Mobile Konami Codes: Analysis of Android Malware Services Utilizing Sensor And Resource-Based State Changes

Boomgaarden, J.

IEEE Computer Society
Mobile Konami Codes: Analysis of Android Malware Services Utilizing Sensor And Resource-Based State Changes

J. Boomgaarden, J. Corney, H. Whittaker, G. Dinolt, and J. McEachen
Departments of Computer Science and Electrical and Computer Engineering
Naval Postgraduate School
Monterey, California, United States of America

Abstract
Challenges in static analysis of mobile malware have stimulated the need for emulated, dynamic analysis techniques. Unfortunately, emulating mobile devices is nontrivial because of the different types of hardware features onboard (e.g., sensors) and the manner in which users interact with their devices as compared to traditional computing platforms. To test this, our research focuses on the enumeration and comparison of static attributes and dynamic event values from sensors and resources within Android runtime environments on physical devices and within several online services’ analysis environments. Utilizing the results from enumeration, we develop two different Android applications that are successful in detecting and evading the emulated environments utilized by those mobile analysis services during execution. When ran on physical devices, the same applications successfully perform a pseudo-malware action and send device identifying information to our server.

1. Introduction
Mobile malware authors are becoming more proficient in the traditional ways of evading detection and are developing new ways of defeating the security roadblocks they encounter as well. Part of their success in doing so comes from the inherent characteristics of mobile devices themselves. Challenges in static analysis of malware have stimulated the need for emulated, dynamic analysis techniques. Unfortunately, emulating mobile devices is nontrivial because of the different types of hardware features onboard (e.g., sensors) and the manner in which users interact with their devices as compared to traditional computing platforms.

1.1. Motivations and goals
Modern mobile devices now come outfitted with a variety of hardware and sensors that users can tap into for sending and receiving input and monitor for state changes. Mobile applications are becoming so dependent on these additional sensors and hardware features that we have come to expect their availability—so much so it has created new paradigms in programming (e.g., context-aware programming) [1].

Unfortunately, context-aware programming and the application programming interfaces (APIs) that enable such functionality also allow for malicious, context-aware attacks on mobile devices. Most can envision how a proximity or location-based attack from a malicious application would be triggered; however, it is just as feasible that the trigger could come from an audio cue, the snapping of a photo, or a change in acceleration indicating the device’s owner is traveling by vehicle vice walking. Equally possible is sensor functionality being utilized for evasion of malware analysis. Our work highlights the challenges faced in conducting dynamic malware analysis—commonly referred to as runtime or sandboxed analysis—by exploring how sensor and other resource-based state changes can be leveraged to evade dynamic analysis tools and services through detection.

Our research efforts were specifically designed with the following questions in mind:
1. Can dynamic analysis tools and services be enumerated (i.e., fingerprinted) through sensors and dynamic resources they simulate, how they simulate them, and which ones they are unable to simulate?
2. In what ways can malicious behavior on mobile devices be triggered by sequences of sensor-based or resource-based state changes (i.e., a mobile Konami code to trigger unwanted behavior)?
3. Can these types of triggering techniques be used to defeat common dynamic analysis tools and services for detecting malware?
1.2. Scope of research

Our research focuses on the development and testing of multiple Android proof-of-concept applications for enumeration and evasion. Although this work is applicable to other mobile platforms, our focus remained on Android rather than being all-inclusive of others such as iOS, Blackberry 10, and Windows Phone.

The proof-of-concept applications were tested against multiple physical devices, as well as multiple mobile malware analysis services, which we became familiar with during an initial literature review. We did not attempt to exploit any known (or unknown) vulnerabilities within the Android platform during our testing, nor within the analysis services that we tested against.

2. Android application component overview

In order to understand the difference between the attack surfaces on traditional software applications and Android applications, one needs to understand the basic components that comprise an Android app. Activities, Services, Broadcast Receivers, and Content Providers all offer unique functionality that malware developers have learned to repurpose for malicious means. Additional details regarding these components can be found on the Android Developers website.

