IntentFuzzer: Detecting Capability Leaks of Android Applications

Kun Yang[1,3], Jianwei Zhuge[2,3], Yongke Wang[4], Lujue Zhou[2], and Haixin Duan[2,3]
[1]Department of Computer Science and Technology, Tsinghua University
[2]Institute for Network Science and Cyberspace, Tsinghua University
[3]Tsinghua National Laboratory for Information Science and Technology
[4]Institute of Information Engineering, Chinese Academy of Sciences

ABSTRACT

Capability leak is a vulnerability in Android applications, which violates the enforcement of permission model and threatens the secure usage of Android phone users. Malicious applications can launch permission escalation attacks with this vulnerability. In this paper, we propose a dynamic Intent fuzzing mechanism to uncover vulnerable applications in both Android markets and closed source ROMs. We built a prototype called IntentFuzzer. With it, we analyzed more than 2000 Android applications in Google Play and hundreds of in-rom applications inside two closed source ROMs. We found that 161 applications in Google Play have at least one permission leak, and 26 permissions in Xiaomi Hongmi phone and 19 permissions in Lenovo K860i stock phone are leaked. Finally, we give several cases of exploitation to verify our analysis result.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Security, Design, Experimentation

Keywords

Smartphone Security, Intent Fuzzing, Capability Leak

1. INTRODUCTION

Android smartphones market share has exploded in recent years. Compared with traditional PC, smartphones are much closer to users. Considering the sensors such as cameras and voice recorder integrated into mobile devices, it's necessary to protect these usage of the sensors. Moreover, smartphones have become the main way to store and handle private data, including SMS messages, call logs, contact information and photos, which are imperative to be protected.

ASIA CCS'14, June 4–6, 2014, Kyoto, Japan.

ACM 978-1-4503-2800-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2590296.2590316 .

Permission model is essential in the design of Android security. In this model, sensitive resources are protected by permissions that other applications don't have. However, because of the existence of vulnerable applications, it is possible for attackers to break through such permission model. Capability leak is such a vulnerability that an application exposes some permissions, by which other applications without these permissions can access protected resources.

Several static analysis systems have already been built to detect capability leaks in applications [7, 6, 10, 5]. DroidChecker [5] has found 6 vulnerable apps including Adobe Photoshop Express 1.3.1. Woodpecker[10] detected 11 permissions are leaked in 8 stock phones from world's leading manufacturers. These tools are effective, but they may all have false positive, and should manually verify how to trigger permission leaks.

To precisely uncover capability leaks in millions of applications automatically, we used an old technique - fuzzing in this new situation. By sending testing Intents to various exposed interfaces, we can detect if any capability leaks happen. Compared with static methods, our method has an advantage in precision. While static analysis only sees the possible calling connections between function calls, dynamic fuzzing can detect permission leaks that really happen, which can be recorded and used to reconstruct all the scenes.

We developed a prototype called IntentFuzzer, and used it to analyze over 2,000 popular applications in Google Play. Our result shows that 161 applications have at least one permission leak. We also apply IntentFuzzer to closed source ROMs that customized by various vendors, and found that 26 permissions in Xiaomi Hongmi phone and 19 permissions in Lenovo K860i are leaked.

The rest of paper is organized as follows: Section 2 briefly introduces background knowledge of Android permission model and Inter Component Communication mechanism. Section 3 and Section 4 describe our system design and implementation respectively. Section 5 presents the detailed evaluation results from our study. Section 6 discusses limitations and future work. Section 7 describes related work and Section 8 summarizes our conclusions.

2. BACKGROUND

2.1 Permission Model

To mitigate security threat related to personal privacy, Google has designed a permission-based model. Android application is prohibited from accessing dangerous permis-

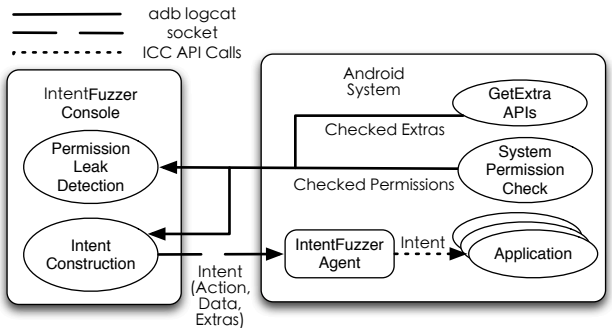

Figure 1: Architecture of IntentFuzzer

sions by default. Users should authorize the privileges that apps request at install-time.

