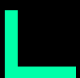
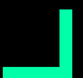
Quokka

ACADEMIC PAPER

Dazed Droids

A Longitudinal Study of Android Inter-App Vulnerabilities





Dazed Droids: A Longitudinal Study of Android Inter-App Vulnerabilities

Ryan Johnson*
Kryptowire
Fairfax, VA, USA
rjohnson@kryptowire.com

Mohamed Elsabagh
Kryptowire
Fairfax, VA, USA
melsabagh@kryptowire.com

Angelos Stavrou*
Kryptowire
Fairfax, VA, USA
astavrou@kryptowire.com

Jeff Offutt
George Mason University
Fairfax, VA, USA
offutt@gmu.edu

ABSTRACT

Android devices are an integral part of modern life from phone to media boxes to smart home appliances and cameras. With 38.9% of market share, Android is now the most used operating system not just in terms of mobile devices but considering all OSes. As applications' complexity and features increased, Android relied more heavily on code and data sharing among apps for faster response times and richer user experience. To achieve that, Android apps reuse functionality and data by means of inter-app message passing where each app defines the messages it expects to receive.

In this paper, we analyze the proliferation of exploitable inter-app communication vulnerabilities using a rich corpus of 1) a representative sample of 32 Android devices, 2) 59 official Google Android versions, and 3) the top 18,583 apps from 2016 to 2017. This corpus covers 91 Android builds from version 4.4 to present. To the best of our knowledge, ours is the first longitudinal study looking into the propagation of vulnerabilities across AOSP builds, between AOSP and a diverse set of devices, and across app versions over a period of 13 months. To identify inter-app vulnerabilities, we developed *Daze* as a swift and fully-automated framework for extracting app components and fuzzing all app interfaces. *Daze* needs only about three hours for full-device analysis or two minutes per app on average. We identified 14,413 vulnerabilities and quantified their exposure time and the number of versions affected. Our findings revealed that 51.7% of Android devices and 49% of the top 300 apps on Google Play contained at least one critical inter-app vulnerability. We found that about 15% of fixed vulnerabilities lived for more than 100 days before being patched, more than 20% of unpatched vulnerabilities have existed for at least 180 days, and 45% of unpatched vulnerabilities persisted through the latest two to four consecutive app versions in our dataset.

KEYWORDS

Android System Fuzzing, Denial of Service, Data Disclosure

ACM Reference Format:
Ryan Johnson, Mohamed Elsabagh, Angelos Stavrou, and Jeff Offutt. 2018. Dazed Droids: A Longitudinal Study of Android Inter-App Vulnerabilities. In *ASIA CCS '18: 2018 ACM Asia Conference on Computer and Communications Security, June 4–8, 2018, Incheon, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3196494.3196549

1 INTRODUCTION

Smart devices and appliances have evolved, striving to be fully integrated with our day-to-day activities, to the point that various research efforts [19, 23, 28] are currently engaged in studying the effects of *nomophobia*, the fear of being without a mobile device. Flurry Analytics reported that the United States population now spends at least five hours a day on their mobile devices [4]. Moreover, according to International Data Corporation, 373.1 million mobile devices were shipped worldwide in the third quarter of 2017 alone, representing a 2.7% year-over-year increase [1]. Endeavouring to offer a rich user experience that can fulfill the growing reliance on mobile devices, mobile platforms are heavily built around a design paradigm that encourages mobile applications — hereinafter referred to as apps — to seamlessly interact. Specifically on Android platforms, apps despite running in isolation, can share their services and data via message passing over inter-app communication channels offered and managed by the platform. We particularly focus on Android platforms in this study because of their popularity (85% market share of mobile devices in the first quarter of 2017 [3]).

Much of the communication between and within Android apps occurs via "intent" objects: a message-like abstraction that provides a fundamental communication mechanism that facilitates data exchange. Unfortunately, developers can (un)intentionally expose interfaces in their Android apps, making them accessible to other apps co-located on a mobile device. An exposed interface offers a potential entry point into an app that can be abused. This can be caused by improper handling of received intents, for instance, making assumptions about the presence of certain data in a received intent. In addition, inadequate exception handling of inter-app messages can enable an app to force-crash other apps and services or even the Android Operating System (OS) itself. System crashes can be intentionally and repeatedly caused by a malicious app on the

*Also with George Mason University.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASIA CCS '18, June 4–8, 2018, Incheon, Republic of Korea
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5576-6/18/06…$15.00
https://doi.org/10.1145/3196494.3196549

device to create local Denial of Service (DoS) attacks or perhaps a crypto-less ransomware. Although the Android OS offers mechanisms to force-remove third party apps, these removal methods may not be available on all devices or may require wiping all user data on an infected device [17].

The Android Open Source Project (AOSP) developed by Google provides the official version of Android code.[1] Android vendors fork AOSP and modify it to add features and provide a custom user experience. Faults in a version of AOSP are severe since they will propagate to all Android devices that run or extended that particular version. Vendors also risk introducing new vulnerabilities in addition to the ones inherited from AOSP. Several research efforts (e.g., [2, 6, 16, 18, 20, 22, 27, 29]) have analyzed Android apps for the existence of inter-app communication vulnerabilities. However, existing solutions have been largely manual, incurred unpractical analysis time, required special permissions and knowledge of the target device and OS version, and focused on examining the resilience of statically-declared Android app components against DoS attacks caused by malformed intents. As we show in this paper, several classes of vulnerabilities — including DoS, privilege escalation, and data leakage — do exist outside the intent model as well as in dynamically-declared app components (see Sections 2 and 4).

To help address these shortcomings, we designed and implemented *Daze*, a novel system to automatically trigger and analyze inter-app communication vulnerabilities in Android platforms. *Daze* uncovers inter-app developer errors and their effects by efficiently enumerating and null-fuzzing all statically and dynamically accessible app components on a device, including these of the Android OS itself. *Daze* fuzzes exported app interfaces using both null and not-null but empty payloads, among others, to expose developer errors of omission (e.g., not checking for the absence of inter-app message data at runtime or not protecting privileged components). These errors can result in process crashes, system crashes, privilege escalation, and data disclosure. *Daze* identifies instances of privilege escalation by monitoring the files created on external storage and changes to the system settings as each component is tested. *Daze* is completely automated and does not require prior knowledge about the target device.

Furthermore, going beyond just discovery, *Daze* automatically generates zero-permission exploits that give direct control over the victim device availability and usability. This can be exploited to craft ransomware via crypto-less attack vectors. Having the ability to crash an app may seem low-risk, but it can enable a malicious app to set itself as the gatekeeper to a vulnerable app, determining when and if a vulnerable app gets to execute. Even worse, persistently exploiting a known fault in an app enables a **controlled crash-loop** DoS attack on Android devices where the Android OS recurrently pops up the app crash dialog (Figure D.1) and restarts the crashing app in the background, but then the attacker crashes it again. This allows the attacker to control overall device usability since the recurring OS app crash dialog box takes the input focus away from other Graphical User Interface (GUI) elements, hindering the user from productively interacting with the device.

[1] AOSP available at: http://source.android.com

Our findings are alarming: using *Daze* to exhaustively scan all unique Android 4.4 to 8.0 platform firmware builds across 32 different devices from low-end to flagship Android vendors, we discovered that more than 50% of current devices contain a system crash vulnerability. Exacerbating the problem, we show that system crashes can result in user data disclosure on certain Android devices due to vendor modifications. Overall, *Daze* discovered 4,972 process crashes and 64 system crashes in the tested devices, providing reproducible test cases and stack traces for error resolution. Some of the identified process crashes belong to processes that provide telephony and Bluetooth services that can be targeted for DoS attacks. Specifically, our testing revealed that pre-installed GApps provide a large DoS attack surface.[2]

In addition, we analyzed a sample set of 18,583 apps downloaded from the top free apps in each category on Google Play between March 2016 and April 2017. We found that once a inter-app vulnerability is introduced, it is likely to be present in subsequent versions of an app. We also discovered that about 15% of patched vulnerabilities in the apps in our dataset persisted for 60 to 600 days. In addition, 50% of unpatched vulnerabilities existed in at least the two latest app versions in our dataset, with around 30% of unpatched vulnerabilities persisting for more than 100 days till the end date of our data collection period. Moreover, 49.6% of the top 300 apps can be easily crashed by a zero-permission external app.

To understand the origin and propagation of the vulnerabilities we discovered, we also tested the robustness of AOSP builds and, by extension, the vendors that modify them. We tested all available AOSP builds for the Nexus 5 and Nexus Player devices and discovered that all Android 5.1, 6.0, and 6.0.1 builds for Nexus 5 were vulnerable to a system crash. We also found that each of the 31 AOSP TV builds from Android 5.0 to 8.0 had at least one critical DoS vulnerability that may require a factory reset to recover. We further discovered an attack vector that can cause any Android device produced to date to run out of memory.

To summarize, we make the following contributions:

- *Daze*, an automatic inter-app vulnerability analysis system to test all exposed app interfaces on Android devices.[3]
- We provide a longitudinal analysis covering 32 different Android devices, all Android AOSP 4.4 to 8.0 builds, and 18,583 free apps from Google Play.
- We discovered several zero-day DoS, data disclosure, and privilege escalation vulnerabilities across different devices and vendors.
- We provide a novel universal DoS attack on any Android platform by tricking a critical system process into running out of memory.

