2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

JGRE: An Analysis of JNI Global Reference Exhaustion Vulnerabilities in Android

Yacong Gu*†, Kun Sun‡, Purui Su*§, Qi Li†, Yemian Lu*, Dengguo Feng*, Lingyun Ying*§
*Institute of Software, Chinese Academy of Sciences †Graduate School at Shenzhen, Tsinghua University
‡Department of Information Sciences and Technology, George Mason University
§University of Chinese Academy of Sciences
{guyacong, luyemian, feng}@tca.iscas.ac.cn, ksun3@gmu.edu, {purui, lingyun}@iscas.ac.cn, qi.li@sz.tsinghua.edu.cn

***Abstract*—Android system applies a permission-based security model to restrict unauthorized apps from accessing system services; however, this security model cannot constrain authorized apps from sending excessive service requests to exhaust the limited system resource allocated for each system service. As references from native code to a Java object, JNI Global References (JGR) are prone to memory leaks, since they are not automatically garbage collected. Moreover, JGR exhaustion may lead to process abort or even Android system reboot when the victim process could not afford the JGR requests triggered by malicious apps through inter-process communication.**

In this paper, we perform a systematic study on JGR exhaustion (JGRE) attacks against all system services in Android. Our experimental results show that among the 104 system services in Android 6.0.1, 32 system services have 54 vulnerabilities. Particularly, 22 system services can be successfully attacked without any permission support. After reporting those vulnerabilities to Android security team and getting confirmed, we study the existing ad hoc countermeasures in Android against JGRE attacks. Surprisingly, among the 10 system services that have been protected, 8 system services are still vulnerable to JGRE attacks. Finally, we develop an effective defense mechanism to defeat all identified JGRE attacks by adopting Androids low memory killer (LMK) mechanism.

I. Introduction

Smartphones are playing an increasingly important role in our daily life. As one of the two most popular mobile OSes on smartphones, Android has been actively used by over one billion users. Due to its popularity and open source feature, Android becomes the target for various malicious attacks such as private data leakage [23], [27], [38], [47], application repackaging [26], [32], [49], and components hijacking [20], [25], [36].

Android system adopts a permission-based security model to restrict unauthorized requests from accessing the critical system services. However, this permission model can only provide a coarse-grained access control on if an app is allowed to access system resources, but it cannot enforce a fine-grained control on how many system resources one app may consume. Due to this limitation, Android system suffers from resource exhaustion based Denial of Service (DoS) attacks [16], [24], [33], [35], [42].

Java Native Interface (JNI) [8] is a programming framework that enables Java code to call and be called by native applications and libraries written in other languages such as C, C++, or assembly. JNI enables programmers to write native methods to handle situations when an application cannot be written entirely in Java. Java objects are passed by reference between Java and native code. The Java VM uses those references to keep track of all objects that have been passed to the native code.

There are two types of JNI references. *JNI local references* are valid for the duration of a native method call, and they are automatically freed after the native method returns. *JNI global references* (JGR) remain valid until they are explicitly freed, which makes the use of JGR prone to memory leaks. What makes it worse is that malicious apps may exhaust JGR of victim processes via inter-process communication (IPC), and system services in Android provide thousands of IPC interfaces that can be misused by malicious apps. When the number of JGR in one process's runtime exceeds a system upper bound threshold (i.e., 51200), this victim process aborts. This JGR exhaustion (JGRE) attack poses a serious threat to Android, since it may lead to process abort or even Android system reboot. The Android team has noticed this problem and has fixed a few JGRE vulnerabilities in system services such as WiFi service [3] and Notification service [2]. However, those ad hoc defenses cannot guarantee to identify and fix all potential JGRE vulnerabilities.

In this paper, we first perform a systematic analysis of all potential JGRE vulnerabilities in Android. We develop a four-step JGRE analysis method to identify all vulnerable IPC methods in system services and apps that can be exploited by malicious apps. We first find all IPC interfaces from Android source code. Next, we identify all the operations of adding JGR entries in both Java code and native code. After obtaining both IPC methods and JGR entry adding points, we can build a call graph and then use static analysis to filter out all potential vulnerable IPC methods. Finally, we perform dynamic tests on each vulnerable IPC method to further verify its correctness.

We study the JGRE vulnerabilities in Android 6.0.1 and find that 32 out of 104 (30.8%) system services contain 54 vulnerable IPC interfaces that may be exploited by third-party apps to launch JGRE based DoS attacks. In addition, we find 2 pre-built core apps contain 3 vulnerable IPC interfaces. 44 out of the total 57 vulnerable IPC interfaces have not been fixed; even among the 13 vulnerable interfaces that have been

2158-3927/17 $31.00 © 2017 IEEE
DOI 10.1109/DSN.2017.40

protected by Android, 10 interfaces still suffer from JGRE attacks. We have submitted all the vulnerabilities to Android Security Team and received multiple Android Bug ID.

Next we perform a study on why Android team has not succeeded on completely fixing the JGRE attacks. We classify those threshold based ad hoc defenses into two categories, either performing JGR checking in system service helpers or performing JGR checking in the system services, and then point out their limitations. We also point out two major challenges. First, due to the fragmentation of the Android ecosystem, it is challenging to choose specific thresholds suitable for all apps and all devices. Moreover, since different interfaces provide different functions, the thresholds vary in those interfaces. If the thresholds cannot be correctly set, Android system will have a severe usability problem. Second, since most system services run in the system server process and share one JGR table, one vulnerable interface in any system service can abort the system server process and crash the system.

Finally, we propose a real time defense mechanism against JGRE attacks. It consists of three phases. First, we dynamically capture a victim process incurring creation of a large number of its JGR entries that exceeds an alarm threshold. According to our observation, the number of JGR for each system service used by benign apps is stable and small, we can safely set a fixed alarm threshold on JGR entry creation. Second, we rank each app's impact on victim process's JGR creation by performing correlation analysis on the runtime log that records the time when the corresponding IPC method is invoked and the time of JGR entry creation and deletion, which is based on another observation that, for all vulnerable IPC interfaces, the duration from invoking an IPC call to creating a JGR entry varies within a small value. Finally, we recover the system by killing the top ranking apps before the victim process's JGR entries are exhausted. The recovery phase is similar to Android's low memory killer (LMK) [41] and comply with the Android system specification [13] that specifies that any app can be killed to release the resources if certain system resources are almost exhausted. We implement our solution on an Nexus 5X phone installed Android 6.0.1. The experimental results show that it can successfully defend against all the identified 57 vulnerabilities and can even detect the JGRE attacks from multiple colluding malicious apps.

In summary, we make the following contributions.

- We systematically study the JNI global resource exhaustion (JGRE) vulnerabilities in Android. We develop a toolset to analyze the newest Android system. We discover 54 JGRE vulnerabilities in 32 system services and 3 JGRE vulnerabilities in 2 pre-built apps. We also find some vulnerable apps in Google Play.
- We study the state-of-the-art defense mechanism against JGRE attacks. We find out that current Android system adopts an ad hoc way to selectively fix a small number of vulnerable system services, and it lacks a defense solution to efficiently protect all system services from JGRE attacks.
- We develop a new defense mechanism that can prevent JGRE attacks with small performance overhead. The experimental results show that it can defend against all identified vulnerabilities and can detect multiple colluding malicious apps.

