ANDROID SECURITY VIA STATIC ANALYSIS
TECHNIQUES

by

Feng Shen
September 1st, 2018

A dissertation submitted to
the faculty of the Graduate School of
the University at Buffalo, State University of New York
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

Department of Computer Science and Engineering
Acknowledgement

First and foremost I want to thank my advisors Prof. Steven Y. Ko and Prof. Lukasz Ziarek. It has been an honor to work with them. They have taught me so much, both consciously and unconsciously, in both research and life. I appreciate all their contributions of time, ideas, and funding to make my Ph.D. experience productive and stimulating. The joy and enthusiasm they have for research was contagious and motivational for me, even during tough times in the Ph.D. pursuit. I am really thankful for the excellent examples they have provided as successful computer scientists. They are knowledgeable, patient and willing to help. Without their help and advice, I could not make it through my research life. They have guided me with their invaluable suggestions, lightened up the way and encouraged me a lot in my academic life.

The members of the RMS group have contributed immensely to my personal and professional time at UB. The group has been a source of friendships as well as good advice and collaboration. I am really grateful to be part of this group and it has been a great pleasure to work with or alongside all group members. I would like to acknowledge Kyungho Jeon, who helped a lot when I first came to UB. We worked together on HybrEx and PigOut projects, and I very much appreciated his hardwork, kindness and patience. I am also very much thankful to Taeyeon Ki, Yin Yan and Sharath Chandrashekhara. It has been a great pleasure to work with them. They have always enlightened me with their knowledge and expertise on research and are always willing to provide good advice and help on both research and personal life. I would like to address special thanks to Justin Del Vecchio. Justin made great contributions to my research work by providing valuable advice, running experiments and paper review/writing. I would like to thank all former or current students who helped me on my research work.

In my later work of Complex-Flows, I would like to address special thanks to Prof. Aziz Mohaisen. He provided me the modern Android malware dataset as well as the initial code for N-gram analysis. The Complex-Flow analysis would not have been possible without his expertise and great contribution. For this dissertation, I would like to thank my committee members: Prof. Steven Y. Ko, Prof. Lukasz Ziarek and Prof. Bharat Jayaraman for their time, interest and helpful comments.

My time at UB was made enjoyable in large part due to the many friends that became a part of my life. I am grateful for time spent with roommates and friends, for all the support through good and bad times. I am fortunate to have them who have opened their hearts to me in turn motivating me many a times with their extraordinary brilliance and analytical perceptions. I am glad to share my research experiences with them since it was a joy and enlightenment to me.

Lastly, I would like to thank my family for all their love and encouragement, for my parents who raised me with love and supported me in all my pursuits, for my brother and sister taking care of everything back home while I am far away. All of you made it possible for me to reach this last stage of my endeavor. Thank You.

Feng Shen
Aug. 2018
Abstract

Android is a popular platform designed for mobile devices. It consists of a customized Linux kernel, middleware, and a few core applications such as the Phone application. The middleware, commonly referred to as the Android framework, provides libraries and runtime services to applications. Applications in Android are written mainly in Java. Once compiled, Android transforms its applications into the Dalvik Executable (or DEX) format to minimize the memory footprint. Android uses a Java VM called Dalvik to execute DEX bytecode.

Unlike other mobile OSes, Android has a unique permission mechanism. At development time, an application developer needs to explicitly request permissions by including them in an application configuration file (AndroidManifest.xml). We refer to this configuration file simply as the manifest in the remainder of the paper. At installation time, each user needs to review the permissions that the application requests and explicitly grant them.

Android currently has over 130 permissions applications can request in API level 17. These permissions are API-oriented and access-based, i.e., permissions control access to sensitive APIs (referred to as protected APIs). Generally, an application can ask for permissions to use protected APIs for phone resources (e.g., storage, NFC, WiFi, etc.) or information available on the phone (e.g., contacts, location, call logs, etc.).

While this permission mechanism is effective in pinpointing which sensitive APIs that an application uses, it does not provide any insight into what the application actually does with the APIs. Thus, our goal is to complement the existing mechanism by providing both behavioral information of a single application as well as the interactions among multiple applications.

This thesis proposes Flow Permissions, an extension to the Android permission mechanism. Unlike the existing permission mechanism, our permission mechanism contains semantic information based on information flows. Flow Permissions allow users to examine and grant per-app information flows within an application (e.g., a permission for reading the phone number and sending it over the network) as well as cross-app information flows across multiple applications (e.g., a permission for reading the phone number and sending it to another application already installed on the user’s phone). Our goal with Flow Permissions is to provide visibility into the holistic behavior of the applications installed on a user’s phone. In order to support Flow Permissions on Android, we have developed a static analysis engine that detects flows within an Android application. We have also modified Android’s existing permission mechanism and installation procedure to support Flow Permissions.

Along with rapid growth of Android market, both Android malware and benignware have evolved and become more complicated. Due to the diverse functionalities modern apps provide, the benign apps are more complex and it is common for a benign app to leverage multiple sensitive data sources for normal usage. Besides, malware apps disguise themselves as benign apps and hide the malicious code among benign code. It becomes more and more difficult to distinguish malware apps from benign apps. As a result, mobile malware detection continues to be a challenging problem, with security researchers estimating new malware being created and deployed every 4.2 seconds. To combat this problem, there have been many different proposed approaches and tools proposed in recent years. However, all these tools are evaluated on hand selected or private data sets, making comparison across tools and techniques very difficult. The only common comparison point is a public malware benchmark set gathered in 2012. To tackle these issues, this paper introduces a new benchmark app set for comparing and contrasting Android malware detection strategies. We begin with a survey and systematic study of 56,000 modern malware apps. We discuss current Android malware detection tools and synthesize a set of features/metrics that these tools leverage. Next, we statistically analyze our dataset based on these metrics. We consider the evolution of both malware and benign applications with respect to these metrics. Based on these studies and comparisons we select a representative 1,000 malware apps and 1,000 benign apps as a modern app benchmark.

Along with the explosive growth of smartphone sales, the threat of Android malware is spreading rapidly, especially those repackaged Android malware. This thesis proposes a new
technique to detect mobile malware based on information flow analysis. Our approach examines the structure of information flows to identify patterns of behavior present in them and which flows are related, those that share partial computation paths. We call such flows Complex-Flows, as their structure, patterns, and relations accurately capture the complex behavior exhibited by both recent malware and benign applications. N-gram analysis is used to identify unique and common behavioral patterns present in Complex-Flows. The N-gram analysis is performed on sequences of API calls that occur along Complex-Flows’ control flow paths. We show the effectiveness and precision of our technique by applying it to multiple different data sets.
Contents

Abstract \hspace{1cm} iv

1 Introduction \hspace{1cm} 1

2 Literature Review \hspace{1cm} 3
  2.1 Android Permissions, Analysis, and Tools \hspace{1cm} 3
  2.2 Android Malware Detection \hspace{1cm} 3
  2.3 Other Android Security tools \hspace{1cm} 4

3 Information Flow as a Permission Mechanism \hspace{1cm} 4
  3.1 Motivation \hspace{1cm} 4
    3.1.1 Flows as Permissions \hspace{1cm} 5
    3.1.2 Interaction Between Apps \hspace{1cm} 6
  3.2 Flow Permissions \hspace{1cm} 6
    3.2.1 Permission Domain Types \hspace{1cm} 6
    3.2.2 Flow Permission Mechanism \hspace{1cm} 7
    3.2.3 Statically Deriving Flow Permissions \hspace{1cm} 7
  3.3 System Design \hspace{1cm} 8
    3.3.1 Entry Point Discovery \hspace{1cm} 8
    3.3.2 Call Graph Restructuring \hspace{1cm} 10
    3.3.3 Content Provider Resolution Analysis \hspace{1cm} 11
    3.3.4 Intent Resolution Analysis \hspace{1cm} 13
    3.3.5 Binder Resolution Analysis \hspace{1cm} 14
    3.3.6 Interprocedural Permission Flow Analysis \hspace{1cm} 15
    3.3.7 Augmented Package Installer \hspace{1cm} 16
  3.4 Results and Discussion \hspace{1cm} 16
    3.4.1 Statistics of Flow Permissions \hspace{1cm} 17
    3.4.2 Uses of the Statistics \hspace{1cm} 20
    3.4.3 Performance \hspace{1cm} 22
    3.4.4 Manual Validation \hspace{1cm} 23
    3.4.5 Flow Frequency in Malicious Apps \hspace{1cm} 24
    3.4.6 Comparison with FlowDroid over DroidBench \hspace{1cm} 25
    3.4.7 Case Studies: Flow Permissions in the Real World \hspace{1cm} 30
    3.4.8 User Study \hspace{1cm} 30
    3.4.9 Threats to Validity and Discussion \hspace{1cm} 31