Activities can be thought of as a container for the user interface of a single page or screen of the app. Activities typically are tied to layouts and house the buttons, widgets, content, and images seen by a user during the Activity’s life cycle.

Services are components that do not have a user interface element, but instead are utilized for performing long-running or task-driven operations in the background [2]. An example of this is a music playing application that defines a Service to continue playing music even when a playing application that defines a Service to continue playing application is given over to something different such as using a browser for navigating a website [2].

Broadcast Receivers are components that help implement the event-driven programming paradigms on Android. An application’s Broadcast Receiver takes a defined action after it has received a specific Intent object (i.e., an event) broadcast by the system (such as the device completing its boot sequence) or by a user-defined broadcast.

Content Providers allow for a structured interface to the data that an application has created and stored in text files, SQLite databases, or any other persistent storage solutions. Utilizing this interface, other apps can query and potentially modify an application’s data based on the behaviors and permissions defined by the provider. A common example of this is the default Contacts application. By providing a structured interface to the contact data stored by this app, other applications (e.g., an email app) can easily query names and addresses for use.

Intent objects are worth briefly mentioning for their role as the primary means for activating and passing data between all application components, with the exception of Content Providers. The Android Developers website specifies: “Although Intents facilitate communication between components in several ways, there are three fundamental use-cases: (1) to start an Activity, (2) to start a Service, and (3) to deliver a Broadcast.” [3].

2.1. Application security and permissions

As described and detailed on the Android Developer website, applications are sandboxed from one another and are given access to only a narrow array of system resources. These accesses and the restriction of access to certain resources are implemented through multiple mechanisms. One of the mechanisms provided by the Android APIs is intentional lack of API support for certain sensitive functionality, such as directly manipulating the SIM card. Another mechanism is to place restrictions on access to sensitive resources through Android application permissions [4].

For instance, access to a device’s camera is protected using this permission mechanism. The same goes for location data (e.g., GPS), wireless network connections, SMS sending and reading, making phone calls, and enabling Bluetooth functionality. To use the protected APIs for accessing these sensitive resources or hardware features, an application (prior to Android API Level 23) must define and request the permissions it requires in its AndroidManifest.xml file, which is a control and configuration file required by all Android applications [5].

3. Android malware analysis services

As the number of Android devices increased in availability and popularity, so too did the number of new applications being published in the Google Play Store. In December 2014, it was estimated over 23,000 new applications were introduced into the Google Play Store [6]. The large number of new submissions each month requires a scalable solution to analyze applications for malicious or benign behavior. Google recognized this early on and publicly announced the use of Google Bouncer, an automated service capable
of scanning existing and newly submitted applications, in February 2012 [7]. Prior to Google Bouncer however, there were efforts both in commercial and academic realms to create similar sandboxing analysis services. AASandbox was likely one of the first to do so [8], and Neuner et al. recently presented an excellent comparison of several of those tools and services [9].

All of the analysis services we investigated provided some combination of static and dynamic analysis techniques. Additionally, the services all provide some sort of downloadable or web-accessible report based on their analysis of an Android APK submission. These reports highlight various static or dynamic findings. Some services provide additional details such as network packet captures of traffic generated by the submitted APKs, or verbose android log files based on the Android logcat tool. Each service provides search functionality across their reports, typically by MD5 hash signatures.

Below is a listing of the Android malware services we utilized. Some services were not available (e.g. site unreachable/offline) during various phases of our testing, and Joe Sandbox specifically blacklisted our efforts at a certain point after informing us their Terms of Use prohibits detecting and/or manipulating runtime environment data.

- Android Sandbox
- APKScan
- Joe Sandbox
- SandDroid
- VisualThreat
- Andrubis
- CopperDroid
- Mobile-Sandbox
- TraceDroid

3.1. Static analysis techniques

Static analysis techniques performed by the services we investigated typically involved unpacking the submitted APK file and parsing information from its AndroidManifest.xml file. Information provided from this action typically includes Android permissions specified for the application and which application components were declared. For some services, this listing of application components was used to guide later dynamic analysis efforts in stimulating specific behaviors.