2.2 Inter Component Communication

ICC(Inter Component Communication) is a mechanism to help apps interact with each other. Each app is made up of several components, or logical building blocks. There are 3 types of components that use ICC: *Activity*, *Service*, and *Broadcast Receiver*. They can communicate with each other using *Intent* objects[3]. Intent is a data structure for carrying messages from one component to another.

Developers could set the "exported" attribute of the application components as "true" to accept Intent from other components, or "false" to deny. If the attribute is not set, it's also possible that system marks it as "exported" by default. We focus on the exported components to detect capability leaks.

2.3 Capability Leaks

`Capability Leak`, also known as Permission Re-Delegation [7], occurs when a vulnerable application performs a privileged action on behalf of a malicious application without that permission. For example, a benign application has been authorized with permission P. If one of its exposed component C fails to check the caller's permission, a malicious application may perform unprivileged actions by constructing and sending Intent to component C.

To avoid capability leaks, a component can specify that an Intent sender must have a certain permission by either adding a permission requirement to component's manifest file, or by calling checkPermission API.

However, many developers don't fully understand the potential risks in Android application components. They either expose the components unintentionally, or expose them intentionally but failed to check the component caller's permissions.

3. SYSTEM DESIGN

We aim at detecting capability leaks in Android applications using fuzzing technique. We dynamically generate appropriate Intents and send them to the components to see if any capability leak indeed happens.

Figure 1 depicts IntentFuzzer's architecture. IntentFuzzer Console is a fuzzing control center and responsible for constructing Intents and processing feedbacks from IntentFuzzer Agent. IntentFuzzer Agent is an Android application running in Android system, taking charge of sending real Intents to the application components to be diagnosed. Several modules in Android system are modified in order to generate feedbacks for the Console. With these feedbacks, Console will amend the Intent for deeper execution path of the receiving components, and decide if any potential permission leak has actually happened.

We will discuss the details in the rest of this section.

3.1 Fuzzing Strategy

There are three types of components that can receive Intents: Activities, Services, and Broadcast Receivers. Activities will always provide a screen for users to interact with. So we need to automatically generate correct interaction with the user interface when fuzzing Activities, which is not a well-studied topic[9]. Although malicious applications have some possibilities to perform an escalation attack stealthily, expecting users to make right interaction to trigger harmful action, users may notice such kinds of attacks and close the activity interface to avoid attacks. So, this kind of threat is not very serious.

In terms of Services and Broadcast Receivers, they both run in the background. Permission escalation attack can be performed without any awareness of users. Capability leaks of Services and Broadcast Receivers are principal threats. So we focus on fuzzing Services and Broadcast Receivers.

There are two forms of services: Started service and Bound Service. Started services receive Intents, while Bound Services don't. The exposed interface of Bound Service is defined by a Messenger, or AIDL(Android Interface Definition Language), which is not as simple as Intent delivery. Arguments transferred between callers and callees are not like Intents that have some construction rules to follow, they can be various types and numbers[4]. Our motivation is to heuristically generate Intents to audit Android components for capability leaks. So we further narrow down the fuzzing targets to Started Service and Broadcast Receivers.

Intents can be divided into two groups: explicit Intent and implicit Intent[3]. Explicit Intents designate the target component by its name while implicit Intents do not name a target. Implicit Intents will be resolved to proper component by system according to the Intent filters defined by each component.

There are two possible situations for permission leaks. Components intend to receive all the Intents including implicit Intents, but failed to check caller's permission; Components intend only to receive intra-application Intents which mean explicit Intents, but are exported to the public. So in both situations, vulnerable components will receive explicit Intents. We use explicit intents to do fuzzing.