4 BlueBench: Modern Android Malware Characterization \hspace{1cm} 31
  4.1 Survey Structure and Starting Assumptions \hspace{1cm} 32
    4.1.1 Analysis Metrics \hspace{1cm} 32
    4.1.2 Modern Malware Family Statistics \hspace{1cm} 34
  4.2 Android Permission \hspace{1cm} 34
  4.3 Reflection analysis \hspace{1cm} 36
  4.4 Dynamic Execution and Native Code Usage on Android \hspace{1cm} 39
  4.5 Android Specific Constructs \hspace{1cm} 42
    4.5.1 Inter Component Communication in Android \hspace{1cm} 42
    4.5.2 Content Provider in Android \hspace{1cm} 44
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.6</td>
<td>Simple Information Flow Analysis</td>
<td>46</td>
</tr>
<tr>
<td>4.7</td>
<td>Complex-Flow Information Analysis</td>
<td>51</td>
</tr>
<tr>
<td>4.8</td>
<td>Dataset App Selection</td>
<td>53</td>
</tr>
<tr>
<td>5</td>
<td>Complex-Flow for Malware Detection</td>
<td>55</td>
</tr>
<tr>
<td>5.1</td>
<td>Motivation</td>
<td>56</td>
</tr>
<tr>
<td>5.2</td>
<td>Complex-Flows</td>
<td>57</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Multi-Flows</td>
<td>58</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Complex Flows</td>
<td>58</td>
</tr>
<tr>
<td>5.3</td>
<td>System Design</td>
<td>59</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Multi-Flow Discovery</td>
<td>60</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Complex-Flow Extraction with API Sequences</td>
<td>60</td>
</tr>
<tr>
<td>5.3.3</td>
<td>N-gram Feature Generation</td>
<td>63</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Classification</td>
<td>63</td>
</tr>
<tr>
<td>5.4</td>
<td>Evaluation</td>
<td>64</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Evaluation Methodology and Metrics</td>
<td>65</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Runtime Performance</td>
<td>65</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Play_2014 Apps versus MalGenome Apps</td>
<td>66</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Google Play Apps versus Modern Malware Apps</td>
<td>67</td>
</tr>
<tr>
<td>5.4.5</td>
<td>General Google Play Apps vs. Modern Malware Apps</td>
<td>69</td>
</tr>
<tr>
<td>5.4.6</td>
<td>Comparison to MudFlow</td>
<td>70</td>
</tr>
<tr>
<td>5.4.7</td>
<td>Classification Precision and Recall Rates</td>
<td>72</td>
</tr>
<tr>
<td>5.4.8</td>
<td>Classification based on SVM with Balanced Training</td>
<td>73</td>
</tr>
<tr>
<td>5.4.9</td>
<td>Manual Validation</td>
<td>74</td>
</tr>
<tr>
<td>5.4.10</td>
<td>Discussion</td>
<td>75</td>
</tr>
<tr>
<td>5.5</td>
<td>Threats to Validity and Limitations</td>
<td>75</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion</td>
<td>76</td>
</tr>
<tr>
<td>Reference</td>
<td></td>
<td>77</td>
</tr>
</tbody>
</table>
List of Figures

1 The BlueSeal Android app analysis framework architecture. Shaded boxes represent components already present in Soot. ......................................................... 9
2 A code snippet illustrating the methods that comprise the control flow of an async tasks in Android and the implicit flow of arguments provided by the Android framework. ......................................................... 10
3 The execution flow of async task methods in their respective threads at runtime. 10
4 Flows based on pairing message sends to the appropriate message handlers. ........ 12
5 The data flow of an URI object that identifies which CP is being utilized. Dashed arrows indicate information derived from dataflow analyses and block arrows how that information is used to disambiguate the CP. .......................................................... 12
6 The data flow of an URI object initialization that is resolvable statically. ............... 13
7 The data flow of an Intent passed from one app to another. ................................. 13
8 Data and control flow relations between a binder activity and service. ...................... 14
9 Flow Permission example screenshot ............................................................... 17
10 Distribution of Flow Permissions generated for each app. 630 apps do not generate any Flow Permission and are not shown. ........................................................ 18
11 Distribution of raw flows generated for each app. (1046 apps range from 100 to 13,646 and 630 apps with 0 flow are omitted.) .................................................. 19
12 Scatter plot showing the time taken to analyze all apps in seconds. ......................... 24
13 Performance of our cross-app analysis. ............................................................... 25
14 Flow Frequency in Different Categories ............................................................ 27
15 Flow Frequency Example in DroidKungfu Apps ............................................. 28
16 Flow Frequency Example of Sending Private Information via SMS ....................... 28
17 Code Snippet in Smallest Modern Malware App. ............................................. 32
18 BlueBench App Requested Permission Family Distribution .............................. 37
19 Reflection Code Snippet in Modern Malware App. .......................................... 38
20 Reflection Code Snippet in MalGenome Malware App. .................................... 38
21 App Reflection Usage Distribution ................................................................. 39
22 App Reflection Usage Family Distribution ...................................................... 39
23 DexClassLoader Code Snippet from a Malware App. ...................................... 40
24 App Dynamic Class Loader Usage Distribution ............................................... 41
25 App Runtime Execution Usage Distribution .................................................... 42
26 Native Code Usage in Modern Malware ........................................................ 43
27 Native Invocation Usage Family Distribution .................................................. 44
28 BlueBench App Intent Usage Distribution ...................................................... 45
29 BlueBench App Intent Usage Family Distribution ........................................... 45
30 BlueBench App Content Provider Usage Distribution ....................................... 46
31 BlueBench App Content Provider Usage Family Distribution ............................ 46
32 MalGenome Information Flow Usage Distribution .......................................... 47
33 BlueBench Information Flow Usage Distribution ............................................. 47
34 BlueBench App LOC Per Information Flow Distribution ................................. 48
35 MalGenome App LOC Per Information Flow Distribution ............................... 49
36 BlueBench App Information Flow Family Distribution ..................................... 49
37 App Behavior Comparison in Benign and Malware Apps .................................. 53
38 BlueBench Complex Flow Usage Normalized Distribution .............................. 53
39 BlueBench Complex Flow Count Against App LOC. ....................................... 54
List of Tables

1. Table listing Android apps and their requested permission examples. 5
2. Table listing Android apps and our proposed Flow Permission extensions. 6
3. A brief overview of experiment results. 18
4. Ten most common flows in normal apps. 19
5. Ten most common flows in malicious apps. 20
6. Top-10 sources in normal apps. 20
7. Top-10 sources in malicious apps. 21
8. Top-10 sinks in normal apps. 21
9. Top-10 sinks in malicious apps. 22
10. Distinct malicious and normal apps using a flow, TelephonyManager:getLine1Number to (network)OutputStream:write. 22
11. Distinct malicious and normal apps using a flow pair compared to distinct apps within a category. 23
12. Distinct malicious and normal apps using the SmsManager:sendTextMessage sink. 23
13. Label-Flow Mappings for Fig. 14 26
14. Label-Flow Mappings for Fig. 15. 26
15. Label-Flow Mappings for Fig. 16. 27
16. Comparison between BlueSeal and FlowDroid. 29
17. User survey result showing how likely the user is to install the app. 31
18. Top 10 Permissions in MalGenome 36
19. Top 10 Permissions in BlueBench 36
20. Top 10 Permissions in Play Store Apps 37
21. Top Five Intents Apps Receive 44
22. MalGenome Top Data Flows 50
23. BlueBench Top Data Flows 50
24. Top Data Flows in Play Store Apps 51
25. Top 10 Sources in PlayStore Apps 51
26. Top 10 Sources in BlueBench 52
27. Information Flows in Both Benign and Malicious Apps 52
28. Top 10 Size-1 Sequences in BlueBench 54
29. Top 10 Size-2 Sequences in BlueBench 55
30. Top 10 MultiFlows in BlueBench 55
31. Final API Call Sequence Output 62
32. Example of API Sequence and its 2-grams 63
33. Gram Based Classification Results of Play_2014 and MalGenome Apps 67
1 Introduction

Modern mobile OSes such as iOS and Android provide permission mechanisms, allowing users to review how an application ("app") accesses the resources on a mobile device. Android, in particular, has a comprehensive permission mechanism; at development time, an app writer needs to explicitly request permissions by statically declaring them in an app configuration file (AndroidManifest.xml). During installation, a user needs to review the permissions that an app requests and explicitly grant them.

