AppSeer: Discovering Flawed Interactions among Android Components

Vincenzo Chiaramida
University of Illinois at Chicago
Chicago, Illinois, USA
vchiar2@uic.edu

Francesco Pinci
University of Illinois at Chicago
Chicago, Illinois, USA
fpinci2@uic.edu

Ugo Buy
University of Illinois at Chicago
Chicago, Illinois, USA
buy@uic.edu

Rigel Gjomemo
University of Illinois at Chicago
Chicago, Illinois, USA
rgjome1@uic.edu

ABSTRACT

We identify several reliability issues arising from interactions between components of system-defined Android apps and components of third-party apps. These issues are generally caused by incorrect assumptions that system apps make about the behavior of third-party apps, resulting in significant vulnerabilities in system apps. For instance, it is possible for a third-party app to make many system applications to crash, including the *Phone* app used to make and receive phone calls, the *Settings* app used to configure a mobile device, and several other apps that expose a so-called *started service*. Our findings indicate that additional automated tools for integration testing and static analysis of Android apps are in order. Here we discuss AppSeer, a toolset that automatically detects vulnerabilities of system apps and third-party apps. Preliminary precision and recall results for AppSeer are quite encouraging.

CCS CONCEPTS

• **Security and privacy** -> **Malware and its mitigation**; **Denial-of-service attacks**; • **Software and its engineering** -> *Software reliability*;

KEYWORDS

Mobile platform security; Denial-of-service attacks; Software testing.

ACM Reference Format:
Vincenzo Chiaramida, Francesco Pinci, Ugo Buy, and Rigel Gjomemo. 2018. AppSeer: Discovering Flawed Interactions among Android Components. In *Proceedings of the 1st International Workshop on Advances in Mobile App Analysis (A-Mobile '18), September 4, 2018, Montpellier, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3243218.3243225

A-Mobile '18, September 4, 2018, Montpellier, France

ACM ISBN 978-1-4503-5973-3/18/09…$15.00
https://doi.org/10.1145/3243218.3243225

1 INTRODUCTION

Applications running on Android platforms are typically organized as a set of communicating components. This kind of design not only supports a high degree of modularity but also allows for interoperability among different apps. Components in a given application can invoke components in that application as well as components of other applications.

Here we explore reliability and security issues arising from the interactions between components in third-party applications and system-defined applications built in Android. We identify a number of defects arising from faulty interfaces among components of different apps. We conclude that the development of automated tools for detecting interface faults among Android components is highly desirable. Finally, we report on the design and implementation of AppSeer, a toolset for automatically detecting certain interface defects among Android components.

We focus specifically on faults affecting two types of app components, namely *activities* and *started services*. In brief, activities are components that define the interface to be displayed on a device's screen and that manage the interactions between displayed objects and the user. Started services are components intended to perform long-running operations, such as synchronizing with a remote server in the cloud or playing a music segment.

First, we investigate failures that may occur when preconditions on the execution of system components are not satisfied and yet the components are invoked by third-party apps. Component invocation occurs through special messages named *intents* sent by a calling component to the component being called. We identify two ways in which a third-party app can cause crucial system apps to crash including the all-important *Phone* app, which allows a user to make and receive phone calls. While the Android OS automatically restarts the *Phone* app after a crash, repeating the behavior leading to the crash will eventually cause the hosting device to shut down and reboot, which is clearly undesirable.

We detected these faults by identifying automatically activities and services that system apps expose to third-party apps, and by checking assumptions that these components implicitly make when they are called into action via an intent. We then designed and implemented our own apps that invoke system-app components when their assumptions do not hold, causing those system apps to crash. We also discovered that new functionality introduced in Android's

most recent version as of this writing, V8.0 (nicknamed Oreo), potentially makes every app exposing a started service susceptible to crashes. This is quite problematic because it affects a multitude of apps that expose a background service. See Section 2 below.