As part of the static analysis, most services also attempted to convert DEX files back to the Java.class files from which they were derived. Open source tools such as dex2jar were likely used for this process.

The majority of the services we investigated also performed some hash-based signature comparisons against known malware in repositories such as VirusTotal.

3.1. Dynamic analysis techniques

In the services we investigated, dynamic analysis was performed in a variety of environments and at differing inspection levels. The inspection levels included analysis done fully outside the Android OS (i.e., at the hypervisor level), strictly within the Dalvik virtual machine running the application, or some blend of both. Most service frameworks were likely setup on QEMU hypervisors based on our findings of various services utilizing default Android SDK Android Virtual Devices (AVD) as foundations for their runtime environments. At least one service indicated physical devices were sometimes used in a distributed analysis approach.

Data taint propagation (commonly referred to as taint tracking) was used across several services to detect potential sensitive data leakage of items such as phone contacts and unique device numbers. TaintDroid is a popular tool for this and was identified as being used in several of the services we investigated.

Event stimulation—such as changing the emulator’s GPS location, performing touchscreen presses, or modifying battery levels—was also used as a dynamic analysis technique in several services. Tools such as monkey and MonkeyRunner, which come as part of the Android SDK, were identified in some services as the primary way in which event stimulation was achieved. Other dynamic analysis techniques included method tracing, code coverage comparison, monitoring of system level calls, phone call and SMS event monitoring, network traffic captures, and file access logging.

4. Environment enumeration

Our tests focused on exploring how Android malware analysis services attempt to emulate not only physical hardware features (e.g., location data from a GPS receiver), but dynamic resources on a phone as well (e.g., contacts, call logs, SMS, etc.). By enumerating and observing what data values and attributes are provided (or not provided), we hoped to discern notable differences between analysis services and physical devices that would aid in our detection and evasion techniques.

4.1. Methodology

Each Android application we developed attempts to capture basic data values and attributes of a sensor, hardware feature, or other dynamic resource. These data values and attributes are directly accessible through proper Android API method calls and, when
required, proper Android permissions declared in the AndroidManifest.xml configuration file. We did not attempt to circumvent or exploit any Android platform feature for purposes of these tests.

To account for differing API levels supported by the four physical devices and nine analysis services tested, we were required to support Android API levels 8 through level 21.

In order to capture results of the tests being performed by each application, we setup an Amazon EC2 instance running a basic Apache web server with PHP. Each application included method calls to perform HTTP POST requests to our EC2 instance, with all results being stored in name-value pairs. Simple PHP scripts on the server parsed out POST parameters and wrote them to individual log files for later retrieval via SFTP.

Each proof-of-concept application had a unique APK for physical devices we tested and for the analysis services. The only difference across the versions for an individual proof-of-concept application was the URL chosen for submitting the data values and attributes via HTTP POST parameters. This also provided us with separate MD5 hash values for each application built per malware service.

In addition to sending all results to our EC2 instance via HTTP for logging, results are displayed on the Android device or emulator screen using a TextView object. Several of the services provided screen captures of submitted applications during runtime as part of their resulting report, so this served as a secondary measure for capturing data generated from each service.

During runtime of each Android application, we also captured various static heuristics about the device or emulator performing code execution. The Android android.os.Build API provides various public member fields such as Build.DEVICE, Build.BRAND, Build.MODEL, Build.PRODUCT, Build.MANUFACTURER, Build.BOARD, and Build.VERSION.SDK_INT, which all give some insight as to the type of hardware and operating system version (i.e., Android API level) utilized for executing an application. This design allowed us to observe several instances where an application we submitted to a single analysis service was executed by at least two separate environments with differing features (e.g., different API levels).

In total, we created 15 applications to perform environment enumeration: SensorList, LocationGPS, LocationNetwork, Accelerometer, Magnetic Field, Orientation, Temperature, Proximity, Battery, Bluetooth, Audio, PhoneState, SMS, CallHistory, and Contacts.