Although considered to be robust, the current permission mechanism of Android provides little contextual information on how permissions of an app are leveraged by the app. For example, it is unclear if an app with the permission to access the Internet, as well as the phone’s SIM card, exposes the private telephony data stored on the SIM card to the outside world. Apps can also communicate with one another via Android’s IPC mechanisms to effectively gain permissions they were not explicitly given, thereby bypassing the current permission mechanism [30].

Contribution 1: To address these issues, this thesis proposes a new permission mechanism, called Flow Permissions, that extends the existing Android permission mechanism with information flows between permission domains (e.g., reading from the SIM card and sending over the network). Our Flow Permissions identify single-app flows, i.e., information flows within an app, as well as cross-app flows, i.e., information flows across apps via IPC mechanisms. In order to synthesize single-app flows, we develop an automated static analysis engine that detects information flows within an Android app. To synthesize cross-app flows, we modify Android to perform cross-app permission analysis when installing a new app. This cross-app permission analysis compares information flows within the new app to those of already-installed apps and derives new Flow Permissions. This combination of static and installation-time analysis comprises BlueSeal, our Flow Permission synthesis system.

The Android market has been growing dramatically in the past decade. Along with it, both Android malware and benignware have been evolved and become more complicated. Due to the diverse functionalities modern apps provide, the benign apps are more complex and it is common for a benign app to leverage multiple sensitive data sources for normal usage. Besides, malware apps disguise themselves as benign apps and hide the malicious code among benign code. It becomes more and more difficult to distinguish malware apps from benign apps. In this case, a better understanding on the evolution of Android apps, especially malware apps, is needed. To achieve this, we would like to analyze modern malicious apps systematically and build up a benchmark dataset for future researchers on Android malicious apps.

Contribution 2: In this thesis, we propose a systematical study on modern Android malware to characterize them from various aspects. More specifically, we will present a large collection of 1,000 modern Android malware apps, which will cover the majority of existing Android malware. We intend to release the whole dataset to the research community. Secondly, we will analyze these collected malware samples and thoroughly characterize them based on their detailed behavior from different aspects. Eventually, we would like to perform a evolution-based study by comparing old Android malware and new ones.

Now we understand the evolution of both Android malware and benign apps and benign and malware apps exhibit different behavior patterns. By doing data-flow analysis via BlueSeal, we have a better understanding about app behavior on sensitive data inside apps and this can be a very important factor to indicate apps’ maliciousness. In this case, we want to leverage this information and apply on distinguishing malicious apps from benign apps. According to security experts [3], over 37 million malicious applications (apps) have been detected in only a 6-month span in the beginning of 2016. Clearly, malware detection is crucial to combat this high-volume...
spread of malicious code. Previous approaches for malware detection have shown that analyzing information flows can be an effective method to detect malicious apps [14, 31, 68]. This is not surprising, as one of the most common characteristics of malicious mobile code is collecting sensitive information from a user’s device, such as a device’s ID, contact information, SMS messages, location, as well as data from the sensors present on the phone. When a malicious app collects sensitive information, the primary purpose is to exfiltrate it, which unavoidably creates information flows within the app code base.

Many previous systems have leveraged this insight and focused on identifying the existence of simple information flows – i.e. considering an information flow as just a (source, sink) pair. A source is typically an API call that reads sensitive data, while a sink is an API call that writes the data read from a source. These previous approaches use the presence or absence of certain flows to determine whether or not an app is malicious and can achieve 56%-94% true negative rates when applied to known malicious app data sets.

In this thesis, we show that there is a need to look beyond simple flows in order to effectively leverage information flow analysis for malware detection. By analyzing recently-collected malware, we show there has been an evolution in malware beyond simply collecting sensitive information and immediately exposing it. Modern malware performs complex computations before, during, and after collecting sensitive information. More complex app behavior is involved in leveraging device sensitive data. A simple (source, sink) view of information flow does not adequately capture such behavior.

Furthermore, mobile apps themselves have also evolved in their sophistication and in the number of services they provide to the user. For instance, most common apps now leverage a user’s location to provide additional features like highlighting points of interest or even other users that might be nearby. Augmented reality apps go a step further, leveraging not only a user’s location, but also their camera and phone sensors to provide an immersive user experience. Phone identifiers are now commonly used to uniquely identify users by apps that tailor their behavior to the user’s needs. This means that benign apps now use the same information that malicious apps gather. As a direct result, many of the exact same simple (source, sink) flows now exist in both malicious and benign apps.

**Contribution 3:** In general, the key to distinguish malicious apps and benign apps is to discover the difference of app behavior on sensitive data usage in apps. We propose a new representation of information flows, called Complex-Flows, for a more effective malware detection analysis. Simply put, a Complex-Flow is a set of simple (source, sink) flows that share a common portion of code in a program. For example, a program can read contact information, encrypt it, and store it in storage as well as send it over the Internet. This means that this program has two simple flows—a (contact, storage) flow and a (contact, network) flow—that share a common portion of code in the beginning of each flow (i.e., reading and encryption). Our Complex-Flow then represents both flows together as a set that contains both flows. Complex-Flows give us the ability to distinguish different flows with same sources and sinks based on the computation performed along the information flow as well as the structure of the flows themselves. We leverage this insight and develop a new classification mechanism for malware detection that uses Complex-Flows as the basis for classification features.

This thesis is organized as follows: we first review the literature and discuss different research techniques on Android security in Section 2. Second, we present detailed discussion on Flow Permissions and BlueSeal system in Section 3. Next, we show systematical study results on Android malware apps in Section 4. We then present detailed discussion on Complex-Flows and experimental results to show effectiveness on distinguishing Android malware apps from benign apps in Section 5. Conclusion are discussed in Section 6.
2 Literature Review

In this section, we review previous research work on Android security problems. We discuss how they ensure security on Android and show how our system is different from them.

2.1 Android Permissions, Analysis, and Tools

The growing popularity of Android has resulted in many tools, case studies, and analysis engines. Stowaway [29] is an automated tool that detects unused permission through static analysis and a pseudo-manually created permission map. Recently, PScout [13] released a permission map that was automatically generated via a static analysis of all the Android sources that was shown to be more precise than Stowaway’s. The most closely related work to ours is CHEX [46], which provides a tool for detecting highjack enabling flows within an app. It is the first tool to tackle analysis of Android’s constructs such as async tasks and handlers, though it uses a brute force permutation approach for disambiguation. Our call graph restructuring can refine CHEX’s approach since we identify implicit calls in Android’s constructs whenever possible. AndroidLeaks [64] is a static analysis tool implemented in WALA that can find leaks of sensitive information sent over the network from Android apps. It does not support analysis of async tasks, intents, nor content providers and is unable to track cross-app flows. SCanDroid [33] first proposed a methodology for analyzing intents statically, but was never tested on real-world apps. The approach also required the original Java source of the programs. Mann et al. created a framework to identify privacy leaks from the Android APIs [47], but the framework has not been evaluated on real-world applications. DroidChecker [21] is a static analysis tool aimed at discovering privilege escalation attacks and thus only analyzes exported interfaces and APIs that are classified as dangerous. ScanDal [43] is an abstract interpretation framework for tracking information flows within apps. Currently, their framework is able to track flows between location information, phone identifiers, camera, and microphone exported to the network and SMS.

Besides static analysis tools, there is a plethora of tools that perform dynamic analyses. Alazab et al. [10] provide a dynamic analysis technique that runs apps in a sandbox and can detect malicious apps. MockDroid [17] is a tool that protects users’ privacy by supplying mock data instead of sensitive data. Aurasium [67] provides user-level sandboxing and policy enforcement to dynamically monitor an app for security and privacy violations. Notably, Aurasium does not require modifications to the underlying OS. We believe that Aurasium is complementary to BlueSeal, as our Flow Permissions can provide a specification of possible malicious leaks. CrowDroid [18] is an offline analysis over traces that can be leveraged to identify malicious apps through examining their behavior via crowdsourcing. Moonsamy et al. [49] provided a thorough investigation and classification of 123 apps using static and dynamic techniques over the apps’ Java source code. Grace et al. [37] showed that ad frameworks opportunistically scan and leverage permissions granted by the app they are called from. AdDroid [53] introduced a new advertisement framework with privilege separation, accomplished through a new set of advertising APIs and permissions. We believe our tool can be extended to analyze their framework through extensions to the permission map BlueSeal takes as parameter. PiOS [24], a static analysis tool for iOS, leverages reachability analysis on control-flow graphs to detect leaks.