2 BACKGROUND

Android apps are compartmentalized into components to facilitate code reuse. An app component is an entry-point in a mobile app that serves a particular purpose within the context of the app. App components are typically declared and statically-registered in a

[2] GApps are the proprietary Google apps that come pre-installed on most Android devices, such as the Play Store, Gmail, and Maps.
[3] *Daze* is available at: https://github.com/Kryptowire/daze

```
public void onReceive(Context context, Intent intent) {
   ...
   String action = intent.getAction();
   if (action.equals("com.sec.android.intent.action.SSRM_MDNIE_CHANGED")) {
      ...
      Bundle bundle = intent.getExtras();
      int value = bundle.getInt("value"); // NullPointerException if `bundle` is null
   ...
```

Listing 1: Recreated source code of a vulnerable broadcast receiver.

manifest file (`AndroidManifest.xml`) that provides the app specifications and configuration data. Android provides four different app components from which an app can be built: **activity**, **service**, **broadcast receiver**, and **content provider**. An **activity** provides a GUI for direct user interaction via GUI elements. An activity is the only app component that provides a GUI to the user. A **service** performs long-running tasks in the background. A **broadcast receiver** registers for particular events and responds to them as they occur. A **content provider** acts as an archive of structured data.

2.1 Accessing App Components

Of the four app components, all but the content provider are accessed by sending "intent" objects. An intent is an abstraction for a message that is sent by a source app component to one or more destination app components. Intents are the primary means with which an app communicates with itself and other apps. The destination of an intent is indicated by including an action and/or a unique app component address (i.e., package name and component class name).[4] An action is a string that generally indicates the purpose of the intent (e.g., `android.provider.Telephony.SMS_RECEIVED` for a received text message). App components can register for specific actions in the manifest file. An action may resolve to one or more app components that can handle it. An intent often carries embedded data, including primitive types and complex objects.

An app component declared in the manifest file can set the access requirements for other apps to interact with it. A component can be "exported," exposing its functionality to the system and other apps on the device.[5] App components are generally not exported by default (i.e., accessible only from within the same app) except dynamically-registered broadcast receivers, which are always exported. The Android OS will export an app component (potentially against the developer's intentions) if the app component declares at least one action that it can handle and does not explicitly state that it should not be exported. Additionally, apps can control access to their exported components by declaring and using self-declared permissions or permissions declared by the Android OS.

2.2 Errors of Omission

Once an intent is sent to an app component, various errors of omission can occur. If the destination app component is declared but not actually implemented, the receiving app will crash due to an uncaught `ClassNotFoundException`. An undefined app component is a component that has a valid entry in the app's manifest file but the component itself is not implemented in the app's code. If the component receiving an intent is implemented, there are various exceptions that can be encountered by further errors of omission resulting in an uncaught exception and process termination if appropriate error handling is absent. Even before processing the received intent, an app component may encounter an `UnsatisfiedLinkError` if a required native library is missing. When an intent is received by an app component, one or more callback methods specific to the app component type are executed. During the processing of a received intent, the most common error of omission manifests as a `NullPointerException` due to accessing null objects. Certain apps do not gracefully handle the absence of expected data, resulting in an unexpected crash. Most of the exceptions can be addressed through proper input validation and exception handling at runtime.

In some cases, a received intent will not contain any embedded data. In other cases, the sole purpose of the app component is process embedded data from an intent. Internally, most embedded data within an intent is stored in a `Bundle` object, which is a map data structure that allows the storage and retrieval of key-value pairs. A recipient app component may also extract the action string or embedded Uniform Resource Identifier (URI) of an intent. When an app component with inadequate error handling and null-checking *assumes* these values will not be null, a `NullPointerException` can occur. The Android OS itself declares components accessible to third-party apps. An uncaught exception occurring in a component *within the Android OS* can lead to the crashing of critical system processes, triggering the OS to reboot in an attempt to recover. We provide a motivating example in Listing 1 that shows a source code snippet from a broadcast receiver app component within the Android OS that will encounter a system crash if the received intent does not carry a `Bundle` object. The snippet was recreated from disassembled bytecode from a Samsung Galaxy S6 Edge running Android 6.0.1 with a build number of `MMB29K.G925AUCS5DPK5`.

In addition to inadequate error handling, certain app components will themselves throw a `RuntimeException` if an unexpected data item is not present in the intent. Various other exceptions can occur, such as errors in handling the app lifecycle (`IllegalStateException`), forgetting to call a base method (`SuperNotCalledException`), and referencing classes that are not defined (`NoClassDefFoundError`). The full list of exceptions we encountered in our experiments are provided in Tables D.3 to D.5.

3 OVERVIEW OF *DAZE*

Figure 1 illustrates the workflow of *Daze*. We developed *Daze* to automatically determine if certain classes of concrete failures or

[4] A package name is a unique identifier for an Android app. No two apps can have the same package name on the same Android device.

[5] The exported property is controlled by setting the `android:exported` attribute for the component in the manifest file. See: https://developer.android.com/guide/topics/manifest/receiver-element.html#exported

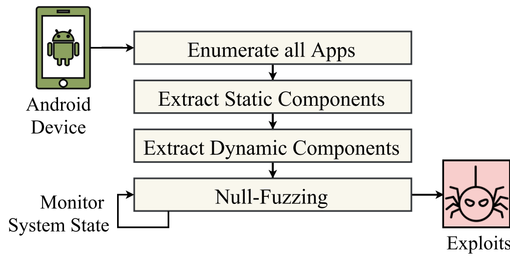


Figure 1: Workflow of Daze.

unexpected behavior exist in Android apps and the Android OS on a given device. *Daze* tests all four types of app components and provides the user with a list of faulty processes, stack traces, discovered behaviors, and an exploit composed of a trace of events and Application Programming Interface (API) calls that can be replayed to trigger a discovered vulnerability.

Daze is implemented in Java in 8.1K SLOC and associated scripts. *Daze* was not extended from any other previous software project. Scanning is done in an automated manner where the user selects desired options and starts the analysis. *Daze* uses null-fuzzing — including not-null but empty objects — to fuzz app components. We opted for null-fuzzing over data-fuzzing since random data-fuzzing offered a negligible improvement (~1%) over null-fuzzing of Android components as witnessed by prior studies [2, 22] and elaborate data-fuzzing techniques suffer from impractical runtime requirements (*hours per app* [15]) making them unsuitable for testing an entire device.

3.1 Identifying Statically Registered Components

Daze extracts statically-registered components from an app by querying the OS package manager for all installed packages, then iterating through each package information looking for components declared with the activities, services, receivers, and providers tags. The package manager fills in this information from the app's manifest file, which cannot be modified once the app has been installed. *Daze* ignores components that are not exported or require a permission to access. The focus is for zero-permission reproducible test cases to crash apps or the system, so only open components with no permission requirements are considered. In addition, users may be more willing to download an app with no permissions. Since *Daze* is open-source, it can be easily modified to request all available third-party permissions and test components that are protected with permissions declared by the Android OS.

3.2 Identifying Dynamically-Registered Broadcast Receivers

.

Broadcast receivers are the only app components that can be both statically registered and dynamically registered. At runtime, an app can create and register a broadcast receiver to be eligible to receive one or more action strings. Dynamically-registered broadcast receivers can only be addressed by using an action string. *Daze* enumerates dynamically-registered broadcast receivers by executing the command `` `dumpsys activity broadcasts` ``. For security reasons, third-party apps are not allowed to obtain this list of dynamically registered broadcast receivers or read from the system-wide Android log. To workaround this limitation, *Daze* is granted the system development permissions using the Android Debug Bridge (ADB) command `` `adb pm grant <package> <permission>` ``.[6] The `READ_LOGS` and `DUMP` permissions can only be externally granted via ADB for development or testing. After these two development permissions are granted to *Daze*, it can obtain a listing of the active broadcast receivers, including those that are dynamically-registered, and access the system-wide Android log. *Daze* parses the Android log to detect app crashes, native crashes, and system crashes.

3.3 Testing Intent-Accessible Components

Daze identifies statically- and dynamically-registered components residing within all apps on the device and sends intents to all discovered components that are exported and not permission-protected. For all components except content providers, the system sends up to four intents: First, it sends an intent containing the minimum data required to be delivered to the target component — namely, the package name and class for an activity, broadcast receiver, or service, or the action string for dynamically-registered broadcast receivers. If a crash is not encountered due to this intent, then *Daze* adds an empty `Bundle` object to the intent and sends it again. If an error is still not encountered for components that are not dynamically-registered, it adds an empty action string and sends the intent again. Lastly, an intent with a shcemeless URI is sent. We chose to focus on the action string, Bundle, and URI since, based on our experience, these are commonly used data items in intents. To cover all code sites reachable by an intent, the system sends intents with the `FLAG_ACTIVITY_SINGLE_TOP` flag to also force the delivery of the intent to the `onNewIntent` method of activity components. *Daze* also calls the `stopService` method to trigger cleanup routines that may access the intent that started the service.

Daze monitors the Android log to record the system-wide effects of issued intents. Once it finishes sending intents to all exported components, it examines the log file recorded for each sent intent. To avoid side-effects and to isolate different runs, *Daze* separately replays each intent that resulted in a fatal exception or a system crash to verify that it indeed triggers a failure condition to provide accurate attribution to individual intents.

3.4 Testing Content Providers

Unlike other app components, content providers are not directly accessed via intents. Any content provider will have to implement a set of methods from the abstract `ContentProvider` class provided by the platform. Apps can read and write data to a content provider using a platform-managed content resolver which has the most safeguards with regard to handling null object references and invalid input. In addition, content providers are not exported by default and tend to be protected by permissions since they act as data repositories. Content providers are generally backed by an SQLite database and must implement the following operations:

[6] ADB is an Android SDK tool that allows a computer to interact with Android devices.

delete, insert, query, and update. *Daze* tests content providers by null-fuzzing all callable methods in their classes.

Testing content providers provides some difficulty as a crash in a content provider causes any app connected to the crashing provider to be killed. Specifically, `ActivityManagerService` within the Android OS will terminate any process with an ongoing connection to a crashing content provider. *Daze* will stop testing a content provider after this occurs 3 times and note that the content provider encounters a fatal error during testing.