3.2 Permission Leak Detection

IntentFuzzer Agent is the application that sends Intents. We request no permissions for IntentFuzzer Agent. If a privileged action is triggered by the Agent, a permission leak happens.

Although the permissions of Android applications are authorized during the installation time. Permission checking is enforced during runtime. To detect if any permissions related actions are performed after sending intents, we modified the permission checking module in Android System. The inserted code will inform Console of what permissions

are passed in the checking process at runtime. Details of system modification will be clarified in Section 4.

3.3 Intent Construction

On receiving explicit Intents, components will execute from their entries. For Started Services, entry method is onStartCommand(); for Broadcast Receivers, entry method is onReceive(). We don't consider Bound Services as mentioned above.

To detect more permission leaks, we should expand the execution path coverage as much as possible. So it's important to deliver appropriate Intent to the components.

An Intent object contains a bundle of information:

- *Action* is a string naming the action to be performed. There are predefined constants of generic actions. Developers can also define their own action strings for activating their components. Self-defined Action string should include the application package as a prefix to avoid naming conflict.
- *Data* is represented by the URI, which also implies the MIME type.
- *Category* is a string containing additional information about the kind of component that should handle the intent. So Category is just for implicit Intent resolution which we don't construct Intent with.
- *Extras* are key-value pairs for additional information that should be delivered to the component handling the Intent.
- *Flags* are predefined values for instructing the Android System how to launch an Activity. It's not related to Broadcast Receivers and Services. We don't construct Intents with Flags.

How to construct an Intent with Action, Data, and Extras? Details are explained in the subsections as below.

3.3.1 Action Construction

To generate Action of Intents for fuzzing, we consider two aforementioned situations of vulnerable applications separately.

For the components that intend to be exported to receive implicit Intent, they will define *Intent Filters* in their manifest file. Intent Filters are used to inform the system which implicit Intents they can handle. So proper intents attributes can be inferred using Intent Filters. By sending appropriate Intents that meet the Intent resolution rule, components can handle it well and execute into deep path. Each Intent Filter may consists of three types of rules: Action, Category, and Data. An Intent Filter may specify more than one Action, but an Intent object names just a single Action. So we construct Intents with each Action in each Intent Filter. We call the Action inferred from Intent Filter as *explicit Action*.

For the exported components that do not intend to be exported, they will not contain Intent Filters, because these components are developed only for intra-application usage and accept explicit Intents. They may compare the Action in the Intent with some predefined Actions to perform different tasks, which may lead to potential capability leaks if the attacker can specify a correct Action. Below is a code example of a Broadcast Receiver entry method.

```
public void onReceive(Context context, Intent intent) {
    SmsManager smsManager = SmsManager.getDefault();
    String action = "com.example.test.action.SEND_SMS";
    if (action.equals(intent.getAction())) {
            smsManager.sendTextMessage("10086", null, "test",
                 null, null); }
}
```

The mistakenly exported Broadcast Receiver above will not specify any Intent Filter in the manifest file, because it's only for self-use. To get our fuzzer deep into the code line of the API sendTextMessage, we must construct the correct Action. We assume that all the Action strings will be defined by const string rather than be generated by runtime code, which is just how the most programers do. Then we use a conservative static method. We first get all the strings from the string pool of dex or odex file of the app[1]. Among such strings, we choose the ones including a prefix of the application package as potential Actions since Google recommends developers to use the package name as a prefix to ensure uniqueness, and we also include the strings that are standard Actions defined by Android(e.g. android.intent.action.DELETE). Thus we get a list of potential appropriate Actions. We call the Action extracted from bytecode as *implicit Action*.

Thus we can generate Intents with both explicit Actions and implicit Actions.

