From System Services Freezing to System Server Shutdown in Android: All You Need Is a Loop in an App

Heqing Huang[†],Sencun Zhu[†], Kai Chen[‡], Peng Liu[†]
[†]The Pennsylvania State University, University Park, PA, USA
[‡]Institute of Information Engineering, Chinese Academy of Sciences, China
{hhuang, szhu}@cse.psu.edu, {chenkai}@iie.ac.cn, pliu@ist.psu.edu

ABSTRACT

The Android OS not only dominates 78.6% of the worldwide smartphone market in 2014, but importantly has been widely used for mission critical tasks (e.g., medical devices, auto/aircraft navigators, embedded in satellite project). The core of Android, System Server (SS), is a multi-threaded process that contains most of the system services and provides the essential functionalities to support applications (apps). Considering the complicated design of the SS and its easily-accessible system services (e.g., via Android APIs), we conjecture that the SS may face DoS attacks. As the SS plays the important role in Android, serious DoS attacks could cause single-point-of-failure to the phone system. By studying the source code, we discovered a general design trait in the concurrency control mechanism of the SS that could be vulnerable to DoS attacks. To validate our hypothesis, we design a tool to cost efficiently explore high-risk methods in the SS. After a systematic analysis of 2,154 candidate-risky methods, we found four unknown vulnerabilities in critical services (e.g., the ActivityManager and the WindowManager), which are named the Android Stroke Vulnerabilities (*ASVs*). Exploiting the ASVs would continuously block all other requests for system services, followed by killing the SS and soft-rebooting the OS. Results of a further threat analysis show that by writing a loop to invoke Android APIs in an app, an attacker can continually freeze (reboot) the device at targeted critical moments (e.g., when patching vulnerable apps). Furthermore, ASVs can be exploited to enhance malware with anti-removal capability or to design the ransomware by putting the devices into continuous DoS loops. After being informed, Google confirmed our findings promptly. We also proposed to their Android framework team several improvements in their concurrency control design and a fine-grained failure recovery mechanism for the SS.

Categories and Subject Descriptors

D.4.1 [Operating Systems]: Process Management–*Concurrency; Synchronization*; **D.4.6 [Operating Systems]: Security and Protection**–*Access controls; Invasive software*; **C.4 [Performance of Systems]: Reliability, availability, and serviceability**

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

ACM 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813606 .

Keywords

Mobile Security; Denial of Service; Vulnerability Detection; Risk Measurement

1. INTRODUCTION

Android is an open-source OS, based on a modified Linux kernel tailored for mobile devices. Android not only has a significant portion of the current mobile market [8], but has been widely used in mission critical scenarios (e.g., used for medical devices [1, 11, 21], aircraft navigation [14, 15, 48], Android-auto navigation [2] and satellite project [18]). One of the fundamental reasons that Android is so popular among users and application (app) developers, is the unique design of its middleware. This middleware has a set of system services that abstract away the low-level OS details and provide the easily-accessible, rich application programming interfaces (APIs) to support app development for the Android eco-system.

No app can run without the System Server (SS), a special multi-threaded process of the Android middleware, which contains most of the core Java-based system services. The SS threads provide the essential services to Android apps through Binder IPC/RPC. First, the *ActivityManager* manages the life cycle of Android app components, so it maintains the complete running context for each app. Second, the *PackageManager* manages the (un)installation of app packages and parses the apps' manifest files, as well as registering app permissions and other static meta data. Third, maintaining rich application context information, the SS becomes a critical building block to support the design of access control mechanisms for app layer (e.g., Android default permission[1] checks [24], ASM [36], SEAndroid [52] userspace hooks).

When the *ActivityManager* is frozen, no app component can be started. When the *WindowManager* is frozen, no app can correctly present its UI onto the screen. In this sense, the SS can be thought of as the core of Android. Without these functionalities from various system services, it is obvious that the Android system cannot run any app in a way acceptable to the user. When part of the SS is frozen, the smartphone is (partially) broken and becomes unreliable to support any mission critical tasks.

Considering the importance of the SS, one would expect its code to be meticulously developed and thoroughly tested so that Android is robust to failures and resilient to attacks. In this sense, it is expected that an attacker should take a great deal of effort to freeze the SS.

In this study, we examine the attack-resilience of the SS from the angle of denial-of-service (DoS). While researchers have discov-

[1]Permissions that grant access to low-level capabilities (e.g., access camera driver or sdcard) are mapped to Linux group identifiers (GIDs), which is not checked in the SS

ered malware samples [9] and vulnerabilities [20] that cause DoS in the SS, the attack-resilience of the SS has yet to be analyzed systematically. Due to the complicated design of the SS and the easily-accessible nature of its system services, we conjecture that the SS could face various DoS attacks. Considering the important role the SS plays in Android, the DoS problems, if existing, could cause various damages to the system (e.g., single-point-of-failure).

By scrutinizing the design of the SS through source code analysis, we find the following common design traits in the Java-based system services. First, most of these services use basic Java concurrency control mechanisms (i.e., the Java *synchronized block*) to coordinate multiple threads (e.g., binder threads in the SS). Second, most of these services only use a few monitor locks (very coarse grained) in their services to protect many critical sections or synchronized methods. Note that most of these methods or critical sections have different functionalities that may access different global variables. This coarse-grained concurrency control design provides a chance to cause vulnerable scenarios that lead to DoS attacks on the SS. We further characterize the exploiting requirements of these vulnerable scenarios and build a tool to help cost-effectively identify relevant vulnerabilities. With the help of the tool, we identified four highly exploitable and previously unknown vulnerabilities in two critical services of the SS, namely the *ActivityManager* and *WindowManager* services within a week (about 2 hrs/day).

When exploiting these vulnerabilities, the relevant system services are frozen, followed by a soft-reboot of the Android middleware layer (the SS and zygote are reinitialized). This indicates that a single-point-of failure exist. We name the newly discovered vulnerabilities as *Android Stroke Vulnerability (ASV)*. We further design several proof-of-concept (PoC) attacks by exploiting these ASVs. The results show that one could write a simple loop to call normal Android APIs to easily craft several exploits for the ASVs, which causes various damages in Android. For instance, they can turn a phone into continuous freezing/rebooting loops at mission critical moments (e.g., navigating, patient status monitoring), prevent the removal of malware and roll back the patching of app vulnerabilities, etc. As these ASVs are easily exploitable and could cause various consequences, the Google security team has made a quick response and confirmed our findings, and accepted some of the proposed suggestions. Our contributions are summarized as follows:

- *New Understanding and Discovery.* Based on new understanding of the core component, the SS in Android, we discover a general type of design flaw that opens the door for various DoS attacks.
- *Identifying New Vulnerabilities.* We characterize the problem and design a tool, called *ASV-Hunter*, to assist the hunting of ASVs. Our tool cost-effectively processed all the methods in Android framework, which enables us to further analyze 2,154 high-risk candidate methods within various system services. This helps us identify and confirmed four *unknown* ASVs, which are exploited on Google/Samsung devices (with OS version 4.0-5.0.2).
- *Performing Threat Analyses.* We design and implement several PoC attacks on Android by leveraging the discovered ASVs.
- *Defenses.* We not only propose short-term remediation for normal users but also design several improvements for the concurrency control mechanism and the failure recovery scheme in the SS.

2. ANDROID SYSTEM SERVER

2.1 Android Overview

Android is a mobile operating system built on the top of the Linux OS. When it boots up, its *bootloader* first performs low-level system initialization and loads the Linux kernel. The kernel further initializes the drivers and the file system, and then starts the kernel daemons and the first userspace process */init*. The */init* then starts the Android specific *Zygote* process, which is a warm-up template process that preloads all the relevant libraries, classes and resources for all the virtual machine (VM) based processes that start later. After that, *Zygote* starts the System Server (SS) process, which contains most critical system services in Android. Finally, various apps are started to serve users.

An Android app often consists of four types of components, namely, the *activity* (user interface), *service* (background task), the *broadcast receiver* (mailbox for broadcast), and the *content provider* (local database server). The components of different apps communicate with each other via the Android specific *Binder* interprocess communication (IPC) by exchanging *intent* messages. Apps also calls APIs to perform remote procedure calls (RPC) to the system services via the Binder mechanism, the SS process maintains a binder thread pool to handle multiple RPC-based requests simultaneously.[2] Different from apps, the SS does not contain any user interface. Note that only after the SS starts such critical services (Java-based threads) as *activitymanager, windowmanager* and etc.,[3] Android can serve requests from apps. The *activitymanager* manages the running status and life-cycle of all apps. When a user tries to launch an app for the first time, the API stub, *startActivity()* is invoked, which creates an RPC into the method implementation within the *ActivityManagerService* (AMS) class and relevant app records are stored in a ProcessRecord list in AMS for references.