3.5 Monitoring System State

Prior to the sending of each intent, *Daze* enumerates all the files on external storage (SD card) to obtain a snapshot of the current state. After sending intents to a component, it will again take a snapshot of the files and compare. If a file has been removed or added, the change will be detected and the file path and file size will be recorded. External storage can be read by any app that requests the `READ_EXTERNAL_STORAGE` permission, so sensitive data should not be written to it. At the end of the analysis, the changes are presented to the user which can view the newly added files to examine their contents. This capability can detect the taking of screen snapshots and dumping of log files to external storage which was observed during the testing of devices (see Section 4.3 for details). Screen snapshots are flagged since they are stored in a known directory on external storage with a known file extension.

Changes to device settings are also recorded before and after testing each component to determine if a privileged process modifies them during its execution. This is accomplished by querying system properties and the secure, global, and system settings.[7] The capability shows whether the tested component has made changes, such as enabling or disabling various communication capabilities. For example, the enabling and disabling of Wi-Fi is automatically detected by monitoring changes to the value of `wifi_on` key in global settings.[8] In addition, a component may extract an expected field from an incoming intent and write its value to device settings. This may be observed when a value of null is written to system settings instead of a concrete value.

Once an app encounters a fatal exception, the Android OS displays a dialog box to the user indicating that an app has crashed. Once this dialog box is present, no additional components can be launched within the crashed app until the dialog box is dismissed or until a crashed service in the app restarts. To be able to quickly launch additional components in a crashed process, *Daze* obtains the current window handle by using the `dumpsys` command and uses the `input` command via ADB to inject key events to programmatically dismiss the dialog box. This allows for the crashed process to be restarted when testing a different app component, without requiring user intervention.

4 STUDY 1: DEVICE EVALUATION

We tested *Daze* on a representative set of 32 low-end to flagship Android devices from 21 vendors covering Android 4.4 to Android 8.0. In this section we discuss our findings for the number of concrete process crashes, system crashes on specific Android devices, instances of privilege escalation, and data disclosure on the tested devices.[9] 72% of the tested devices contained at least one system crash vulnerability. We tested all of the pre-installed apps present on the 32 Android devices. Table 1 provides the total number of process and system crashes on the tested devices. *Daze* triggered 4,972 unique app crashes and 64 unique system crashes across all devices, taking about three hours on average to scan an entire device.

Table 1: Unique app and system crashes per device. 📱 indicates the vulnerability was introduced by the vendor. 🤖 indicates it was introduced by AOSP.

Device	OS Version	App Crashes		Sys Crashes	
		📱	🤖	📱	🤖
Alcatel A30	7.0	66	146	0	0
Alcatel A30 Plus	7.0	77	135	0	0
Amazon Fire 7.0 Inch Tablet	5.1.1	66	34	0	1
Amazon Fire TV Stick 2	5.1	18	2	0	1
BLU Advance 5.0	5.1	41	76	0	1
BLU Grand M	6.0	36	94	0	1
BLU Grand XL	7.0	34	135	0	0
BLU R1 HD	6.0	43	139	1	1
Cubot X16S	6.0	66	113	0	1
Doogee X5	6.0	59	102	0	1
Figo Atrium 5.5	5.1	37	90	1	1
Figo Virtue 4.0	6.0	33	133	0	1
Google Pixel	8.0	0	137	0	0
Juning TV Box	5.1.1	22	78	2	1
Juning Z8	5.1.1	69	102	0	1
Kata C2	6.0	75	125	0	1
Leagoo Z5C	6.0	13	88	0	1
LG Phoenix 2	6.0	112	160	1	0
NPOLE Tablet	5.1.1	14	35	0	1
Nvidia Shield Android TV	7.0	39	74	0	1
Plum Axe Plus 2	6.0	51	125	0	1
Plum Compass	6.0	21	84	0	1
RCA Q1	6.0	55	117	0	1
RCA Voyager Tablet 2	5.0	28	121	1	0
Samsung Galaxy S5	6.0.1	57	145	14	1
Samsung Galaxy S6 Edge	6.0.1	98	98	17	1
Samsung S8+	7.0	45	70	0	0
Sony Bravia Android TV	6.0.1	86	65	0	2
Ulefone Power 2.0	7.0	43	128	0	0
Xiaomi Redmi 4	6.0.1	100	82	0	2
Yuntab	4.4.2	92	145	0	0
ZTE Maven 2	6.0.1	74	124	1	2
TOTAL		1,670	3,302	26	38

We attributed each of the vulnerabilities discovered in the tested devices to either vendor apps or AOSP apps (see Table 1; ratios are plotted in Figure D.2).[10] We identified AOSP apps by recording the package names of the apps present in AOSP builds for smartphones, tablets, and Android TV. We attributed a fault to an AOSP app if it occurred within an app in the AOSP apps list. For attributing system crashes, we manually examined the stack trace and checked whether it occurred in AOSP by examining AOSP source code.

[7] Secure settings are present only on devices running API level 17 and above.
[8] Additional items in global settings are found in https://developer.android.com/reference/android/provider/Settings.Global.html

[9] All findings have been responsibly disclosed to Google and affected vendors prior to the publication of this document.
[10] We made no distinction between GApps and AOSP apps in this study.

Table 2: Processes with the highest number of crashes.

Process Name	# of Crashes
com.google.android.gms.ui	1,011
com.android.settings	611
com.android.phone	572
com.google.android.setupwizard	228
com.android.contacts	111
com.google.android.gms	104
com.google.android.gms.persistent	94
com.android.mms	84
com.android.cts.priv.ctsshim	84
com.google.android.gms	81
com.android.bluetooth	67
android.process.media	66

4.1 App Crash Vulnerabilities

We discovered that more than 50% of fatal exceptions were present in AOSP apps. Amazon devices were an exception since Amazon maintains its own Android version called Fire OS that primarily uses Amazon's apps instead of GApps. For the Google Pixel device, we consider all apps on the device to be AOSP/GApps. Table 2 shows the top 12 crashed processes across all devices. These were either Google or AOSP processes, with Google `gms.ui` (a process within the Google Play Services app) topping the list at a total of 1011 crashes. A particularly important process, `com.android.phone`, is surprisingly vulnerable to being crashed by an external app with a total of 572 crashes. A crash of the `com.android.phone` process can deny telephony functionalities, including the ability to receive or make calls, which can have dire consequences in times of emergency. There were, on average, 17.87 vulnerable components on each device that crash the `com.android.phone` process. This ranged from the Google Pixel running Android 8.0 with 2 vulnerable components to the Yuntab running Android 4.4.2 with a total of 54 distinct components or broadcast actions to crash the `com.android.phone` process. The `com.android.bluetooth` process has the 11th most crashes with a total of 67. Launching continual DoS attacks using intents can restrict the user's access to wireless communication capabilities on the device, or hinder the usability of the device by causing an app crash-loop as discussed in Section 1.

4.2 System Crash Vulnerabilities

For each system crash that *Daze* uncovered (see Table 1), we investigated the cause to attribute it to either AOSP code or vendor code. All of the devices were running the most recent Android versions available to them at the time of testing. The majority of system crashes were caused by vendor modifications (62.5%). If the system crashes from Samsung were excluded, AOSP code would be responsible for 85% of the system crashes. Vendor modification was responsible for system crashes in three component types: broadcast receiver (34 crashes), service (1 crash), and content provider (1 crash). AOSP code was responsible for crashes in two component types: activity (22 crashes) and broadcast receiver (6 crashes). This breakdown is provided in Table D.1. All of the reported system crashes were triggered by a zero-permission third-party app.

Implementation errors in AOSP code are particularly severe due to their ubiquitous nature due to code inheritance. We tested all 27 factory builds for Nexus 5 and discovered two app components that do not properly perform null-checking before operating on data. We discovered that all AOSP 5.1 to 6.0.1 builds contain a vulnerable activity named `com.android.internal.app.IntentForwarderActivity` in the `android` package that crashes the system when the intent contains a null action string. In addition, we discovered that all Nexus 5 AOSP 6.0.1 builds contain a vulnerable broadcast receiver that will crash the system when receiving a broadcast intent with an action of `android.net.conn.CONNECTIVITY_CHANGE_SUPL` and an empty body. These two components are explained in more detail in Section 6. These vulnerabilities have been propagated to the vendors' implementations of Android as shown in Table 1.

Of all the devices we tested, Samsung contained the most exposed interfaces that can be used to make the device encounter a failure state via a system crash. We initially reported that the Samsung Galaxy S6 Edge (AT&T) running Android 6.0.1 with a build number of `MMB29K.G925AUCS5DPK5` contained 18 different vulnerable components. We received 6 Samsung Vulnerabilities and Exposures (SVEs) for vulnerabilities discovered using *Daze*. After another disclosure for the Samsung Galaxy S8+ running Android 7.0 with a build number of `NRD90M.G955USQU1AQD9` containing 7 vulnerable broadcast receivers. Samsung fixed all the vulnerable components in their current Android devices. This is particularly relevant since Samsung Android devices were also disclosing user data during a system crash as discussed in Section 4.4 and also held the greatest global market share of smartphones in Q3 2017 [1]. Interestingly, despite null-checks present in content providers, *Daze* crashed Juning TV running Android 5.1.1 with a null pointer exception in the `HdmiControlService$SettingsObserver` class of the `com.android.server.hdmi` package.

4.3 Privilege Escalation Vulnerabilities

We discovered that various components on the tested devices could be used for privilege escalation in the form of a confused deputy attack [14]. This occurs when a process uses the exposed interface of a privileged process to perform an action on its behalf. The confused deputy attack is well-known on Android and research has been conducted to mitigate its impact [8, 11, 13]. Our findings show that this issue still persists in Android on a range of devices.