For purposes of this publication’s conciseness we have only detailed three of them. Full details and results can be found in our original thesis work\(^1\).

4.2. The SensorList application

As stated on the Android Open Source Project (AOSP) website: “Android sensors give applications access to a mobile device’s underlying physical sensors. They are data-providing virtual devices defined by the implementation of sensors.h, the sensor Hardware Abstraction Layer (HAL).” [10]

Data provided by these types of virtual sensors comes from corresponding physical hardware features onboard systems running Android such as accelerometers, gyroscopes, magnetometers, barometer, humidity sensors, pressure sensors, light sensors, and proximity sensors.

Each Android sensor has an associated sensor type, which is accessed via the Sensor.getType method call. Official Android sensor types are defined in sensors.h (currently 25 official types) and are documented within the Android SDK APIs, describing what data attributes and methods can be called utilizing that sensor [10]. Manufacturers implementing Android on their devices may utilize hardware not defined by Android open-source specifications and therefore assign new temporary types to support corresponding sensors [10]. Our testing of the SensorList application on physical devices also gave evidence to this.

Our results provided several insights based upon what sensors were listed as being implemented by each individual analysis service.

First, none of the services implemented any manufacturer-specific sensor types (i.e., unofficial sensor types with numbers above 25). Each physical device tested had at least one manufacturer-specific sensor type, with the LG Nexus 5 indicating eight different ones.

Second, five of the nine services provided the exact same five sensor types: ORIENTATION, ACCELEROMETER, GEOMAGNETIC_FIELD, TEMPERATURE, and PROXIMITY. The other services provided one or zero sensor types as implemented. The physical devices all gave results indicating at least 17 different sensor types. Unsurprisingly, the aforementioned five sensor types are exactly the same five sensor types provided by an Android SDK default virtual device (AVD), which we confirmed by running SensorList in a default AVD. This suggests the runtime environments for these

\(^1\) Thesis work: [https://calhoun.nps.edu/handle/10945/45163](https://calhoun.nps.edu/handle/10945/45163)
services are likely built upon a default-provided AVD as a starting foundation.

Third, none of the physical devices we tested provided the TEMPERATURE sensor type (it has been deprecated for some time now); however, five of the analysis services did.

Lastly, the sensor names associated with each of the sensors provided by the malware analysis services all began with the string Goldfish. For instance, the VisualThreat sensor name for its accelerometer is Goldfish 3-axis Accelerometer, whereas the Samsung Galaxy S4 sensor name for its accelerometer is K330 Acceleration Sensor. This Goldfish string is likely an artifact left over by the analysis services building upon an Android SDK-provided Android Virtual Device (AVD) as a starting foundation for their runtime environments.

4.3. The LocationGPS and LocationNetwork applications

The Android platform provides developers two separate methodologies (i.e., framework APIs) for accessing and utilizing location data (e.g., GPS, cell tower, or Wi-Fi location). Their recommended method for utilizing location services is via the recently published Google Location Services API, part of Google Play Services [11]. This method requires developers to include the additional Google Play Services SDK while building their applications. Additionally, devices (or emulators) that run location-aware apps using Google Location Services API require Google Play Services to be installed on the device. For two of the more common Android emulators (the default Android SDK emulator and GenyMotion), this presents problems as Google Play Services is not typically installed by default, nor can it be easily installed based on our testing. The second (and more traditional way) of accessing and utilizing location data is via the android.location API. This is the method we chose for developing our LocationGPS and LocationNetwork applications.

At a high level, the android.location API has three primary classes for dealing with location attributes and information: Location, LocationManager, and LocationProvider. All locations generated by a LocationManager are guaranteed to have a valid latitude, longitude, and timestamp; other parameters like accuracy or altitude are optional for a given location generated [12]. In addition to the higher-level API classes for utilizing location data, our LocationGPS application makes use of the GPSSatellite API for accessing attributes of the satellite data received when a GPS location fix is obtained.