Second, we examine vulnerabilities arising from Java constructs, such as *reflection* and the class loader. These constructs allow a third-party app lacking system privileges to modify system data structures holding information about that app. For instance, an Android app is identified by a so-called *package name*, which is unique throughout the entire Android ecosystem. By cleverly using the class loader and reflection, we were able to access and modify an app's package name stored by the system. The ability for an app to "impersonate" a different app can be exploited for malicious purposes to gain access to protected information and resources on the device, for instance, to start the device's camera.

Finally, we reported the faults that we discovered along with app examples to Google, using their company-provided quality assurance system [3], and to a global vulnerability database [1]. Issues involving activities were assigned CVE-2018-9447; issues involving services are under active investigation as of this writing.

In summary, this paper makes these contributions:

(1) We introduce techniques and an automated toolset for automatically detecting interface flaws involving Android activities and started services.
(2) We discovered a defect involving started services and their clients, which could be different apps from the app that defines the service. The defect causes the application exposing a started service to crash unless it raises its service to foreground status.
(3) We discovered a number of defects when certain activities in system apps *Phone* and *Settings* are invoked by third-party apps. These defects will cause *Phone* and *Settings* to crash.
(4) We discovered a backdoor way for a third-party app to access and modify information about the app maintained by the operating system.

This paper is organized as follows. In Section 2 we summarize key Android concepts. In Section 3 we discuss our method for identifying interface issues among Android components and our automated toolset. In Section 4 we discuss defects that we detected through our analyses. Finally, in Section 5 we summarize related work.

2 BACKGROUND ON ANDROID

Android applications consist of one or more components, of which there are four kinds. The main process is responsible for executing specific callback methods to manage component lifecyle. Specifically, an *activity* is an application component aimed at managing the app's user interface and user interactions. A *service* typically manages long-running operations, such as playing music or synchronizing with a remote server. Services generally run in the background; however, a service can raise its status to foreground, for instance, if it has an effect on the user experience of the device (e.g., a music playing service). Background services are good candidates to be killed by the system in a low memory situation.

Application components can invoke and interact with other components through special messages called *intents*. An *explicit* intent is targeted to a specific application by including the application's unique *package name* and the name of a component in that application. Alternatively, the Android system can infer the recipient of an intent through a so-called intent resolution process. In this case, an *implicit* intent will include only a generic description of the target component; a component whose *intent filter* matches that description will receive the intent.

An application can start activities and services of other applications by calling such system methods as *startActivity(Intent)* and *startService(Intent)*. These methods will cause the execution of the *onCreate()* callback on the target activity or service. The targeted component will be called into action even though its context may not be properly initialized, possibly leading to such run-time errors as null pointer exceptions. This can happen, for instance, if the component in question is supposed to be called by its hosting application after other components have completed execution; however, the component is also exported to other applications. We discuss defects arising from this situation, which we call *unexpected intents*, in Section 4 below.

When a service is started, the system sets the service to the background state by default. Android version V8.0 (Oreo) introduced a new method for starting a service in the foreground state, namely *startForegroundService(Intent)*. However, the target service will still be started in the background state. It is the service's responsibility to raise its status to foreground state by calling method *startForeground()* during the execution of such callbacks as *onCreate()*, *onStartCommand()* or *onHandleIntent()*. Failure to do so will cause the application containing the service to crash. Table 1 below summarizes four possible scenarios involving the *client* application that requests the service to be started and the *server* application that defines the service.

Table 1: Starting services—Possible use cases.

Client App	Server App	Behaviour
startService()	*onStartCommand()*	Normal: BG
startService()	*onStartCommand()* + *startForeground()*	Normal: FG
startForegroundService()	*onStartCommand()*	***Server crash***
startForegroundService()	*onStartCommand()* + *startForeground()*	Normal: FG

In Case 3 above an application explicitly requests a service in the foreground but the server app fails to execute *startForeground()*. After the expiration of a timeout, the service application will freeze while displaying the infamous *Application Not Responding (ANR)* dialog. The system will then proceed to the termination of the process hosting the server app, closing all the currently active connections with client apps. The programmatic exploitation of this mechanism can lead to *Denial of Service (DoS)* attacks. We argue that this behaviour is the result of an improper design choice missing two key aspects:

(1) How the client application can infer if the target service is ready to be started in the foreground;

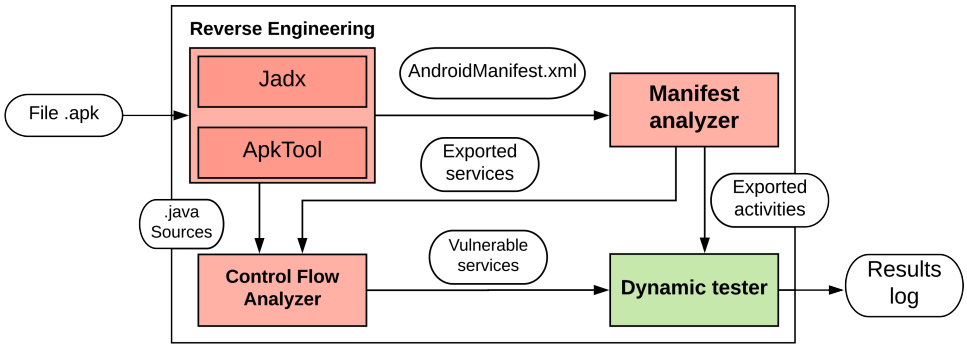

Figure 1: Flow diagram of AppSeer.

(2) How the server application can execute *startForeground()* only when required.

3 APPROACH AND IMPLEMENTATION

Now we describe our approach and AppSeer.

3.1 Approach

As we mentioned earlier, the new mechanism for starting services in the foreground makes it difficult for existing apps to know how to start their services. Furthermore, such confusion may pave the way for attacks whereby a malicious app can issue requests to start foreground services and crash legitimate apps. In order to check if an app is vulnerable to this attack, we developed AppSeer, a tool that (1) statically analyzes apps and classifies their services as vulnerable or safe, and (2) tests whether exposed activities and services can actually crash due either to unexpected intents (e.g., resulting in uninitialized objects) or to the foreground service issue.

Overview. AppSeer combines a static analysis phase with a dynamic testing phase to verify the results of the static analysis. In particular, given an apk file of an application, AppSeer first extracts the app's code and *manifest file*. An app's apk file is an archive containing compiled code, other app resources (e.g., media files), and the app's manifest file, which declares all the app components. Next, AppSeer statically analyzes the code to determine if services comply with the new service-starting mechanisms by performing an interprocedural control flow analysis. This analysis yields a set of potentially-defective services. Finally, in the dynamic phase, we use the information gathered during static analysis to test the activities and services identified as vulnerable by generating code that invokes those activities and services. As a result, we obtain a set of true positives and false positives of our analysis of services and activities. Our static analysis is conservative, meaning that we explore all paths concerning a service component, in order to avoid false negatives. A high level overview of AppSeer is shown in Figure 1. Next, we describe its four main components.

Reverse Engineering Component. This component receives as input the apk file and produces as output the manifest file and Java sources of the application. We use the *Jadx* [10] tool to translate Android bytecodes into Java source code and the *apktool* tool [9] to extract the manifest file.

Fetcher. This component is responsible for analyzing the manifest file and extracting the exposed activities and services. In particular, it checks in the manifest file if an activity has the `exported` flag set to `true` and if there are declared intent-filters associated with an activity or service. In addition, it also builds a template for intents that can be sent to the activity or service.

Control Flow Analyzer (CFA). The CFA is the core component of AppSeer. It takes as input the code of a service and it determines whether that service is safe or unsafe. CFA performs a static analysis of the code to determine if there exist paths from *sources* (e.g., calls to `onStartCommand()`) to *sinks* (calls to `startForeground()` or the end of the service handling methods). If paths exist between a call to `onStartCommand()` and `startForeground()`, then the service is marked as safe. Otherwise, the service is marked as unsafe. To perform this analysis, the CFA needs to solve several challenges, listed below.

Interprocedural Analysis. A service may use helper functions, each dedicated to a particular task, which may interact with one another in complex ways. The analysis must take this fact into account and be able to follow paths across multiple function invocations.