2.2 Android Malware Detection

There are many general malware detection techniques proposed for Android. Some of these leverage textual information from the app’s description to learn what an app should do. For example,
CHABADA [35] checks the program to see if the app behaves as advertised. WHYPER [52] leverages NLP techniques to analyze the app description from the market and a semantic model of a permission and determine why an app uses a permission. Meanwhile, AsDroid [41] proposes to detect stealthy malicious behaviors in Android apps by analyzing mismatches between program behavior and user interface. All these techniques rely on either textual information, declared permissions in the manifest file, or on specific API calls, while our approach focuses on analyzing app behaviors based on the app code related to device sensitive data.

Machine learning techniques are also very popular among researchers for detecting malicious Android apps. However, most of these solutions train the classifier only on malware samples and can therefore be very effective to detect other samples of the same family. For example, DREBIN [11] extracts features from a malicious app’s manifest and disassembled code to train their classifier, whereas MAST [20] leverages permissions and Android constructs as features to train their classifier. We believe these coarse features are great mechanisms to filter many apps prior to leveraging techniques like our own, which require more analysis of the app internals. There are many other systems, such as Crowdroid [19], and DroidAPI Miner [8], that leverage machine learning techniques to analyze statistical features for detecting malware. Similarly, researchers developed static and dynamic analyses techniques to detect known malware features. Apposcopy [31] creates an app signature by leveraging control-flow and data-flow analysis. RiskRanker [36] performs several risk analyses to rank Android apps as high-, medium-, or low-risk. Sebastian et al. [54] analyze dynamic code loading in Android apps to detect malicious behavior. [27], [22], and [38] are all signature-based malware detection techniques and are designed to detect similar malware apps.

2.3 Other Android Security tools

In addition, researchers have explored many other ways to ensure security on Android. These tools leverage techniques on different aspects to protect sensitive data, such as compositional analysis, sandbox mining, partitioning app code to handle confidential data or UI examination [57] [42] [56] [40].

3 Information Flow as a Permission Mechanism

In this section, we discuss our Flow Permission concept and BlueSeal system, our engine for static information flow analysis on Android. Flow Permissions are an extension to the Android permission mechanism that characterizes the implicit interactions between data and APIs protected by standard permission. We derive fine-grained information flows using static analysis. We also derive cross-app flows by combining per-app flows across different apps. This is done by matching one app’s sinks to another app’s sources.

3.1 Motivation

To motivate the necessity of extending the Android permission mechanism, we examine four apps in detail: MyCalendar, MySpace, Blackmoon File Browser, and Gmail. MyCalendar is a third-party calendar app, MySpace is a social networking app with multimedia support, Blackmoon File Browser is a popular file manager, and Gmail is a well-known email app from Google. Although these apps have widely varying functionality, MyCalendar and MySpace request similar permissions.
<table>
<thead>
<tr>
<th>Android App</th>
<th>Category</th>
<th>Permissions Requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>MyCalendar</td>
<td>Productivity</td>
<td>STORAGE, LOCATION, NETWORK, PHONE CALLS</td>
</tr>
<tr>
<td>MySpace</td>
<td>Social</td>
<td>STORAGE, NETWORK, PHONE CALLS</td>
</tr>
<tr>
<td>Blackmoon File Browser</td>
<td>Productivity</td>
<td>STORAGE, SYSTEM, TOOLS</td>
</tr>
<tr>
<td>Gmail</td>
<td>Communication</td>
<td>NETWORK, STORAGE</td>
</tr>
</tbody>
</table>

Table 1: Table listing Android apps and their requested permission examples.

A partial and stylized set of permissions each app requests is given in Table 1. Notice that MyCalendar and MySpace both request PHONE CALLS and NETWORK. The PHONE CALLS permission grants the app a set of more fine grained permissions, which we omit for brevity, including permissions to read the phone number, device ID, and the phone state. Similarly, the permission NETWORK allows the app to access the internet, either through wifi or cellular networking.

Savvy users may notice that by granting permissions to read from the phone’s log and phone state as well as access to the internet, they are also implicitly granting permission to transmit data stored within the call log and phone state over the internet to an external source. Once the app has permission to read from a given piece of data stored on the phone (i.e. a data source) as well as permission to send data outside of the app (i.e. a data sink), the app also implicitly has permission to export the source data via the sink. Importantly, the permissions offer no insight if the apps leverage the APIs to exfiltrate data.

3.1.1 Flows as Permissions

The goal of the Flow Permission mechanism is to show whether or not an app contains a flow between a source and a sink. The general structure of a Flow Permission is of: source → sink. From Table 2, we can see that, even though MyCalendar and MySpace are granted the same permissions (PHONE CALLS and NETWORK), MyCalendar is augmented by our tool to contain the Flow Permission: PHONE NUMBER → NETWORK. This Flow Permission indicates that data read from the stored phone number is subsequently exported through the use of the network. Additionally, we can deduce that MySpace does not contain such a flow as it does not report such a permission. The MySpace app does, however, transmit the International Mobile Equipment Identity (IMEI) number of the device, which is indicated by the IMEI NUMBER → NETWORK Flow Permission.

In this manner, Flow Permissions provide the user additional context on how the standard Android permissions and the resources / data they protect are leveraged by the apps. Nevertheless, it is up to the user to decide if these behaviors should be allowed or not. The existence of a flow does not indicate that the app is necessarily malicious. For example, a social networking app might be expected to contain a flow from the IMEI number to the network as this provides the app a mechanism to uniquely identify the device for analytics. However, some users may not be comfortable providing such information to the app developer, as other mechanisms (e.g. manual login screens) can be used without exposing such data. In contrast, a calendar app should not have such
Table 2: Table listing Android apps and our proposed Flow Permission extensions.

<table>
<thead>
<tr>
<th>Android App</th>
<th>Flow Permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>MyCalendar</td>
<td>PHONE NUMBER → NETWORK</td>
</tr>
<tr>
<td>MySpace</td>
<td>IMEI NUMBER → NETWORK</td>
</tr>
<tr>
<td>Blackmoon File Browser</td>
<td>STORAGE → NETWORK</td>
</tr>
</tbody>
</table>

3.1.2 Interaction Between Apps

Consider a more complicated case that highlights how multiple apps can expose data sources and sinks to one another, thereby acquiring additional implicit permissions [30]. The Blackmoon File Browser app includes functionality to send a file as an email attachment. However, the app cannot access the network to send an email as it does not have the NETWORK permission. Instead, the Blackmoon File Browser leverages Gmail’s public interface to send files over the network. In other words, the Blackmoon File Browser is implicitly granted permission, if Gmail is also installed, to use the network without overtly requesting such a permission. Flow Permissions, on the other hand, highlight the flow between the Blackmoon File Browser and the network, accomplished through the RPC mechanism leveraged to transmit the file, as shown in Table 2.

3.2 Flow Permissions

Flow Permissions are an extension to the Android permission mechanism that characterizes the implicit interactions between data and APIs protected by standard permissions. This interaction is determined by the existence of an information flow between the permission domains. Although there may be multiple Android permissions dealing with a domain (i.e. READ_SMS, WRITE_SMS, RECEIVE_SMS, and SEND_SMS, etc.), we only consider the domains themselves (e.g. SMS). Domains are split into three categories: source domains, which can be viewed as sources of data; sink domains, which can be viewed as data export mechanisms; and cross-app domains, domains which act as inputs or outputs between apps.

3.2.1 Permission Domain Types

We display these flows to users at installation time so that users can examine the flows present in an app. Since it is possible that an app has many flows, we categorize sources and sinks into domains to reduce the number of flows that need to be shown to the users.

Out of over 130 Android permissions, we have identified thirteen canonical source domains: SMS, STORAGE, HISTORY BOOKMARKS, USE DICTIONARY, FINE LOCATION, COARSE LOCATION, CALENDAR, ACCOUNTS, PHONE STATE, CONTACTS, CALL LOG, VOICEMAIL, and LOG. Similarly, we have identified five canonical sink domains: NETWORK, LOG, SMS, STORAGE, and INTENT. The third type of domain (cross-app) consists of Android’s IPC mechanisms that allow apps to share data or provide services to one another. For example, Gmail exposes its email service to other apps via an

---

1This number is used to uniquely identify the user, phone, and subscription plan. Networks use this to establish roaming policies and charges associated with non local network usage.
IPC mechanism as mentioned in Section 3.1.2. These IPC mechanisms can bridge a source domain in one app to a sink domain in another app.