3.3.2 Data Construction

For the occasion of explicit Action, each component will have Intent Filter defined in the manifest file. As Intent Filters may contain rules for Data, we can infer corresponding data type that the component is able to handle. Data rule can be specified by scheme, host, port, and path for each part of the URI scheme://host:port/path. So we prepare common data types as common form of URI before fuzzing. For example, we store pictures of common type in both websites and system content providers, providing possible URIs such as http://example.com/a.jpg and content://media/external/images/media/1. If any URI we prepared fits the rule of Data in the Intent Filter, we construct an Intent with it. If no rule of Data is specified in Intent Filters, we don't construct any Data URI in Intents.

For the occasion of implicit Action, there is no corresponding Intent Filter, so we also don't construct Data URI in Intents.

3.3.3 Extras Construction

Extras are key-value pair information in Intents. Extra keys are strings while values can be any Java primitive type or Class. They are not specified in Intent Filters. Intent recipient may check the Extra data for later use in performing privileged actions. There is a code example of a Service entry point method.

```
public int onStartCommand(Intent intent, int flags, int
     startId) {
    SmsManager smsManager = SmsManager.getDefault();
    String smsContent = intent.getStringExtra("sms");
    if (smsContent != null && smsContent != "") {
        smsManager.sendTextMessage("10086", null, smsContent,
             null, null); }
    ...
}
```

The Android API getStringExtra() will extract the string value mapped by key "sms" from the Intent. If Extra is not

included in the Intent, the Service above will not trigger sendTextMessage method, and the permission leak of android.permission.SEND_SMS could not be detected. In this situation, false negative will be produced.

To decrease this kind of false negative, we built a runtime feedback system to construct Extra data with appropriate key and type that components can handle. Console will keep a set of the requested Extra keys and value types. Components that receive Intents must call APIs(e.g. getStringExtra) to get Extra data. We instrumented these APIs and the inserted code will inform Console of what key and value type is requested. Console will add this Extra key and type of value to the list. So in the next round of fuzzing, the detected new Extra data with detected key and type will be randomly generated and feed to the component. By using such a feedback based iteration method of Extras construction, execution can go deeper.

We only construct Extra data of Java primitive types and don't consider types of Java Class, because it's non-trivial to deal with various Classes defined in various libraries or by app developers.

3.4 General Fuzzing Steps

We define the component to be fuzzed as C. IntentFuzzer first checks if there exists any Intent Filter in C. If so, C is exported on purpose and we do fuzzing in steps as follows:

1. For each explicit Action defined in each Intent Filter, do the step 2 to 5;
2. Construct Data only when the Intent Filter contains Data tag;
3. Initialize an empty Extras data set E;
4. Send an Intent with constructed Action, Data, and all the Extras in E;
5. Wait for several seconds. Scan for permission leak logs and getExtra API logs. If new Extra data requests are found, add them to E, and goto step 4; If not, then exit the iteration.

If no Intent Filter is specified for C, IntentFuzzer takes it as a mistaken exported component and do fuzzing in the following steps:

1. Construct implicit Actions, and for each implicit Action, do the steps 2 to 4;
2. Initialize an empty Extras data set E;
3. Send an Intent with constructed Action, and all the Extras in E;
4. Wait for several seconds. Scan for permission leak logs and getExtra API logs. If new Extra data requests are found, add them to E, and goto step 3; If not, then exit the iteration.

We record all the permission leak results and their corresponding Intents that trigger the them for post-analysis.

4. IMPLEMENTATION

To accelerate prototype development, we reused existing excellent open source tools.

4.1 Architecture

We built our prototype IntentFuzzer on top of Drozer[2] - an open source security assessment framework for the Android platform. Drozer prompts users a console to dynamically interact with the ICC endpoints exported on a device. Drozer employs the similar architecture as IntentFuzzer, which consists of an agent installed in Android system and an server-side console on PC. All modules in the server-side console are written in Python.

IntentFuzzer benefits a lot from Drozer's modular design. We implemented our IntentFuzzer Console by inserting a single module in it.