Inheritance and polymorphism. Related to the first challenge, services inherit from the Android *Service* class and its subclasses. Developers may also introduce a class hierarchy and polymorphic functions. The main challenge for static analysis is identifying precisely the correct function implementation inside a hierarchy. In addition, to perform correct function name resolution all the functions along the class hierarchy must be available.

Code Obfuscation. The code obtained by the reverse engineering step is usually obfuscated. Therefore, different function names in the original code could be replaced with the same name in obfuscated code, while relying on overloading resolution rules to keep the original program behavior. This fact may increase the false positive rate.

To address these challenges, we use various heuristics. Given an exposed service Σ, we define the associated class hierarchy H of Σ as follows:

- H is a sequence of classes: $H = \{\Gamma_1, \Gamma_2, ..., \Gamma_n\}$;
- $\Gamma_1 \in$ *{Service, IntentService}*;
- $\Gamma_n = \Sigma$;
- $\forall i \in \{1, ..., n-1\} : \Gamma_{i+1}$ **extends** Γ_i;

- $H(k) = \Gamma_k$, meaning that we can extract a specific class from the hierarchy given its index k, with $1 \leq k \leq n$.

The first class Γ_1 in the hierarchy is either *Service* or *IntentService*, the root superclasses of all service classes. In addition, we associate each class Γ_i with a set M_i of method declarations $m_{i,j}$, each representing the name of a method and an ordered list of its parameter types.

With the help of hierarchy H and the set of of method signatures M_i associated with each class Γ_i, the CFA scans the Java code, starting from the source class Γ_n, and expanding the search along all possible paths. In particular, upon encountering a function whose invocation matches the declaration of a known function, the CFA expands the search interprocedurally to the body of that function.

The algorithm for the search is shown in Listing 1. Identifier *cursor* denotes a file cursor that navigates the class hierarchy of a given service component. The source methods are *onCreate()*, *onStartCommand()*, and *onHandleIntent()* depending on the kind of service under consideration (i.e., a *Service* or *IntentService* instance). Method *MatchDeclaration()* is responsible for extracting the signature of the method invocation and for using the class hierarchy and the associated method signatures to retrieve the correct method declaration that matches the invocation.

```
search(MethodDecl)

    if cursor.currentClass == H(1) {
       /* terminate at hierarchy root */
        return }
    // terminate because we found the sinkMethod
    if cursor.currentMethod == sinkMethod {
        cursor.componentState = SAFE
        return }

    let D = { m | m is a MethodDecl whose signature
                        matches cursor.currentMethod }

    // cursor.currentMethod not in cursor.currentClass
    if D is empty {
        cursor.hierarchyUp()
        search(MethodDecl)
        cursor.hierarchyDown()
    }
    else
        for each d in D {
            for each method call mc in method d
                search(mc)
        }
```

Listing 1: Pseudocode of the search algorithm.

To deal with the challenge of *obfuscation* the CFA expands the search into every possible (obfuscated) method name. While this may increase the rate of false positives, our next step of dynamic evaluation, which tests the results produced by the CFA, identifies the false positives. In practice we found the number of false positives to be quite modest relative to true positives.

Dynamic Tester. This component uses the Android Debug Bridge (ADB), which is part of the Android development kit, to test the results found by the previous two components. In particular, the tester uses the fetcher's results to determine the type of *unexpected intents* to send to exposed activities and to services labeled as unsafe by the CFA. After sending an intent to an exposed activity or an unsafe service, the tester scans the so-called logcat (i.e., the Android console log) to determine if the corresponding activity or service has crashed.

3.2 Exploitation

In this section, we describe how a malicious application can exploit these vulnerabilities to crash arbitrary applications on the phone (DoS attacks).