The first noticeable dynamic heuristic was none of the analysis services provided more than one update to location data when location providers were enabled. During executions of the LocationGPS app on all four physical devices, multiple new location updates were provided usually within a short period of time (i.e., under a minute).

Although it is feasible the GPS receiver on a device could be turned off or could lose signal immediately after one update, we feel confident that the testing for multiple location updates in a short period of time would allow detection of an emulated runtime environment, specifically when a GPS location provider is initially enabled. The same assertion is harder to make when locations are generated from a Network provider. Location updates may not happen if a device’s location is associated with a Wi-Fi access point and never leaves the access point’s proximity. Our testing of the LocationNetwork app shows this when all four physical devices never receive multiple updates.

The next noticeable heuristic showed all locations generated by GPS providers in the analysis environments lacked any satellite data attributes. In fact, when an analysis environment provided a GPS location, the number of satellites used in the location fix was zero. In contrast, all four physical devices we tested provided all satellite data attributes and the location fixes for each device ranged between six and twelve satellites being used.

Another noteworthy heuristic was how believable the locations generated by GPS or Network providers were. Of the three GPS locations obtained, one service gave coordinates in the ocean off the coast of Somalia (Mobile-Sandbox), another gave coordinates in the middle of the Arabian Sea (SandDroid), and a third service gave coordinates in Antarctica (CopperDroid), as shown in Figure 2. Interestingly, ANDRUBIS and TraceDroid both provided the exact same location data (using a Network provider). The latitude and longitude for this location placed it near an urban location of China. Also noteworthy was how much accuracy was given in the latitude and longitude values. Most values went out to 14 or 15 decimal places; however, SandDroid’s locational data gave latitude of 14.0N and longitude of 64.0E.
Table 1. LocationGPS application results

<table>
<thead>
<tr>
<th>Attribute Tested or Implemented</th>
<th>Galaxy S4</th>
<th>Galaxy S5</th>
<th>Galaxy Note 4</th>
<th>Nexus 5</th>
<th>Android Sandbox</th>
<th>Andrubis</th>
<th>APKScan</th>
<th>CopperDroid</th>
<th>Joe Sandbox</th>
<th>Mobile-Sandbox</th>
<th>SandDroid</th>
<th>TraceDroid</th>
<th>VisualThreat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is GPS Provider enabled?</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Location provided?</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Multiple updates provided?</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Location accuracy (m)</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Location altitude (m)</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Location bearing (degrees)</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Location latitude (degrees)</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Location longitude (degrees)</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Location speed (m/s)</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Location time (POSIX time)</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Number of Satellites</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Satellite azimuth (degrees)</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Satellite elevation (degrees)</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Satellite signal-to-noise ratio</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

The last dynamic heuristic we observed was the value provided by the `Location.getTime` method for each location obtained. Both ANDRUBIS and TraceDroid provided a value of 1342232104000, which equates to 14 July 2012 02:15:04 UTC. CopperDroid provided a value of 1343174400000, which equates to 25 July 2012 00:00:00 UTC. SandDroid provided a value of 1417449600000, which equates to 01 December 2014 16:00:00 UTC. Only, Mobile-Sandbox’s value was reasonably accurate at 1423036800000, which equates to 04 February 2015 08:00:00 GMT.

Although we submitted APK files to each of the nine analysis environments at least twice, further submissions would likely provide more accurate results. For instance, initial submissions for CopperDroid did not result in any location information, while later submissions did. This was similar across several services and likely indicates different runtime environments being used for scaled analysis, including physical devices vice emulated environments.

4.4. The Accelerometer Sensor application

The `android.hardware.SensorManager` and `android.hardware.Sensor` APIs provide access to all sensor objects and their generic attributes such as the sensor’s vendor name and version number. The `android.hardware.Sensor.Event` and `android.hardware.SensorEventListener` APIs provide access for capturing new sensor events (i.e., changes in sensor values) and the corresponding data for the sensor that caused an event to trigger.