4.2 System Modification

Android's permission system is enforced by both Android system services and Linux kernel[11]. Most permissions are checked by Android system services and finally handled by *checkPermission(String permission, int pid, int uid)* in *ActivityManagerService*. File system and network related permissions are enforced by the GID isolation mechanism in Linux kernel, such as android.permission.INTERNET and android.permission.WRITE_EXTERNAL_STORAGE. Combining manual analysis of Android sources and some tests, such kernel enforced permissions are also passed to ActivityManagerService in Android 4.2, which is not the case in old version of Android[8]. So we only need to instrument the method checkPermission to record what UID of Android application is checked, and what permission is checked. Console can use the UID in the log to look up leaked permissions.

We also modify Extras getting APIs to catch components' requests for extra data. There are a bunch of Extras getting APIs for different data types, such as getStringExtra(String name), getIntExtra(String name, int defaultValue). We outputted type and key of each Extra request. Leveraging logcat logs, IntentFuzzer Console will generate corresponding new Extras in the next round of Intent fuzzing until the iteration ends.

5. EVALUATION

5.1 Experiment Design

To evaluate IntentFuzzer, we use it to detect capability leaks of applications in both Google Play and closed source ROMs. We downloaded 2183 free applications from Google Play, all of which are among top 200 most popular apps in each category in www.appbrain.com. We install them and do fuzzing one by one in our modified Android 4.2.2 Samsung Galaxy Nexus phone. Galaxy Nexus phone is a device supported by Android Open Source Project, which we can build ROM from source and directly flash the device.

In terms of closed source ROMs, it's not easy to migrate all in-rom applications to other environment due to dependency problems. So we choose to directly modify closed source ROMs by rewriting framework bytecode and do fuzzing inside themselves. We selected Xiaomi Hongmi phone and Lenovo K860i phone for our evaluation. Xiaomi Hongmi phone runs Android 4.2.2, and Lenovo K860i runs Android 4.2.1. Both phones use updated system version from their vendors.

After analyzing an application package, a report will be produced recording leaked permissions and Intent attributes that trigger the corresponding permission leak.

5.2 Results

It took 2240 minutes to fuzz all 2183 apps from Google Play, 90 minutes for 104 packages in Xiaomi Hongmi phone, and 95 minutes for 105 packages in Lenovo K860i phone.

From analysis reports, we detected 161 application packages that have at least one permission leak, and found 26

Table 1: Leaked Permissions in Google Play Applications

Permissions	Packages	Components
ACCESS_NETWORK_STATE	91	86
READ_PHONE_STATE	42	39
WAKE_LOCK	22	30
INTERNET	11	14
ACCESS_FINE_LOCATION	9	8
ACCESS_WIFI_STATE	8	8
GET_ACCOUNTS	6	9
VIBRATE	4	4
SYSTEM_ALERT_WINDOW	3	5
CHANGE_WIFI_STATE	3	3
ACCESS_COARSE_LOCATION	2	2
GET_PACKAGE_SIZE	2	2
READ_CONTACTS	2	2
READ_SMS	2	2
READ_EXTERNAL_STORAGE	1	1
WRITE_SMS	1	1
WRITE_CALL_LOG	1	1
GET_TASKS	1	4
RESTART_PACKAGES	1	1
CLEAR_APP_CACHE	1	1
BLUETOOTH_ADMIN	1	1

permissions in Xiaomi Hongmi phone and 19 permissions in Lenovo K860i phone are leaked. We organized reports into three tables. In each table, leaked permissions are listed. For each permission, we counted the total numbers of packages and components that have the corresponding permission leakage. We can see results of 2183 Google Play applications in Table 1, results of Xiaomi Hongmi phone in Table 2, and results of Lenovo K860i in Table 3. Permissions' standard package prefix is omitted in the tables.

Except for capability leaks, we also detect the runtime exceptions occurred during fuzzing via default logcat information. There are 11 components from 9 packages in Xiaomi Hongmi phone, 13 components from 12 packages in Lenovo K860i, and 141 components from 123 apps in Google Play crashed during fuzzing. This information may also help developers to diagnose the robustness of their apps.

5.3 Exploitation Analysis and Case Study

Capability leaks don't mean that attackers can do every action authorized by the leaked permission. Exploitation depends on how vulnerable components deal with the Intents.