To send *explicit* intents to the victim components, the *Context* and *Class* objects of the component are needed. To obtain the former, an app can call *createPackageContext(String, int)*, which retrieves the context of the application specified by its package name. To obtain the *Class* object we used a combination of Java class loaders and reflection constructs. In particular, a subclass of *ClassLoader*, called *PathClassLoader*, can be used by a malicious app to load every class object of the operating system, including objects stored in locations owned by the *root* user (e.g., */system/frameworks/services.jar* and apk files). This code will of course have only the privileges granted to the malicious app, but this will still be enough to send intents to start services and activities. This method would allow an attacker to target every vulnerable component of every application at their own will, making the interaction flaws exploitable. Moreover, the use of reflection methods on these newly accessible Java class objects will provide a standard application with a new set of executable methods that were previously unavailable.

In addition to explicit intents, we also discovered a way to use *implicit* intents, which were disabled starting from API version 21, to start exposed services and activities. By performing a down cast from abstract class *Context* to the concrete subclass *ContextImpl*, we can access and modify fields of *Context* objects. In particular, reflection allows us to modify the field *targetSdkVersion* that holds the API level of the malicious app. The run-time modification of this field will change the target API level of the calling application, while still allowing the exploitation of constructs introduced in later versions. Thus, method *startForegroundService()* introduced in API level 26 can be invoked even though the application's target API version is modified to a value lower than 26. Specifically, by providing a value less than 21, corresponding to Android *Lollipop*, implicit intents will become available again to start a service, providing yet another way to exploit the interaction flaws we described earlier.

4 EVALUATION

To evaluate AppSeer, we studied both third-party and system applications, and we found that every analyzed application suffers from one or more of the interface flaws we discovered. Since one vulnerable component is enough to crash the entire application, we distinguish between apps that successfully protect *all* their activities and services through the Android permission mechanism and apps in which at least one component can be freely invoked by other applications. We reported all our findings to the Android Security Team through the *Google IssueTracker* platform. The interaction flaws caused by the vulnerable activities were recognized with the assignment of the CVE-2018-9447, while the foreground service defect was under active investigation as of this writing.

4.1 Results on Third-Party Apps

We analyzed the top 30 free applications available on *Google Play*, the official Android app store. Among the 27 applications that exposed at least one service, we made the following observations:

- Only 3 applications protect all their services through `SIGNATURE` level permissions. These permissions are only granted to applications published from companies sharing the same signature certificate;
- 24 applications expose at least one vulnerable service that can be started by external applications.

Overall we analyzed 120 total services. We found that only 3 services were labelled and confirmed to be safe according to the *onStartCommand() + startForeground()* pattern. These services belong to popular applications like Facebook, Facebook Messenger and Telegram. However, these apps also exposed vulnerable services. Thus, these apps are not completely safe. In fact, only the 3 apps that protect all their services with `SIGNATURE` level permissions should be considered safe, because a malicious application cannot acquire such permissions.

The testing phase of AppSeer revealed no false positives on the third-party apps. False positives are possible if two conditions are met: (1) the CFA finds paths from such service methods as *onCreate()* and *onStartCommand()* to *startForeground()*, and (2) the service takes longer than 5 seconds to reach *startForeground()* at run-time. In practice, Condition 2 is quite unlikely to occur. Given the absence also of false negatives, we report 100% precision and recall on third-party apps, which is quite encouraging.

4.2 Results on System Apps

Surprisingly, we found that system apps are also vulnerable to DoS attacks exploiting interaction flaws on activities and services. Regarding activities, we found 195 total exposed activities that fall into three groups:

(1) Activities requiring no permissions to be started (172 activities);
(2) Activities requiring `NORMAL` or `DANGEROUS` level permissions (6 activities); and
(3) Activities requiring `SIGNATURE` level permissions (17 activities).

The absence of the required preconditions leads to the killing of the target application in 14 cases for Group 1 activities and in 4 cases for Group 3 activities. This means that 14 activities can be crashed by a malicious app with an unexpected intent and without any permissions.

Regarding services, we focused on 10 of the most popular system applications including Camera, Contacts, Google Photo, Google Maps, Google Music, Phone, and Settings. We made the following observations:

- The Camera app does not export any service;
- Only the Contacts and Settings apps protect all their services through permissions. These are usually `SIGNATURE` level permissions, except for Contacts, which also uses one `DANGEROUS` permission;
- The remaining 7 system apps expose at least one service that can be started with either no permissions or with `NORMAL` level permissions (granted automatically).