The most significant dynamic heuristic discovered during our testing was that each attribute showed the same values across all analysis services that provided sensor event updates. Specifically, the values provided for acceleration force were always 0.0 m/s², 9.77622 m/s², and 0.813417 m/s² along the x, y, and z-axes, respectively.
Another significant dynamic heuristic discovered during our testing was the vendor name associated with each analysis service’s implemented accelerometer sensor. The four physical devices tested had vendor names of InvenSense (Samsung Galaxy S5, Samsung Galaxy Note 4, LG Nexus 5) and STMicroelectronics (Samsung Galaxy S4). All analysis services provided a vendor name of The Android Open Source Project.

Other noteworthy dynamic heuristics discovered were associated with sensor power values and sensor maximum range values. The power consumption indicated by the accelerometer sensor on each of the physical devices was less than or equal to 0.4 milliamperes (mA), while each of the analysis services indicated a power consumption of 3.0—roughly 7.5 to 12 times the value of the physical devices. The sensor maximum range value for each of the analysis services was 2.8 m/s²—a value significantly lower than the physical devices’ maximum sensor range (with 19.6 m/s² being the lowest of those). More telling, however, is that the maximum value of 2.8 m/s² given by the analysis services suggests incorrect implementations.

Values given along the y-axis at 9.77622 m/s² are above this maximum value (and were likely implemented so as to be close to the force of gravity, thereby suggesting a device at rest).

Interestingly, the Android Sandbox analysis service gave sensor attribute data but did not provide any actual sensor updates based on accelerometer value changes. Additionally, CopperDroid was thought to implement an accelerometer based on our findings after executing the SensorList application; however, no device or sensor data was logged to our EC2 server. It is possible in both cases the analysis services executed our application, generated accelerometer data, and for some reason did not send the data across their networks or that the data was interrupted in transit. Network PCAP data provided by each of the analysis services’ reports gave no further insight.

5. Android emulator evasion

Both Android applications we developed for evasion testing first attempt to capture basic values and attributes of sensors, hardware features, or other dynamic resources. These values and attributes are directly accessible through proper Android API method calls and, when required, proper Android permissions declared in the AndroidManifest.xml configuration file.

Both applications attempt to determine if they are executing on a physical device, and, if so, retrieve the device’s unique ID (i.e., its IMEI, ESN, or MEID value) and send that value to our EC2 server. If the application determines it is executing in an emulated environment, it will not send the device’s unique ID. Of
note, sending the device’s unique ID is a common trend associated with malware and aggressive spyware, as described in [13] and evidenced in a profile of the NickiSpy.A malware instance detected in 2011 [14].

5.1. Konami Code application (static version)

The first version of the Konami Code application we developed focused on testing runtime environments against known static attribute values based on our enumeration findings. For example, in testing the accelerometer sensor, we compared attribute values such as its max range, power, and vendor name. This is performed by a method called from the application’s `onCreate` method (i.e., its initial entry point). We chose nine different tests to perform involving various values from the `SensorList`, `Accelerometer`, `Geomagnetic Field`, `Orientation`, `Proximity`, `Battery`, `Bluetooth`, `Audio`, and `PhoneState` applications. Each test is called sequentially from `onCreate`, and if any one of them fails, the application will not send the device’s unique identification number to our EC2 server.

5.2. Konami Code application (dynamic version)

The second version of the Konami Code application we developed focused on testing runtime environments against known dynamic attributes and their behavior over multiple events. For example, in testing the accelerometer sensor, we compared whether given force values along each of the three axes ever changed over the range of ten accelerometer sensor events. We chose three different sensor types for testing (`Accelerometer`, `Geomagnetic Field`, and `Proximity`). Unlike the static version of the Konami Code application, we could not perform tests in a sequential manner because we had no control over which sensor event occurred at any given time. Instead, any time a sensor event was detected, we determined what sensor type caused it and then looked to see if the values given by this event differed from the previous event values for that sensor type. If they differed, the assumption became that the runtime environment was a physical device. If they remained the same across ten distinct sensor events of a given sensor type, we stopped testing that sensor and assumed the runtime environment was an emulated environment.