II. JGRE Attacks

A. Background

The Java Native Interface (JNI) is part of the Java Software Development Kit (SDK). It allows Java code use code and code libraries written in other languages. On the other hand, the Invocation API, which is part of JNI, can be used to embed a Java virtual machine (JVM) into native applications, thereby allowing native code to call Java code [8]. JNI divides the objects referenced by native code into two categories: *local reference* and *global references* [8]. JNI local references are valid for the duration of a native method call, and they are automatically freed after the native method returns. JNI global references (JGR) remain valid until they are explicitly freed.

System services are the core components of Android to support various functions to apps, and Android provides Binder mechanism to achieve inter-process communication (IPC) between apps and system services. A number of IPC interfaces opened by system services to third-party apps will create new JGR entries after receiving service requests from third-party apps. In Android, each process has its own dedicated Android runtime (Dalvik VM or ART runtime) along with individual runtime resource management. There is an upper bound JGR threshold (i.e., 51200), hard-coded in *art/runtime/java_vm_ext.cc*, on the number of JNI Global Reference (JGR) that each Android Runtime can sustain. Therefore, when a malicious app can make one victim process have more JGR entries than the upper bound threshold, the victim process's runtime will abort. We name this type of DoS attack as *JNI Global Reference Exhaustion (JGRE)* attack.

We know that most system services run as threads inside the *system_server* process, so the total number of JGR entries created by all those system services is constrained by the JGR threshold of system_server, which is also 51200. When the runtime of the *system_server* process aborts, the entire Android system crashes, followed by a soft reboot. For instance, *clipboard service* runs in *system_server* process as a thread, and it has one vulnerable IPC interface *addPrimaryClipChangedListener()*, which adds a clipboard listener that will be notified when the content in system's primary clipboard changes. Once an app invokes this interface, the clipboard service allocates new JGR entries in its own process, and those JGR entries will not be released until the corresponding app process exits. Therefore, a malicious app can invoke a large number of IPC calls to exhaust the JGR entries of the *system_server* process, leading to soft reboot of Android system.

B. Challenges on Defeating JGRE

Android permission-based security model cannot successfully prevent JGRE attacks since it does not provide a fine-grained control on the process resources such as JNI global

reference that may be consumed by each authorized app. Android team has noticed JGRE attacks and fixed a few vulnerable system services such as WiFi service [3] and Notification service [2]. However, JGRE attacks have not been well studied. Since there is no systematic work on identifying potential JGRE attacks against all system services in Android, the threat of JGRE attacks has not been fully recognized.

There are three challenges to discover all JGRE vulnerabilities in a given Android system. First, we should find all IPC methods that may be accessed by third-party apps. The number of system services and prebuilt apps increases along with the evolvement of the Android system. For instance, Android version 6.0.1 contains 104 system services and 88 prebuilt apps, which provide thousands of IPC methods. Most IPC interfaces are defined in *AIDL* files and can be easily located; however, some IPC interfaces are not, and we have to analyze both Java code and native code, since 5 system services are implemented in native code. Second, we need to identify all attack paths on JGR. JGR's *add entry* and *remove entry* interfaces are implemented in native code; however, most of the real call stack that triggers *add* and *remove* operations are from Java code. Thus, it is important to investigate all Java code and native code to find the operations on JGR. Third, we should identify the vulnerable IPC methods. There may exist a large number of interfaces that may allocate JGR resources; however, since JGRE can succeed only when a large number of JGR can be allocated but cannot be released quickly, we need to develop a mechanism to narrow down to the real JGRE vulnerabilities.

Due to the lack of fully understanding of JGRE attacks, existing solutions against JGRE attacks are ad hoc and only pertinent to specific system services. Android has not provided one generic countermeasures against all JGRE attacks.

III. JGRE Analysis Methodology

Figure 1 shows the methodology for systematically analyzing JGRE vulnerabilities that may be exploited by malicious third-party apps to launch DoS attacks in Android system. We target at identifying all vulnerable IPC methods, which can be accessed by third-party apps to exhaust JNI global references. Our analysis consists of four major components, namely *IPC method extractor*, *JGR entry extractor*, *Vulnerable IPC detector*, and *JGRE Verification*. IPC method extractor is responsible for locating all IPC interfaces from Android source code. JGR entry extractor focuses on identifying all the *add* operations of JGR entries in both Java code and native code. After obtaining both IPC methods and JGR entry adding points, we can build a call graph and then use static analysis in the vulnerable IPC detector to find out all potential vulnerable JGRE locations. Since only JGR entries that cannot be revoked efficiently may lead to JGRE attacks, we further filter out those innocent JGR operations. Finally, to improve the report accuracy on JGRE locations, we perform dynamic tests on each vulnerable JGRE location to verify its correctness in the JGRE verification step.

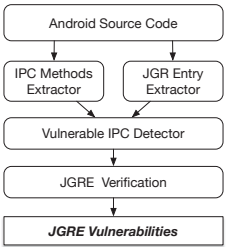

Fig. 1. JGRE analysis methodology

A. IPC Method Extractor

The goal of this component is to find out a complete list of IPC methods provided by both system services and prebuilt apps. First, we can find most IPC interfaces from *ServiceManager*, which records the IPC interfaces from the registered system services. We use SOOT [46] to study the compiled class files of Android Open Source Project (AOSP) version 6.0.1. We extract all classes' hierarchy and each class's methods. Particularly, we find out all the system services' classes that register IPC interfaces to *ServiceManager* through the *addService* and *publishBinderService* Java methods. In addition, we discover 5 native system services whose classes provide IPC interfaces through the *ServiceManager::addService* native method. Next, we find IPC interfaces from the classes in both Java code and native code. Among all classes that have AIDL definitions or are inherit classes based on *IInterface* class, if one class has one method overriding AIDL definition or IInterface, we consider this method as one IPC method.

Second, some base service class for app extension provides some default IPC interface implementation. Several pre-built core apps (e.g. Bluetooth, TTS) extend those base service class and can provide services to third-party apps through IPC interfaces. To find this type of IPC methods, we locate all abstract service classes and look up the *IBinder* interface returned from *asBinder()*. The methods contained in the IBinder interface are marked as IPC methods.

B. JGR Entry Extractor

JGR entry extractor targets at finding all Java and native methods that call the JGR add method in the native code. The method to add JGR entry in the native code *IndirectReferenceTable::Add(uint32_t cookie, mirror::Object* obj)*. However, in Android, most IPC methods are implemented in Java code, which calls the "IndirectReferenceTable::Add()" method through JNI mechanism. For instance, as Figure 2 shows, since the JNI method *android_os_Parcel::android_os_Parcel_readStrongBinder()* calls IndirectReferenceTable::Add() through a call chain, we record the corresponding Java method *Parcel.nativeReadStrongBinder()* as an JGR entry.

1) Native JGR Entry: We use static call graph analysis [7] to find all the calling paths in the native code that start from a JNI method and end at the *IndirectReferenceTable::Add(uint32_t cookie, mirror::Object* obj)* method. We find 147 paths in the native code. However, through manual

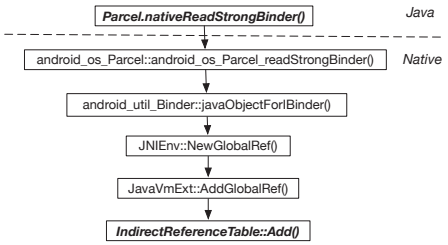

Fig. 2. An example of JGR entry in Java code

analysis, we filter out 67 paths that cannot be exploited by third-party apps, since those paths can only be gone through during the Runtime initialization stage for resource allocation, such as the *WellKnownClasses::CacheClass()* for caching classes.