3.2.2 Flow Permission Mechanism

Flow Permissions can be viewed as relations between the three types of permission domains. There are four potential types of Flow Permissions:

- **Source → Sink**: A flow from a source domain to a sink domain.
- **IPC → Sink**: A flow from an IPC source domain to a sink domain.
- **Source → IPC**: A flow from a source domain an IPC sink domain.
- **IPC → IPC**: A flow from an IPC source domain to an IPC sink domain.

Of the four types of Flow Permissions, those which deal with flows to and from IPC, are not reported directly to the user by default. Instead, these Flow Permissions, along with meta-data to disambiguate the IPC, are leveraged at installation time and during cross-app analysis to synthesize cross-app as well as deployment flows. Abstractly, cross-app and deployment flows are characterized by one app having a Flow Permission of the form: Source → IPC, another app having a Flow Permission of the form: IPC → Sink, and any number of apps having Flow Permissions of the form: IPC → IPC.

3.2.3 Statically Deriving Flow Permissions

In order to present Flow Permissions at installation time, we statically analyze Android apps to derive them. In doing so, we overcome a set of challenges unique to Android.

As with other GUI frameworks, Android’s programming model is highly event-driven. Many of the Android APIs are essentially event handler interfaces that an app needs to implement to handle various events via callbacks. The Android framework calls these event handlers not only for user-generated events such as a button click, but also for framework events such as app start, stop, and pause. At the minimum, an app is required to extend one framework component class that defines handlers for framework events. In addition, Android has introduced many new constructs, including new thread types (e.g., AsyncTask), messages and message handlers (e.g., Intent and Handler), and IPC mechanisms (e.g., Binder). We detail the usage scenarios of these constructs further in Section 3.3.

Android’s unique programming model and constructs present the following set of challenges for static analysis.

- All entry points for an app must be identified to leverage standard analyses like deadcode elimination. The Android standard library has over 1,700 possible entry points.
- Methods registered as callbacks and listeners for various external and internal events must be identified, and their invocation points tracked. Android allows callbacks and listeners to be registered not only in the app code, but also in configuration files. Thus, configuration files must also be analyzed.

2Our tool can be configured to emit these as well.

3Android defines four framework components—activities, services, broadcast receivers, and content providers. An app extends Activity to handle UI events; Service to perform background tasks; BroadcastReceiver to handle broadcast events (e.g., a battery low event) from either the Android framework or apps; and ContentProvider to provide a custom storage with a database-like interface.
• Android provides new classes and methods for inter-thread communication in a message passing style, necessitating the pairing of possible send and receive points of messages.

• The Android IPC mechanisms require disambiguation to distinguish which apps communicate to other apps and through which mechanisms.

• Android manages the execution of asynchronous tasks, implicitly invoking methods during specific points its lifetime. These methods must be handled explicitly and their implicit invocation sites discovered.

3.3 System Design

Our system design is built on top of the Soot Java Optimization Framework [62, 63]. Since Soot is originally developed for analyzing Java bytecode, Soot integrated the Dexpler Dex to Java bytecode translator [16] to transform Dex bytecode into Soot’s own intermediate representation (Jimple). In addition, we leverage the PScout Permission Map [13]; abstractly, a permission map is a mapping between Android API calls and the permissions required to enact those calls. The PScout Permission Map was generated by statically analyzing the entire Android source code and to our knowledge is the most complete among known permission maps. Our compiler leverages this precomputed mapping internally within the analyses to associate specific permission to API calls.

At its core, our BlueSeal leverages classic forward and backward intraprocedural dataflow analysis as well as interprocedural dataflow analysis based on graph reachability. As outlined in Fig. 1, BlueSeal leverages six main analysis passes to generate Flow Permissions: 1) entry point discovery, 2) call graph restructuring, 3) unused permission analysis, 4) resolution of intents, content providers, as well as uses of the binder, 5) interprocedural permission flow analysis, and 6) cross-app permission flow analysis. Abstractly, BlueSeal uses analyses 2, 3, and 4 to disambiguate Android specific constructs and identify source and sink points, prior to tracking flows between sources and sinks in analysis 5. Since BlueSeal is built from classic analysis techniques, we tailor our discussion on Android specific linguistic constructs, libraries, and IPC mechanisms and how to modify standard analyses to support them. Currently, BlueSeal is not path or context sensitive. BlueSeal implements Stowaway’s unused permission analysis [29] to remove unnecessary permissions, the details are omitted for brevity.

3.3.1 Entry Point Discovery

The Android platform is event driven and almost all apps have multiple entry points. Prior to static analysis, precise entry point detection should be performed to improve precision. As we describe below, there are three ways that an app can register entry points and our entry point discovery covers all three cases. Our approach complements the entry point analysis given in CHEX [46] with support for entry points specified in layout configuration files as well as discovering potential entry points from the Android API documentation.

• Framework Components Any Android app is required to implement one of the four main framework components (Activity, Service, BroadcastReceiver, and ContentProvider). These components have standard entry points and are declared in the manifest. BlueSeal discovers framework components’ entry points by analyzing this file.

• UI Layout A developer can also declare entry points handling UI events such as button clicks in layout configuration files. Discovering these entry points requires extra analysis
on the layout configuration files; this is due to the fact that an app can contain multiple layout configuration files, one for each layout it uses. Android internally maintains the mappings between layouts and their configuration files by generating another configuration file at compile time. BlueSeal analyzes this internal configuration file to match classes corresponding to the layouts the app uses, each of which is identified by unique int, to handlers defined in that layout’s configuration file.

- API Callbacks The third entry point option is implementing callbacks pre-defined in the Android APIs. In order to discover these pre-defined entry points, we have implemented a crawler that parsed the API documentation and discovered that the current Android API documentation (API 17) has 1,738 callback methods that can serve as potential entry points.
public class MainActivity extends Activity {
    protected void onCreate(Bundle savedInstanceState) {
        ...
        new Task().execute("http://www...");
        ...
    }
    ...
    private class Task extends AsyncTask<String, String, Integer> {
        ...
        protected void onPreExecute() {
            ...
        }
        protected Integer doInBackground(String... strs) {
            ...
            publishProgress("intermediate result");
            ...
            return intObj;
        }
        protected void onProgressUpdate(String... strings) {
            ...
        }
        protected void onPostExecute(Integer intObj) {
            ...
        }
    }
}

Figure 2: A code snippet illustrating the methods that comprise the control flow of an async tasks in Android and the implicit flow of arguments provided by the Android framework.

UI Thread

Implicit Thread

MainActivity.onCreate()
...
Task.execute(...);
...
Task.onPreExecute()
...
Task doInBackground()
...
Task.onProgressUpdate()
...
Task.onPostExecute()
...
Task onPostExecute()
...

Figure 3: The execution flow of async task methods in their respective threads at runtime.

3.3.2 Call Graph Restructuring

The Android framework is responsible for implicitly invoking methods associated with many of the constructs it provides. To correctly analyze an app, we must infer the association of user-called methods to their corresponding framework-invoked methods. We discuss the two most common cases below.

- **Async Tasks** AsyncTask is a new threading class introduced in Android. It provides a simple way to write a short lived thread that communicates with the UI thread in an asynchronous fashion. An AsyncTask can implement five methods—onPreExecute, doInBackground, onProgressUpdate, onPostExecute, and onCancelled, which dictate the control flow of the asynchronous task. As an example, consider the code snippet in Fig. 2 and the cor-
responding control flow given in Fig. 3. The doInBackground method performs the actual computation for the async task. The methods onPreExecute and onPostExecute run before and after doInBackground and typically include pre- and post-processing. The onCancelled method is called when the async task is cancelled by another thread. Notice that onPreExecute will execute in the implicitly created thread backing the asynchronous task, but onPostExecute callback will be executed by the UI thread. Similarly, onProgressUpdate gets executed as a callback in the UI thread after there is a call to publishProgress within doInBackground. An app writer can call AsyncTask's execute and executeOnExecutor to start an AsyncTask. Obviously, a typical call graph generation process does not understand this execution flow; hence, we identify all AsyncTask instances and augment the call graph to include edges corresponding to the async task control flow. We do this by effectively replacing the invoke of execute with invoke calls to onPreExecute, doInBackground, and onPostExecute. Similarly, a call to publishProgress is replaced with a onProgressUpdate call. Notice that doInBackground implicitly passes its return value as an argument to onPostExecute. publishProgress also passes its arguments as arguments to onProgressUpdate. The call graph and method bodies are updated accordingly.