The Android OS will export an app component in certain circumstances even if this is not what the developer has intended. Even if an app component does not have the `android:exported` attribute set to true, the OS will *still export the component* if it contains at least one `intent-filter`. An `intent-filter` is used by an app component to register for action(s) that it it expects to receive. An exported component will be accessible to all external apps if the component does not use the `android:permission` attribute in its manifest file. The `android:permission` attribute creates an access requirement that only allows processes with the specified permission to interact with the app component. Android app developers have a tendency to unintentionally export app components, which makes them accessible to third-party apps [10]. Exported components can lead to privilege escalation and local DoS attacks [13, 22]. Table 3 displays our findings showing the device, Android version, and the capability obtained by sending an intent.

Table 3: Discovered privilege escalation vulnerabilities in common Android devices and Android OS versions.

Device	OS	Build ID	Privilege Escalation Action
Alcatel A30	7.0	NRD90M	Take screenshot
Alcatel A30 Plus	7.0	NRD90M	Take screenshot
Amazon Fire TV Stick 2nd Gen.	5.1	LMY47O	Enable/disable Wi-Fi
BLU Grand XL	7.0	NRD90M	Device shutdown
Doogee X5	6.0	MRA58K	Video record screen
Juning TV Box	5.1.1	LMY49F	Take screenshot
Leagoo Z5C	6.0	MRA58K	Factory reset
LG Phoenix 2	6.0	MRA58K	Device shutdown
MXQ TV Box	4.4.2	KOT49H	Factory reset; brick the device
Plum Compass	6.0	MRA58K	Factory reset
Samsung S6 Edge (AT&T)	6.0.1	MMB29K	Initiate firmware update; forget Wi-Fi networks; device shutdown
Ulefone Power 2	7.0	NRD90M	Device shutdown; kill foreground app
Xiaomi Redmi 4	6.0.1	MMB29M	Take screenshot; leak bug report

Of particular concern are devices that can be "factory reset" simply by sending an intent, a capability that is supposed to be reserved for system apps and enabled Mobile Device Management (MDM) apps. A factory reset will wipe all user data. The most severe privilege escalation we noticed occurred in the MXQ TV Box which has an exported broadcast receiver named `SystemRestoreReceiver` that when called will brick the device, making it nonfunctional even after a factory reset. This component modifies the system partition so that the device will not boot properly.

Certain components can cause data leakage when receiving an intent. For example, via a intent with only an action string, the Xiaomi Redmi 4 device will dump the text of active notifications and system log into a bug report on external storage. The Xiaomi Redmi 4 device and three others devices contained an open interface to a privileged process that will take a screenshot and write it to external storage when it receives an intent with a specific action string. Obtaining the contents of the screen is regarded as sensitive and not granted to third-party apps. Using this vulnerability, a malicious app can, for example, send an intent to open a messaging app or an email app, then take a screenshot and dismiss the app by sending an intent requesting the home screen. If needed, the malicious app can cause a system reboot to remove any notifications that a screenshot was taken.

4.4 Data Disclosure Vulnerabilities

A system crash is an exceptional event since a fatal error occurs within a critical Android OS process. Vendors may be interested in recording the cause, so it can be identified and fixed in future releases. Certain vendors record the Android log and write it to a file during or after a system crash. Information such as unique device identifiers, the user's email address, phone number, Global Positioning System (GPS) coordinates, the body of text messages, and sensitive log messages from other apps can be present in the Android log. A system crash can result in information leak of sensitive data if this log file is not adequately protected. Therefore, any app on a vulnerable device can deliberately cause a system crash to obtain and process the log file for sensitive user data.

We discovered that Samsung devices running Android 5.0 to 7.0 create a world-readable file that contains the kernel log and Android log whenever a system crash occurs.[11] Samsung introduced a "special" system process called `bootchecker` to ease the collection of needed debugging information after a system crash. However, `bootchecker` failed at setting the proper file permissions of the file in which it collects the logs, leaving it world-readable to any app on the device. Some Android devices with a MediaTek chipset have a modified `debuggerd` binary and non-AOSP system binaries such as `aee_dumpstate` and `aee_archive`. The `debuggerd` process sets the signal handlers for each process and will obtain debugging information by attaching to the process before it terminates. When a system crash occurs, it will attach to the `system_server` process. The `system_server` process is a critical system process that provides services to apps. On certain devices with a MediaTek chipset, it will then write a world-readable archive file containing the Android log, the kernel log, and various other logs to the `/data/aee_exp` directory or to the `/sdcard/mtklog/aee_exp` directory. The generated archive file is password-protected, but the password was hard-coded in the `debuggerd` binary as `X4rLa8f3`. Examples of vendors that exhibited this behavior are BLU, RCA, Kata, Yuntab, Ulefone, and Figo. The two information disclosure vulnerabilities we discovered and reported have been fixed.

[11] https://nvd.nist.gov/vuln/detail/CVE-2017-7978

4.5 Analysis Time

Figure 2 provides the time taken to analyze each of the 32 devices in hours and provides the average (3.28 hours) over all devices. The Amazon Fire TV Stick 2nd Generation took the least amount of time (15.81 minutes) to analyze due to it having the lowest amount of exported components (310). Of particular note are the Samsung S5 and Samsung S6 Edge devices which took an extended amount of time to analyze due to an aggressive OS policy that repeatedly killed *Daze* for power management purposes.

5 STUDY 2: GOOGLE PLAY APP TESTING

Google Play is the official app distribution channel for the Android platform, facilitating the installation of apps. To determine the prevalence of inadequate exception handling during inter-app communication within Android apps, we tested a representative sample of 18,583 free Android apps from Google Play. These apps were the most popular apps from each app category on Google Play that were downloaded over the time period of March 2016 to April 2017, once every four weeks. The 18,583 apps were comprised of 4,972 unique package names each of which had four different versions

chart created with amCharts | amCharts file:///Users/user/Downloads/amcharts.editor.html

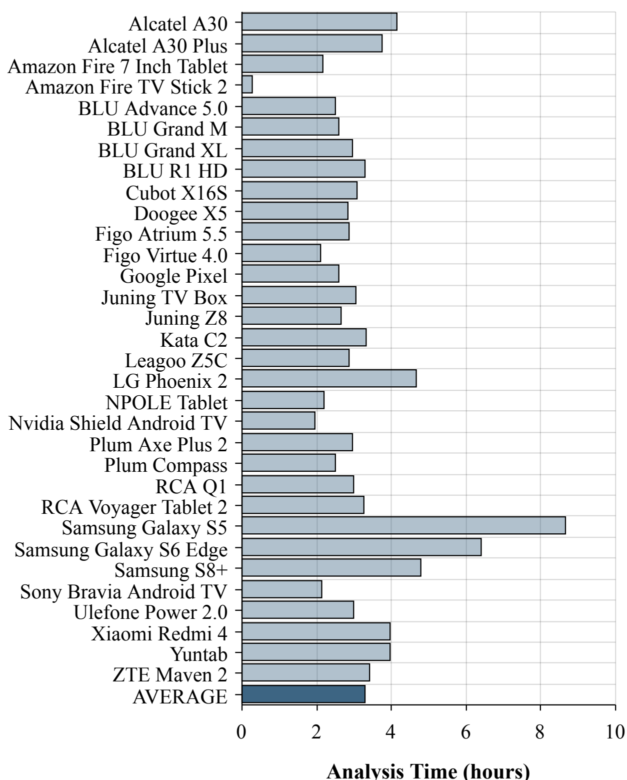


Figure 2: Analysis time in hours of all tested devices.

1 of 1

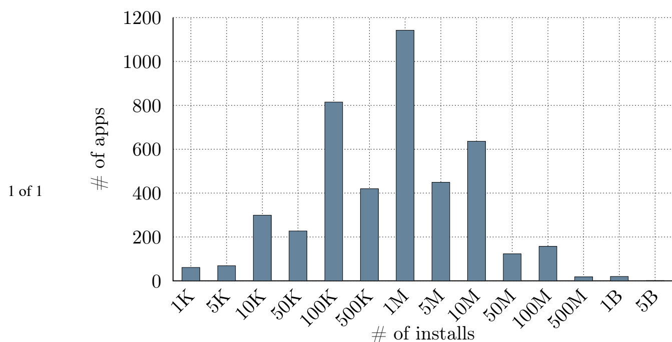


Figure 3: Distribution of apps in our dataset. (Apps with multiple versions were counted only once.)

on average (rounded up). Figure 3 shows the distribution of apps in our dataset grouped by unique package name (apps with multiple versions were counted only once). We also separately tested the top 300 most popular free apps on Google Play to determine the exposure of the apps that have the largest number of users. The top 300 apps were downloaded on November 27, 2017.

Daze tested 18,583 apps and discovered that 34.7% of apps (6,463) in the sample can be crashed externally by a zero-permission Android app co-located on the device. *Daze* found a total of 14,413 fatal exceptions covering 53 types of exceptions. Table D.4 presents

Table 4: Breakdown of the number of vulnerable components for the Google Play study.

Type	#Exported	#Vulnerable	%Vulnerable
Activity	62,328	7,483	12.0
Static Receiver	50,453	4,625	9.2
Dynamic Receiver	16,041	1,449	9.0
Provider	3,749	82	2.2
Service	11,219	774	6.9
TOTAL	143,790	14,413	10.0

a count of all fatal exceptions by type. The most common exception encountered during testing was `NullPointerException` with 10,862 instances, accounting for 75.3% of all fatal exceptions. During testing, *Daze* leaves various intent fields set to null, causing a receiving process to crash if it does not perform proper null-checking. The next most common reason for fatal exceptions (7.5%) was the failure of the developer to implement a particular class (`ClassNotFoundException` and `NoClassDefFoundError`) which causes an uncaught exception when the class loader fails to find the class. This is generally caused by an app component that was registered in the app's manifest not being implemented. The third most common reason is the developer failing to call the appropriate superclass method when executing an app component life-cycle method, resulting in a `SuperNotCalledException`, occurring 650 times (4.5%). The 126 instances (0.8%) of `SecurityException` were due to apps performing permission-protected functionality without the corresponding permission.