6. Final results and concluding thoughts

Our research had the initial goal of enumerating physical Android devices and multiple Android malware analysis services through their implementation of sensors and other dynamic resources. To accomplish this goal, we developed 15 different Android applications that captured various sensor and resource attributes, in addition to dynamic values associated with sensor and resource events.

Utilizing the results from these 15 applications, we explored our next goal of creating triggering techniques to perform pseudo-malware actions, or to detect and evade emulated Android runtime environments. To do this, we created two separate applications based on (1) comparing static attributes of sensors and resources with known emulated values, and (2) monitoring the dynamic behaviors of sensors within the runtime environments.

Our final results from these two applications demonstrated how trivial it is to detect and evade emulated runtime environments utilized by several popular Android malware analysis services. Each physical device we tested executed our pseudo-malware action, while each analysis service tested was successfully evaded.

6.1. Related work

Although our research was not seminal in nature, we do feel it contributes to an ongoing area of interest within the mobile security community. Foundational work in 2012 by J. Oberheide and C. Miller discussed several ways they fingerprinted Google’s Play Store Android Bouncer and its analysis environment, as well as their subsequent methods for evading its runtime analysis [15]. At the time of our testing, Android Bouncer did not allow outbound network traffic to pass their network perimeter, thereby eliminating any ability we would have at logging behaviors from that environment. Interestingly, Google recently moved to include human review for all applications submitted to the Play Store for publication [16].

Work presented last year at EuroSec detailed how various heuristics from static, dynamic, and hypervisor (i.e., the virtualization software itself) attributes could be used for analysis evasion [17]. Similar online services were tested, and our research extends this work by exploring and detailing additional sensor and data sources within Android runtime environments that could be used for analysis evasion.
### Table 3. KonamiCode application results (static version)

<table>
<thead>
<tr>
<th>Sensor/Resource Tested</th>
<th>Devices</th>
<th>Malware Analysis Services</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Galaxy S4</td>
<td>Galaxy S5</td>
</tr>
<tr>
<td></td>
<td>Galaxy Note 4</td>
<td>Nexus 5</td>
</tr>
<tr>
<td>Test 1: Number of Sensors²</td>
<td>✔️ ✔️ ✔️ ✔️</td>
<td>🚧 🚧 🚧 🚧 🚧</td>
</tr>
<tr>
<td>Test 2: Accelerometer Sensor³</td>
<td>✔️ ✔️ ✔️ ✔️</td>
<td>🚧 🚧 🚧 🚧 🚧</td>
</tr>
<tr>
<td>Test 3: Geomagnetic Field Sensor⁴</td>
<td>✔️ ✔️ ✔️ ✔️</td>
<td>🚧 🚧 🚧 🚧 🚧</td>
</tr>
<tr>
<td>Test 4: Orientation Sensor⁵</td>
<td>✔️ ✔️ ✔️ ✔️</td>
<td>🚧 🚧 🚧 🚧 🚧</td>
</tr>
<tr>
<td>Test 5: Proximity Sensor⁶</td>
<td>✔️ ✔️ ✔️ ✔️</td>
<td>🚧 🚧 🚧 🚧 🚧</td>
</tr>
<tr>
<td>Test 6: Battery⁷</td>
<td>✔️ ✔️ ✔️ ✔️</td>
<td>🚧 🚧 🚧 🚧 🚧</td>
</tr>
<tr>
<td>Test 7: Bluetooth⁸</td>
<td>✔️ ✔️ ✔️ ✔️</td>
<td>🚧 🚧 🚧 🚧 🚧</td>
</tr>
<tr>
<td>Test 8: Audio⁹</td>
<td>✔️ ✔️ ✔️ ✔️</td>
<td>🚧 🚧 🚧 🚧 🚧</td>
</tr>
<tr>
<td>Test 9: Phone State¹⁰</td>
<td>✔️ ✔️ ✔️ ✔️</td>
<td>🚧 🚧 🚧 🚧 🚧</td>
</tr>
<tr>
<td>Was Device ID (IMEI) sent?</td>
<td>✔️ ✔️ ✔️ ✔️</td>
<td>🚧 🚧 🚧 🚧 🚧</td>
</tr>
</tbody>
</table>