Since Android highly depends on the system services in the SS to fulfill various tasks, **the SS acts as the core of Android**. When an app is going to send a broadcast to relevant receivers, the *activitymanager* queries the *packagemanager* for all the statically registered receivers, so that the broadcasts can get both dynamically and statically registered receivers. Because the SS maintains most running contexts of apps, the access control mechanisms on Android also rely on it. By default, Android enforces part of its permission checking via the SS. It grants permissions to apps through the *packagemanager* and performs permission validations through *activitymanager* and other services [24]. Moreover, a series of enhanced mandatory access control mechanisms (e.g., ASM [36], SEAndroid [52] and XManDroid [26]) also rely on the SS to perform their userspace enforcements. However, the security and reliability of the SS itself has not been scrutinized yet.

2.2 System Server Concurrency Control

System services in the SS are usually started as Java threads, and they help maintain the system-wide states for various aspects (e.g., app life-cyles, window states, and registered app permissions). To guarantee correctness of the program logic (e.g., providing synchronized access to global variables), the SS enforces concurrency control (CC) among threads (e.g., the *binder threads*, service threads and etc.) that handle lots of concurrent IPC requests. Since no detailed study has been done on its CC design, we analyze its source code to understand its mechanism.

Synchronized Code Blocks in the SS. The most frequently used CC mechanism in Android SS is based on the *synchronized* block mechanism from the Java library, namely, the *java.util.concurrent*. $synchronized(\mathbf{monitorlock})\{...access/modify(data)...\}$ is an

[2]Some system services (e.g., MediaService) are designed to join the binder thread pool and handles the RPC directly

[3]After Android 4.0, SurfaceFlinger becomes a native service, which starts right before the Zygote and runs independently from other services in the SS. This design renders the boot animation much faster, as it is responsible for displaying the image surfaces

Table 1: **Statistics of synchronized critical sections (CSs) with watchdog monitored locks in different system services**

System Services Checked by Watchdog (Android Versions Ranges)	Watchdog Monitor Lock	Monitored CS Ratio (v5.1)
ActivityManagerService (v1.5-v5.1)	AMS.this	279/345=81%
WindowManagerService (v1.5-v5.1)	mWindowMap	185/198=94%
PowerManagerService (v1.5-v5.1)	mlock object	49/60=82%
MountService (v4.0-v5.1)	MountService	41/41=100%
InputManagerService (v4.0-v5.1)	mInputFilterLock	4/26=16%
NetwkManagementService (v4.0-v5.1)	mDaemonLock	4/7=57%
PackageManagerService (v4.4-v5.1)	mPackages	171/243=70%
MediaRouterService (v5.0-v5.1)	mlock object	13/13=100%
MediaSessionService (v5.0-v5.1)	mlock object	21/21=100%
MediaProjManagerService (v5.0-v5.1)	mlock object	16/16=100%
Total # of monitored CSs/total # of CSs	The overall ratio	783/970=80%

example of a *synchronized block*. This built-in locking mechanism, namely the **synchronized** block (for critical sections) is heavily used in the SS. *Monitor locks* are acquired by a thread before it enters an explicitly-programmed synchronized block (each such code block conducts certain operations on certain global variables). The Java synchronized block mechanism holds two properties: 1) only one thread can execute the block of code at each time; 2) each thread entering a synchronized block of code sees the effects of all previous modifications guarded by the same lock. This type of synchronization is necessary for mutually exclusive access to global variables when handling multiple threads' concurrent requests. For example, the *getPackageUid* method from PackageManagerService.class (PMS) returns the UID based on a given app's package name and *PMS.removePackageLI* removes an app package. Although these two methods run two different synchronized blocks, they operate on the same *mPackage* global variable and require the same lock, namely the *mPackage* lock. Hence, these blocks are mutually exclusive, though they could be called by different apps/services simultaneously (via binder threads in the SS). In Android 5.0.0, there are 1,917 synchronized code blocks in all these system services (about 20 services in total), which indicates synchronized code blocks are frequently used in the SS.

Lock-Suffixed Method. Another observation on the current Android SS design is that lots of mission critical methods are suffixed with "Locked" and other key strings (e.g., "LP", "LI", "Lw" and etc.). After further analysis, we find that these suffixes are used to remind system developers that the corresponding methods can only be invoked from a thread holding a particular *lock* to guarantee the atomicity of corresponding computations. For instance, the *processStartTimedOutLocked()* method in the *ActivityManagerService.class* is such a method with the "Locked" suffix. We find that this method is only invoked when being wrapped in $synchronized(\mathbf{AMS})\{..processStartTimedOutLocked()..\}$ structure. For any thread to call this method, it must first hold the *monitor lock*, (i.e., *AMS.this* object), which exclusively prevents other threads from executing the critical sections guarded by the *AMS.this* lock. The similar design pattern (i.e., syntax reminder) is used all over other system services.

2.3 System Server Problem Overview

Android developers use the common design patterns (e.g., synchronized locks) in Java to neatly fulfill multi-threaded concurrency control requirements. However, without a more careful design, it might cause unexpected consequences in systems.

Denial-of-Service. The column 3 in Table 1 shows the fraction of critical sections (CSs) guarded by the primary lock in each system service. Because within each service only one primary monitor lock is used to guard a large portion (783/970, 80%) of critical sections, the current CC design is easy to raise a potential Denial-of-Service (DoS) problem. Since a very small portion of critical sections (CSs) are protected by other locks, which are not monitored by the watchdog thread, once one of these monitor locks is held by a thread, a large part of the CSs guarded by the same primary lock cannot be entered. Since the SS provides almost all critical services to the whole Android system, any app can easily request these system services through the Binder IPC (e.g., call the Android APIs). These client-side requests can eventually trigger the executions on some of the CSs (i.e., synchronized blocks) or atomic critical methods in the SS. Since the inputs for these APIs are provided by third party apps, a malicious app may invoke a CS (or a method) with high computational complexity to cause a long holding of the relevant service *lock*, effectively starving the targeted system service. For instance, when the malicious app invokes the *registerReceiver* API stub, the corresponding CS (or method) in the server side implementation (i.e., *AMS.registerReceiver()*) is executed. *During its execution, the AMS.this lock will be held, which prevents other threads of the SS from entering any other CSs in the AMS. During the starvation, the targeted system services become irresponsive, so we call this consequence* **ss_freezing**.

Single-point-of-failure in the SS. After further analysis of these services, we find that the primary locks used to guard critical sections (80%) in these system services are monitored by a separate thread in the SS, called *watchdog*. This thread is used to recover the system from any deadlock, starvation and other failure situations. The first column in Table 1 shows that the number of services monitored by the watchdog increases from three (in Android 1.5) to ten (in Android 5.1). This indicates that more and more complicated system services are added into the main branch of Google Android code-base. Note that other vendors can add more system services into the SS, which makes it more likely to be attacked.

The high level idea of this watchdog design is similar to the traditional watchdog idea in the Linux and Choices OSes [29, 30]. It sets timer-based monitors for various critical system components and kills the corresponding services when its timer expires. Similarly, the Android *watchdog* is initiated as a singleton thread inside the SS process, which is designed to be simple and reliable. Whenever a watchdog-monitored system service starts, it adds a new monitor into the global list for the watchdog thread. The watchdog creates one **HeartbeatHandler** instance (also called *HandlerChecker* in some OS versions) to proactively check the status (**heartbeat**) of different critical system services by requesting the primary monitor locks of system services (e.g., AMS.this for *activitymanager*). $monitor()\{synchronized(AMS.this)\{\}\}$ is the pattern of this monitoring method for performing the heartbeat checks. Once any one of the monitor locks cannot be acquired in a preset period (e.g., 1 minute), the watchdog considers that the corresponding system service is in the starving/deadlock situation. However, we find that, instead of killing the relevant service thread, it kills the whole SS and forces the Android userspace (e.g., the zygote, the SS and other processes) to be rebooted. *This seemingly normal design can cause another consequence: an app can potentially control the watchdog bite on the whole SS process. For instance, the app can invoke some specific RPC into the critical sections to cause the ss_freeze effect for a long period, so as to raise a false alert to the watchdog thread. Since the System Server can be controlled to be shut down, we name this attack consequence* **SS_shutdown**.

We come to a hypothesis that these consequences reveal a general flaw in Android, which is rooted in the fundamental design of the SS, the core of Android. We name this class of vulnerability *Android Stroke Vulnerability* (ASV), when being exploited, the normal service-request flows within the system services are blocked. This eventually causes the shutdown of the SS and a userspace reboot. Similarly, the effect of brain *strokes* block the blood flow.

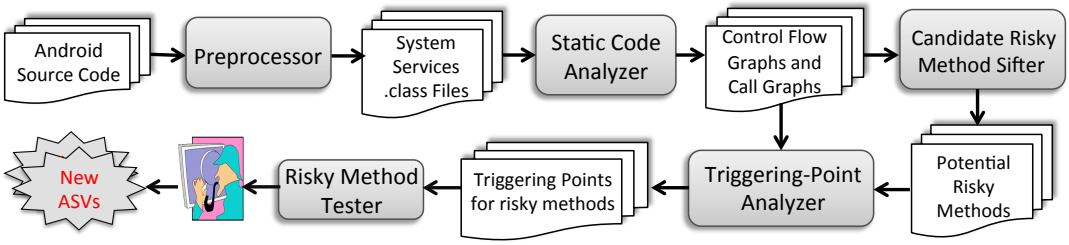

Figure 1: **The ASV-Hunter framework**

3. HUNTING ASVS

To help further validate our hypothesis about the vulnerable *system server* (SS) in Android and explore unknown ASVs in its system services, we design an ASV-Hunter.