2) Java JGR Entry: After locating the JGR entries in the native code, we can map them to the calling methods in the Java code. According to JNI specification [10], JNI methods are registered by using *AndroidRuntime::registerNativeMethods(JNIEnv* env, const char* className, const JNINativeMethod* gMethods, int numMethods)* in Android, where the *className* points to the calling Java class and *gMethods* contains the mapping relationship between the native method and the Java method. Therefore, we can enumerate all corresponding Java methods that trigger those JNI paths identified in the native code. For instance, some critical mappings include *Binder.linkToDeath()* to *JavaDeathRecipient::JavaDeathRecipient()*, *Thread.nativeCreate()* to *Thread::CreateNativeThread()*, and *Parcel.nativeReadStrongBinder()* to *android::ibinderForJavaObject()*.

C. Vulnerable IPC detector

With the collected JGR entry methods in the Java code, the IPC detector searches an IPC call graph to identify IPC methods that may trigger those JGR entry methods. Since not all risky IPC methods may be misused to launch a JGRE attack, we need to further narrow down the real vulnerable IPC methods.

1) IPC Call Graph Generator: We use SOOT toolset [46] to build a method-level call function graph (CFG) for each IPC method. Then we use PScout [18] to parse the indirect dependency such as Message Handler. Since Android 6.0, AOSP adopts a new Java Android Compiler Kit (Jack) toolchain [4] to generate .jack and .dex files as build target. Since PScout uses .jar file as default input, we need to use dex2jar tool [6] to convert .dex files to .jar files before inputting them into PScout.

2) Risky IPC Detector: When searching the IPC call graph of each IPC method, if the graph contains any Java JGR entry, we mark this IPC method as a risky IPC method. There are two special Java JGR entries, *Parcel.nativeReadStrongBinder()* and *Parcel.nativeWriteStrongBinder()* that are not included in the IPC method's call graph, since they are typically called by the Binder on Transact() in the Binder framework.

To solve this problem, we first enumerate all four scenarios when these two methods are called: *transmit Binder object through IPC*, *transmit IInterface object through IPC*, *transmit object containing Binder or IInterface through IPC*, and *transmit a array (or List) of Binder or IInterface through IPC*. We identify all classes that inherit from Binder or IInterface, contain Binder or IInterface, or contain one IPC method that uses Binder or IInterface as parameters. After recognizing an array type, we can use a similar method to locate all risky methods. For the type of List, since the detailed type information is unknown in code static analysis due to *Type Erasure* [15], we have to manually check if the List contains Binder or IInterface elements.

3) Risky IPC Sifter: All the risky IPC methods identified in the IPC detector can lead to the increase of JGR when they are running. However, if those JGR entries will be revoked quickly, such IPC methods can hardly be used to trigger JGRE attacks. Therefore, we sift out the following special IPC methods:

1) only contains Java JGR method *Thread.nativeCreate()*. The corresponding native JGR method *Thread::CreateNativeThread()* immediately releases the JGR entry after running.
2) Binder or IInterface as parameter will only be used inside one IPC method and won't be transmitted to other methods. The Binder or IInterface object will be collected by Garbage Collector after the IPC method ends.
3) Binder or IInterface as parameter will only be used as the read-only interface for data structures including Map, Set, or RemoteCallBackList. The Binder or IInterface object will be collected by Garbage Collector after the IPC method ends.
4) Binder or IInterface as parameter will be assigned to one Service's class member variable. Though the Binder or IInterface object will not be automatically collected after the IPC method ends, when the IPC method is called again with a different Binder or IInterface object, the last Binder or IInterface object will be revoked.

We further sift out IPC methods that cannot be accessed by third-party apps according to the permission map generated by PScout.

D. JGRE Verification

We use dynamic testing to verify the exploitability of the identified IPC methods. We first modify the Android source code to monitor IPC communication and JGR add/remove processes. Particularly, we use *Thread.currentThread().getStackTrace()* to record IPC's call stack along with Binder object's Object ID. We use AOSP's *CallStack* library to record the call stack for *JGR add* and *JGR remove* along with the JGR number into a log file.

For each risky IPC interfaces identified in the previous stage, we use a semi-automatic method to generate the test cases for verification. We manually extract parameters, e.g., package name and binder object, and feed them to IPC interfaces so as

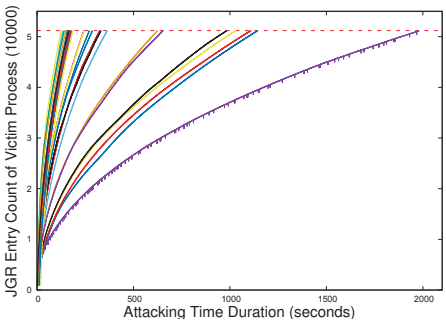

Fig. 3. Misuse effectiveness of 54 vulnerable IPC interfaces

to invoke JGR allocation. Meanwhile, we utilize Javapoet [9] to generate test code with only small manual changes, i.e., feed the analyzed parameters. Also, by analyzing the reference of framework.jar file, our test cases can directly use the hidden class or methods in the standard Android SDK. We trigger 60000 IPC requests and use DDMS [5] tool to trigger target process's Garbage Collector periodically. From the JGR number recorded in the log, we can verify if one IPC method is vulnerable and exploitable.

IV. JGRE Analysis Results

We perform a study on the JGRE attack in Android 6.0.1. In summary, we discover that 32 out of 104 (30.7%) system services contain 54 vulnerable IPC interfaces that may be exploited by malicious third-party apps. In addition, we find 2 prebuilt apps containing 3 vulnerable IPC interfaces.

A. Vulnerabilities in System Services

Figure 3 shows the increasing JGR entry counts for 54 vulnerable interfaces in system services when they are under JGRE attacks. The attacks succeed when the JGRE entry counts are beyond the 51200 threshold. Due to distinct implementation logics, the JGR incremental rates vary on different vulnerabilities. For instance, it takes 1800 seconds to launch the DoS attack when exploiting *INotificationManager.enqueueToast(String, ITransientNotification, int)* interface, while it takes only 100 seconds when using *IAudioService.startWatchingRoutes(IAudioRoutesObserver)* interface.

By scrutinizing the 54 vulnerable IPC interfaces one by one, we find that 44 vulnerable interfaces have not been protected at all. Even among the 13 IPC interfaces that have been protected, 10 interfaces are still vulnerable to JGRE attacks.

B. Unprotected Vulnerable IPC Interfaces

Table I summarizes 44 unprotected vulnerable IPC interfaces in Android. Among the 26 unprotected vulnerable system services, 19 system services can be exploited without acquiring any permission, 4 system services require Normal level permissions, and only 3 system services require Dangerous level permissions.

Android team was well aware of JGRE attacks [2], [3], [35], but why can we still find so many unprotected system services? The major reason is that Android system adopts an ad hoc way to selectively fix a small number of vulnerable system services. There are two major challenges when extending this method on all vulnerable IPC methods. First, because of the fragmentation of the Android ecosystem, it is challenging to choose specific thresholds suitable for all apps and all devices. Moreover, since different interfaces provide different functions, the thresholds vary in those interfaces. For example, *WifiManager* sets the threshold to 50 per process, while *InputManagerService* sets the threshold to only 1 per process. If the thresholds cannot be correctly set, Android system will have a severe usability problem.