Components form the skeleton of an Android app where the developers implement components to perform specific functions. Table 4 presents the aggregate number of crashes by component type and the corresponding ratio of externally-crashable components to the total number of exported components. Activity components are the most numerous and also the most vulnerable (12.0%) to fatal exceptions. Content providers were the least vulnerable (2.2%). The 18,583 apps had a total of 143,790 total exported components with 14,413 apps being vulnerable (10.0%) to having an external app crash the process containing the vulnerable component.

10/25/17, 5:02 PM

To our knowledge, *Daze* is the only system that tests dynamically-registered broadcast receivers. Of the 14,413 fatal exceptions *Daze* identified, 1,449 were due to a dynamically-registered broadcast receivers registered by a component. Certain dynamically-registered broadcast receivers register for actions that can only be sent by the Android OS itself, which *Daze* is not able to send.

5.1 Longitudinal Vulnerability Analysis

We examined all apps that had between three to eight (inclusive) different versions with the same package name — a total of 1,451 package names comprising 5,491 app versions. We determined, in each version of an app, whether identified exceptions were introduced in that version or inherited from the previous version of the app. We consider all exceptions found in the 1st version (lowest version code) of an app to be introduced in this version. (Note that results reported in this section are conservative; see Appendix B).

We considered a recurring exception to be a specific exception introduced in a particular version of an app that propagates to a subsequent version. Within the 5,491 apps subset, *Daze* discovered

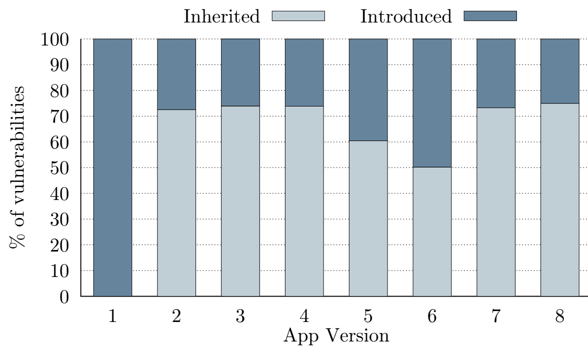


Figure 4: Vulnerabilities evolution as the relative percentage of inherited and introduced exceptions in apps across eight versions.

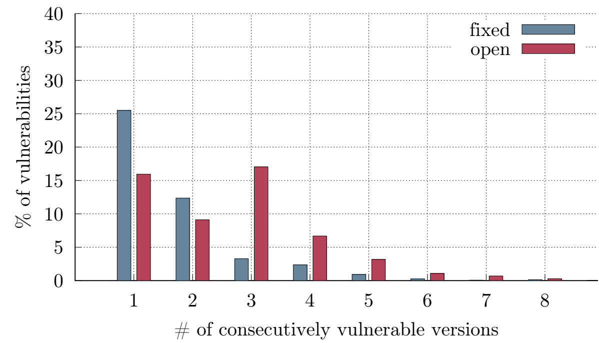


Figure 5: Distribution of the exposure window in terms of the number of consecutively vulnerable versions to fixed and open vulnerabilities (till April 2017).

6,427 fatal exceptions. Of these exceptions 2,706 were unique and the remaining were recurrences of the same exception in different versions of the same app. We found that the majority of the exceptions were inherited from previous app versions instead of being newly introduced as shown in Figure 4. Of all apps with at least 3 versions in our sample, 2,085 apps (37.9%) contained at least one recurring exception and 1,008 apps (18.3%) contained at least one fatal exception that was non-recurring. There were 1,580 apps (28.7%) in the sample that contained the same exception recurring through all versions (covering 419 apps with 1,580 different versions).

We categorized the 2,706 exceptions into those had been fixed (1,241) and those that were still vulnerable (1,465) as of the last app versions available in our sample. Figure 5 illustrates how many consecutive versions a vulnerability persisted through in our dataset. Around 40% of the vulnerabilities were present in a single version and then fixed in the subsequent version of an app. An open vulnerability is a vulnerability that is still present in the last version (most recent) of an app contained in our sample. More than 50% of the open vulnerabilities persisted in at least the latest two app versions in our sample and about 10% of vulnerabilities persisted without patching through at least the latest four versions of the same app. Interestingly, 16% of the vulnerabilities were introduced in the last version of apps in the sample (about 30% of open vulnerabilities).

We also examined the exposure window of vulnerabilities with respect to time. We used AppBrain[12] to determine when an app version was updated on Google Play. AppBrain did not contain data for all versions, so in those cases we relied on the date we downloaded the app. The exposure window starts when a vulnerable app version was uploaded to Google Play. Certain apps that were infrequently updated increased the size of the exposure window since the available app version at the time of downloading may have been uploaded prior to the beginning of the collection period. For fixed vulnerabilities, the time window ends when the vulnerability is first fixed in a subsequent app version contained within our sample. For open vulnerabilities, we conservatively assumed that the vulnerability would be fixed in the version released after the last version in our sample. If the last vulnerable version in our sample was the current version on Google Play, we set the end date of the exposure window to December 10, 2017.

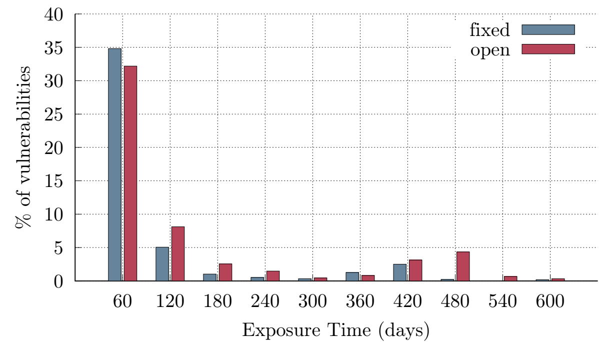


Figure 6: Distribution of the exposure time of fixed and open vulnerabilities (till April 2017).

Figure 6 provides the exposure window of fixed and open vulnerabilities in days. About 35% of the vulnerabilities were fixed with a subsequent update occurring within 60 days, and about 15% of the vulnerabilities persisted for 60 to 600 days before being fixed. More than 30% of vulnerabilities have been unpatched for 60 or more days in our sample, about 30% of these have been unpatched for more than 100 days, and 5% remained unpatched for more than 360 days (about 15% of the open vulnerabilities).

5.2 300 Most Popular Free Apps on Google Play

The top 300 free apps on Google Play are the most widely-used apps available to Android devices with some apps having billions of installations. One might presume that the top 300 apps are more carefully coded to be resilient to inter-app vulnerabilities due to their popularity and user base from which to obtain feedback. *Daze* tested the top 300 free apps and found that 149 of the 300 apps (49.6%) contained at least one vulnerable component that can be crashed externally. There were a total of 310 fatal exceptions in the 300 most popular apps on Google Play. Table D.3 provides all the fatal exceptions in the 300-app sample ranked by their number of occurrences. The top 300 apps have a higher ratio of apps contain at

[12] AppBrain can be accessed at: https://www.appbrain.com

least on vulnerable component than the 18,583 app sample (34.7%). This is likely due to the fact that the top 300 free apps have a higher ratio of exported app components (14.84) than the 18,583 app sample (7.75). The larger attack surface due to a higher average number of exported components in the top 300 free apps, may result in additional chances for developer error. The larger number of components to test also affected the time to test each app for the two samples as discussed in section Section 5.3.

The most popular app (as of November 30, 2017) named *Rules Of Survival* (`com.netease.chiji`, version code 221929) had five fatal exceptions. Two components (`PushServiceReceiver` and `PushService`) encountered a `SecurityException` when creating a shared preferences file with a mode of `MODE_WORLD_READABLE`. This is an Android version compatibility issue since the API call that threw the exception has behavior dependent on the Android version of the device in which it runs. It is also a security issue since the shared preferences file the app tries to create is world-readable and may contain sensitive data. *Google Photos* (`com.google.android.apps.photos`, version code 1992480) contained four fatal `NullPointerException` exceptions. In the top 300 apps, there were 43 instances of `ClassNotFoundException` due to not implementing a particular class. This occurred in 32 apps (10.7% of the sample) with *AdVenture Capitalist* (`com.kongregate.mobile.adventurecapitalist`, version code 2040016345) encountering the `ClassNotFoundException` five times.

5.3 App Processing Overhead

The primary factor influencing the amount of time *Daze* takes to test an app is the number of exported components that it contains. Android apps have a wide variance in regard to complexity and the number of interfaces it exposes externally. Basic apps can contain a single exported activity component, whereas more complex apps can contain hundreds of components. The number of dynamically-registered broadcast components also increases overhead since each will be tested. The average time to test an individual app for the 18,583 app sample was 76.5 seconds, whereas the average time to test an app in the top 300 free app sample was 126.1 seconds. The apps in the top 300 free apps on Google Play had 14.84 exported components per app on average and the 18,583 had an average of 7.75 exported components per app. Figure 7 shows the analysis time for the two samples. The 18,583 app sample contained outliers that were due to apps with numerous components to test and retest. There were over 101 apps that contained 60 or more exported components. Some of the apps in the both samples trigger the system to kill background processes to free up resources, collaterally terminating *Daze* and causing it to incur delays in retesting the component to attribute this behavior to the responsible component. We discuss other fault propagation cases in Appendix A.