3.1 ASV-Hunter Design Logic

Due to the large set of APIs from the platform SDKs and the huge code-base of the Android system services, one can hardly test every method of the SS comprehensively. Therefore, we consider the key for efficient ASV hunting is to distill a set of high-risk methods in various system services. Based on our earlier observation, a general way to exploit the ASVs is to trigger one of the system services to execute a *complicated* method with critical code sections. This causes the monitor lock to be held for a long period. As a result, other threads that need to acquire the same monitor lock will be starved (referred to as *SS_freezing*). Eventually, this long period (30s-60s) of resource starvation can cause a watchdog bite on the whole SS (*SS_shutdown*). We call the methods that can cause the *SS_freezing* or *SS_shutdown* as *risky methods*.

To sift out a set of *candidate-risky methods* in different system services and better assist further analysis, we design the ASV-Hunter, which consists of four components: 1) a lightweight but efficient *static code analyzer* to construct the control-flow graphs (CFGs) and call graphs for each system service in the SS, 2) a *candidate-risky-method sifter* to sift out the potentially risky methods and provide more targeted analysis, 3) a *triggering-point analyzer* to identify a set of triggering methods that may invoke the execution of corresponding candidate-risky methods, and 4) a *candidate-risky-method tester* to test whether the candidate-risky method is actually triggerable. Once a candidate-risky method is tested to be triggerable from the triggering point method, it will be reported to the security analysts for further analysis and stress testing.

3.1.1 Analyze the Android System Server Bytecode

To perform a detailed analysis on the System Server (SS) of Android, we extract the relevant Jar files from the compiled Android AOSP source code of v5.0.0, the Lollipop. We then preprocess all the compiled Java bytecode (*.class* files) of the system services implementation. During the preprocessing, we find that the SS contains 1,570 class files, 10,812 methods, 993,782 lines of bytecode instructions, and 70,204 method invocations. To fully analyze all the critical system services in the SS, we design an efficient *static code analyzer*. It first takes the decompiled Java bytecode files of the system services as inputs and uses the Soot [17] analysis framework to translate the Java bytecode instructions into an intermediary representation (IR), namely the *Jimple IR*. With the support of Soot, we construct the method-level CFGs efficiently for each *.class* file, which takes less than an hour on a Ubuntu machine with 4G RAM and an Intel Core i5 processor. Second, we generate the call graphs based on an existing tool [24] for the whole Android framework, which takes about 4 hours. Building the method level CFGs and the whole call graphs is a one-time effort. Third, we stitch the CFGs with the edges in the constructed call graphs (i.e., adding invocation edges between callers and callees in the corresponding CFG nodes) for all services. As a result, we have a comprehensive approximation of how these system services interact with each other and all the potential control flows within each service. This helps us further measure the complexity of different candidate-risky methods in the corresponding services and assist our *triggering-points* analysis. The generated graphs also help us easily identify some attack scenarios (described in Section 4), which cause various damages in Android.

3.1.2 Sift Out Methods for Analysis

To cost-effectively identify and analyze the candidate-risky methods in the SS that lead to the ASVs, we design a *candidate-risky-method sifter* to identify a set of highly-exploitable methods in various system services. The sifting criteria for each candidate-risky method are as follows: 1) if a method contains more complicated control flow structures (e.g., loop structures), it is more likely to cause starvation; 2) if a method contains synchronized blocks that contain more bytecode instructions, it is more risky to cause ASV; 3) if a method calls more other methods (with more *invoke-direct/.../static* instructions) or 4) appears in the call-sites of other methods more frequently, it has a higher chance to cause problems; 5) if it is a lock-suffixed method, it is considered risky, as this method can only be called in critical code blocks (i.e., one of its caller methods in the call stack should contain a synchronized block, which is wrapped with *monitorenter/monitorexit* opcode pair); 6) if the method is called while holding a lock that is monitored by the watchdog thread, it is more risky as it can potentially cause the *SS_shutdown* consequence. All the criteria are encoded as a *RiskMethodVector*. The first four numerical fields in the vector measure the complexity of each method, which is relevant to the *SS_freezing* consequence. Also, the last two fields are boolean values, which are relevant to the *SS_freezing* and the *SS_shutdown*.

Based on these criteria, we design Algorithm 1 (in Appendix A) to collect relevant information for all methods in the system services. First, our *static code analyzer* returns the stitched CFGs; then, we set the system-service method under analysis as the entry node for a depth first search (DFS). We design an approach to efficiently compute the number of loops for each method. During this DFS process, we also count the number of synchronized code blocks and collect the number of *invoke* instructions and other relevant information, so that all the fields in the *RiskMethodVector* are filled. This whole process results in a vector database for all the methods in the SS, which enables us to sift out candidate risky methods gradually. Specifically, in the first batch, we issue strict queries to the database. The last strict query that we used is $[(\delta > 1) \wedge (\gamma > 40) \wedge (\sigma > 3) \wedge (\epsilon > 3) \wedge true \wedge true]$, which tries to match and return all the candidate-risky methods that each contains at least 2 loops, has more than 40 instructions in critical sections, contains at least 4 invoke-opcodes, appears at least 4 times in the callsites, is a locked suffixed method, and is guarded by a monitor lock checked by the watchdog thread. The first batch of queries return totally 171 methods from system ser-

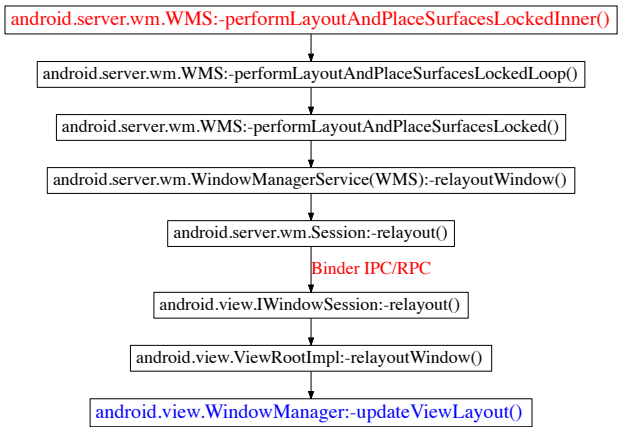

Figure 2: One example triggering point for ASV #3

vices, which yield *two* unknown ASVs after further testing. In the second batch, we issue more relaxed queries by reducing the thresholds for the first four fields and using 5 ORs relation among these 6 querying criteria. For instance, the last query in the second batch is $[(\delta > 0) \vee (\gamma > 10) \vee (\sigma > 0) \vee (\epsilon > 0) \vee true \vee true]$. These relaxed queries return about 1,983 risky methods gradually, which lead to the identification of *two* new ASVs later on. We then stopped our querying operations, because the returned methods are either too simple to cause any sufficiently long *SS_freezing* consequence or not capable of causing the *SS_shutdown* consequence. However, we do not claim that the criteria and heuristics used here are perfect. Note that the purpose of the sifter is to gradually reveal candidate methods with higher risk to prioritize the cost of further analysis effort (e.g., dynamic testing). This risk measurement idea is similar to the iterative approach proposed for high-risk attack-surface distillation [39] and efficient vulnerability-impact ranking [58, 40]. Our current configurations capture most of the traits in the candidate risky methods that are most likely causing the ASV. Detailed results are shown in Section 3.2.

3.1.3 Triggering-Point Analyzer

After sifting out a set of candidate-risky methods in the relevant system services, we want to identify their corresponding triggering points to better assist further dynamic testing. The triggering points are those methods that can trigger the execution of corresponding candidate-risky methods from a third party app. Therefore, we design a *triggering-point analyzer*, which uses *backward reachability analysis* on the constructed call graphs. As we want to find candidate-risky methods that can be directly triggered from any app, when traversing the call graphs, our analyzer prunes the traces with inter-service calls. That is, a traversal will terminate when it identifies a trace contains two methods from different system services. Note that inter-service calls might be feasible paths too, but they require more effort to trigger during runtime testing. This, on the other hand, also means it is even harder to be exploited. This pruning strategy reduces low-risk-cyclic traces among services.

Figure 2 contains one sample trace corresponding to the identified triggering point (in blue) of the risky method (in red). Note that due to the fundamental limitations of static analysis and our conservative design to find more traces statically, some of the identified triggering points might not lead to risky method execution. Also, since we want to obtain the direct accesses to most of the triggering points through Android APIs, we further match the returned triggering points against the specified third-party APIs in *frameworks/base/api/current.txt* from Android source code. Finally, our *triggering-point analyzer* is able to statically identify triggering points for 1,677 (out of 2,154 (171 plus 1,983)) candidate-risky methods.