Second, most system services run in one common *system_server* process, and they share one JGR table. Therefore, one vulnerable interface in any system service can be used to attack the *system_server* process. It is a challenge to locate all vulnerable methods. Moreover, new Android versions will provide some additional IPC methods in system services. Thus, even if all vulnerable IPC methods have been fixed, new version may introduce new vulnerable IPC methods.

C. Protected but still Vulnerable IPC Interface

The first JGRE attack had been identified and fixed in 2009 [3]. Since then, Android Security Team has selectively fixed 13 vulnerable IPC methods by adopting two types of defense mechanisms. The first one depends on certain service helper classes to constrain third-party apps' JGR requests. However, this approach can be totally circumvented. The second approach enforces per process constraint by directly modifying the System Services to limit the JGR entries that can be requested by each third-party app. Improper implementation can still fail the protection enforced by this approach.

1) Circumventing System Service Helpers: Android system provides a number of system service helper classes to help developers access system services by encapsulating the functions in system services to interfaces that are more developer-friendly. App developers usually invoke the methods from system service helper instead of from the service directly. For instance, *ActivityManager* is the helper class of *ActivityManagerService*, and *AudioManager* is the helper class of *AudioService*. Table II lists 9 vulnerable methods that have been protected by service helper classes to constrain JGR IPC requests.

This defense approach can effectively prevent ignorant developers from unintentionally requesting too many JGR entries; however, a malicious app can easily bypass it by directly calling Binder interface to communicate with the system services. We verify that all 9 vulnerable interfaces in Table II still can be exploited. Note that among 9 vulnerable interfaces, only 2 interfaces of WifiManager have explicit code with comments dedicated for DoS mitigation. The wifi service is implemented in $WifiServiceImpl.java$ and its helper class is $WifiManager$. Because the *acquireWifiLock()* interface in wifi service is vulnerable, its helper class provides a specific protection as shown in Code-Snippet 1 [1]. When a

[1]All code we present in this paper has been simplified for brevity.

TABLE I
UNPROTECTED VULNERABLE IPC INTERFACES

Service Name	Vulnerable IPC Interface	Required Permission (Protection Level) in AOSP 6.0.1
location	addGpsStatusListener	ACCESS_FINE_LOCATION (dangerous)
sip	open3	USE_SIP (dangerous)
	createSession	USE_SIP (dangerous)
midi	registerListener	-
	openDevice	-
	openBluetoothDevice	-
	registerDeviceServer	-
content	registerContentObserver	-
	addStatusChangeListener	-
mount	registerListener	-
appops	startWatchingMode	-
	getToken	-
bluetooth_manager	registerAdapter	-
	registerStateChangeCallback	BLUETOOTH (normal)
	bindBluetoothProfileService	-
	bindBluetoothProfileService	-
audio	registerRemoteController	-
	startWatchingRoutes	-
country_detector	addCountryListener	-
power	acquireWakeLock	WAKE_LOCK (normal)
input_method	addClient	-
accessibility	addAccessibilityInteractionConnection	-
print	print	-
	addPrintJobStateChangeListener	-
	createPrinterDiscoverySession	-
package	getPackageSizeInfo	GET_PACKAGE_SIZE (normal)
telephony.registry	addOnSubscriptionsChangedListener	READ_PHONE_STATE (dangerous)
	listen	READ_PHONE_STATE (dangerous)
	listenForSubscriber	READ_PHONE_STATE (dangerous)
media_session	registerCallbackListener	-
	createSession	-
media_router	registerClientAsUser	-
media_projection	registerCallback	-
input	vibrate	-
window	watchRotation	-
wallpaper	getWallpaper	-
fingerprint	addLockoutResetCallback	-
textservices	getSpellCheckerService	-
network_management	registerNetworkActivityListener	CHANGE_NETWORK_STATE (normal)
connectivity	requestNetwork	CHANGE_NETWORK_STATE (normal)
	listenForNetwork	ACCESS_NETWORK_STATE (normal)
activity	registerTaskStackListener	-
	registerReceiver	-
	bindService	-

TABLE II
VULNERABLE IPC INTERFACES PROTECTED BY SERVICE HELPER CLASSES

Service Name	Service Helper Class	Vulnerable IPC Interface
clipboard	ClipboardManager	addPrimaryClipChangedListener
accessibility	AccessibilityManager	addClient
launcherapps	LauncherApps	addOnAppsChangedListener
tv_input	TvInputManager	registerCallback
ethernet	EthernetManager	addListener
wifi	WifiManager	acquireWifiLock
		acquireMulticastLock
location	LocationManager	addGpsMeasurementsListener
		addGpsNavigationMessageListener

third-party app calls *WifiManager.acquire()* multiple times, the WifiManager examines whether the total number of requests exceeds the maximum lock number that an app can acquire. If the number exceeds the threshold, WifiManager will release the lock immediately. As seen in the comments in AOSP, this mechanism aims to defend against DoS attacks [2].

Code-Snippet 2 shows a piece of malicious code that can directly communicate with WifiService through IPC without going through the helper class. After we reported this type of attacks to Google, Android Security Team confirmed this vulnerability as a "resource exhaustion issue". The other seven methods also constrain the number of JNI entries and can be totally circumvented. Android Security Team classifies them into the same cluster of "resource exhaustion issues" after we submitted the corresponding bug reports.

2) Implementation Flaws on Per Process Constraint: An alternative defense approach is to limit the request number per process at the system service side. We find 4 interfaces

[2]The comments explicitly say "prevent apps from creating a ridiculous number of locks and crashing the system by overflowing the global ref table"

Code-Snippet 1 Protection code in WifiManager.java.

```
/* Maximum number of active locks we allow. This
 * limit was added to prevent apps from creating a
 * ridiculous number of locks and crashing the system
 * by overflowing the global ref table.
 */
private static final int MAX_ACTIVE_LOCKS=50;
public void acquire(){
    mService.acquireWifiLock(mBinder, mTag);
    if (mActiveLockCount >= MAX_ACTIVE_LOCKS){
        mService.releaseWifiLock(mBinder)
        throw new Exception("Exceeded
            maximum number of wifi locks");
    }
    ...
}
```

Code-Snippet 2 Sample attack code on wifi service.

```
IWifiManager wifiService = IWifiManager.Stub
    .asInterface(ServiceManager.getService("wifi"));
for (int i = 0; i < 51200; i++) {
    wifiService.acquireWifiLock(new Binder(),
        1, "test" + i,  null);
  }
}
```

protected by this approach, as shown in Table III. We can see that "per process constraint" is an effective defense mechanism against JGRE attacks when it is implemented correctly. However, after studying the detailed implementation code, we find that one interface of Notification service still can be attacked.

Code-Snippet 3 Protection code in NotificationManagerService.java.

```
public void enqueueToast(String pkg, ...) {
    boolean isSystemToast = isCallerSystem()
        || ("android".equals(pkg));
    //Limit the number of toasts that any given
    //package except the android package can enqueue.
    //Prevents DOS attacks and deals with leaks.
    if (!isSystemToast){
        if (count >= MAX_PACKAGE_NOTIFICATIONS){
            Slog.e(TAG, "Package has already posted"
              + count + " toasts. Not showing more");
            return;
        }
    }
    mToastQueue.add(record);
    ...
}
```

The protection on *NotificationManagerService.enqueueToast()* interface limits the number of Toasts that can be enqueued by each process, except for the system toasts. As shown in Code-Snippet 3, the *NotificationManagerService.enqueueToast()* takes the first parameter as caller's package name, and it considers the toast as a system toast if this parameter is set to "android". However, an attacker can bypass this restriction by directly invoking *INotificationManager.enqueueToast()* method and passing "android" as the first parameter instead of its own package name to this method. Thus, a malicious app with zero permission can enqueue enough toasts to eventually exceed the limit of JGR table. We reported this vulnerability to Android Security Team and received a confirmed Bug ID.