6 STUDY 3: STABILITY OF AOSP BUILDS

To determine the robustness of AOSP builds and, by extension, the vendors that modify them, we tested all available AOSP factory images (i.e., builds) for the Nexus 5 and Nexus Player devices.[13] The 27 builds for the Nexus 5 ranged from Android 4.4 (KRT16M)

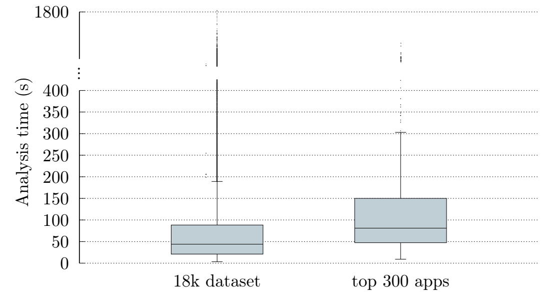


Figure 7: Analysis time statistics of the 18K apps and the top 300 apps datasets.

to Android 6.0.1 (M4B30Z). We installed each firmware available for the Nexus 5, and used *Daze* to determine the exposed interfaces of the core Android package (i.e., `system_server`). According to Google's Android Dashboards, as of November, 2017, 51.7% of all Android devices (e.g., 5.1 to pre-7.0) contain a core component that allows any app co-located on the device to quickly crash the system by sending a single intent message.[14] Although some of these vulnerabilities have been fixed in later Android releases, a majority of current Android devices are vulnerable due to the Android fragmentation problem [21].

6.1 DoS on AOSP Android 5.1 to Pre-7.0

We discovered that the activity `IntentForwarderActivity` in the `com.android.internal.app` package of the `system_server` process can crash the system in all Android versions from 5.1 to versions prior to 7.0 (20 builds in total for the Nexus 5). This occurs since the component blindly operates on the action string from the intent without null checking, causing an uncaught exception by calling the `equals` method on a null string reference.

We discovered that all 13 AOSP 6.0.1 builds for Nexus 5 were vulnerable to system crash when an app broadcasts an intent for the action `android.net.conn.CONNECTIVITY_CHANGE_SUPL` without supplying any embedded data in the intent. The root issue resides in the `GpsLocationProvider` class in `system_server` which dynamically registers a broadcast receiver that assumes every broadcast intent it receives will not contain a null `Bundle` object, resulting in a null pointer exception exception when it tries to handle an intent with no data. The vulnerable broadcast receiver is dynamically registered in the `GpsLocationProvider` constructor and an object of this type will be created in the `LocationManagerService`.

6.2 DoS on AOSP Android TV 5.0 to 8.0

Daze tested all 31 AOSP factory images for the Asus Nexus Player device to determine the prevalence of vulnerable system components within the Android TV device. We discovered that each of the 31 factory images from Android 5.0 (LRX21M) to 8.0 (OPR6.170623.021) had at least one vulnerable broadcast receiver. Our results are presented in Table D.2. The primary cause of the system crashes in Android TV were inadequate input validation and error handling

[13]Factory images for Nexus devices can be downloaded from: https://developers.google.com/android/images#hammerhead

[14]Google Dashboards can be accessed at: https://developer.android.com/about/dashboards/index.html

for Bluetooth and telephony-related intents. The Bluetooth-related intents caused uncaught exceptions in the RemoteControlService class due to inadequate null-checking and the assumption a Bundle will be not be null in the received intent. All the Android TV devices we tested (Sony Bravia XBR-43X830C, Nexus Player, and Nvidia Shield) would crash when certain bluetooth or telephony-related broadcast intents were sent without an embedded Bundle object.

We examined the AOSP code and found that system_server had registered broadcast receivers to listen for specific telephony-related broadcast intents, even though the device does not have telephony capabilities. Generally, the phone app (com.android.phone) uses the protected-broadcast tag in its manifest file so that only the system can send these telephony-related intents. Since the phone app is not installed on Android TV devices starting with Android 6.0, a third-party app can send these broadcast intents and cause system_server, and thus the device, to crash. Android TV devices usually lack safe mode, so a persistent local DoS attack against system_server can result in the user having to perform a factory reset to recover the device if ADB was not enabled prior to the DoS attack, possibly resulting in data loss.

7 A GENERIC ANDROID DOS ATTACK

Certain Android devices will not have exposed system components that allow a single intent to crash the system. We have discovered a novel approach to trigger a controlled boot loop attack on Android by making the system_server process in the Android OS encounter an OutOfMemoryError condition, leading to a system restart. This is accomplished by a zero-permission app repeatedly using a specific API method call where a parameter to the method call will end up being stored on the heap of the system_server process. When a process is started in Android, including system_server, it is allocated a fixed maximum heap size. Once system_server allocates all of its heap memory, it will eventually crash if it cannot free any memory.

Apps can dynamically register a broadcast receiver by providing an object that inherits from BroadcastReceiver and an IntentFilter object that contains one or more action strings. system_server manages all app components in the ActivityManagerService class. When an action string is provided during broadcast receiver registration, it is stored in a variable that can hold an arbitrary amount of data (a HashSet instance variable named mfilters in the IntentResolver class). Therefore, the app can provide large strings to be stored on the heap of the system_server process to exhaust its memory, causing a system crash. The app registers broadcast receivers that have an IntentFilter with a unique action string containing 55,405 characters and a integer value that is incremented to ensure the action is unique. The registration of broadcast receivers is quickly repeated and eventually the system_server process throws an OutOfMemoryError as it tries to allocate more memory while aggressively performing garbage collection.

Most of the OutOfMemoryErrors that repeatedly occur will be caught by the underlying Binder implementation. Binder is part of the Android architecture enabling Inter-Process Communication (IPC) via a kernel module. Once the heap memory of system_server is exhausted, the app can wait for an uncaught error to occur or perform additional action(s) to facilitate an uncaught exception such as starting a large number of activities using the startActivities(Intent[]) API method call. An app can determine the maximum heap size for apps by obtaining value for the dalvik.vm.heapsize system property. system_server will generally have a maximum heap size of either 256 MB or 512 MB. Although current heap size of system_server varies depending on its current workload, the maximum heap size multiplied by a factor of 17.7 generally yields the appropriate number of actions to register to exhaust its heap.

8 COMPARISON TO PRIOR WORK

It has been shown that vendor customization can introduce vulnerabilities via their pre-installed apps [25], hanging attribute references [5], and device driver customization [30]. Previous studies have found that apps and ad libraries tend to be over-privileged, with more unneeded permissions often added to updates without regard to the increased risk these permissions pose [7, 9, 24].

Previous approaches that generated reproducible crash test-cases have relied on two different methods to obtain the statically-registered app components: parsing the apps' manifest directly [12, 22, 27, 29] or relying on the OS package manager [2, 20]. Unfortunately, these approaches inherently suffer from poor inter-app coverage as they overlook dynamically-registered components in apps and in the Android OS. As shown in Section 4, more than 62% of system crash DoS vulnerabilities detected by *Daze* in the 32 devices resided in dynamically-registered components. Sasnauskas et al. [22] compared null-intent fuzzing to data fuzzing of intents and discovered that data fuzzing only yielded a 1% increase in code coverage in their evaluation. Ye et al. [27] focused exclusively on fuzzing the category, data, and action fields of activity components. Yang et al. [26] created a system to detect privilege escalation events. Our approach overcomes key limitations of the other approaches for null-intent testing in regard to completeness (covering all types of app components), automation (e.g., clearing the crash dialogue and restarting the app), and settings and file system monitoring. Maji et al. [20] proposed a semi-manual approach to test exported app activities on three specific versions of Android.

Recently, Garcia et al. [12] created LetterBomb to detect inter-app vulnerabilities. They used complex backward data-flow analysis algorithms to discover DoS vulnerabilities due to missing null-checks when accessing intent fields, then dynamically generated intents to see if the discovered vulnerabilities were indeed exploitable. They tested on 10,000 apps and discovered only 104 exploitable DoS vulnerabilities, taking three minutes per app on average. Similarly, Hay et al. [15] proposed IntentDroid as a full data-fuzzing approach to detect inter-app vulnerabilities. IntentDroid identified 31 intent-related DoS vulnerabilities in 55 apps among the top-rated apps on Google Play in 2014, taking an average of 25 minutes per app. The work in [12] also compared LetterBomb to IntentDroid and found that IntentDroid detected only two thirds of the DoS vulnerabilities detected by LetterBomb in a sample of 40 apps. In contrast, Daze detected 219 vulnerabilities for the oldest versions in our dataset for the same 55 vulnerable apps reported in [15] compared to only 108 detected by IntentDroid (presuming IntentDroid successfully detected all java and native crashes in Table 1 in [15]).

Although we did not measure code coverage during testing (mainly since collecting coverage metrics from stock firmware images and apps requires tampering with their packaged code and re-signing them, which could influence the outcomes of our study), we argue that the main difficulty of inter-app data-fuzzing on Android is that Android enforces type-safety on intent fields by returning a null-like value when a field is accessed using the wrong value type. This type-safe access means that data-fuzzing approaches must use only correct value types in order to achieve any meaningful coverage beyond null-fuzzing. However, the overhead incurred by data-fuzzing prohibits large-scale vetting due to the large input space and complex algorithms involved. *Daze* differs from these solutions in that it approaches the problem directly by null-fuzzing exported components, allowing it to discover significantly more vulnerabilities in significantly less time: 14,413 exploitable vulnerabilities in 18,583 apps, taking two minutes per app on average.

9 CONCLUSION

We presented *Daze*, an automated system to identify fatal exceptions within Android apps and the Android OS. Using *Daze*, we discovered that more than 50% of the current Android devices are vulnerable to a persistent system crash DoS attack, enabled by inadequate exception handling in the Android base code. *Daze* created reproducible test cases for over 20,000 fatal errors within the three datasets spanning 13 months across 59 different versions of AOSP from the smartphone and Android TV distributions. Furthermore, our longitudinal analysis quantified the exposure period of fatal exceptions in consecutive versions of apps. The results showed that the majority of fatal exceptions in an app were inherited from the previous app version and 20% of unpatched vulnerabilities have existed more than 100 days. Moreover, 10% of app components tested were susceptible to attacks causing fatal crashes by an external app. Going beyond DoS attacks, we discovered that system crashes facilitated data disclosure vulnerabilities on certain popular Android devices. Lastly, we presented a novel and universal attack to force any Android device to encounter a system crash by exhausting its heap memory using standard APIs from a zero-permission app.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Zhiyun Qian for their insightful comments that helped improve this paper. We thank Dimitris Tsiounis for providing access to a repository of apps.