Table 2: 4 Android devices used in the dynamic testing

Device Name	OS/Kernel v	CPU core/speed	RAM
Nexus 7 Tablet	5.0.2L/3.4.0	Quad/1.3kMHz	1G
Nexus 4 Phone	5.0.0L/3.4.0	Quad/1.5kMHz	2G
Samsung S4	4.3JB/3.4.0	Quad/1.6kMHz	2G
Samsung Ns	4.0.4/3.0.36	Single/1kMHz	0.5G

3.1.4 Candidate-Risky Method Testing

We perform dynamic testing to further evaluate the 1,677 returned candidate-risky methods with triggering points. To confirm that the candidate-risky methods can indeed be triggered through Binder IPC/RPC calls from an app, we instrument the Android Binder IPC mechanism. Most of the remote procedure calls from an app to the SS in Android are handled by the Binder IPC/RPC mechanism. When a process invokes an RPC, our instrumentation call will generate a specific *calling ID*, which will be included in the Binder transaction data (i.e., *parcel* object). The whole parcel will be sent to the remote binder thread that actually executes the candidate-risky method. All the operations performed by the binder thread are logged with the *calling ID*, so that our further analysis can easily map the invocation of risky methods with the corresponding calling process based on the testing logs. This helps automatically confirm the candidate risky methods are indeed triggerable.

With this binder instrumentation scheme, we then generate test cases to analyze the candidate risky methods in two phases. First, we run a configured test case to check if the corresponding risky method can be triggered by analyzing the testing logs. Our testing case is based on a normal Android app, so that the test units are directly reusable to construct proof-of-concept (PoC) attacks. The testing app template contains one normal main activity component (needed for most Android apps). The main activity sends intents to initiate different tasks at a number of IntentService components (for the execution of long running tasks) as test units. Within each IntentService, we test the reported risky methods by invoking the triggering-point to trigger the corresponding risky method. We write a parser with the *antlr* framework [6] to customize the IntentService components, so that security analysts only have to load the configured test-cases into Android Studio IDE (AS-IDE) and specify the relevant parameters for the developer-APIs. Second, if a method is triggered, we then perform a stress test by further configuring different testing parameters (e.g., register different numbers of receivers and configure different parameters values in the APIs) to trigger various conditions.

Using the AS-IDE, we were able to configure and fully test all the 1,677 candidate risky methods by leveraging its type inference within a month (2 hrs/day). Note that our candidate-risky-method sifter is very effective, since all the four ASVs are identified within the first week of our analysis. Also, the first two ASVs are confirmed within the first batch returned by our stricter queries and the other two ASVs are also returned very early when performing the second batch of queries, which were also tested earlier. Also, the second batch return about 10 times more methods than the first batch, but yields the same amount of ASVs, which demonstrates that our sifter indeed returns first the methods with high chance of causing ASVs. Hence, we envision our effective sifter can be extended into a generic framework to support analysis of similar performance issue of other Android system components (e.g., the native daemons), the Apple iOS middleware and etc.

Table 3: Discovered new ASVs in the ActivityManagerService and WindowManagerService

ASV	Risky Method in the System Service	Triggering Points
1	ActivityManagerService(AMS).broadcastIntentLock()	ActivityManager.send(Ordered)Broadcast(AsUser)()/...
2	ActivityManagerService(AMS).cleanUpApplicationRecordLocked()	android.os.Binder.sendDeathNotice()/...
3	WindowManagerService(WMS).performLayoutAndPlaceSurfacesLockedInner()	WindowManager.addView()/updateViewLayout()
4	WindowManagerService(WMS).removeWindowInnerLocked()	android.os.Binder.sendDeathNotice()/...

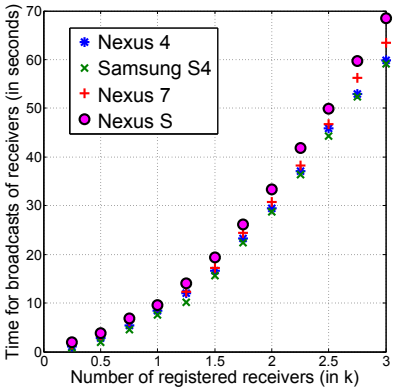

Figure 3: The freezing-time trend of ASV #1 when different # of receivers are registered

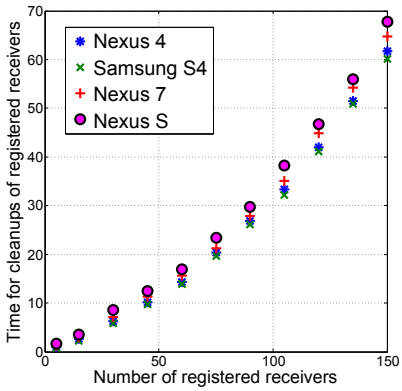

Figure 4: The freezing-time trend of ASV #2 when cleaning up different # of receivers

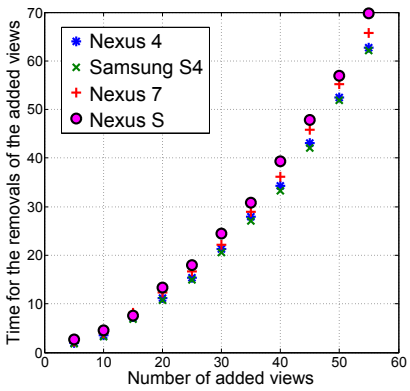

Figure 5: The freezing-time trend of ASV #4 when removing different # of views

3.2 ASV Hunting Result

Our *ASV-Hunter* helps us identify four new ASVs in two batches of queries (listed in Table 3), which are confirmed exploitable by any third party app. All of the ASVs have been further tested and confirmed in four different Android devices (listed in Table 2) with Android version (4.0.4–5.0.2). Given the giant code-base of system services, the result indicates that our candidate-risky-method sifter can help gradually return the most risky methods that lead to ASVs and our two-phase dynamic testing indeed helps security analysts reduce the analysis effort.

From all the tested system services, *activitymanager* and *windowmanager* are confirmed vulnerable and each contains two ASVs. Table 1 shows that these two system services are monitored by the watchdog thread in all Android versions (v1.5-v5.1), indicating that they are actually very essential to Android. Later, we can easily build several PoC attacks by exploiting them to cause various threats to Android users.

Next, we discuss the discovered ASVs by focusing on their exploitation requirements and their direct consequences to the system when exploited. Section 4 provides a more detailed threat analysis on these ASVs with several proof-of-concept (PoC) attacks.

3.2.1 Vulnerable Message Broadcasting in AMS

One of the risky methods that are discovered by the ASV-Hunter is the *broadcastIntentLock()* in the *activitymanager* service (AMS), which is executed to resolve receivers and deliver the actual broadcast message to them. In Android, any app/service is able to broadcast a message to a set of specific components of apps/services that have registered the matched action strings for the message. For example, if a battery-status-monitoring app needs to receive messages from the *powermanager* service, which indicate the changes of battery status, it needs to register a broadcast receiver with action.*battery_status_low*. Both receiver registration and message broadcasting are handled by methods in the AMS class. When any app/service registers a receiver by calling the *registerReceiver()* API stub, a Binder RPC call happens and the corresponding method *registerReceiver()* in the AMS is invoked. Within the *registerReceiver()*, the corresponding *receiver* fields (e.g., action, type, scheme and permission) are parsed into a *broadcastFilter* object, which are further added into a global object *mRegisteredReceivers*. This global object includes all the dynamically registered receivers[4].

The *broadcastIntentLocked* method is a lock-suffixed method in the *activitymanager*, which indicates that it can only be called when the *AMS.this* lock is held. The *AMS.this* lock is checked by the watchdog, so the *SS_shutdown* consequence can happen when the lock is held for a duration longer than the preset timeout of this lock in the watchdog (usually set as 1 minute). Later, we observe that this method is indeed very complicated with several nested loops. When a broadcast request is sent to the SS, a binder thread will call *queryIntent()* to resolve a set of relevant receivers by matching the broadcast fields against the fields of each registered receiver in the global object *mRegisteredReceivers*. This process is actually very time-consuming, as each broadcast usually contains several fields and matching each field to resolve the receivers is a complicated task. Hence, it is actually returned by our *risky method sifter* with strict queries. Our *triggering-point analyzer* also reports a set of relevant triggering points to trigger this method, including *sendBroadcast, sendBroadcastAsUser*, *sendOrderedBroadcast* and other similar APIs stubs, which are later confirmed triggerable during our testing.

Exploitation requirements and consequences. To exploit the ASV #1, our test units register many receivers into the global intent resolving object, namely the *mRegisteredReceivers* in the AMS. By triggering the risky method *broadcastIntentLocked()* from one of the triggering points (e.g., *sendBroadcast()*), it can cause the *ss_freezing* consequence to the whole *activitymanager* service.