TABLE III
IPC INTERFACES PROTECTED BY PER PROCESS CONSTRAINT

Service Name	IPC Interface	Protected?
notification	enqueueToast	No
display	registerCallback	Yes
input	registerInputDevicesChangedListener	Yes
	registerTabletModeChangedListener	Yes

D. Vulnerabilities in Apps

In addition to system services, we also study the vulnerable apps that may crash under the JGR DoS attacks. Among 88 prebuilt core apps, We find three vulnerabilities in two apps, namely, Bluetooth and PicoTts, as shown in Table IV. For instance, PicoTts's PicoService inherits from the *android.speech.tts.TextToSpeechService* interface, which is a base service class that provides default implementation of *ITextToSpeechService* IPC methods. The *setCallback()* method in the default implementation increases JGR entry number whenever it is called, and all the JGR entries can be revoked only when the requesting third-party app exits. Thus, a malicious app can abuse the usage of this method to crash the app. Note all apps that extend *android.speech.tts.TextToSpeechService* and open IPC interface to third-party apps are vulnerable to JGRE attacks, including "Google Text-to-speech Engine" app that has been installed 10^{10} times.

TABLE IV
VULNERABLE PREBUILT CORE APPS

App	Code Path in AOSP	Vulnerable IPC Method
PicoTts	external/svox/pico	PicoService.setCallback()
Bluetooth	packages/apps/Bluetooth	GattService.registerServer()
		AdapterService.registerCallback()

We also extend our study to find vulnerable third-party apps. Comparing to system services, third-party apps have fewer JGR vulnerabilities, since few apps open IPC interface to other third-party apps. We download 1000 Android apps from Google Play marketplace and find only three apps are vulnerable to JGRE attacks. Table V shows these three apps and the corresponding vulnerable IPC interfaces.

TABLE V
VULNERABLE THIRD-PARTY APPS

App	# of downloads	Vulnerable IPC Interface
Google Text-to-speech	$1{*}9^{10}$-$5{*}9^{10}$	TextToSpeechService.setCallback()
Supernet VPN	$1{*}10^{6}$-$5{*}10^{6}$	IOpenVPNAPIService.registerStatusCallback()
SnapMovie	$1{*}10^{6}$-$5{*}10^{6}$	IMainService.a()

V. JGRE COUNTERMEASURE

Based on two key observations on JGR usages, we develop a JGRE countermeasure that can effectively defeat all identified JGRE attacks.

Observation 1. The number of JGR for each system service used by each benign app is stable and small.

We study the number of system services' JGR by downloading the top popular free apps from Google Play marketplace and installing them in an Nexus 5X. Due to the limitation of the 16 GB internal storage, we can install up to 100 apps on

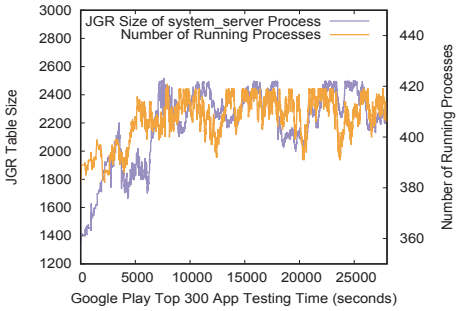

Fig. 4. The size of system_server process's JGR table (left Y axis) and the number of running processes (right Y axis).

this platform simultaneously. Therefore, we measure 300 top popular apps in three rounds. We use MonkeyRunner [12] to run these apps. For each app, we run it for two minutes and then switch it to a background process by simulating pressing the *HOME* button.

We record the JGR number of system_server process as well as the number of running processes in Android. Figure 4 shows that the number of JGR changes between 1000 and 3000, which is small compared to the JGR threshold 51,200. Also, the number of running processes is between 382 and 421. There are 382 processes running on stock Android that has not installed any third-party apps. Because the Android low memory killer will automatically terminate processes, after running all 100 top popular apps, we can see at most 39 apps running in the system simultaneously. When one process is terminated, its corresponding JGR entries will be released.

Observation 2. For all vulnerable IPC interfaces, the duration from an IPC call being invoked to the creation of a JGR entry varies within a small value.

The duration can be expressed as $Delay + \Delta$, where $Delay$ is a constant, which indicates the minimum latency between IPC call request and JGR creation, and Δ is a variable where $\Delta \geq 0$, which indicates the deviation of $Delay$. Since it is difficult to accurately measure Delay, we instead measure the duration from execution of a vulnerable IPC method to the creation of a JGR entry triggered by the method, which does not impact the accuracy of measuring Δ values. Figure 5 shows that the distribution of execution duration of *telephony.registry.listenForSubscriber()* under a JGRE attack. The entire attack process invokes the vulnerable interface 50,236 times. The execution time increases along with the increasing number of the interface invoked, since more lookup time is required to search the stored data. When the total number of the invoked interfaces is smaller, the duration for each execution is stable.

We measure the execution duration of all 54 vulnerable interfaces, which are invoked 1,000 times individually. Since different services have various process to generate a JGR entry, instead of measuring the duration from the interface being invoked to the creation of a new JGR entry, we only measure execution duration of the interfaces. Figure 6 shows cumulative distribution function (CDF) of execution for each

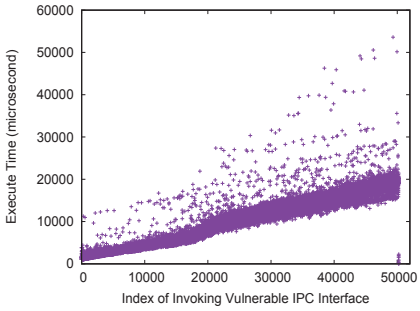

Fig. 5. The distribution of the execution duration of telephony.registry.listenForSubscriber() during an attack.

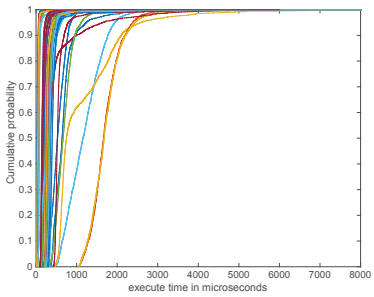

Fig. 6. The cumulative probability of all the 55 vulnerable IPC methods' execution time during 1000 IPC calls.

interface. We can see that the Δ of execution durations from various interfaces are close.

A. JGRE Defense Mechanism

Our countermeasure includes three phases to defend against JGRE attacks. First, we capture a victim process when the number of its JGR entries exceeds an alarm threshold, which is set according to our first observation. When a process is under attack, the number of new JGR keeps increasing and the number of deleted JGR lags behind the newly created one.

Second, we record the time when the corresponding IPC method is invoked and the time of JGR entry creation and deletion. Based on our second observation, we can use recorded data to infer each app's impacts on the victim process's JGR creation. We calculate the maximum value of Δ of all vulnerable services. Then, based on the time of IPC calls and creation of JGR entries for each IPC interface, we can compute all possible $Delay$ values.