REFERENCES

[1] *Gearing Up for a Flagship-Filled Holiday Quarter, Smartphone Shipments Grew 2.7% Year-Over-Year in the Third Quarter, According to IDC.* Retrieved November 7, 2017 from https://www.idc.com/getdoc.jsp?containerId=prUS43193517.

[2] *Intent Fuzzer.* Retrieved June 8, 2017 from https://www.nccgroup.trust/us/about-us/resources/intent-fuzzer/.

[3] *Smartphone OS Market Share, 2017 Q1.* Retrieved October 2, 2017 from https://www.idc.com/promo/smartphone-market-share/os.

[4] *U.S. Consumers Time-Spent on Mobile Crosses five Hours a Day.* Retrieved June 8, 2017 from http://flurrymobile.tumblr.com/post/157921590345/us-consumers-time-spent-on-mobile-crosses-5.

[5] Yousra Aafer, Nan Zhang, Zhongwen Zhang, Xiao Zhang, Kai Chen, XiaoFeng Wang, Xiaoyong Zhou, Wenliang Du, and Michael Grace. 2015. Hare hunting in the wild Android: A study on the threat of hanging attribute references. In *ACM SIGSAC Conference on Computer and Communications Security.* ACM.

[6] Alessandro Armando, Alessio Merlo, Mauro Migliardi, and Luca Verderame. 2012. Would you mind forking this process? A denial of service attack on Android (and some countermeasures). In *IFIP International Information Security Conference.*

[7] Theodore Book, Adam Pridgen, and Dan S Wallach. 2013. Longitudinal analysis of android ad library permissions. *arXiv preprint arXiv:1303.0857* (2013).

[8] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. 2012. Towards Taming Privilege-Escalation Attacks on Android.. In *Network and Distributed System Security Symposium.*

[9] Bogdan Carbunar and Rahul Potharaju. 2015. A longitudinal study of the Google app market. In *Advances in Social Networks Analysis and Mining (ASONAM), 2015 IEEE/ACM International Conference on.* IEEE, 242–249.

[10] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing inter-application communication in Android. In *international conference on mobile systems, applications, and services.* ACM.

[11] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. 2011. Permission Re-Delegation: Attacks and Defenses.. In *USENIX Security Symposium*, Vol. 30.

[12] Joshua Garcia, Mahmoud Hammad, Negar Ghorbani, and Sam Malek. 2017. Automatic Generation of Inter-component Communication Exploits for Android Applications. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017).* ACM, New York, NY, USA, 661–671.

[13] Michael C Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. 2012. Systematic Detection of Capability Leaks in Stock Android Smartphones.. In *Network and Distributed System Security Symposium*, Vol. 14.

[14] Norm Hardy. 1988. The Confused Deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review* 22, 4 (1988).

[15] Roee Hay, Omer Tripp, and Marco Pistoia. 2015. Dynamic detection of inter-application communication vulnerabilities in Android. In *International Symposium on Software Testing and Analysis.* ACM.

[16] Heqing Huang, Sencun Zhu, Kai Chen, and Peng Liu. 2015. From system services freezing to system server shutdown in Android: All you need is a loop in an app. In *ACM SIGSAC Conference on Computer and Communications Security.*

[17] Ryan Johnson, Mohamed Elsabagh, and Angelos Stavrou. 2016. Why Software DoS Is Hard to Fix: Denying Access in Embedded Android Platforms. In *International Conference on Applied Cryptography and Network Security.*

[18] Ryan Johnson, Mohamed Elsabagh, Angelos Stavrou, and Vincent Sritapan. 2015. Targeted DoS on Android: How to disable Android in 10 seconds or less. In *10th International Conference on Malicious and Unwanted Software (MALWARE).*

[19] Anna Lucia Spear King, Alexandre Martins Valença, Adriana Cardoso Silva, Federica Sancassiani, Sergio Machado, and Antonio Egidio Nardi. 2014. âĂIJNomophobiaâĂİ: Impact of cell phone use interfering with symptoms and emotions of individuals with panic disorder compared with a control group. *Clinical Practice & Epidemiology in Mental Health* 10, 1 (2014).

[20] Amiya K Maji, Fahad A Arshad, Saurabh Bagchi, and Jan S Rellermeyer. 2012. An empirical study of the robustness of inter-component communication in Android. In *International Conference on Dependable Systems and Networks (DSN).* IEEE.

[21] Patrick Mutchler, Yeganeh Safaei, Adam Doupé, and John Mitchell. 2016. Target fragmentation in Android apps. In *Security and Privacy Workshops (SPW).* IEEE.

[22] Raimondas Sasnauskas and John Regehr. 2014. Intent fuzzer: Crafting intents of death. In *Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA).* ACM.

[23] Marie-Pierre Tavolacci, G Meyrignac, L Richard, P Dechelotte, and J Ladner. 2015. Problematic use of mobile phone and nomophobia among French college students. *The European Journal of Public Health* 25, suppl 3 (2015).

[24] Vincent F Taylor and Ivan Martinovic. 2017. To Update or Not to Update: Insights From a Two-Year Study of Android App Evolution. In *Asia Conference on Computer and Communications Security (ASIA CCS '17).* ACM, 45–57.

[25] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. 2013. The impact of vendor customizations on Android security. In *ACM SIGSAC Conference on Computer and Communications Security.*

[26] Kun Yang, Jianwei Zhuge, Yongke Wang, Lujue Zhou, and Haixin Duan. 2014. IntentFuzzer: Detecting Capability Leaks of Android Applications. In *Asia Conference on Computer and Communications Security (ASIA CCS '14).* ACM.

[27] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. Droidfuzzer: Fuzzing the Android apps with intent-filter tag. In *International Conference on Advances in Mobile Computing & Multimedia.* ACM.

[28] Caglar Yildirim, Evren Sumuer, Müge Adnan, and Soner Yildirim. 2016. A growing fear: Prevalence of nomophobia among Turkish college students. *Information Development* 32, 5 (2016).

[29] Aimin Zhang, Yi He, and Yong Jiang. CrashFuzzer: Detecting input processing related crash bugs in Android applications. In *Performance Computing and Communications Conference (IPCCC), 2016 IEEE 35th International.*

[30] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. 2014. The peril of fragmentation: Security hazards in Android device driver customizations. In *2014 IEEE Symposium on Security and Privacy (SP).*

APPENDIX

A FAULT PROPAGATION

We have witnessed some cases where testing an app component will result in the analysis app getting killed. If this happens five times, the component is skipped. The skipping of an app component is fairly uncommon, except with content providers that crash during testing. Of the total of the 143,790 tested components in the 18,583, only 105 components needed to be skipped to due the analysis app being terminated. We manually examined the logs to determine the cause of the *Daze* being terminated. This was due to crashing content providers, apps killing background processes, and apps creating the conditions necessary to activate the Low Memory Killer module which terminates processes to free memory. There may be additional faults in the components exported by the `system_server` process that can be found by additional fuzzing of the inputs in intent objects. Our analysis can be extended to provide a more complete analysis using static analysis to determine the key names and value types stored in intents to provide aid with the fuzzing of inputs. *Daze* can be configured to propagate content provider faults into calling apps by exporting a content provider that throws a null pointer exception when queried. Therefore, if the SQLite operation on the client app side is not caught (i.e., not within a `try-catch` block), the client app will itself crash.

B APPS DATASET FIDELITY

Our reported exposure measurements for the 18k dataset might be conservative (under-approximations) in some cases due to potentially missing versions of apps that were updated outside our market sampling interval. Compared to the version updates history on AppBrain.com, we found that 119 apps had two to three missing updates between the last vulnerable version and the fixed version in our dataset. This may lead to under-approximation of the number of consecutively vulnerable versions and the exposure time window if any of these missing versions were still vulnerable. Though we were unable to find download links for these missing versions, these 119 apps comprised about 30% of fixed vulnerabilities that persisted in a single app version in our dataset (total 50$ of all fixed vulnerabilities existed in a single app version; see Figure 5) and if we assume an equal probability that one of the missing versions were still vulnerable, that would drop the percentage of fixed vulnerabilities existing in only one app version from 56% to around 26%, and the difference would be redistributed over vulnerabilities that existed in two and three consecutively vulnerable versions.

C UNTESTED ANDROID PLATFORMS

We have not experimented with custom Android platforms such as Android Auto, Android Wear, and Android Things devices mainly due to the lack of any commercial versions of these devices that could be re-flashed with different versions of the OS. These platforms could very well be as vulnerable as, or even more so than, phones and tables. For instance, while tinkering with a Moto 360 1st generation watch, we found that it has the vulnerable `ForwardIntentToUserOwner` activity that immediately crashed it when called without an action string. Inspecting these custom Android platforms remains an open research area.

D RES

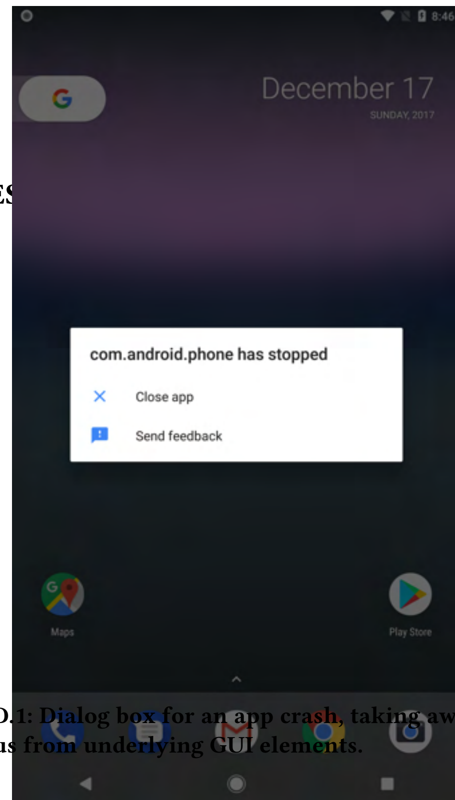


Figure D.1: Dialog box for an app crash, taking away the input focus from underlying GUI elements.