Figure 3 shows the freezing time as a function of the number of registered receivers on four testing devices. When invoking the

[4]Apps/services can register static receivers in the AndroidManifest.xml file, which is actually maintained by the *packagemanager* by parsing all the manifests and bookkeeping those receivers when the system boots up. Note that all these static ones have lower priority than the same receiver types that are registered dynamically.

broadcastIntentLocked() to exploit ASV #1 with 1,500 identical receivers registered, it can cause the *ss_freezing* effect for about 20 seconds, during which most of the critical sections in *activitymanager* cannot be accessed to other threads. Furthermore, as most of Android services and apps rely on the *activitymanager*, there are potentially many other consequences. One scenario is that the launcher app in the system will not be able to respond, so the user cannot interact with the screen and the system becomes totally unresponsive for the user. Also, since the *activitymanager* manages the life cycle of apps, it cannot help any app perform any task (e.g., start(stop)Activity, start(stop)Service and etc.).

When triggering the *broadcastIntentLocked()* to exploit ASV #1 with 3,000 receivers registered in the *activitymanager*, it can actually cause *ss_freezing* for more than 60 seconds. By analyzing the recorded logs in our *ASV-Hunter*, we find that it indeed causes the *SS_shutdown* consequence every time when the lock has been held for more than 60s. This is because the watchdog keeps monitoring the *AMS.this* lock, and it kills the SS process to force the system to take a soft reboot when it cannot obtain the lock within a minute. Note that during the experiment, we disable the watchdog bite capability, so that all the highest data points in Figures 3, 4 and 5 can reach 70 seconds without the impact of system reboot.

3.2.2 Vulnerable Application Record Cleanup in AMS

The *cleanUpApplicationRecordLocked()* method is another risky method in the AMS confirmed by our ASV-Hunter as vulnerable. After further checking, we find that it is lock-suffixed and contains several nest loops. Since the AMS manages the life-cycle of all Android apps/components, it has to remove the relevant fields from its global object (e.g., *mRegisteredReceivers*) when certain app components died. The *cleanUpApplicationRecordLocked* method will invoke the *AMS.removeReceiverLocked()* method multiple times when an app is killed. Within this method, the *removeFilter* method of the IntentFilter object is further called to remove all the previously registered receivers. Therefore, besides matching the intent fields, it has to match the relevant intent fields as well as removing the corresponding registered receivers from the global object, thus making it even more complicated than the previous risky method, *broadcastIntentLocked()*.

Exploitation requirements and consequences. To exploit the ASV #2, our *risky-method tester* confirms that one only has to write a simple loop to dynamically register 75 receivers with tens of fields in each receiver into the *mReceiverResolver* object. This can cause the *ss_freezing* consequence for about 20 seconds when triggered. Since the triggering-point of this method is the *sendDeathNotice()*, when designing our testing app, we find that it can be triggered indirectly by calling the *System.exit()* to cause removal of all the previously registered receivers. Figure 4 shows different freezing time periods caused by different numbers of registered receivers on different Android devices. We can observe that the trend is superliner, which means as the number of registered receivers increases, the freezing time will increase dramatically. To cause the *SS_shutdown* consequence, one only needs to register 75 more receivers to freeze the *activitymanager* service for an extra 40 seconds. The potential direct consequences of this ASV are quite similar to that of the ASV #1. Our further investigation find that this ASV can be naturally leveraged to design anti-removal techniques for malware.

3.2.3 Vulnerable Screen Layout Manipulation in WMS

Our *risky method sifter* returns the *performLayoutAndPlaceSurfacesLockedInner()* method in the WMS with a strict querying criteria. Later, we find that it is a complicated method and it needs the *mWindowMap* object lock before execution. The *mWindowMap* object is one of the critical resources in the WMS, which contains the *WindowState*s of all apps, so it is monitored by the watchdog thread. Almost all the critical tasks of the WMS that manipulate the screen layout need to hold this lock.

One of the direct triggering points leading to this risky method is the *windowmanager.addView()* API stub. We only have to configure the required permission for calling this API and then test the exploitability of this risky method through our *risky-method tester*.

Exploitation requirements and consequences. We find that calling this method repetitively can cause continuous freezing of the system UI. After further analyzing the internal design of this risky method, we find that it is a critical method for the Android system that keeps reorganizing the screen-layout components when changes happen. It helps calculate the layout components on the screen and fresh the relevant view pixels by interacting with the SurfaceFlinger service, which feeds the actual pixels to the memory (i.e., the framebuffer). Therefore, by adding views onto the screen or keeping modifying the screen-layout components, the *performLayoutAndPlaceSurfacesLockedInner()* method is forced to keep recalculating the components and refreshing the screen with the SurfaceFlinger. This can prevent normal user's interaction with the system and cause *ss_freezing* repeatedly (around 1 second each time). Since the *mWindowMap* resource is not held continuously for 60 seconds, a watchdog bite cannot happen in this case. However, when our testing app adds around 30 tiny (almost invisible) views on the screen, the Android userspace will be rebooted (i.e., zygote and the SS killed) due to a failure in the *HWComposer*[5] of the SurfaceFlinger. This is another form of *SS_shutdown* consequence, which shows that the heuristics used in our *hunter* can help identify risky methods with problematic design in the system.

3.2.4 Vulnerable WindowState Removal in WMS

Another vulnerable method in the WMS returned by our hunter is the *removeWindowInnerLocked()* method. When triggered, this method recursively calls itself and each recursive invocation will further call a couple of complicated code blocks to refresh the screen. Our *triggering-point analyzer* finds that it can be triggered when the corresponding binder object in an app is removed, which means we can use the *System.exit()* on the testing app itself to trigger this RPC in the *windowmanager*[6].

Exploitation requirements and consequences. To exploit the ASV #4, our test app first adds about 50 tiny views (human invisible) onto the screen.[7] Later, at any critical moment, the test app can choose to trigger the recursive call on *removeWindowInnerLocked()* to remove all the previously added views. This operation holds the lock *mWindowMap* of the windowmanager continuously for a long enough period and eventually causes a watchdog bite on the SS. The corresponding time trend is shown in Figure 5. Therefore, this exploitation can cause both the *SS_freezing* and *SS_shutdown* consequences. Furthermore, no further user interactions can be accepted by the system, since the window layout rendering mechanism is paralyzed.

[5] HWComposer does several different things: sync framework (vsync callback), mode-setting, compositing layers together using features of the display controller, displaying frames on the screen.

[6] The *windowmanager* uses the death recipients mechanism of Android. When the app is killed while its windows are still showing, the *windowmanager* will receive a death notification callback to clean up everything

[7] In this test, we temporarily fixed the HWComposer vulnerability by disabling it in the SurfaceFlinger and rebuilt the system

3.2.5 Human Intelligence

Currently, our ASV-Hunting process still needs human intelligence to understand the semantics of the tested API in order to eventually confirm the actual exploitability of ASVs via further stress testing. For instance, we discover that by calling *System.exit()* in an app/component, one can trigger the ASV #2 indirectly. Also, the ASV #4 can only be exploited when the PixelFormat is not set as *TRANSPARENT*. Furthermore, the ASV #2 can register less receivers when all the fields (e.g., action, data and etc.) in the receivers are given *non-null* values. All these heuristics that helped us confirm the final exploitability of ASVs are not very straightforward. Therefore, how to further guide the API testing for vulnerability analysis is an interesting but challenging task for our future work. We note that recently, similar observations have appeared in the UI vulnerability analysis [25].

4. PROOF-OF-CONCEPT ATTACKS

In this section, we perform threat analyses by engineering several proof-of-concept attacks (PoCs) exploiting the discovered ASVs under various application contexts. With a thorough understanding on the consequences of the ASVs and their easily-exploitable nature, we design the PoCs to cause various user impacts. The recorded videos of some PoCs can be found in the following link: **https://sites.google.com/site/heartstrokevulnerability/**.

4.1 Attack Design Overview

In our current design, we implement the PoCs in a third party app, which can be installed on the victim device through repackaging popular apps [27, 57, 41] or other social engineering approaches. This is a common requirement for Android-based attacks [38, 51]. Remote exploitations through other code injection channels (e.g., vulnerable HTML-5-based Android apps [42]) are also applicable, due to the easily-exploitable nature of the ASVs. Exploiting ASV #1 and #2 requires no permission, and exploiting ASV #3 and #4 requires only one normal permission to add window. The design of some specific attacks might need one/two common permissions (e.g., the *internet* or *boot_complete* permission).

4.2 Attack Design and Implementation

4.2.1 Hindering Critical Application Patching

Attack Scenario. Due to the evolving nature of mobile systems, mobile apps (e.g., antivirus, banking and other apps) have to patch vulnerabilities [7, 3, 16, 19]) very frequently. We show that by exploiting some of the ASVs, the app-update task for the vulnerability patching can be manipulated (either being delayed or completely failed). Suppose that two banking apps need to be updated (patched) in an Android device: one is the Citi bank client side app (not vulnerable) and the other is the Bank of America app (BoA-app). Also assuming that only the BoA-app has a serious vulnerability, which can be leveraged to steal users' account credentials. Also, the adversary only wants to cause the patching failure in the targeted BoA app, so that the vulnerability in the BoA app can be continuously leveraged without effecting the patching process of other apps. Next we show how this can be achieved.