By iteratively choosing a value between 0 and the maximum delay and evaluating which IPC call's execution delay is equal to the value, we can obtain different sets of IPC calls consuming the same $Delay$. It counts the number of each suspicious IPC interface's calls and computes the total number of IPC calls triggered by an app, and computes *jgre_score* for each app according to the number of IPC calls invoked by it. A higher *jgre_score* of an app indicates that it has more impacts on victim process's JGR creation. Algorithm 1 shows the pseudo-code of JGR scoring algorithm. When there are

more than one app colluding to construct JGRE attacks, we can still identify them since these apps must trigger much more JGR entry creation than other apps to successfully launch the attacks.

Algorithm 1 JGR Scoring Algorithm

Input: Δ, time of IPC calls, time of creation of JGR entries
Output: jgre_score of an App

$IPCTypes$ = type of IPC interfaces triggered by an App;
$IPCCalls$ = time of IPC calls triggered by an App;
$JGRAdds$ = time of JGR entry creation in victim process;
$TimeLen$ = time interval of data collection;
$JgreScore$ = 0 ▷ jgre_score: number of max suspicious IPC calls

for each $IPCType$ in $IPCTypes$ **do**:
  $IPCCallOfType = IPCCalls.get(IPCType)$
  $ThisType_{max}$ = 0
  $Delay[TimeLen]$ = 0
  for each $IPCTime$ in $IPCCallOfType$ **do**:
    for each $JGRTime$ in $JGRAdds$ **do**:
      $MinDelay = JGRTime - IPCTime$
      $MaxDelay = JGRTime - IPCTime + \Delta$
      for each $delay$ in $[MinDelay, MaxDelay]$ **do**:
        $Delay[delay]$++;
      end for
    end for
  end for
  $ThisType_{max} = maxOf(Delay[TimeLen])$
  $JgreScore = JgreScore + ThisType_{max}$
end for
return $JgreScore$

Note we cannot identify malicious apps by simply finding the highest number of IPC calls since IPC calls may not trigger the creation of new JGR entries. Instead, our countermeasure accurately detects JGR consumption by inferring the execution duration of each IPC call, which cannot be faked by adversaries.

Finally, we will continue to kill the top ranking apps until the number of victim process's JGR back to a normal value, which is consistent with the Android system specification [13]. In other words, if system resources are exhausted, we can kill any apps to release the resources used by the apps. Our defense mechanism is triggered when the processes or the system run short of resources, similar to Android's low memory killer (LMK) [19]. However, it is more difficult to track IPC calls triggering JGR operations than to monitoring normal system resources in Android LMK. Detailed comparison can be found in Section VII.

B. Implementation

We present a prototype of our defense on an Nexus 5X phone installed Android 6.0.1. Figure 7 shows the architecture of our defense, which extends Android Runtime to collect JGR creation information of each app and extends Binder driver to collect IPC call information. We build a JGRE Defender as a system service that analyzes the data collected from Android Runtime and binder driver to identify malicious apps.

We extend Android Runtime to monitor the creation and deletion of JGR entries triggered by each app. Once the number of created JGR entries exceeds 4,000, it starts to record the time of the events. It delivers the information to JGRE defender when the number of new JGR entries exceeds 12,000. The extended Android binder driver records the related data

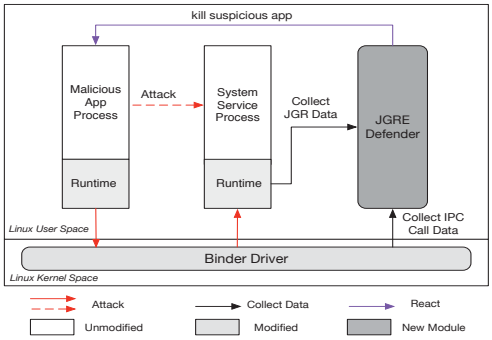

Fig. 7. JGRE defense architecture.

of IPC calls on *from_pid*, *to_pid*, *target_handle*, *to_node* and *timestamp*. It creates a file */proc/jgre_ipc_log* in memory to store the data. Via the proc filesystem (procfs) [14], it can quickly write and read the data between the Linux kernel space and the user space. Also, we set the permission of the file so that it can be only accessed by system service but not third-party apps. JGRE Defender runs as a standalone service. Android Runtime of a process notifies JGRE Defender by sending the JGR data if it is under attack. The defender reads the IPC call data from */proc/jgre_ipc_log* for further analysis.

The JGRE Defender uses Algorithm 1 to compute *jgre_score* for all apps. Then it issues a command of "am force-stop" to kill the top ranking apps until the number of victim process's JGR back to a normal value. Note that although JGR data delivered from untrusted apps can be faked, a malicious app cannot trick JGRE Defender into killing benign apps since the IPC data is reported by the kernel.

C. Effectiveness Analysis

To validate the effectiveness of our scheme, we perform two experiments. First, we verify if our scheme can effectively defend against all the identified 54 vulnerabilities in system services and 3 vulnerabilities in the prebuilt apps. Second, we verify if our scheme can detect the JGRE attacks constructed by multiple colluding malicious apps, when four colluding malicious apps target on accessing different vulnerable system services and one benign app generates a large number of invulnerable IPC calls.

Detect Single Malicious App. We install top 100 apps downloaded from Google Play marketplace. The malicious app runs in the background. In the meanwhile, we use MonkeyRunner [12] to launch the benign apps. In this experiment, we set Δ to the average value of all system services, i.e., 1.8 ms. The experimental results are shown in Figure 8. Our scheme can accurately detect this attack since the number of suspicious IPC calls triggered by the malicious app is significantly larger than the number triggered by the benign app.

Detecting Multiple Colluding Attacks. We construct an attack with four colluding apps, each one leveraging one vulnerable interface. In the meanwhile, a benign app launches a large number of benign IPC calls. During the attacks, the

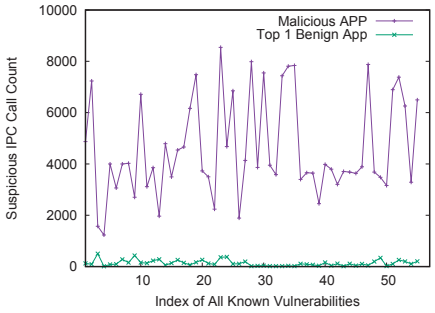

Fig. 8. The number of IPC calls: malicious vs. benign Apps.

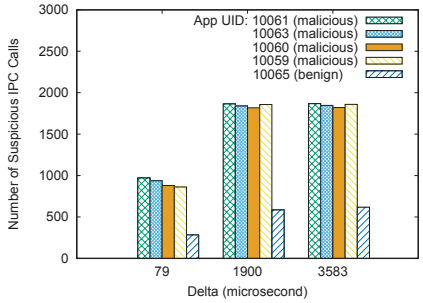

Fig. 9. The number of suspicious IPC calls triggered by top five apps with three different Δ.

benign app keeps triggering IPC calls with the interval between two IPC calls varying between 0 and 100 ms. We evaluate the impacts of choosing different Δ on the detection accuracy.