- Handler Android also provides a message mechanism for communicating between threads within an app, called Handler (depicted in Fig. 4). Threads can communicate through a shared Handler object. Receiving threads implement the handleMessage method to process received messages and sending threads communicate through the sendMessage* family of methods. Similar to async tasks, BlueSeal effectively replaces a call to sendMessage* with a call to handleMessage to restructure the call graph.

### 3.3.3 Content Provider Resolution Analysis

After restructuring the call graph, BlueSeal performs additional analyses to identify permission domains discussed in Section 3.2. One mechanism for interaction between apps is the Content Provider (CP). An app can provide content to itself or other apps, can consume content hosted by a CP, or both. CPs are uniquely identified by an URI object (android.net.Uri) and to correctly pair uses of CPs these objects must be tracked and disambiguated to the extent possible by static analysis. To identify uses of CPs we track the Content Provider API calls as well as the URI Objects (as shown in Fig. 5). Our CP Resolution Analysis (CPRA) is based on an interprocedural dataflow analysis that leverages a backward intraprocedural dataflow analysis that leverages a backward intraprocedural dataflow analysis. Abstractly, we track backward flows from uses of the CP mechanism to the definitions of URI objects and from the definitions of URI objects to the strings that uniquely identify them.

CP are accessed through two separate classes in Android: ContentResolver and ContentProviderClient. Within these classes the methods from which we begin tracking flows are: insert, query, and update. Each of these methods takes an URI object as an argument. Our analysis identifies the creation points of the URI objects passed into these methods. URI objects can be created in one of two ways: they can be provided by the Android libraries or they can be constructed by the app itself. In the former, the identifying URI string is hidden. For precision, our analyses leverages PScout, which also provides a mapping between framework provided URI objects and their URI strings as shown in Fig. 5. For app created URI objects we attempt to discover this information in the compiler. Once the app created URI object is created, BlueSeal can effectively replace a call to sendMessage with a call to handleMessage to restructure the call graph.

---

11

---

4By method family we mean any methods of similar form defined by the same class (e.g. setData and setDataAndType belong to the method family setData*).
public class HandlerActivity extends Activity {
    ...
    private Handler mHandler = new Handler() {
        public void handleMessage(Message msg) {
            Bundle data = msg.getData();
            ...
        }
    };
    public void onClick(View v) {
        new Thread(new Runnable() {
            public void run() {
                try {
                    msg.setData(data);
                    mHandler.sendMessage(msg);
                    ...
                } catch (InterruptedException e) {
                    ...
                }
            }
        }).start();
    }
}

Figure 4: Flows based on pairing message sends to the appropriate message handlers.

public class ContentProviderExampleActivity extends Activity {
    ...
    public void onButtonClick(View v) {
        Uri uri = ContactsContract.Contacts.CONTENT_LOOKUP_URI;
        ...
        results = getContentResolver().query(uri, ...);
        ...
    }
    ...
}

Figure 5: The data flow of an URI object that identifies which CP is being utilized. Dashed arrows indicate information derived from dataflow analyses and block arrows how that information is used to disambiguate the CP.

identified, the analysis tracks the construction of this object. There are two ways to construct an URI object. One is to use Uri.parse and the other is to use Uri.Builder. The first case is simple as the argument to the parse method is the URI string. If Uri.Builder is used, then Uri.Builder.scheme is used to set the scheme and Uri.Builder.authority is used to set the authority. For example, ‘content://edu.buffalo.cse.provider’, is a valid CP identifier where the authority is ‘edu.buffalo.cse.provider’ and ‘content’ is the scheme. After the scheme and the authority are set, Uri.Builder.build returns the actual URI object. Thus, our analysis tracks calls
public class UriExampleActivity extends Activity {
    private final Uri mUri;
    ...
    protected void onCreate(Bundle savedInstanceState) {
        mUri = buildUri("content", "edu.buffalo.cse.provider");
        ...
    }
    ...
    private Uri buildUri(String scheme, String authority) {
        Uri.Builder uriBuilder = new Uri.Builder();
        uriBuilder.scheme(scheme);
        uriBuilder.authority(authority);
        return uriBuilder.build();
    }
    ...
}

Figure 6: The data flow of an URI object initialization that is resolvable statically.

public class IntentExampleActivity extends Activity {
    ...
    public void onClick(View v) {
        Intent intent = new Intent(Intent.ACTION_SEND);
        intent.putExtra(Intent.EXTRA_TEXT, "Test String");
        ...
        startService(intent);
    }
    ...
}

public class IntentExampleService extends Service {
    ...
    public int onStartCommand(Intent intent, int flags, int startId) {
        Log.v("value", intent.getExtras().getString(Intent.EXTRA_TEXT));
        ...
    }
    ...
}

Figure 7: The data flow of an Intent passed from one app to another.

3.3.4 Intent Resolution Analysis

Intents are message objects that can be used to send data between components within a single app as well as across different apps. An app can receive intents in two ways, either statically or dynamically. Static intents are declared in the app’s manifest file on a per component basis. An app can also register itself to receive intents dynamically at run time without declaring it in its manifest file.

To resolve dynamic intents on the receiver side, our Intent Resolution Analysis (IRA) must first discover all classes that are registered to receive the intent via Context.registerReceiver. The call to registerReceiver requires an intent filter that identifies which intents the class is able to receive. Intents that are to be received by the filter can be specified at initialization time via the intent filter constructor or dynamically via the add method. In much the same was as CPRA, IRA also performs an inter-procedural backwards flow analysis to disambiguate between intents by tracking strings. An example is shown in Fig. 7.
Once the intents are disambiguated, the analysis must identify possible sources and sinks related to the intents. Intent sinks are identified by any API call that inserts data into the intent. In Android, this can be done through the intent constructors `Intent(...)` as well as the method call families `put*Extra` and `setData*`. Intent sources are identified by any API call that retrieves data from the intent, namely the method family `get*`. Intents themselves can be sent between apps. Sinks related to the intents used in this manner are the methods that can send out intents to other apps and include the following method families: `send*Broadcast*`, `startActivity*`, `startIntentSender`, `startService`, and `stopService`. All the methods listed above require an `Intent` object. Similar to intent filter, there are two main ways to specify an action string that uniquely identifies the intent. First is at the initialization time via the constructor. The second is by using `setAction`. IRA does not track what data flows through or from intents, it only serves to identify and disambiguate how the app leverages intents. Effectively, IRA computes points in the intermediate representation that act as sources and sinks for the subsequent compiler passes.

3.3.5 Binder Resolution Analysis

Binder/IBinder, commonly referred to as just Binder, is the default IPC mechanism on Android. It can be used for inter-component communication within the same app (e.g., activity-to-service communication) as well as inter-process communication between different apps. Android provides multiple ways to use the Binder mechanism, such as simply extending the base Binder class or using AIDL (Android Interface Definition Language) to define a customized interface. Regardless of which method is used, a Binder server (i.e., an IPC callee) implements all the IPC methods in the Binder class. A Binder client (i.e., an IPC caller) uses an IBinder object which is the proxy for the server-side Binder. Fig. 8 shows an example.

Although Binder calls are mostly identical to local calls, there are two cases to handle for correctness of our analysis. First, for inter-component communication, we need to match each call with an IBinder object to the corresponding Binder implementation. Second, for inter-process communication, each client-side IBinder call is a potential sink, which might result in a server-side Binder call which then becomes a potential source.

A variation of Binder is Messenger, which allows a process to send a message to another...
process. It relies on Binder/IBinder to implement its functionalities underneath, but is simpler to use from the programmer’s point of view. In order to receive a message, a server needs to create a Messenger object; it also needs to implement a Handler as described in Section 3.3.2 and pass it to the Messenger object. In order to send a message, a client can use Messenger’s send method. We handle these implicit calls by matching calls to send with Handler’s handleMessage. If matches cannot be enumerated we treat them as a potential sink (for send) or a source (for handleMessage).

3.3.6 Interprocedural Permission Flow Analysis

To synthesize Flow Permissions we leverage an interprocedural forward flow analysis to track flows between sources and sinks. Our analysis is fixed point based, leveraging the standard work list model and method summaries. The flow analysis is parameterized by a listing of sources and sinks. Sources and sinks are specified directly from API calls via the PScout Permission Map or synthesized by CPRA, IRA, and BRA. Sources and sinks synthesized by CPRA, IRA, and BRA correspond to uses of IPC. The goal of this analysis is to track data flows originating at sources and terminating at sinks.