Both ICC of Broadcast Receivers and Started Services do not return results. So the leaks of permissions that are related to accessing some data are not easy to exploit. Attackers cannot find a channel to receive sensitive data protected by the leaked permission. Maybe attacks should combine other permission leaks, or other vulnerabilities.

For the leaks of permissions that are related to changing some status, they are easy to exploit and often result in great harm, e.g. sending messages, changing system settings. Some detailed exploitation cases are given below.

Clean Master is an app that pre-installed in the Xiaomi Hongmi phone. It can help user to kill all background processes and get memory freed. However, the permission android.permission.RESTART_PACKAGES is leaked and all other apps could invoke the exposed component to kill background processes. The vulnerable component is com.cleanmaster.appwidget.WidgetService. Exploiting method is just starting the service using an Intent with Action com.cleanmaster.appwidget.ACTION_FASTCLEAN. This case is similar to another app called Smart RAM Booster. Sending

Table 2: Leaked Permissions in Xiaomi Hongmi Phone

Permissions	Packages	Components
CHANGE_COMPONENT_ENABLED_STATE	9	11
ACCESS_NETWORK_STATE	8	9
READ_PHONE_STATE	8	9
WAKE_LOCK	6	7
INTERNET	5	7
GET_ACCOUNTS	4	5
DEVICE_POWER	3	4
STATUS_BAR	3	4
ACCESS_ALL_DOWNLOADS	2	4
READ_CONTACTS	2	2
READ_DREAM_STATE	1	1
READ_CALENDAR	1	3
UPDATE_DEVICE_STATS	1	1
WRITE_CALENDAR	1	1
RESTART_PACKAGES	1	1
DELETE_PACKAGES	1	1
READ_CALL_LOG	1	1
READ_SMS	1	1
GET_PACKAGE_SIZE	1	1
MODIFY_PHONE_STATE	1	1
INSTALL_PACKAGES	1	2
com.android.email.permission.ACCESS_PROVIDER	1	1
WRITE_SECURE_SETTINGS	1	1
VIBRATE	1	1
ACCESS_WIFI_STATE	1	2
DELETE_CACHE_FILES	1	1

an intent with Action com.anttek.rambooster.action.BOOST to the Service com.anttek.rambooster.service.BoostService will also get background processes killed.

Package com.popularapp.fakecall is an application called Fake Call & SMS, which can help user to fake coming calls and messages. WRITE_SMS and WRITE_CALL_LOG permissions are leaked in receivers com.popularapp.fakecall.incall.MessageAlarm and com.popularapp.fakecall.incall.CallAlarm respectively. Attacker with no permission can reproduce any messages and calls which is preset in this application at any time, by sending Intents with an "id" Extra.

We detected 4 apps that leak android.permission.VIBRATE Permission. Package cn.etouch.ecalendar2 is a calendar application. Sending an Intent with Action ACTION_SUISENT_ECALENDAR_ShowNotice to the receiver cn.etouch.ecalendar.service.NoticesReceiver will vibrate the phone. Package com.azumio.android.sleeptime is an alarm app. If one of its component com.azumio.android.sleeptime.alarm.AlarmReceiver receives an Intent with Action value of com.azumio.android.sleeptime.WAKEUP will make the phone vibrate and beep. Package br.com.gerenciadorfinanceiro.controller is a personal finance manager app. Any empty Intent sending to the component br.com.third.utils.GerarNotificacao will also make the phone vibrate. The last one is the app in Xiaomi Hongmi phone that is similar with Apple Siri, which targets at assisting users via voice recognition. The exposed component is com.miui.voiceassist.SiriReceiver. Broadcasting an Intent to with Action com.miui.voiceassist.alarm and an string Extra, will read out the string and vibrate the phone.