Figure 9 illustrates the top five apps that trigger the most numbers of suspicious IPC calls with three different values of Δ. We observe that the top four numbers of the suspicious IPC calls triggered by the malicious apps are significantly larger than the fifth app with the same setting of Δ. We confirm that the top four apps in the table are all malicious apps. Though the benign app triggers a large number of IPC calls, the number of IPC calls is still much smaller than the number triggered by the attacks.

D. Performance Overhead

We evaluate the response delays of our defense approach by measuring the delays of identifying attack sources. Also, we measure the overhead incurred by recording JGR and IPC calls. We use the Android default build as our baseline for comparison.

1) Response Delay: We test all vulnerable IPC interfaces among 57 vulnerable services (i.e., 54 in system services and 3 in prebuilt apps). We observe that most of detection delays are within one second except that it takes more than one second to detect attacks on the three vulnerable interfaces. In particular, it takes around 3.6 seconds to detect the attacks to *MidiService.registerDeviceServer()*. Figure 3 shows that the least time duration to construct an JGRE attack is around 100 seconds, which is much larger than 3.6 seconds. Thus, an JGRE attack cannot evade our defense.

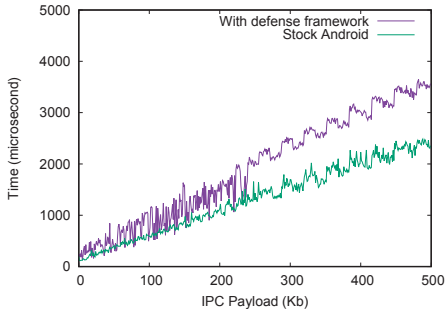

Fig. 10. The delays incurred by IPC call executions.

2) Computation Overhead: It is important to evaluate the impacts of the frequent operations on recording IPC calls and adding/removing JGR operations. To evaluate the impacts on the IPC operations, we measures the delays of IPC calls by delivering byte array via IPC methods. In total, we conduct 500 rounds of byte array delivery. In each round of data delivery, the size of the array keeps increasing with 1,024 bytes. As shown in Figure 10, our scheme incurs maximal 1.247 milliseconds for each IPC call, and the overhead increase is about 46.7%.

We measure the overhead of recording JGR operations by running a pair of attack app and victim app. Our scheme does not interfere with the JGR operation when the number of JGR entries is less than 4,000. We do not observe any obvious delays. After the number of JGR entries reaches to 4,000, the recording operations incur around 1 μs delay.

We apply several optimizations to reduce the overhead and improve the performance of our defense mechanism. First, according to Observation 1, our defense begins to interfere with IPC calls and collect data only when the number of JNI creation triggered by the victim processes exceeds a threshold. Second, the JGRE Defender is triggered only when the number of new JGR entries exceeds 12,000 to reduce the size of recorded data for analysis. Furthermore, we leverage a *segment tree* [39] data structure to implement Algorithm 1 to reduce the memory overhead. *Segment tree* is a efficient tree data structure for storing intervals (or segments), which perfectly meets our requirement that Algorithm 1 needs to store and process various $intervals$ such as $IPCCalls$ and $JGRAdds$.

VI. Discussion

This paper studies JGRE vulnerabilities that can lead to DoS attacks against a large number of system services and some prebuilt apps on Android, and then we develop a defense mechanism to mitigate the JGRE attacks. Now we discuss some limitations of our defense and point out that the root cause of JGRE attacks may result in other security problems.

False negatives in analyzing JGRE. Similar to the traditional static analysis approaches, our approach may incur false negatives on not monitoring all IPC interfaces. First, though most apps communicate with system services via Binder based IPCs, there still exists other IPC techniques, such as

unprotected broadcast receivers, Anonymous Shared Memory (ASHMEM), as well as IPC via Linux sockets, pipes, and signals. Though these techniques are rarely used by system services to provide interfaces that can be accessed by third-party apps, they may exist and be misused to launch JGRE attacks. Second, we compute the parameters according to our analysis on each IPC method during the dynamic test stage, so we cannot guarantee to find a complete list of IPC vulnerabilities due to the limitation of dynamic testing.

Exploiting JGRE vulnerability via multiple attack paths. Observation 2 assumes each IPC method only has one attacking path and thus the execution time is stable; however, attackers may exploit the vulnerabilities via multiple attack paths. Our defense scheme can still detect this attack by first classifying different IPC calls triggered by the same IPC method according to code execution paths and then counting the total number of IPC calls in the same categories. Then the attacks can be identified and throttled according to the number.

DoS attack towards other resources. JGR is one of the critical resources of Android Runtime. The root cause of JGRE attack is that the security enforcement in the current Android system cannot prevent authorized apps from sending excessive service requests to consume the limited resources allocated for system services. There may exist other resource consumption vulnerabilities related to other system resources such as memory, file descriptor, and internal storage. Our defense cannot be directly applied to prevent the DoS attacks against other resources; however, it may trigger some of our future research.

VII. Related Work

JNI Security. JNI security has been well studied. Tan et al. [45] analyze the native code in JDK 1.6 and find some bugs such as unexpected control flows, bugger overflows, JNI misusing, etc. Lee et al. [34] use state machines to detect foreign function interface (FFI) violations. They build a bug detection tools for JNI named *Jinn* to dynamically detect constraint violation during a java program running. Qian et al. [37] detect information leakage by tracing information flows going through JNI in Android apps. Robusta [43] uses software-based fault isolation (SFI) to isolate the native code of a java program to a single sandbox. FlexDroid [40] builds JNI sandboxing of Android apps using Hardware Fault Isolation (HFI). It introduces two memory domains named JNI domain and Java domain to isolate JNI code and Java code to different spaces. NativeGuard [44] isolates the native library of an Android app to a separate service app, and the rest of the original app composes the client app.

DoS Attacks. A series of DoS attacks by exploiting vulnerabilities of Android system services have been identified [24], [33], [42]. Huang et al. [33] discover a design flaw in the concurrency control of Android system services and carry out a series of DoS attack based on this flaw. Shao et al. [42] focus on security enforcement inconsistencies in Android framework. Cao et al. [24] focus on the vulnerabilities happened in the input validation of Android SS interfaces. In addition, DoS attacks could also be triggered by the depletion of some Android system resources. Lineberry et al. [35] successfully launch a DoS attack by popping up a lot of toasts which results in the restart of the device. This vulnerability has been fixed since Android 4.0. The use of Flash SMS can also leads to DoS attack [21], causing the phone restart and the disconnection of the network. The vulnerability has been fixed in Android 4.4.2. Armando et al. [16] trigger DoS attacks through forking a lot of Zygote processes. Viadyanathan et al. [22] extract and monitor the key variables occurring in the early part of DoS attacks. Our work focuses on the DoS attacks on the JNI global reference resources that may be manipulated through IPC interfaces.

Android Out Memory Management. Linux adopts out-of-memory (OOM) killer [11] to handle low memory conditions. However, since it incurs significant performance degradation [19], Android uses low memory killer (LMK) to recover the system during low memory circumstances. LMK classifies process into different groups using *oom_score_adj*. In low memory situation, it kills apps with a victim selection process based on *oom_score_adj*. There are some works on predicting apps that users may still want to use to improve the effectiveness of LMK [48], [50]. Furthermore, Baik et al. [19] presents a policy-extendable LMK filter framework to enhance LMK victim selection mechanism.