Design and Implementation. The app-update task performed by the *packagemanager* service (PMS) has three sequential subtasks, including removing the old package, adding a new app package and configuring the new app. It needs to grab the *AMS.this* lock three times to send three broadcast intents sequentially to other components (e.g., *installd*) with action strings *package_removed*, *package_added* and *package_updated*, respectively. In this way, other system components can collaborate in this atomic process to help in various sub-tasks. Through triggering the ASV#1 or ASV#2 of the AMS, one can block the following sub-tasks (adding package or configuring package). We test different timings to trigger the exploit and find the best moment is right after the removal of the old package. Therefore, we implement our PoC-App by registering one receiver with the *package_removed* action.

Note that this attack is highly targeted, as it only performs the exploitation for the system that has installed the targeted vulnerable app (here the BoA-app), but not the other apps. This can be done by obtaining a list of vulnerable apps' version codes and package names through offline reverse-engineering. Our app will only plan the ASV (i.e., register enough receivers according to the results in Figures 3 and 4) when the installed package name (*boa.app*) and version number are matched with that of the targeting app. This can be verified by calling *packagemanager.getInstalledPackages()*. Also, the planned ASV will only be triggered by receiving the broadcast with the *package_removed* action and the updated package is *boa.app*.

Result. When the *AMS.this* monitor lock is being held *over one* minute, the system reboots. The PMS of Android has a failure-recovery mechanism, which rolls back the unfinished app-update task to ensure its atomicity. This mechanism provides a consistent state for the whole system; however, it unexpectedly leaves the vulnerable app unpatched. Therefore, the malware can keep leveraging the known loophole in the app (e.g., keep on stealing users' banking credits). By exploiting the similar scenario for *less than* a minute (the watchdog bite will not be triggered), attackers can control the exploit to only delay the update of critical apps. Based on this observation, we implemented a PoC that extend the *null-protection window* length described in the **engine-update attack** against the mobile antivirus apps discovered in a previous work [38]. Both PoCs are tested for 10 times on four Android devices listed in Table 2 with 100% success rate with no Android permission needed.

4.2.2 Anti-Removal Techniques

Attack Scenario. In this scenario, any malicious app can be potentially equipped with an anti-removal technique by naturally exploiting the ASV#2. By checking the control flow of the AMS, we find that an app removal operation includes two tasks. First, the system needs to kill the relevant active processes of the target app and deregister its running status through *activitymanager*. Second, the package-removal task is then executed.

Design and Implementation. In the first task, to clean up the dynamic app record, the system has to invoke the risky method *AMS.cleanUpApplicationRecordLocked()* of ASV#2. Now, if the malicious app manages to exploit the risky method of ASV#2 (registered certain amount of receivers), its package can never be removed. Because the whole removal process is guaranteed to be blocked at the first task, it cannot proceed to the second package-removal task. We design a PoC app for this case. Our app registers 150 receivers, which blocks the removal operation at the first task for over a minute and eventually cause a watchdog bite on the SS, followed by a soft reboot. When the system reboots, the PoC app listens on the system boot-up broadcast message and exploits ASV#2 again.

Result. When testing the anti-removal technique, we use a third-party removal app as well as the system *setting* (removal) app to perform the removal operation for five times on four various devices. The test result is quite consistent. All the 40 trials are blocked for one minute and followed by a soft reboot, and our PoC app package remains untouched after the soft reboots. Note that Google remote uninstall mechanism will not help on this either, as it still needs to kill the running PoC process(es) on the device lo-

cally before actually removing the malicious package. Therefore, an average person will have a hard time dealing with malware that is equipped with this technique.

4.2.3 Repeated DoS Attacks

Attack Scenario. Many ransomware families [4, 10, 13, 5, 12] are now attacking the mobile devices. We find that the discovered ASVs can serve as ideal building blocks to construct ransomware. One straightforward way is to exploit these ASVs to cause continuous DoS effects on the Android devices and then ransomware writers can request for a ransom to remove the consequences.

Design and Implementation. We design two PoCs. One PoC is to cause soft reboots continuously. When the system performs a soft reboot or a complete restart, Android broadcasts a message with action.*boot_complete*. Our PoC app registers the *boot_complete* receivers to get notified and then plans the next exploitation. The ransomware writers can add an activity view on top of other views right before triggering the exploits to present information for requesting a ransom. After receiving the ransom, he/she can remotely command the ransomware to stop triggering these ASVs. Basically, before triggering the next exploit, the ransomware will try to connect to a remote server for a command to release the attacks. The other PoC will continuously freeze the device using the discovered ASVs (each freezing period is less than 50 seconds). Both PoCs will not allow the user to interact with the screen.

Result. The result shows that the designed PoC apps are quite effective at causing *ss_freezing* with a ransom view presented on the screen. Also, when combined with the *anti-removal technique*, it becomes more difficult to clean up.

4.2.4 Remote Exploits via HTML-5 Code Injection.

Remote-code-injection attack [42] on HTML-5-based Android apps is a well studied problem. Since most of the plugins (e.g., Apache Cordova plugins) for these apps can retrieve the *context* object, the injected malicious code can invoke the Android APIs directly. For example, one can call *sendBroadcast(intent)* through *((CordovaActivity)this.cordova.getActivity()).sendBroadcast(intent)*. Thus, one can inject JavaScript code to construct the ASV exploits remotely. We use an Apache Cordova plugin and write relevant payload (20 lines) in JavaScript to exploit ASVs.

4.2.5 Attack at Mission Critical Moments

Android OS is widely used in various mission critical scenarios, for instance, serving medical devices [1, 11, 21], aircraft navigation [14, 15], Android-auto navigation [2] and embedded in nano-satellites [18]. To be more stealthy, the ASVs may be exploited only at some critical moments (e.g., aircraft navigation [48], monitoring patient and etc.) when the above apps are running. Here critical moments can be inferred from the running status of the mobile devices/apps based on various side channels discovered in previous works [38, 48].

5. POSSIBLE DEFENSE MECHANISMS

We reported all the identified ASVs to Google's security team, who confirmed our findings and acknowledged our contributions. CVE IDs for the ASVs will be generated after fully patching. Based on the understanding of the root cause of this general design flaw, we are now proposing mitigation to end users and defenses to Android framework developers.

5.1 Apply User Side Remediations

If a vigilant user identifies the suspicious app, he/she can learn to boot the system into the "safe mode" to remove it. However, a user can hardly identify the remote exploits launched via HTML5-based vulnerability in trusted apps. A more knowledgeable and determined user can perform a factory reset of the phone through fast-boot. This, however, requires that most of the important data have already been backed up somewhere else (e.g., in the cloud). Note that an app can still repackage/disguise itself as *benign* and only exploit the system at a few mission critical moments to cause deadly damages. Thus, we need system-level defense solutions.

5.2 Use Access Control Mechanisms

In the system level defense, one may easily think of leveraging Android permissions and access control mechanisms. However, they may only help partially mitigate the problem. First, not all the critical resources or APIs are currently protected by permissions. For example, two of the discovered ASVs can be exploited without any permission. Second, users must have a good understanding of the added Android permissions. This, however, has been shown to be very ineffective according to a study [34] (only 3% of the surveyed users have a good comprehension of permissions). Third, the general access control mechanisms (e.g., ASM [36], SEAndroid [52] and etc.) are insufficient either to prevent the DoS attacks because they are mostly stateless. Once the permissions/capabilities are granted to a vulnerable trusted/malicious app, it is difficult to prevent them from being misused.

5.3 Define Resource Usage Thresholds

One stateful approach is to use *thresholds* to restrict the number of resources that each app (component) can register. For instance, one may limit the number of receivers an app can register or control the number of view add/update operations allowed for a normal app. However, the appropriate thresholds are not easy to define for market-level malware detection. The static analysis can hardly be precise to determine the number of resource registrations. Dynamic testing may not trigger all the execution scenarios well enough. Hence, inappropriate thresholds can cause lots of false alarms on legitimate apps that need to use the resources frequently. Also, the market-level malware detection cannot prevent remote exploits that are loaded dynamically.

5.4 Retrofit the Concurrency Control Design

One crucial observation from our research is that every system service (in Table 1) uses only one primary monitor lock for most of the critical sections that manipulate different variables in different services (e.g., *AMS.this* lock in the *activitymanager*). Thus, one approach to retrofit the SS design is to provide a fine-grained concurrency control design in system services. The high level design principle is similar to that on the Big Kernel Lock refinement [45].

Based on our observation, the requests for *startActivity()*, *startService()* and *sendBroadcast()* should not compete for the same *AMS.this* lock; otherwise, it can easily cause the freezing of the whole *activitymanager*. Hence, instead of using the *AMS.this* lock, we use the *AMS.mRegisteredReceivers* object (which contains all the dynamically registered receivers) as a fine-grained lock. Accordingly, only the threads that dynamically resolve or manipulate the receivers have to compete for the *AMS.mRegisteredReceivers* object lock before entering relevant critical sections. This prevents the blocking of other types of requests in the *activitymanager* (e.g., *startActivity()*). Our experiments show that other tasks now become responsive even when the ASVs #1 and #2 are exploited. For instance, activities and services can still be started, so the user is able to handle the ransomware cases through the launcher app and kill the suspicious apps.

