamic analysis and static analysis, we are able to eliminate more than 8% of those apps that are categorized as overprivileged by mistake.

2) Top Overdeclared Permisions: Table IV lists the top 10 overdeclared permissions that are declared (PR) but are not used (PU). Interestingly, more than 76% of the overprivileged apps declare the *WRITE_EXTERNAL_STORAGE* permission but do not use it either in the code or dynamically. We investigate the reasons why it happens on so many apps. One possible reason might be that due to API changes, we cannot find the relevant APIs corresponding to this permission. Another possible explanation is that it is actually no longer necessary for an app to request the *WRITE_EXTERNAL_STORAGE* permission when it writes its own app-specific directories on the external storage since Android 4.4 (API level 19) [12], although the permission is required for API level 18 and lower. Since we are performing experiments on Android 4.4.2, this might be one of the possible reasons.

D. Implications

We believe our study can help app developers precisely request permissions to guarantee the correct execution of their apps while fulfilling the principle of least privilege, and help mobile app users understand and manage the permissions requested by installed apps in a better way.

App developers can utilize the detection results to make their apps follow the principle of least privilege, thus reducing the attack surface. Our study can also help mobile users manage their mobile app privacy preferences. Users can toggle permissions on and off in a more reasonable way. Unused permissions can be toggled off in order to reduce the attack surface, while necessary permissions used by the main functionalities of app could be toggled on in order to keep the app run properly.

V. Discussions

In this section, we examine possible limitations of our approach and potential future improvements.

Developing a better permission map. Our static analysis relies on the permission map of STOWAWAY [4], which would be partially outdated due to the evolution of permissions and APIs. If the permission map is incomplete and inaccurate, it may lead to the false negatives when detecting used permissions, thus leading to false positives when detecting permission gaps.

Improving dynamic exploration. Dynamic analysis relies heavily on the coverage of execution traces, which is almost impossible to reach 100% with automate testing techniques. Prior studies have proposed techniques of more advanced testing of mobile apps, such as UI fuzzing [13] and targeted event sequence generation [14], which can be leveraged in our dynamic analysis in the future.

VI. Related Work

State-of-the-art permission gap detection approaches [4, 5, 3, 6, 15] typically involve first building a permission map that identifies what permissions are needed for each API calls, then performing static analysis to identify permission related API calls.

In order to build the permission map, STOWAWAY [4] uses automated testing techniques to generate unit test cases for API calls, Content Providers and Intents. PScout and COPES [5, 3, 15] perform call-graph based analysis on the Android framework to build the permission map. The idea is to compute a call graph for each entry point of the framework and then to detect whether or not permission checks are present in the call graph.

Felt *et al.* [4] use STOWAWAY to study more than 900 apps. They found that about one-third apps are overprivileged. Wu *et al.* [6] use the permission mapping result of PScout to study apps on 10 different devices, the result shows that more than 85% of the apps are overprivileged. It is surprising to see that more than 90% apps on one of Google's own devices are overprivileged.

VII. Conclusion

This paper presents a new approach to detect permission gaps in Android apps. Our approach leverages both dynamic and static analysis, which could complement each other to achieve significantly higher coverage. Experiments on more than 1,000 apps show that our approach could detect significantly more permission usage information than the state-of-the-art static analysis approaches. We also find that more than 8% of the overprivileged apps detected by previous approaches are false positives.

Acknowledgment

This work is partly supported by the High-Tech Research and Development Program of China under Grant No.2015AA01A202, the National Basic Research Program of China (973) under Grant No. 2011CB302604, the National Natural Science Foundation of China under Grant No.61421091, 91118004, 61103026.

References

[1] "1.5 million Android devices activated every day," http://www.androidcentral.com/larry-page-15-million-android-devices-activated-every-day.

[2] AppBrain, "Google play stats," http://www.appbrain.com/stats/stats-index.

[3] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Automatically securing permission-based software by reducing the attack surface: An application to Android," in *Proceedings of ASE'12*, 2012.

[4] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of CCS'11*, 2011, pp. 627–638.

[5] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the Android permission specification," in *Proceedings of CCS'12*, 2012, pp. 217–228.

[6] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, "The impact of vendor customizations on Android security," in *Proceedings of CCS'13*, 2013, pp. 623–634.

[7] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," in *the 13th Intl Conf on Information Security*, 2011, pp. 346–360.

[8] P. K. Manadhata and J. M. Wing, "An attack surface metric," *IEEE Trans. Softw. Eng.*, vol. 37, no. 3, pp. 371–386, 2011.

[9] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth, "Slicing droids: Program slicing for smali code," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, 2013, pp. 1844–1851.

[10] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan, "IntentFuzzer: Detecting capability leaks of Android applications," in *the 9th ACM Symposium on Information, Computer and Communications Security*, 2014, pp. 531–536.

[11] "Ui/application exerciser monkey," http://developer.android.com/tools/help/monkey.html.

[12] "uses permission," http://developer.android.com/guide /topics/manifest/uses-permission-element.html.

[13] C. Hu and I. Neamtiu, "Automating gui testing for Android applications," in *the 6th International Workshop on Automation of Software Test*, 2011, pp. 77–83.

[14] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *Proceedings of ISSTA'13*, 2013, pp. 67–77.

[15] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon, "Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android," *IEEE Transactions on Software Engineering (TSE)*, 2014.