- **Computing and Applying Method Summaries** The intraprocedural forward flow analysis, leveraged by our interprocedural analysis, builds a method summary for each reachable method. The intraprocedural analysis is standard and builds in-flow and out-flow sets for each statement in the method body. The method summary constructed during this analysis is a flow graph representing the flows between sources and sinks within the method itself as well as arguments, returns, and class variables the method reads or writes. We add nodes to the graph for every argument, return statement, statement containing a class variable read/write, and statements identified as sources or sinks. Edges between nodes are added when a flow is determined by the intraprocedural forward flow analysis. Argument nodes and source nodes can have only outgoing edges. Sinks and return nodes can have only incoming edges. Nodes which represent class variable reads/writes can have both incoming and outgoing edges. Thus, there are four types of possible flows contained within the flow graph comprising the method summary: 1) generative flows: flows from a source to a return or class variable, 2) terminating flows: flows from an argument or class variables to a sink, 3) local flows: flows from a source to a sink, and 4) transitive flows: flows from arguments or class variables to other class variables or returns. Orphan nodes, nodes with no incoming or outgoing edges, are pruned.

At a method call site the analysis applies the summary for that method. If the method summary contains transitive flows, we add the arguments supplied at the call site to the out-flow set for the call. For both generative and terminating flows, we add a place holder node into the method summary. This placeholder node represents potentially multiple sources and/or sinks, one for each generative and terminating flow. The place holder nodes will be used to synthesize a global flow graph once all method summaries have been computed and the interprocedural analysis reaches a fixed point. Edges between nodes in the flow graph and the place holder node are added as if the place holder node was a source and/or sink node.

- **Synthesizing a Global Flow Graph** Once the interprocedural analysis reaches a fixed point, we synthesize a global flow graph from the per method summaries. Place holder nodes that were inserted when method summaries were applied and class variable nodes serve as merge points for combining method summaries. Once all method summaries are merged, paths that do not originate from a source and terminate in a sink are pruned. Flow Permissions can be generated from the graph by enumerating all paths and removing duplicates.
(e.g. an app may send contact data over the network in multiple code blocks). Lastly, we remove any Flow Permissions that correspond to permissions the app does not request. This step is necessary because ad libraries [37] check to see which permissions an app has been granted and perform computation based on these permissions. Thus, an app may contain code that contains flows, but will never be executed at runtime. In general, any flow that requires a permission the app has not been granted cannot be executed at runtime. The Flow Permissions are then added to the manifest file for the app.

Special consideration must be given to apps that leverage Android’s shared user ID mechanism. This mechanism allows for multiple apps to execute as a single process. This process is granted the union of the permissions requested by the apps. In this case, we do not remove any Flow Permissions, regardless if the app requests the necessary permissions or not.

• Cross App Permission Flows Analysis Cross-app permission flow analysis simply enumerates permutations of pairs of Flow Permissions between apps such that one app has a source to IPC flow and another has a IPC flow to a sink, and the IPC mechanism is the same. In the case where an IPC is not disambiguated statically in either app, our analysis is conservative and assumes any IPC mechanism of the same type can be potentially utilized. For file reads/writes to external storage we track the file name(s) if they are deducible statically. Notice that when an app is installed on a phone, its Flow Permissions and associated meta-data (e.g. types and identifiers of IPCs) can be compared to the Flow Permissions of installed apps to synthesize implicit deployment Flow Permissions.

3.3.7 Augmented Package Installer

In order to display Flow Permissions and perform cross-app analysis, we have augmented Android’s package installer in three aspects. First, we have added all our Flow Permissions to the source manifest file (the framework’s AndroidManifest.xml) that the package installer accesses during installation. This is the global list of all permissions available in the system. Second, we display our Flow Permissions by modifying the package installer. Fig. 9 shows an example of how we display Flow Permissions to users.

Lastly, we implement cross-app analysis in the package installer in its PackageInstaller-Activity. In our implementation we synthesize cross-app permissions by performing all-to-all matching between all existing flows from already-installed apps, and flows from the app being installed. For example, while installing app2, if app1 (already installed) has a flow from the device ID to a file, and app2 has a flow from the same file to a socket, a new cross-app Flow Permission, PHONE_STATE → NETWORK, is created and displayed. A similar matching is done between the derived sources and sinks of already-installed apps. For this analysis, our new package installer stores all permissions from all already-installed apps in its storage. Our performance results in Section 3.4 demonstrate that our all-to-all comparison is still practical and feasible to run on a smartphone.

3.4 Results and Discussion

To validate our approach, we tested BlueSeal on 2,992 of the top-rated free apps available on the Google Play Store, with 571 apps from January 2013, 2,421 apps from January 2014, and on 1,047 known malicious apps from the MalGenome Project 5 [69]. We ran BlueSeal on Amazon EC2 [6] using an 13-ECUs and 4-vCPUs node instance with 15GB of RAM. In the set of apps, there are 107

---

5http://www.malgenom project.org.
apps not analyzed because the Soot framework, which BlueSeal relies on, threw exceptions when performing intermediate representation transformation. BlueSeal, thus, was able to analyze 2,885 apps. Our full data set and results can be found at http://blueseal.cse.buffalo.edu/data.html.

The main purpose of our evaluation is to assess the analysis capability and usefulness of BlueSeal. We do this in four ways. First, we present aggregated as well as categorized statistics regarding information flows and Flow Permissions. Second, we present the analysis performance of BlueSeal. Third, we discuss the limitations of BlueSeal with manual validation. Fourth, we show the usefulness of BlueSeal with a user survey.

3.4.1 Statistics of Flow Permissions

Table 3 shows the overview of our results. Although we detect many information flows in an app, the number of Flow Permissions is significantly smaller due to our domain categorization
Results Overview

<table>
<thead>
<tr>
<th></th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total apps</td>
<td>2,885</td>
</tr>
<tr>
<td>Total number of raw flows detected</td>
<td>631,152</td>
</tr>
<tr>
<td>Avg. number of raw flows per app</td>
<td>218.77</td>
</tr>
<tr>
<td>Distinct flows</td>
<td>2374</td>
</tr>
<tr>
<td>Total number of Flow Permissions</td>
<td>17,332</td>
</tr>
<tr>
<td>Avg. number of Flow Permissions per app</td>
<td>6.01</td>
</tr>
<tr>
<td>Distinct Flow Permissions</td>
<td>431</td>
</tr>
</tbody>
</table>

Table 3: A brief overview of experiment results.

Figure 10: Distribution of Flow Permissions generated for each app. 630 apps do not generate any Flow Permission and are not shown.

(described in Section 3.3). Fig. 10 and Fig. 11 show a more detailed view of the flows and Flow Permissions generated by BlueSeal. Most of the apps contain less than 15 Flow Permissions, an amount practical for a user to examine. The app with the maximum number of Flow Permissions (45) heavily uses different content providers, currently distinguished based on their URIs. We plan on categorizing content providers in much the same way as permission domains. The app that contains the maximum number of raw flows (13,646 flows as shown in Fig. 11) generates twenty Flow Permissions.

Table 4 shows the ten most common flows observed from normal apps and Table 5 shows the ten most common flows observed in malicious apps. From these two tables, we make four observations. First, normal apps are more concerned about users’ input data since most normal apps require a user’s login information and many provide social communication functionality that requires a lot of user input. On the other hand, malicious apps are heavily interested in the phone’s unique identifier, DeviceId. Second, both normal and malicious apps read system content providers often. The most commonly accessed content provider in both normal and malicious apps is the contacts. Third, normal apps access the phones’ location data more frequently than malicious apps. Fourth, most of the flows in normal apps indicate that data is often used inside
Figure 11: Distribution of raw flows generated for each app. (1046 apps range from 100 to 13,646 and 630 apps with 0 flow are omitted.)

Table 4: Ten most common flows in normal apps.