However, lock contention for *AMS.mRegisteredReceivers* still exists. For instance, multiple broadcast threads (readers) still need to wait for the first broadcast thread (reader) to finish its work. We observe that currently there is no such optimization based on *readers-writer lock* (RWL) [46] (e.g., the Java *ReentrantReadWriteLock*) in system services to allow concurrent access among readers (broadcast threads). We further propose a fine-grained design to leverage the *read-copy update* [31] (RCU) mechanism whenever possible. The RCU is an improvement over the RWL, which ensures the coherence among readers by copying a new version of objects for the single writer. Comparing to RWL, RCU further ensures that the updating of the *AMS.mRegisteredReceivers* object can no longer block the message broadcasting jobs. Hence, it can be used here to further boost the throughput and prevent some relevant attack scenarios (e.g., the anti-patching attack). This, however, cannot cope with the scenarios when several writers (e.g., the watchdog thread[8]) are starved due to a malicious writer thread (e.g., a binder thread that executes *cleanUpApplicationRecordLocked*). This means some ASVs can still cause some damages in the system, e.g., the anti-removal attack.

5.5 Design a Smart Watchdog Mechanism

To avoid the writer-writer starvation and other failure situations (e.g., the deadlocks), one on-device protection is to set thresholds to monitor and detect a burst of resource usages (e.g., view manipulations). Therefore, a lightweight but *smarter watchdog* is designed help recover the system from failures. One problem with the current *watchdog* design is that it simply kills the whole SS process whenever it fails to grab the lock after about 60s. To better resolve this problem, the *watchdog thread* needs to quickly identify the problematic thread(s) and gracefully resolve the failure situations by only removing those problematic threads.

We observe that when a malicious app exploits the ASV, one *binder thread* in the SS is assigned to fulfill the actual RPC by accessing the critical sections and blocking all the other SS writer threads for the same lock. In our design, once the watchdog thread finds that the monitor lock(s) is not available (e.g., in 20 seconds), it will quickly diagnose the problem by calling *Thread.holdsLock()* and *Thread.getAllStackTraces()* to identify whether it is the binder thread(s)[9] that holds the monitor lock. Also, instead of killing the whole SS, it only helps exit the problematic threads. Since binder threads loop on the commands delivered from the kernel *binder driver* to execute various RPC calls, one may want to send the BR_FINISHED command[10] to deliver a TIMED_OUT to the problematic binder thread. However, we find that the busy (problematic) binder thread cannot take the command. Thus, we choose to let the watchdog thread use a JNI call to invoke the *tgkill()* syscall to send inter-thread signals (i.e., SIGTERM) specifically to the problematic binder thread(s). During our test we find that, the binder thread has to add a signal handler using the sigaction() syscall to handle the SIGTERM signal and gracefully exit itself to resolve the starvation (or deadlocks). Otherwise, the whole SS will still be killed. The other benefit of exiting the problematic binder thread(s) is that by default, the kernel binder driver will spawn new binder threads (via the BR_SPAWN_LOOPER command) and add them into the thread pool whenever the number of binder threads is not enough.

[8]With the RCU setup, we need to modify the watchdog to always check the writers lock and thus become a writer thread

[9]The binder thread is created as a native pthread, which is attached to the Android runtime.

[10]One has to enable the unused BR_FINISHED command in the communication protocol of the binder mechanism.

We have already proposed these defense solutions to the Google Android framework team. They will probably refine the problematic critical sections in the AMS to patch ASV #1, #2 and fix #3 as suggested. They are still evaluating the corresponding pros and cons of other defenses/mitigation based on various design objectives.

6. RELATED WORK

Android attacks and defenses have been proposed in [28, 35, 34, 26, 52, 59, 43]. Chin et al. [28] describe an attack only against the ordered broadcast API of Android (by the default design, Android allows the receiver of the ordered broadcasts to manipulate or drop the broadcasted intent in the middle). Differently, our ASVs #1 and #2 target all types of broadcast APIs (e.g., *sendBroadcast* and etc.) and other functions in the *activitymanager*. Android permission re-delegation attacks [35, 26, 47, 53, 33] have been well studied. Long et al. [47] design a static analysis tool to find permission re-delegation vulnerabilities between app components. Pileup attacks [55] is based on a flaw in the *packagemanager* service that targets system update. We characterize a new type of hazard and then design an efficient tool to help discover the ASVs in various system services. PoC attacks and relevant system-level defenses are designed. Virtualization based defenses [22, 54] on Android can help mitigate some of the discovered ASVs, however, some shared hardware resources (the single user screen among trusted/untrusted VM instances) will still be vulnerable to attacks (e.g., screen view manipulating problem in ASV#3 and #4).

Bugs in Android apps have been studied widely in previous work [37, 50, 49]. Hsiao et al. [37] build a system to identify a number of known and unknown harmful concurrency errors in Android apps. Ravindranath et al. [50] design an Android app instrumentation approach to monitor app performance for developers. Our work primarily focuses on the Android framework side, which contains most critical system services that provide essential support for Android apps. Due to the specific *wake-lock* design in Android that makes it extremely hard for developers to use, Pathak et al. [49] characterize the problem and detect a new type of energy bugs in Android apps. Our work characterized and detected a new type of vulnerability and then fully analyzed the discovered vulnerabilities.

DoS Attacks are a well explored research area [44, 56, 23, 32]. Armando et al. [23] abuse the loosely protected Unix socket permission in the Zygote process to fork an unbounded number of processes to mount a DoS attack on Android and fix the problem by setting the right access control bits. However, our ASVs are a more general type of vulnerability caused by the design of the *system server*. It is deeply rooted in the fundamental tension between program complexity and security and is much easier to exploit than to fix, which can appear in other systems with similar design. Previously, Martin et al. [32] use formal method to identify deadlock vulnerability that causes DoS attacks in IEEE 802.11w protocol. Detecting deadlock based ASVs on Android is our future work.

7. CONCLUSION

Our new understanding of the SS reveals a general design flaw in its concurrency control scheme and failure recover mechanism. We further characterize it as a general vulnerability (i.e., ASV). We then build a tool to help cost-effectively analyze 2,154 risky methods, which helps reveal four previously unknown ASVs in two critical services (i.e., *activitymanager* and *windowmanager*). We further easily craft several PoCs by exploiting the ASVs under various scenarios. Google immediately confirmed our findings, and we proposed short-term mitigation for users, refinement for the CC mechanism and a *smarter* watchdog scheme for the SS. Google

also takes some of our defense proposals. Our study shows that some seemingly neat and robust design choices can lead to unexpected flaw. Hence, the secure design for critical components in the rapidly-evolving mobile systems becomes very important.

8. ACKNOWLEDGMENTS

We greatly appreciate the insightful comments and constructive feedback from the anonymous reviewers. We would like to give special thanks to Dr. William Enck for his detailed instructions for preparing our camera ready version. This work was partially supported by NSF CCF-1320605, AROW911NF-13-1-0421 (MURI), NSF SBE-1422215 and NSFC 61100226 and Beijing Natural Science Foundation 4144089. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation and Army Research Office.

9. REFERENCES

[1] Android and RTOS together: The dynamic duo for today's medical devices. http://goo.gl/StURzu.

[2] Android Auto: Driven by Android. https://www.android.com/intl/en_us/auto/.

[3] Android banking apps vulnerable to cash theft by CAS hole hackers. http://goo.gl/LerfXT.

[4] Android Phones Hit by Ransomware. http://goo.gl/W0TBy3.

[5] Android ransomware 'Koler' turns into a worm. http://goo.gl/bpo66T.

[6] ANTLR (ANother Tool for Language Recognition). http://www.antlr.org/.

[7] CVE-2014-3500: Cordova cross-application scripting via Android intent URLs. http://goo.gl/YhyrRw.

[8] Despite iPhone 6 hype, Android continues to dominate iOS market share. http://goo.gl/xmKfrP.

[9] Icon vulnerability causes mobile-system crashes. http://goo.gl/SjvVVB.

[10] iPhone Ransomware. http://goo.gl/58CsT2.

[11] Medical Device Manufacturers Improve Their Bedside Manner with Android. http://goo.gl/d2JF3.

[12] Mobile ransomware campaigns. http://goo.gl/4aRzqT.

[13] Mobile ransomware scarepakage. https://blog.lookout.com/blog/2014/07/16/scarepakage/.

[14] Northrop grumman news release: DARPA ASPN project. http://goo.gl/3IUSXu.

[15] Northrop to demo darpa navigation system on android. http://goo.gl/bgRggD.

[16] Six vulnerabilities found in lots of banking apps. http://goo.gl/tBT8po.

[17] Soot: a java optimization framework. http://www.sable.mcgill.ca/soot/.

[18] Strand-1 satellite launches Google Nexus One smartphone into orbit. http://goo.gl/r5zkrs.

[19] The Lookout AVD v8.17-8a39d3f for Android allows attackers to cause a denial of service . http://cve.scap.org.cn/CVE-2013-3579.html.

[20] Trend Micro Discovers Vulnerability That Renders Android Devices Silent, July 30, 2015. http://goo.gl/k2bX0x.

