**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 discovering both the Android Package (APK) and a snapshot of the source tree (src.tar.gz). As frequently discussed at conferences such as Mining Software Repositories (MSR), F-Droid is ideal for many Android research projects.

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

# Finding an Interesting Project

The first challenge encountered was understanding the landscape of F-Droid and which applications might be of interest. To address this issue a script was created to enumerate through the store and download each app’s most recent APK and associated src.tar.gz files. The script identified and staged 1610 apps in Amazon S3 storage. These packages were then expanded 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 and 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 of 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 back to Java Assembly with the open source utilities javap and dex2jar-2.0. From the Java Assembly it is possible to determine class shapes, method invocation, etc. The sources files could have been parsed to extract these same details, however there are several challenges to that approach. (1) The src.tar.gz do not contain standardized build scripts. (2) The extensive use of Gradle metaprogramming can change the code in a significant manner. (3) Relying on the source would only work on open source apps.

## Contents of the DEX Files

We know that the classes.dex contains the app code, but what are the major libraries and themes? A random sample of 21% (339) of 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 to become a successful Android developer.

ProGuard is an open source tool for obfuscation and code reduction that was used by 23% of sampled applications. It performed these actions 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.

Given the relative ease of decompiling APK files a scammer might swap out the configuration for the advertising binaries and then “repack and publish” their version into the store. Then users of the repacked version will send all advertising revenue 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 sampled 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 using Gradle plugins to generate code at build time (Wharton, 2019).

The Android operating system ships 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 other encoding errors (Chon & Frankl, 2012).

## 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 misconfiguration.

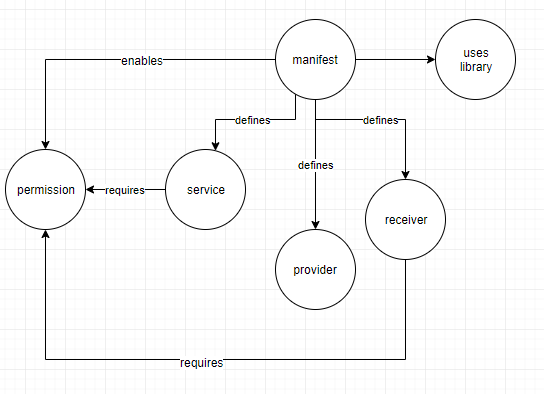


Figure 1: Manifest Graph

# 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 ironically 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, which would have prevented the permission exploitation.

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 symbolic names.

A 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.

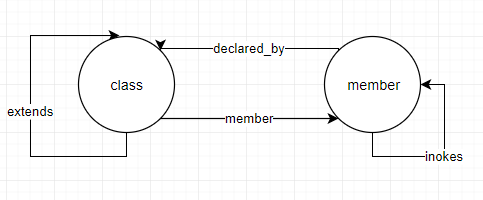


Figure 2: Call Graph

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. This could be addressed with a more complex data model such as adding nodes for each statement and edges to connect them back to the declaring method.

Code that uses runtime binding, Reflection and other dynamicly loaded assemblies can be complex to analyze. Some Android apps only contain a subset of the functionality and require users to buy premium features. When the feature is purchased then additional code is downloaded across the network.

It can also be complex to reliably traverse the code as it crosses the Java Native Interface. Others have also reported that Android applications heavy use of callbacks also increases tracking error when performed at scale (Fan, et al., 2018).

# Usages with Data Storage

Android developers also face challenges with ensuring data is properly stored. This arises from scenarios such as (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 (Elenkov, 2015). To address this issue apps like SMS Secure, persist transient data into SQLite and treat the table as an inbox. Background services query 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. Modern frameworks exist however it can be prohibitively expensive to port legacy code to these technologies. The SMS Secure app has 8,368 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 (Chon & Frankl, 2012). 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 the user input flow. Areas that can be manipulate by user interaction are more likely to encounter issues due to their dynamic nature.

## Encrypting Data

Every app is assigned a unique Linux user id and its data isolated from other software on the system. However, data at rest must always be encrypted as there can be vulnerabilities in the operating system, misconfigured endpoints, or someone could steals the physical device. Developers can use the Cipher class and related crypto APIs to easily protect their information.

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.

## Content Providers

Content Providers represent a standardized mechanism for sharing data between applications. A third-party application can read or write to URI, and then if they are authorized those requests will be delivered to the content provider. These providers can create and authorize temporary Uri for the third-party instead of providing permanent access to all their data.

The permanent access policy that is defined in the manifest can be mined like the service data. Tracking the temporary accesses and ensuring they are properly scoped is a complex issue for static analysis. It might be possible to inject dynamic analysis into debug versions of the app. The QA teams could then review log data as part of their integration tests.

# 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 separate resource files. Due to the scattered nature of the source artifacts it can be difficult to ensure everything aligns correctly on the screen 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 summed across different paths. If a sub graph’s total does not equal an unexpected value, then the developer can be notified before committing the change.

# Other Observations

The SMS Secure app contains a custom implementation of its cryptographic algorithms and uses them for securing text messages. It is generally considered bad behavior to recreate crypto functions as they are very difficult to implement correctly.

