**Week 8: Analyze an Open Source Project**

Nate Bachmeier

April 28th, 2019

TIM-8101: Principals of Computer Science

Northcentral University

# Automated Defect Detection of Android Apps by Graph Query

F-Droid is a Free Open Source Software (FOSS) application store for the Android platform. The store hosts thousands of applications with a consistent method for downloading both the Android Package (APK) and an exact snapshot of the source tree used for the build. As frequently discussed at conferences such as Mining Software Repositories (MSR), it ideal for many Android research projects.

Previous efforts have demonstrated the effectiveness of automated vulnerability detection by creating a representation of the application as a graph, then mining for complex relationships. If the same process were applied to Android packages would it also find interesting results? (Zhang, Deutschbein, Huang, & Sturton, 2018)(Lemieux, Sen, Padhye, & Song, 2018)(Avgerinos, et al., 2014).

# Finding an Interesting Project

The first challenge encountered was understanding the landscape of F-Droid and which applications could be of interest. To address this, a script was authored to enumerate all packages and download the most recent version’s compiled APK and associated src.tar.gz files. These 1610 apps were uploaded to Amazon S3 storage and extracted using Amazon EC2 compute instances.

## Contents of APK

An APK file is based on standard ZIP archive format and contains a manifest definition (AndroidManifest.xml), resources, native libraries, and one or more classes.dex files.

The manifest specifies the entry points into the application, its enabled permission, and endpoint access controls. Each exposed object is categorized as a UI Activity, Data Content Provider, Background Service, or Broadcast Receiver. The system or other applications interact with these objects by publishing Intent data structures. To publish an intent the caller needs to be authorized according to the permissions policy on the object. Permissions come in two flavors Protected Normal and Dangerous. Dangerous permissions are defined as anything which can compromise the integrity or privacy of the user or system (Elenkov, 2015).

Android applications run on the Dalvick virtual machine which has been specifically optimized for low power mobile devices. Developers can write their code in Java, Kotlin, or C/C++ and then compile into Java bytecode and native libraries. The build artifacts are then packaged into Java Archives which are converted into classes.dex files.

This process can be reversed using the utilities javap and dex2jar-2.0 as a mechanism to get Java Assembly. From the Java Assembly it is possible to determine class shapes, method invocation bindings, etc. This information could have also been parsed out the src.tar.gz files however there are several challenges to that approach. For instance, the src.tar.gz do not contain standardized build scripts which becomes tedious for large scale exploration. Another is that a solution which works on open and closed source APKs is preferred.

## Contents of the DEX Files

We know that the classes.dex contains the app code, but what are the major libraries and themes? To answer this question a random sample of 21% (339) classes.dex were decompiled into Java Assembly. Including setup this compute intensive workload took 2 hours on a 128-vcore Amazon EC2 machine.

Android Jet Pack was present in 65% of the sampled applications and is by far the most commonly used framework. According to the developer documentation it removes common boilerplate code and accelerates the time to develop new applications. It covers a wide range of scenarios such as interacting with SQLite to animating layout transitions. Clearly understanding this framework is critical building Android apps.

ProGuard is an open source tool for obfuscation and code reduction that was used by 23.2% of sampled applications. This is performed by mutating the compiled bytecode as a post processing build task. It is also used to prevent app re-packaging which is common practice by malware and scam artists. For example, an app will be repackaged and published back into the store with all advertising revenue redirected to the scammer (Rastogi, Bhushan, & Gupta, 2016) (Hammad, Garcia, & Malek, 2018).

Kotlin is the recommended language for new development with compiler support for transforming into JavaScript, Java, or C/C++. This enables the code to be written once and run anywhere. Only 8.7% of sample applications are using the new language which suggests that most developers are still using Java by default.

Metaprogramming was very prominent across the sampled application using packages such as retrofit, butter-knife, and dagger. Developers can use meta programming at compile or runtime to generate code and make decisions based on the shape of classes. To make runtime decisions Java Reflection is commonly used. There is performance overhead to reflection which many developers mitigate by instead using Gradle plugins at build time (Wharton, 2019).

The Android operating system is shipped with SQLite database server and is universally used for storing state. Apps can share the data with third parties through Content Providers. Developers often build up SQLite queries through string concatenation which introduces SQL Injection and encoding challenges (Chon & Frankl, 2012). Android Room, a component of JetPack, is an Object Relational Mapper (ORM) that can generate the queries and guarantee their safety.

## Exploring the Manifest

The AndroidManifest.xml describes all components of an application, their configuration, and permissions. An attack vector exists where services can be publicly exported and are not properly secured. These configuration errors can allow unprivileged code to publish Intents to a privileged service and then perform actions that would otherwise be blocked (Choi & Yongdae, 2018) (Elenkov, 2015).

To explore this idea a utility was written to parse 1365 manifests into a graph structure. A filter was applied to remove any manifest that did not enable a dangerous permission (783). Another filter was applied to find the subset with misconfigured services (94) and receivers (88). This list was deduplicated resulting in 119 apps that exhibited this trait.

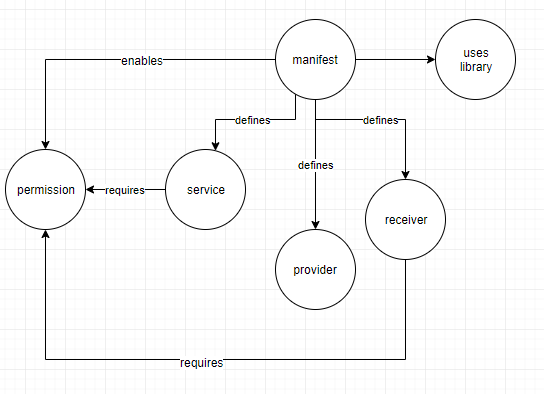


Figure 1: Manifest Graph

# Exploring a specific application

The list of potentially vulnerable apps was subset to only those with the SEND\_SMS permission enabled. This permission was selected as it makes for a clear demonstration of the problem to a general audience. Of the apps available SMS Secure was selected because it had the term ‘secure’ in the name.