Stk(Sim Toolkit Application) app in both Xiaomi and Lenovo phone leaks MODIFY_PHONE_STATE permission. Attackers without this permission can send an Intent with Action android.intent.action.stk.command to the com.android.stk.StkCmdReceiver component in the package com.andro-

Table 3: Leaked Permissions in Lenovo K860i Phone

Permissions	Packages	Components
ACCESS_NETWORK_STATE	11	11
CHANGE_COMPONENT_ENABLED_STATE	8	11
WAKE_LOCK	6	7
READ_PHONE_STATE	5	5
GET_ACCOUNTS	3	4
READ_EXTERNAL_STORAGE	2	2
INTERNET	2	2
READ_CALL_LOG	2	2
READ_DREAM_STATE	1	2
READ_CALENDAR	1	3
DEVICE_POWER	1	1
ACCESS_WIFI_STATE	1	1
ACCESS_ALL_DOWNLOADS	1	1
STATUS_BAR	1	1
BLUETOOTH	1	1
DELETE_PACKAGES	1	1
MODIFY_PHONE_STATE	1	1
BLUETOOTH_ADMIN	1	6
READ_CONTACTS	1	1

id.stk. This intent will kill the current application running on the phone interface. An exploitation case is that attackers can hang up the phone when any call is coming.

6. DISCUSSION

There are some limitations in our work. IntentFuzzer cannot search for capability leaks in Bound Services. IntentFuzzer also cannot handle Broadcast Receivers which are registered at runtime. IntentFuzzer only detects permission leaks when a component performs a privileged operation immediately after receiving a Intent. If a component that receives an Intent internally changes the state of the target app and lead to a capability leak in the future, IntentFuzzer will produce false negative.

We used a heuristic method to construct Intents. To achieve smarter fuzzing and higher path coverage, symbolic execution techniques should be applied to generate correct Intents for each execution path in the future.

7. RELATED WORK

In recent years, much work has been devoted to detecting capability leaks. ComDroid [6] is a tool for developers to analyze their own applications before release. Warnings of security risks in inter-application communication will be raised. Felt et al. [7] created a static path-finding tool to find potential attack path in application components. Woodpecker [10] can detect capability leaks in stock Android smartphones. With consideration of object inheritance and callbacks, Woodpecker has a better accuracy. DroidChecker[5] applies inter-procedural control flow graph analysis and static taint checking to search for capability leaks. All tools above use static method and may contain false positives, and their detection results must be verified manually. To the best of our knowledge, we are the first to apply dynamic fuzzing technique in detecting capability leaks. All the leaks we detected are all real occurrences of permission leaks, without false positives.

8. CONCLUSIONS

We propose a novel fuzzing approach to detect capability leak vulnerabilities of Android applications. We invent a Intent construction strategy to achieve higher execution coverage. With our prototype IntentFuzzer, we analyze more than 2,000 applications in Google Play and find 161 applications with at least one permission leak. We also tested IntentFuzzer in Xiaomi Hongmi phone and Lenovo K860i phone, and find 26 and 19 permission leaks respectively.

Acknowledgements

This work is partially supported by China Core Electronic Devices, High-end Generic Chips and Basic Software Award 2012ZX01039-004, NSFC No.61161140454, The National Key Technology R&D Program under Grant No.2012BAH38B03, and China InformationTechnology Security Evaluation Center under Grant No.CSTC2011AC2143.

9. REFERENCES

[1] Dalvik Executable Format. `http://source.android.com/devices/tech/dalvik/dex-format.html`.

[2] drozer. `https://labs.mwrinfosecurity.com/tools/drozer/`.

[3] Intents and Intent Filters. `http://developer.android.com/guide/components/intents-filters.html`.

[4] Services. `http://developer.android.com/guide/components/services.html`.

[5] P. P. Chan, L. C. Hui, and S. Yiu. Droidchecker: analyzing android applications for capability leak. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 125–136. ACM, 2012.

[6] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252. ACM, 2011.

[7] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Security Symposium*, volume 18, pages 19–31, 2011.

[8] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Trust and Trustworthy Computing*, pages 291–307. Springer, 2012.

[9] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung. Vision: automated security validation of mobile apps at app markets. In *Proceedings of the second international workshop on Mobile cloud computing and services*, pages 21–26. ACM, 2011.

[10] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, 2012.

[11] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 611–622. ACM, 2013.