The app also receives arbitrary multimedia messages which can contain complex audio and video files. If those files are corrupt or malicious then it can crash the application. To mitigate these issues the app is sending an Intent for an external application to preform the work.

This also presents an interesting scenario where it could be useful to have a system graph of the holistic device. Nearly all Android research is scoped to a single application which could represent untested scenarios.

Lastly many Android applications appear to be shipped with native libraries which can be vulnerable to memory allocation attacks. The graph could be queried to determine the entry points and fuzz testing performed to find defects within those methods. There is also low hanging fruit by simply indexing the versions of these binaries and comparing against public vulnerability lists.

# Identify a Potential Venue

This paper discussed themes of (1) repository mining and (2) automated defect detection. Either the Mining Software Repositories (MSR) or Automated Software Engineering (ASE) would be good venues for the solution.

If MSR is chosen, then it would be helpful to focus more on the empirical effectiveness across a breath of applications. For ASE the conversation should focus on more depth of fewer examples.

## ASE Submission

ASE 2019 will be held in San Diego, California in mid-November. Papers can be submitted at <https://2019.ase-conferences.org/track/ase-2019-papers>. There are specific formatting requirements such as font sizes and as a technical research paper cannot exceed 10 pages. If these formatting requirements are not met the document is automatically rejected.

Technical papers need to also be well cited and any claims of novelty be justified. If the works builds on another area it might push it into the category of experience papers. The paper must also identify which track it belongs too. For this effort reverse engineering would be a likely candidate.

## MSR Submission

MSR allows technical papers to be either short or long 4 or 10 pages respectfully. Short is intended for new ideas that are not fully developed. There is also review requirements around the quality of the evaluation and that results can be replicated.

The paper needs to be properly formatted and then submitted through EasyChair at <https://easychair.org/conferences/?conf=msr2019>. All formatting must align with the standardized IEEE guidelines.

# Conclusion

Android Package files are ZIP archives which can be expanded to extract the classes.dex and AndroidManifest.xml files. They can be parsed into graph processing systems and rapidly searched in a contextually rich manner.

The custom utilities demonstrated the effectiveness of converting software into a graph and querying the relationships as a mechanism to find defects. Using the graph, a critical vulnerability was detected and manually confirmed. The graph also identified 118 other applications which maybe exploitable. There are several other use cases for detecting defects within the call graph such as SQL Injections, unencrypted data store, or misaligned UI components.

This works has the potential to fit into either ASE or MSR conference with a little bit of tweaking. The deadline for MSR has passed which makes ASE in November the next available conference. That would be the first one attempted and if the work was rejected then try for MSR in 2020.

# References

Agrawal, G. (1999). Simultaneous demand-driven data-flow and call graph analysis. *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360) Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on.*, 453-462.

Avgerinos, T., Kil Cha, S., Rebert, A., Schwartz, E., Woo, M., & Brumley, D. (2014). Automatic Exploit Generation. *Communications of the ACM Vol 57. No. 2*, 74-84.

Choi, H., & Yongdae, K. (2018). Large-Scale Analysis of Remote Code Injection Attacks in Android Apps. *Security & Communication Networks. 4/17/2018*, 1-17.

Chon, J., & Frankl, P. (2012). AQUA: Android QUery Analyzer. *2012 19th Working Conference on Reverse Engineering Reverse Engineering, Working Conference on Reverse Engineering (WCRE), 2012 19th Working Conference on*, 395-404.

Elenkov, N. (2015). *Android security internals.* San Francisco : No Starch Press.

Fan, L., Su, T., Chen, S., Meng, G., Liu, Y., L, X., . . . Su, Z. (2018). Large-Scale Analysis of Framework-Specific Exceptions in Android Apps. *ICSE ’18, May 27-June 3, 2018, Gothenburg, Sweden*.

Hammad, M., Garcia, J., & Malek, S. (2018). A Large-Scale Empirical Study on the Effects of Code Obfuscations on Android Apps and Anti-Malware Products. *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*.

Lemieux, C., Sen, K., Padhye, R., & Song, D. (2018). PerfFuzz: Automatically Generating Pathological Inputs. *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*.

Rastogi, S., Bhushan, K., & Gupta, B. (2016). Measuring Android App Repackaging Prevalence based on the Permissions of App. *Procedia Technology; 2016, Vol. 24*, 1436-1444.

Shah, Y., Shah, J., & Kansara, K. (2018). Code obfuscating a Kotlin-based App with Proguard. *2018 Second International Conference on Advances in Electronics, Computers and Communications (ICAECC) Advances in Electronics, Computers and Communications (ICAECC), 2018 Second International Conference on. :1-5 Feb, 2018*.

Statista. (2019). *Number of apps available in leading app stores as of 3rd quarter 2018*. Retrieved from Statista: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

Wharton, J. (2019, April 25). *Mechanisms of Metaprogramming.* Retrieved from JakeWharton: https://jakewharton.com/mechanisms-of-metaprogramming/

Zhang, R., Deutschbein, C., Huang, P., & Sturton, C. (2018). End-to-End Automated Exploit Generation for Validating the Security of Processor Designs. *Annual IEEE/ACM International Symposium on Microarchitecture*, 815-827.