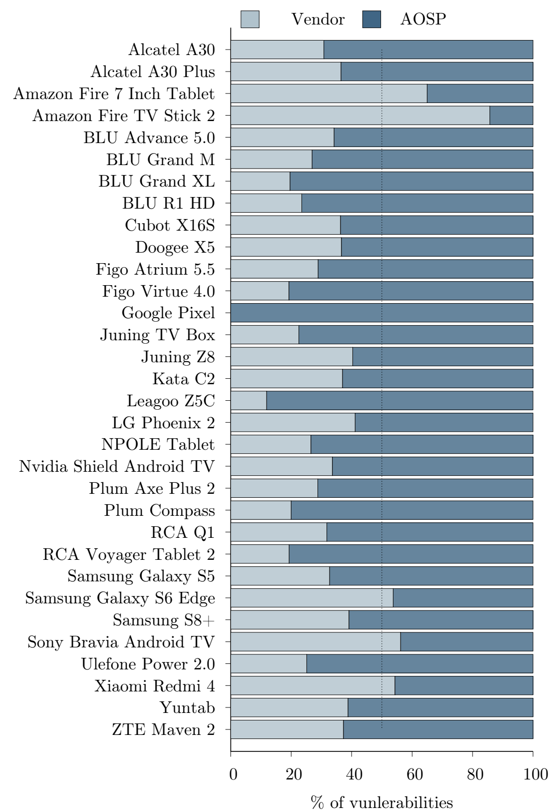


Figure D.2: Attribution of device vulnerabilities to AOSP or vendor customization.

Table D.1: Discovered components by type that will trigger a system crash DoS vulnerability in common Android devices.

Type	System Crash Instances	Cause
Receiver	34	Vendor
Activity	22	AOSP
Receiver	6	AOSP
Service	1	Vendor
Provider	1	Vendor

Table D.2: Broadcast actions causing a system crash for AOSP Android TV.

Broadcast Action	Vulnerable TV Version			
	5.x	6.x	7.0	8.0
android.bluetooth.input.profile.action.HANDSHAKE	•	•		
android.bluetooth.input.profile.action.REPORT	•	•		
SIM_STATE_CHANGED†		•		
EMERGENCY_CALLBACK_MODE_CHANGED†			•	•

† Actions in the default `android.intent.action` scope.

Table D.3: Exceptions in the top 300 Google Play apps.

Exception Name	Freq.
java.lang.NullPointerException	237
java.lang.ClassNotFoundException	44
java.lang.IllegalArgumentException	5
java.lang.ClassCastException	5
java.lang.IllegalStateException	4
java.lang.RuntimeException	4
android.util.SuperNotCalledException	3
java.lang.SecurityException	2
org.xmlpull.v1.XmlPullParserException	1
Signal 11 (SIGSEGV)	1
java.lang.UnsatisfiedLinkError	1
java.io.FileNotFoundException	1
android.view.InflateException	1
android.os.FileUriExposedException	1
TOTAL	310

Table D.4: Exceptions in the 18K apps dataset.

Exception Name	Freq.
java.lang.NullPointerException	10,862
android.util.SuperNotCalledException	650
java.lang.ClassNotFoundException	583
java.lang.NoClassDefFoundError	492
java.lang.IllegalArgumentException	322
java.lang.RuntimeException	252
java.lang.IllegalStateException	211
java.lang.IndexOutOfBoundsException	161
java.lang.UnsatisfiedLinkError	140
java.lang.SecurityException	126
java.lang.ClassCastException	83
content provider crash	82
android.content.res.Resources$NotFoundException	63
java.lang.NumberFormatException	52
java.lang.InternalError	46
java.lang.UnsupportedOperationException	42
android.content.ActivityNotFoundException	35
android.view.WindowManager$BadTokenException	29
java.lang.ArrayIndexOutOfBoundsException	24
android.database.CursorIndexOutOfBoundsException	24
android.database.sqlite.SQLiteException	16
java.lang.InstantiationException	15
java.lang.NoSuchFieldError	14
java.lang.StringIndexOutOfBoundsException	12
android.util.AndroidRuntimeException	10
java.lang.AssertionError	9
android.content.ReceiverCallNotAllowedException	8
java.security.InvalidParameterException	7
android.view.InflateException	5
java.lang.NoSuchMethodError	4
java.lang.IllegalAccessException	4
java.util.MissingFormatArgumentException	3
java.io.FileNotFoundException	3
android.view.ViewRootImpl$CalledFromWrongThreadException	3
signal 6 (SIGABRT)	2
java.lang.AbstractMethodError	2
android.runtime.JavaProxyThrowable	2
java.lang.ExceptionInInitializerError	2
signal 11 (SIGSEGV)	1
org.json.JSONException	1
java.lang.VerifyError	1
java.lang.NoSuchFieldException	1
java.lang.IncompatibleClassChangeError	1
java.lang.IllegalAccessError	1
java.lang.Exception	1
java.lang.ArithmeticException	1
android.support.v4.app.SuperNotCalledException	1
android.database.sqlite.SQLiteCantOpenDatabaseException	1
android.content.pm.PackageManager$NameNotFoundException	1
android.app.RemoteServiceException	1
android.app.Fragment$InstantiationException	1
TOTAL	14,413

Table D.5: Exceptions detected by *Daze* for full device testing.

Device	java.lang.NullPointerException	java.lang.NoClassDefFoundError	java.lang.IllegalArgumentException	java.lang.RuntimeException	java.lang.SecurityException	java.lang.ClassNotFoundException	java.lang.IllegalStateException	android.util.SuperNotCalledException	android.content.ActivityNotFoundException	java.lang.ClassCastException	java.lang.UnsatisfiedLinkError	java.lang.AssertionError	java.lang.InstantiationException	java.lang.ArrayIndexOutOfBoundsException	java.lang.IndexOutOfBoundsException	android.content.res.Resources$NotFoundException	java.lang.NoSuchMethodException	java.lang.UnsupportedOperationException	java.lang.NumberFormatException	java.lang.NoSuchFieldError	android.database.sqlite.SQLiteException	java.lang.IllegalThreadStateException	java.lang.StringIndexOutOfBoundsException	java.lang.IllegalAccessException	android.view.WindowManager$BadTokenException	junit.framework.AssertionFailedError	com.amazon.assertion.AssertionException	com.facebook.FacebookSdkNotInitializedException	unavailable	signal 6 (SIGABRT)	signal 11 (SIGSEGV)
Alcatel A30	131		6	3	12	28	6	13	3	1			5	1	1	1															
Alcatel A30 Plus	125		8	4	12	29	5	13	4	1	2		3	1	1	2				1											
Amazon Fire 7 Inch Tablet	61	3	16	4	1		9		1						1		1		1								1		1		
Amazon Fire TV Stick 2nd Gen	17				1			1																		1			1		
BLU Advance 5.0	74	6	13	4	3		3	1	2	1	3			1	1	2	2												1		
BLU Grand M	98	5	7	4	3				3	1	3		2	2		1													1		
BLU Grand XL	97		8	3	13	21	3	13	3		1		2	3		1															
BLU R1 HD	114	11	10	6	14		4		3	2	2	10	2	4															2		
Cubot X16S	117	16	16	4	4		7	2	3		3		2	1	2			1											1		
Doogee X5	105	17	13	4	4		5		3		3		2	2	1			1											1		
Figo Atrium 5.5	76	15	11	5	4		4		3	1	1			3		2	2												2		
Figo Virtue 4.0	121	8	13	5	5		4		1		3		2	1		1		1											1		
Google Pixel	82		6	5	9	18	4	6	2	1	1		1					2													
Juning Z8	113	18	13	4	5		4	1	3	1	3			1	1	1	2												1		
Juning TV Box	56	8	13	7	5		4	1	2	1							2			1									3		
Kata C2	138	20	14	4	4		5		4	1	3		2	1	1			2											1		
Leagoo Z5C	61	2	5	1	10		4		2	3		10			1			1											1		
LG Phoenix 2	181	29	11	6	5	5	5		5	14						2		3	1		2			1					1	1	
NPOLE Tablet	36	2	3	2	1		1										2											1	1		
Nvidia Shield Android TV	74		3	1	3	19	7	1	2		1											1							1		
Plum Axe Plus 2	102	26	9	3	11		4		3	1	3	10	2	1	1														1		
Plum Compass	75	1	3	2	11		3		3	3		1				1		1											1		
RCA Q1	118	18	13	4	4		4		3		3		2	1	1			1											1		
RCA Voyager Tablet 2	89	16	17	3	5		8			2	3			1			2		1										1		
Samsung Galaxy S5	118	36	11	3	14		5	1	1	1	1	8			1	1	1												15		
Samsung Galaxy S6 Edge	157	6	3	5	3	2	15				1			1	1			1											18		1
Samsung S8+	74		3	4	7	17	2		3	3									1				1								
Sony Bravia Android TV	115	9	5		2		9	3	3		1	3										1							2		
Ulefone Power 2.0	97		10	2	14	19	4	13	3	2	2		2	3																	
Xiaomi Redmi 4	119	22	6	3	12		7	2	3	3		1			3				1										2		
Yuntab	5		18	157	5	20	14	3		4			3	1	1	2	2								1						
ZTE Maven 2	168	10	12	8	4		8	1	2				2	1		1													3		
TOTAL	3,114	304	299	270	210	178	167	75	73	47	43	43	34	30	18	18	16	14	5	2	2	2	1	1	1	1	1	1	64	1	1