A manual inspection of the code confirmed that a configuration error exposed the QuickResponseService and a code path existed to call sendMultipartTextMessage. This is exactly the desired scenario.

## Describe Vulnerable High-Level Flow

1. The QuickResponseService is given the intent ACTION RESPOND VIA MESSAGE.
2. The Intent structure contains the phone numbers and text message contents.
3. The service calls MessageSender to store the message in a local outbox database.
4. The MessageSender creates a SmsSendJob to bind the record and send action.
5. The JobManager is receives the job will schedules it for execution.
6. The SmsSendJob will bind to the SmsManger system service.
7. The system API sendMultipartTextMessage is called to send the message.

## Reducing False Positives

Querying the manifest graph quickly identified potential vulnerabilities but it is susceptible to false positives. Consider the scenario where the service had not called sendMultipartTextMessage, then exploitation of the permission by the third party would not have worked.

To remove some of these false positive scenarios call graph analysis could be performed to ensure a path exists between the service class and the target method. The head and tail can be programmatically determined by inspecting the manifest file. The service node states its implementation class and the target method can be inferred from the enabled permission set.

Since the starting node is publicly registered it cannot be obfuscated. The tail can be obfuscated however algorithms exists to identify modified system methods (Hammad, Garcia, & Malek, 2018). Code along the path can be obfuscated as call graph analysis is only interested in connectivity, not the symbol names.

To explore this idea the ByteCodeMapper utility was written to parse Java Assembly files into GraphML documents. GraphML is an XML based file format which is supported by many open source graph analysis tools. The call graph for SMS Secure contained 9,253 vertices and 45,058 edges. Vertices represent the classes and methods with edges specifying invokes, extends, and declared by relationships.

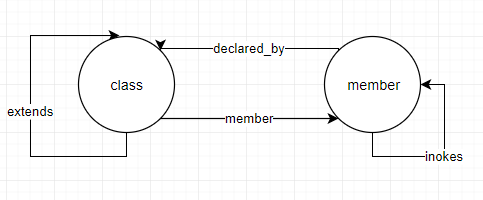


Figure 2: Call Graph

The document was loaded into Apache TinkerPop which confirmed the connectivity of the nodes with query:

graph.traversal()  
.V('org.smssecure.smssecure.service.QuickResponseService')  
.loop(1){it.loops<100}{true}  
.hasId('android.telephony.SmsManager.sendMultipartTextMessage')  
.path{it.name}

## Expanding the Idea

Instead of Java Assembly it would be possible to use .NET CIL or LLVM IR byte codes to form the call graph. The same query could confirm connectivity between different parts of the code base. Another strength of this approach is that it can easily be applied to different static analysis scenarios.

## Limitations

The traversal application is only looking for broad connectivity between methods and can still report false positives. An example might be an if/else branch where its not possible to enter one of the branches.

Another challenge is with code that uses runtime binding such as Reflection or loads dynamic assemblies. Some Android apps only contain a subset of the functionality and require users to buy premium features. When the feature is purchased the additional code is downloaded across the network.

It can also be complex to reliably traverse the code as it crosses between the Java Native Interface. Researchers have also reported that Android applications make heavy use of callbacks which can increase tracking error (Fan, et al., 2018).

# Usages with Data Storage

Android developers must also face challenges with ensuring data is properly stored. This comes arises from scenarios (1) the app can be stopped at any point and (2) third-party apps are actively trying to steal data.

## SQLite

Unlike desktop applications a mobile app can be unloaded by the operating system at any time for any number of reasons. To address this apps like SMS Secure, persist transient data into SQLite and treat the table as inbox. Background services will pull from the inbox and do the needful.

If the developer is not using an ORM like Android Room, then they must write the queries by hand. While modern frameworks exist, it can be prohibitively expensive to port the legacy code to these technologies. The SMS Secure app has 8368 lines of database specific code and upgrading it could easily take several weeks of developer time.

To address this issue data flow analysis could be used to discover which code paths led to invalid SQL. One approach is to extract the call graph information and then use symbolic execution to enumerate all the combinations. The tests can be prioritized based on taint analysis of user input. If users manipulate the query it is more likely to have an issue than more static parts of the system.

## Encrypting Data

Every app is assigned a unique Linux user id so that its data is isolated from other software on the system. However, vulnerabilities in the operating system, an endpoint can be misconfigured, or someone could steal the device. To mitigate this scenario, applications need to encrypt their data at rest. The platform provides the Cipher class to handle these scenarios.

It would be possible to extend the call graph analysis to identify where data is being written to the SD card. Going one step further a Gradle plugin could be written to add transparent encryption around those APIs. This would ensure that all data is always protected.

# Usages with Activities

Android exposes the Activity class as a representation of a UI layout. It can contain simple components like buttons and text boxes or complex structures like fragments. Literal text is often stored in resource files. Due to the scattered nature of the source artifacts it can be difficult to ensure everything aligns correctly and text is not cut off.

This scenario can also be addressed with graph queries if we represent each of the components as a node and edges for the hierarchical structure. Width and height properties can be attached to the nodes and then all sub paths summed. If a summed total equals an unexpected value, the developer can be notified before committing the change.

The graph could be further extended to have nodes for each supported configuration allowing and then all combinations inspected in a single query.