Similar to LMK, our defense mechanism is also triggered when processes or the system run short of resources (e.g., JGR or memory), and then attempts to find and kill suspicious apps based on their scores. However, there is a big difference between LMK and our defense. In Android, it is easy to know how many memory occupied by each process, but it is challenging to find how many JGR creation are triggered by each app. Therefore, we cannot directly leverage LMK to defend against JGRE. In particular, JGRs add entry and remove entry interfaces are implemented in native code, while most of the complex functional logic code of system services is implemented in Java. Moreover, IPC calls are invoked between two processes, which makes it extremely difficult to identify which specific JGR creation is triggered by which IPC call and from which app. Our defense mechanism addresses these challenges by recording and analyzing the behaviors of each suspicious app.

Android static analysis tools. Static analysis technology has been studied for years, and a lot of well designed tools have been proposed [1], [17], [28]–[31]. Kirin [28] examines the permissions of an app to determine whether the app may contain dangerous functions. AdRisk [31] analyzes the permissions used in ad libraries by checking the API calls in those libraries, and then identifies possible data leakage and dangerous paths. Flowdroid [17] is a precise context, flow, field, object-sensitive static taint analysis framework that can be used to detect possible sensitive data leakage in Android apps.

VIII. Conclusion

In this paper, we systematically study the JNI Global Reference resource exhaustion vulnerabilities in Android system. We build a toolkit to analyze IPC methods provided by Android system services that can be accessed by third-party apps. We discover 54 JGRE vulnerabilities in 32 system services and three JGRE vulnerabilities in two prebuilt apps. Android security team confirmed all our findings. Furthermore, we develop a new defense mechanism to defend against the JGRE attacks. We implement our mechanism in Android 6.0.1 and the experimental results show that it can successfully prevent all known JGRE attacks with small overhead.

Acknowledgments

The research is supported by the National Key Research and Development Program of China under Grant 2016YFB0800102, and the National Natural Science Foundation of China under Grant 61572278, 61572483, and 61502468. Kun Sun's work is supported by U.S. Office of Naval Research under Grant N00014-16-1-3214 and N00014-16-1-3216. Q. Li and L. Ying are the corresponding authors.

References

[1] Androguard. http://code.google.com/p/androguard.
[2] Code to Fix a JGRE Vulnerability in Notification Service. https://goo.gl/NvsMVe.
[3] Code to Fix a JGRE Vulnerability in Wifi Service. https://goo.gl/YEobmk.
[4] Compling with Jack. https://goo.gl/o9RYX8.
[5] DDMS. http://goo.gl/J7MBC4.
[6] Dex2jar. https://goo.gl/skfQLl.
[7] Doxygen. http://goo.gl/cy0NJL.
[8] Java Native Interface Specification. http://goo.gl/zqHp29.
[9] Javapoet. https://goo.gl/nsIHR3.
[10] JNI Functions: Registering Native Methods. http://goo.gl/DdZb0o.
[11] Linux OOM Killer. https://linux-mm.org/OOM_Killer.
[12] MonkeyRunner. https://goo.gl/xcy6ha.
[13] Processes and Application Life Cycle. https://goo.gl/Vsed4i.
[14] Procfs. https://en.wikipedia.org/wiki/Procfs.
[15] Type Erasure. https://goo.gl/qJjenO.
[16] Alessandro Armando, Alessio Merlo, Mauro Migliardi, and Luca Verderame. Would You Mind Forking This Process? A Denial of Service Attack on Android (and Some Countermeasures). In *Information Security and Privacy Research*, pages 13–24. Springer, 2012.
[17] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. PLDI '14.
[18] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android Permission Specification. CCS '12.
[19] Kunhoon Baik, Jongseok Kim, and Daeyoung Kim. Policy-extendable LMK Filter Framework for Embedded System. In *Linux Symposium*, page 49.
[20] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the App is That? Deception and Countermeasures in the Android User Interface. S&P '15.
[21] Bogdan. 0class2dos. http://goo.gl/BmM6rR.
[22] J. B. D. Cabrera, L. Lewis, Xinzhou Qin, Wenke Lee, R. K. Prasanth, B. Ravichandran, and R. K. Mehra. Proactive Detection of Distributed Denial of Service Attacks using MIB Traffic Variables-A Feasibility Study. In *Proc. Symp. IEEE/IFIP Int Integrated Network Management*, pages 609–622, 2001.
[23] Liang Cai and Hao Chen. TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion. HotSec'11.
[24] Chen Cao, Neng Gao, Peng Liu, and Ji Xiang. Towards Analyzing the Input Validation Vulnerabilities Associated with Android System Services. ACSAC'15.
[25] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing Inter-application Communication in Android. MobiSys '11.
[26] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the Clones: Detecting Cloned Applications on Android Markets. ESORICS'12.
[27] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Trans. Comput. Syst.*, 32(2):5:1–5:29, June 2014.
[28] William Enck, Machigar Ongtang, and Patrick McDaniel. On Lightweight Mobile Phone Application Certification. CCS '09.
[29] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. Scandroid: Automated Security Certification of Android. Technical report, University of Maryland, 2009.
[30] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. MobiSys '12, 2012.
[31] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe Exposure Analysis of Mobile In-app Advertisements. WISEC '12.
[32] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications. DIMVA'12.
[33] Heqing Huang, Sencun Zhu, Kai Chen, and Peng Liu. From System Services Freezing to System Server Shutdown in Android: All You Need Is a Loop in an App. CCS '15.
[34] Byeongcheol Lee, Ben Wiedermann, Martin Hirzel, Robert Grimm, and Kathryn S. McKinley. Jinn: Synthesizing Dynamic Bug Detectors for Foreign Language Interfaces. PLDI '10.
[35] Anthony Lineberry, David Luke Richardson, and Tim Wyatt. These Aren't The Permissions You're Looking For. *DefCon*, 18:2010, 2010.
[36] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. CCS '12.
[37] C. Qian, X. Luo, Y. Shao, and A. T. S. Chan. On Tracking Information Flows through JNI in Android Applications. DSN'14.
[38] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. NDSS '14.
[39] Hanan Samet. *The design and analysis of spatial data structures*, volume 199. Addison-Wesley Reading, MA, 1990.
[40] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Taesoo Kim, and Insik Shin. FLEXDROID: Enforcing In-App Privilege Separation in Android. NDSS'16.
[41] Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, and Shlomi Dolev. Google Android: A State-of-the-art Review of Security Mechanisms. *arXiv preprint arXiv:0912.5101*, 2009.
[42] Yuru Shao, Jason Ott, Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. In *NDSS'16*.
[43] Joseph Siefers, Gang Tan, and Greg Morrisett. Robusta: Taming the Native Beast of the JVM. CCS '10.
[44] Mengtao Sun and Gang Tan. NativeGuard: Protecting android applications from third-party native libraries. WiSec'14.
[45] Gang Tan and Jason Croft. An Empirical Security Study of the Native Code in the JDK. In *Usenix Security Symposium*, pages 365–378, 2008.
[46] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java Bytecode Optimization Framework. CASCON '99.
[47] Zhi Xu, Kun Bai, and Sencun Zhu. TapLogger: Inferring User Inputs on Smartphone Touchscreens Using On-board Motion Sensors. WISEC '12.
[48] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. Fast App Launching for Mobile Devices Using Predictive User Context. MobiSys '12.
[49] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces. CODASPY '12.
[50] Xun Zou, Wangsheng Zhang, Shijian Li, and Gang Pan. Prophet: What App You Wish to Use Next. In *Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication*, UbiComp '13 Adjunct.