² Emulated environments were known to only indicate five sensors or less as being implemented.
³ Emulated environments provided detectable values for accelerometer max range, power, and vendor name.
⁴ Emulated environments provided detectable values for geomagnetic field sensors’ max range, power, and resolution.
⁵ Emulated environments provided detectable values for orientation sensors’ power, resolution, and vendor name.
⁶ Emulated environments provided detectable values for proximity sensors’ max range, power, resolution, and vendor name.
⁷ Emulated environments provided detectable values for battery temperature and voltage when current status was queried.
⁸ Emulated environments provided detectable values indicating a Bluetooth adapter was not present.
⁹ Emulated environments provided detectable values for initial audio volume levels across seven audio streams.
¹⁰ Emulated environments provided detectable values for a device’s unique identification number (e.g., its IMEI).
¹¹ Emulated environments provided accelerometer events with unchanging values of 0.0 (x-axis), 9.7762 (y-axis), and 0.813417 (z-axis) m/s².
¹² Emulated environments provided geomagnetic field events with unchanging values of 0.0 (x-axis), 0.0 (y-axis), and 0.0 (z-axis) µT.
¹³ Emulated environments provided proximity events with unchanging values of 1.0 cm.

### Table 4. KonamiCode application results (dynamic version)

<table>
<thead>
<tr>
<th>Sensor/Resource Tested</th>
<th>Devices</th>
<th>Malware Analysis Services</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Galaxy S4</td>
<td>Galaxy S5</td>
</tr>
<tr>
<td></td>
<td>Galaxy Note 4</td>
<td>Nexus 5</td>
</tr>
<tr>
<td>Test 1: Accelerometer Events¹¹</td>
<td>✔️ ✔️ ✔️ ✔️</td>
<td>🚧 🚧 🚧 🚧 🚧</td>
</tr>
<tr>
<td>Test 2: Geomagnetic Field Events¹²</td>
<td>✔️ ✔️ ✔️ ✔️</td>
<td>🚧 🚧 🚧 🚧 🚧</td>
</tr>
<tr>
<td>Test 3: Proximity Events¹³</td>
<td>✔️ ✔️ ✔️ ✔️</td>
<td>🚧 🚧 🚧 🚧 🚧</td>
</tr>
<tr>
<td>Was Device ID (IMEI) sent?</td>
<td>✔️ ✔️ ✔️ ✔️</td>
<td>🚧 🚧 🚧 🚧 🚧</td>
</tr>
</tbody>
</table>

¹¹ Emulated environments provided accelerometer events with unchanging values of 0.0 (x-axis), 9.7762 (y-axis), and 0.813417 (z-axis) m/s².
6.2. Future work in countering anti-analysis

Because our research demonstrated how easily emulated mobile environments can be detected and evaded, we believe future work in the area of mobile malware should explore multiple methods of countering anti-analysis techniques.

One such method would be to concentrate on better simulation for sensor events and other hardware events (e.g., battery status, GPS, etc.). We recently came across a program called SensorSimulator, which reportedly can simulate realistic values on an Android emulator and has the ability to store and replay sensor event data from actual physical devices.

Another method might be to abandon using emulated environments altogether for malware analysis, and conduct scaled testing utilizing large numbers of physical mobile devices, similar to commercial endeavors like Bitbar.

Finally, the most promising method we encountered while concluding our research came from SandDroid’s creator, Wenjun Hu. At the 2014 PacSec conference in Tokyo, he advocated for implementing API runtime hooks within an emulated environment to intercept malicious malware attempting to detect the runtime environment [18]. These runtime hooks would then allow analysis environments to provide simulated results in a lightweight manner, as opposed to modifying Android source code.

7. References