[21] Why Android will be the biggest selling medical devices in the world by the end of 2012. http://goo.gl/G5UXq.

[22] ANDRUS, J., DALL, C., HOF, A. V., LAADAN, O., AND NIEH, J. Cells: a virtual mobile smartphone architecture. In *Proceedings of the Twenty-Third ACM SOSP 2011*.

[23] ARMANDO, A., MERLO, A., MIGLIARDI, M., AND VERDERAME, L. Would you mind forking this process? a denial of service attack on android (and some countermeasures). In *Information S&P Research*. 2012.

[24] AU, K. W. Y., ZHOU, Y. F., HUANG, Z., AND LIE, D. Pscout: analyzing the android permission specification. In *ACM CCS* (2012).

[25] BIANCHI, A., CORBETTA, J., INVERNIZZI, L., FRATANTONIO, Y., KRUEGEL, C., AND VIGNA, G. What the app is that? deception and countermeasures in the android user interface. In *IEEE Symposium on SP, 2015*.

[26] BUGIEL, S., DAVI, L., DMITRIENKO, A., FISCHER, T., AND SADEGHI, A.-R. Xmandroid: A new android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04* (2011).

[27] CHEN, K., WANG, P., LEE, Y., WANG, X., ZHANG, N., HUANG, H., ZOU, W., AND LIU, P. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *USENIX Security 15'*.

[28] CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. Analyzing inter-application communication in Android. In *Proceedings of MobiSys 11'*.

[29] DAVID, F. M., AND CAMPBELL, R. H. Building a self-healing operating system. In *Dependable, Autonomic and Secure Computing, 2007*.

[30] DAVID, F. M., CARLYLE, J. C., AND CAMPBELL, R. H. Exploring recovery from operating system lockups. In *USENIX Annual Technical Conference* (2007).

[31] DESNOYERS, M., MCKENNEY, P. E., STERN, A. S., DAGENAIS, M. R., AND WALPOLE, J. User-level implementations of read-copy update. *Parallel and Distributed Systems, IEEE Transactions on* (2012).

[32] EIAN, M., AND MJOLSNES, S. A formal analysis of ieee 802.11 w deadlock vulnerabilities. In *INFOCOM 2012*.

[33] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI* (2010).

[34] FELT, A. P., HA, E., EGELMAN, S., HANEY, A., CHIN, E., AND WAGNER, D. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security* (2012).

[35] FELT, A. P., WANG, H. J., MOSHCHUK, A., HANNA, S., AND CHIN, E. Permission Re-Delegation: Attacks and Defenses. In *Proceedings of the 20th USENIX Sec, 2011*.

[36] HEUSER, S., NADKARNI, A., ENCK, W., AND SADEGHI, A.-R. Asm: A programmable interface for extending android security. In *Sec' 14*.

[37] HSIAO, C.-H., PEREIRA, C. L., YU, J., POKAM, G. A., NARAYANASAMY, S., CHEN, P. M., KONG, Z., AND FLINN, J. Race detection for event-driven mobile applications. In *Proceedings of the 35th ACM SIGPLAN Conference on PLDI 2014*.

[38] HUANG, H., CHEN, K., REN, C., LIU, P., ZHU, S., AND WU, D. Towards discovering and understanding unexpected hazards in tailoring antivirus software for android. In *ACM AsiaCCS 15'*.

[39] HUANG, H., ZHANG, S., OU, X., PRAKASH, A., AND SAKALLAH, K. Distilling critical attack graph surface iteratively through minimum-cost sat solving. In *ACSAC 11'*.

[40] HUANG, H., ZHAO, F., AND YE, M. Estimate the influential level of vulnerability instance based on hybrid ranking for dynamic network attacking scenarios. In *IEEE ISSPA 2010*.

[41] HUANG, H., ZHU, S., LIU, P., AND WU, D. A framework for evaluating mobile app repackaging detection algorithms. In *Trust and Trustworthy Computing*. Springer, 2013.

[42] JIN, X., HU, X., YING, K., DU, W., YIN, H., AND PERI, G. N. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *Proceedings of the 2014 ACM CCS*.

[43] KONG, D., CEN, L., AND JIN, H. Autoreb: Automatically understanding the review-to-behavior fidelity in android applications. In *ACM CCS* (2015).

[44] KUMAR, S., AND SURISETTY, S. Microsoft vs. apple: Resilience against distributed denial-of-service attacks. *Security & Privacy, IEEE* (2012).

[45] LEHEY, G. Improving the freebsd smp implementation. In *USENIX Annual Technical Conference, FREENIX Track* (2001).

[46] LIU, R., ZHANG, H., AND CHEN, H. Scalable read-mostly synchronization using passive reader-writer locks. In *Proceedings of the 2014 USENIX ATC*.

[47] LU, L., LI, Z., WU, Z., LEE, W., AND JIANG, G. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM CCS*.

[48] LUNDBERG, D., FARINHOLT, B., SULLIVAN, E., MAST, R., CHECKOWAY, S., SAVAGE, S., SNOEREN, A. C., AND LEVCHENKO, K. On the security of mobile cockpit information systems. In *Proceedings of the 2014 ACM CCS*.

[49] PATHAK, A., JINDAL, A., HU, Y. C., AND MIDKIFF, S. P. What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys 12'.

[50] RAVINDRANATH, L., PADHYE, J., AGARWAL, S., MAHAJAN, R., OBERMILLER, I., AND SHAYANDEH, S. Appinsight: Mobile app performance monitoring in the wild. In *OSDI* (2012).

[51] SCHLEGEL, R., ZHANG, K., ZHOU, X.-Y., INTWALA, M., KAPADIA, A., AND WANG, X. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *NDSS* (2011).

[52] SMALLEY, S., AND CRAIG, R. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS* (2013).

[53] WEI, F., ROY, S., OU, X., ET AL. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM CCS*.

[54] WU, C., ZHOU, Y., PATEL, K., LIANG, Z., AND JIANG, X. Airbag: Boosting smartphone resistance to malware infection. In *Proceedings of NDSS'14*.

[55] XING, L., PAN, X., WANG, R., YUAN, K., AND WANG, X. Upgrading your android, elevating my malware: Privilege escalation through mobile os updating. In *IEEE S&P 14*.

[56] YAN, G., LEE, R., KENT, A., AND WOLPERT, D. Towards a bayesian network game framework for evaluating ddos attacks and defense. In *CCS 12'*.

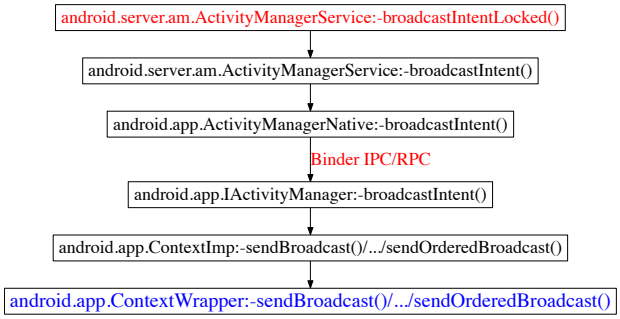

Figure 6: Identified triggering points of ASV #1

[57] ZHANG, F., HUANG, H., ZHU, S., WU, D., AND LIU, P. View-Droid: Towards obfuscation-resilient mobile application repackaging detection. In *ACM WiSec 14'*.

[58] ZHAO, F., HUANG, H., JIN, H., AND ZHANG, Q. A hybrid ranking approach to estimate vulnerability for dynamic attacks. *Computers & Mathematics with Applications* (2011).

[59] ZHENG, C., ZHU, S., DAI, S., GU, G., GONG, X., HAN, X., AND ZOU, W. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *SPSM 2012*, ACM.

APPENDIX

A. APPENDIX

Algorithm 1 Pseudocode to build the vector database for all SS methods and query candidate risky methods from the database

Input: $SSjar$:{Android System Server Jar files};
ϕ:{specify the query criteria};
Output: $SSmethod$: {Ranked top-k Risky Methods};
ν : RiskMethodVector[δ,γ,σ,ϵ,α, ω]
δ : {# of loops in the method, value: [0, n)};
γ : {# of instructions in critical sections, value: [0, n)};
σ : {# of method invocations, value: [0, n)};
ϵ : {# of times appeared as callee, value: [0, n)};
α : {a lock-suffixed method/not, value: 0/1};
ω : {a watchdog monitoring method/not, value: 0/1};
ξ : {risky vector database of all the methods};
$\chi \leftarrow staticCodeAnalyzer(SSjar)$ {χ is the superCFG}
for each system service *S* in the system server **do**
  for each class *C* in *S* **do**
    for each method *M* in *C* **do**
      initialize(ν)
      $\nu \leftarrow$ *DepthFirstSearch*(χ, ν, *C*, *M*)
      //To collect and update relevant information in ν
      $\xi \leftarrow \xi$ + [ν, *C*, *M*]
    end for
  end for
end for
for each system service *S* in the system server **do**
  $SSmethod \leftarrow$ *queryRiskMethods*(*S*, ϕ, ξ)
end for