On the whole, the CFA found 64 unsafe services. The testing phase found 60 true positives and 4 false positives, yielding a precision of 93.75%. The reason is that some system apps (e.g., Music) use multiple Android processes. In such cases, failure to call *startForeground()* by a service does not crash the host application. Recall remains at 100% by the absence of false negatives. We note that the use of `DANGEROUS` level permissions (as the Contacts app does) is risky because these permissions are often granted by device users [7].

4.3 Case Study: Phone App

We now describe a proof of concept that demonstrates how the termination of a system app can significantly harm a device. Suppose that the *Phone* app is under attack. This app has various relevant responsibilities, including managing outgoing and incoming phone calls and sending and receiving SMS messages.

Our study found that the *Phone* app suffers from both activity and service interaction defects. Indeed, sending an unexpected intent to one of its activities generates a *NullPointerException* due to a missing object initialization, causing the *Phone* app to crash. Moreover, the app exposes four services vulnerable to the foreground services interaction defect; one such service does not require any permission to be started. Thus, a malicious application could attack either the vulnerable activity or the vulnerable service.

The process running the *Phone* app is called *com.android.phone*. When this process is unexpectedly terminated, the cellular network signal is immediately lost and every ongoing phone call is automatically closed. Next, Android restarts the process by default, restoring the signal and eventually restarting the phone call. To attack the *Phone* app, a malicious app with the *READ_PHONE_STATE* permission can detect if a phone call is currently active in the device.

Next, a malicious app can start the vulnerable service as a foreground service and crash the *Phone* app. When the phone call is restarted, the malicious app can repeat the same steps to crash *Phone* again, in an infinite loop. The system will try to handle all the consecutive crashes of the *Phone* app by displaying an ANR dialog window every time that the application is killed. However, this process will rapidly drain all the CPU capabilities of the device, eventually leading to a soft reboot of the system.

4.4 Considerations

Our work uncovered a new set of vulnerable components both in third-party apps and in system apps. The analysis executed using AppSeer has been proven to be correct in the quasi-totality of cases. An interesting exception is represented by those applications that use two different processes for specific jobs (like managing the UI and performing heavy computations), like the system app *Google Play Music*. The analysis of this application detected six vulnerable exposed services. However, the testing phase revealed four *false positives*.

By further analyzing the *Google Play Music* app, we discovered that these four services are explicitly executed in a second process that does not manage the UI, thanks to the `android:process`

attribute in the manifest file. The process hosting the services is still affected by the foreground services vulnerability; however, the crash dialog is not visible. (Android calls it a *background ANR*.) Moreover, these services are immediately restarted after they are killed. The result is that the process responsible for the UI of the application is not affected, leaving the entire app still visible on the screen, while the process involved in computations is crashed and immediately restarted without major consequences on the user experience. We rarely observed this behaviour in third-party applications.

The component interaction flaws that we detected beg the question of remedial actions that might sidestep these defects. App failures caused by unexpected intents can be corrected by making components that receive these intents internal in the defining app, rather than exported. Access to system structures describing an app could be stopped by changing the behavior of the class loader in the Android ecosystem. The issue caused by the *startForeground-Service()* call has no simple solution. In Oreo, an exported service cannot determine whether it was started with a regular *startService()* call or a *startForegroundService()* call. If a service could make this determination, it would have at least the ability to raise itself to foreground status, depending on how it was called.

5 RELATED WORK

The reliability and security of the Android OS have been the subject of extensive investigations in the past decade [5–7, 12, 13]. In general, Google has addressed these issues by modifying Android as the system evolved from one API level to the next. The currently released version, corresponding to API level 27, is far more robust and reliable than earlier versions. However, Android users and original equipment manufacturers often delay upgrading their devices to new system versions by several months or years. As of this writing a large minority of Android devices (about 38%) are still running Lollipop (API Level 22), which was released in November 2014, according to the official Google site [2]. Thus, detected defects and vulnerabilities are still affecting a large portion of devices in circulation. For sake of brevity here we report only on a few existing approaches to reliability and security analysis of Android apps.

Fratantonio et al. [8] report on a vulnerability allowing a malicious app to modify the settings of a device by placing an overlay over the display of another app. The user of the device could be tricked into changing the device's settings. Google limited the ability to place overlays over dialogues and over specific activities in the *Settings* app. However, the permission to apply overlays to most application displays is still present, which could lead to further vulnerabilities. Wang et al. [11] obtain similar results to ours, disabling system's apps and causing soft reboots, by passing malicious objects to system methods executing code holding a mutual exclusion lock. Their work applies to Android Version 5.1 (Lollipop); Google has since modified the system services that execute the code in question; devices running the most recent Android versions are not susceptible to this vulnerability.

6 CONCLUSIONS

We explored interactions between third-party apps and system apps in Android, while focusing on activities and started services. Our automated approach first identifies components that an app exposes to client apps. We next examined the ways in which client apps can invoke exposed components in system apps and in third-party apps. Finally, we analyzed the behavior of exposed components and we flagged components as being potentially unsafe.

An advantage of our approach is its accuracy. AppSeer identified various vulnerable components in popular third-party apps and system apps. We tested the components in question at run-time to determine whether the components in question were indeed unsafe. Only about 6% of system apps deemed to be unsafe turned out to be safe when we tested them at run-time. In the future, we plan to extended our analyses to additional Android components.

In the future we plan to continue investigating vulnerabilities arising from interactions among Android components. We are especially interested in exploring Kotlin [4], a new language for programming Android applications, and its effects on component interactions. In addition, we intend to expand the capabilities of AppSeer to detect additional component interface issues for Android.

ACKNOWLEDGMENT

We thank the anonymous referees of our original submission for their valuable comments and suggestions, which helped us improve the content and presentation of our final version.

REFERENCES

[1] 2018. Common Vulnerabilities and Exposures. https://cve.mitre.org/cve/. Online; accessed 2 July 2018.
[2] 2018. Distribution Dashboard. https://developer.android.com/about/dashboards/. Online; accessed on 5 July 2018.
[3] 2018. Google Bughunter University. https://sites.google.com/site/bughunteruniversity/. Online; accessed on 2 July 2018.
[4] 2018. Kotlin Language Documentation. https://kotlinlang.org/docs/kotlin-docs.pdf. [Online; accessed July-2018].
[5] Elias Athanasopoulos, Vasileios P Kemerlis, Georgios Portokalidis, and Angelos D Keromytis. 2016. NaClDroid: Native Code Isolation for Android Applications. In *European Symposium on Research in Computer Security*. Springer, 422–439.
[6] Michael Backes, Sven Bugiel, Derr Erik, Patrick McDaniel, Damien Octeau, and Sebastian (2016) Weisgerber. 2016. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In *Proceedings of the 25th USENIX Security Symposium*. USENIX Association, Berkeley, CA, 1101–1118.
[7] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. 2012. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*. ACM, 3.
[8] Yanick Fratantonio, Chenxiong Qian, Simon Chung, and Wenke Lee. 2017. Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. In *Proceedings of the IEEE Symposium on Security and Privacy*. San Jose, CA.
[9] Wisniewski Ryszard and Tumbleson Connor. 2012. Apktool - A tool for reverse engineering 3rd party, closed, binary Android apps. https://ibotpeaches.github.io/Apktool/. [Online; accessed March-2018].
[10] Skylot. 2013. Jadx - Dex to Java decompiler. https://ibotpeaches.github.io/Apktool/. [Online; accessed March-2018].
[11] Kai Wang, Yuqing Zhang, and Peng Liu. 2016. Call Me Back! Attacks on System Server and System Apps in Android Through Synchronous Callback. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 92–103. http://doi.acm.org/10.1145/2976749.2978342
[12] Lok Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. USENIX, Bellevue, WA, 569–584. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/yan
[13] Suleiman Y Yerima, Sakir Sezer, Gavin McWilliams, and Igor Muttik. 2013. A new Android malware detection approach using Bayesian classification. In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*. IEEE, 121–128.