<table>
<thead>
<tr>
<th>Count</th>
<th>Raw Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>100379</td>
<td>EditText:getText→Intent:putExtra</td>
</tr>
<tr>
<td>49910</td>
<td>ContentResolver:query→Intent:putExtra</td>
</tr>
<tr>
<td>18512</td>
<td>EditText:getText→Log:e</td>
</tr>
<tr>
<td>16937</td>
<td>Location:getLatitude→Intent:putExtra</td>
</tr>
<tr>
<td>16756</td>
<td>Location:getLongitude→Intent:putExtra</td>
</tr>
<tr>
<td>15135</td>
<td>EditText:getText→Log:d</td>
</tr>
<tr>
<td>10660</td>
<td>ContentResolver:query→Log:e</td>
</tr>
<tr>
<td>9191</td>
<td>Location:getLongitude→DataOutputStream:writeShort</td>
</tr>
<tr>
<td>9173</td>
<td>Location:getLatitude→DataOutputStream:writeShort</td>
</tr>
<tr>
<td>9053</td>
<td>EditText:getText→PrintStream:println</td>
</tr>
</tbody>
</table>

the app, while in malicious apps data mainly flows to the network and storage. This observation suggests that normal apps leverage the sensitive data for debugging purposes while malicious apps may store or send the data. Last, the two location flows in Table 4 show that location is often sent within intents; this is in fact a common design pattern that many developers use. An app gets a location update, wraps it in an intent, and sends the intent to itself to display the update. BlueSeal currently does not distinguish whether or not an app sends intents to itself; it just detects that there is a flow to an intent that gets sent out. To improve BlueSeal’s precision, we can leverage existing techniques such as the ones implemented in Epicc [51].

In Tables 6 and 7, we collect top ten sources used by normal and malicious apps, respectively. As shown in these two tables, there are three main categories of sources frequently accessed by
both normal and malicious apps—system content providers, phone identifier, and location. Specifically, normal apps use the users’ location data while malicious apps read the phone identifier and data stored in system content provider, which is often private (e.g., contacts).

Table 8 and Table 9 show the top ten sinks present in normal and malicious apps, respectively. We observe that there are mainly three categories of sinks in normal apps: log, storage, and intent. In malicious apps, the top sinks are log, intent, storage, and network. Logging, which is used for debugging purposes, is the most frequently used sink in both normal and malicious apps.

### 3.4.2 Uses of the Statistics

Using these statistics, we can make observations about the difference between malicious apps and normal apps in terms of information flows. For this purpose, we have classified malicious apps into different categories according to their malware type, e.g., AnserverBot, BeanBot, DroidKungFu3, DroidKungFu4, GoldDream, etc. We then examined what common or distinct flows exist across the categories. A visualization of this data can be found for all categories at http://blueseal.cse.buffalo.edu/flows.png.

With this flow analysis, we have discovered that there are a few flows that highlight the difference between malicious apps and normal apps. Table 10 shows the number of apps in malicious

<table>
<thead>
<tr>
<th>Count</th>
<th>Source Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>205706</td>
<td>EditText:getText</td>
</tr>
<tr>
<td>123870</td>
<td>ContentResolver:query</td>
</tr>
<tr>
<td>77009</td>
<td>Location:getLatitude</td>
</tr>
<tr>
<td>76105</td>
<td>Location:getLongitude</td>
</tr>
<tr>
<td>43911</td>
<td>LocationManager:getLastKnownLocation</td>
</tr>
<tr>
<td>37561</td>
<td>TelephonyManager:getDeviceId</td>
</tr>
<tr>
<td>13820</td>
<td>FileInputStream:read</td>
</tr>
<tr>
<td>5816</td>
<td>AccountManager:getAccountsByType</td>
</tr>
<tr>
<td>4296</td>
<td>AutoCompleteTextView:getText</td>
</tr>
<tr>
<td>3815</td>
<td>TelephonyManager:getLine1Number</td>
</tr>
</tbody>
</table>

Table 6: Top-10 sources in normal apps.
<table>
<thead>
<tr>
<th>Count</th>
<th>Source Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>6697</td>
<td>TelephonyManager:getDeviceId</td>
</tr>
<tr>
<td>4159</td>
<td>TelephonyManager:getSubscriberId</td>
</tr>
<tr>
<td>3939</td>
<td>ContentResolver:query</td>
</tr>
<tr>
<td>1190</td>
<td>Location:getLatitude</td>
</tr>
<tr>
<td>1188</td>
<td>Location:getLongitude</td>
</tr>
<tr>
<td>1169</td>
<td>LocationManager:getLastKnownLocation</td>
</tr>
<tr>
<td>1114</td>
<td>TelephonyManager:getLine1Number</td>
</tr>
<tr>
<td>1106</td>
<td>FileInputStream:read</td>
</tr>
<tr>
<td>883</td>
<td>EditText:getText</td>
</tr>
<tr>
<td>446</td>
<td>TelephonyManager:getCellLocation</td>
</tr>
</tbody>
</table>

Table 7: Top-10 sources in malicious apps.

<table>
<thead>
<tr>
<th>Count</th>
<th>Sink Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>221831</td>
<td>Intent:putExtra</td>
</tr>
<tr>
<td>56584</td>
<td>Log:e</td>
</tr>
<tr>
<td>48814</td>
<td>Log:d</td>
</tr>
<tr>
<td>41747</td>
<td>DataOutputStream:writeShort</td>
</tr>
<tr>
<td>30808</td>
<td>DataOutputStream:writeUTF</td>
</tr>
<tr>
<td>20801</td>
<td>PrintStream:println</td>
</tr>
<tr>
<td>20065</td>
<td>OutputStream:write</td>
</tr>
<tr>
<td>19590</td>
<td>DataOutputStream:write</td>
</tr>
<tr>
<td>15576</td>
<td>Log:i</td>
</tr>
<tr>
<td>14523</td>
<td>Log:w</td>
</tr>
</tbody>
</table>

Table 8: Top-10 sinks in normal apps.

and normal categories. It summarizes the flow analysis result for one specific flow—the flow from TelephonyManager:getLine1Number to (network)OutputStream:write. This flow is used to ex-filtrate the user’s phone number using Java’s Socket IO. We observe that this flow is only used in 11 out of the 2198 normal apps which have more than one flow. This flow exists with a much greater frequency in the DroidKungFu3 and DroidKungFu4 categories. Further, we have found out that 3 out of the 11 normal apps with this flow have been removed from the Play Store since we originally downloaded them; these apps are com.SuperQiang.SexyGirlWallpaper.apk, com.SuperQiang.SexyGirlWallpaper3.apk, and Muli.touch.Sex10009.apk.

These individual flows can also be grouped to make a more specific match. Table 11 shows the number of apps in each malicious category and among all normal apps for the flow pair from TelephonyManager:getLine1Number to (network)OutputStream:write and the flow from TelephonyManager:getDeviceId to HttpClient:execute. The former flow is the same flow that we used in Table 10. The latter flow sends the device ID over to the network with an HTTP API. By flow pair we mean two flows that are held within the same app. In this case, each app counted contains a flow that is capable of sending the phone number off the phone using the Java Socket IO methodology and is capable of sending the phone’s device ID over the network using Android’s HttpClient. We observe that this flow pair is found in 6 out of 2198 normal apps which have any flow. Three of these apps were the same three that had been removed from the Play Store since our initial download. Of the remaining three, one is the free version of a popular guitar simula-
<table>
<thead>
<tr>
<th>Count</th>
<th>Sink Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>6965</td>
<td>Log:d</td>
</tr>
<tr>
<td>2793</td>
<td>Log:e</td>
</tr>
<tr>
<td>1809</td>
<td>Log:i</td>
</tr>
<tr>
<td>1330</td>
<td>Intent:putExtra</td>
</tr>
<tr>
<td>1200</td>
<td>HttpClient:execute</td>
</tr>
<tr>
<td>1197</td>
<td>ContentResolver:insert</td>
</tr>
<tr>
<td>1176</td>
<td>ByteArrayOutputStream:write</td>
</tr>
<tr>
<td>973</td>
<td>OutputStream:write</td>
</tr>
<tr>
<td>939</td>
<td>Log:v</td>
</tr>
<tr>
<td>897</td>
<td>FileOutputStream:write</td>
</tr>
</tbody>
</table>

Table 9: Top-10 sinks in malicious apps.

<table>
<thead>
<tr>
<th>Category</th>
<th>Distinct Count</th>
<th>Total Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnserverBot</td>
<td>1</td>
<td>172</td>
</tr>
<tr>
<td>BeanBot</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>DroidKungFu3</td>
<td>189</td>
<td>236</td>
</tr>
<tr>
<td>DroidKungFu4</td>
<td>73</td>
<td>82</td>
</tr>
<tr>
<td>GoldDream</td>
<td>3</td>
<td>40</td>
</tr>
<tr>
<td>Normal</td>
<td>11</td>
<td>2198</td>
</tr>
</tbody>
</table>

Table 10: Distinct malicious and normal apps using a flow, TelephonyManager:getLine1Number to (network)OutputStream:write.

...tor app. The other two apps, com.applock1 and com.mm.security.androidhider1, require root access and AVG Threat Labs indicate that they are malware. In the DroidKungFu4 category, the flow pair was found with the greatest frequency. More flows, in increasingly larger groups, can be used to narrow down the likelihood of an app being malicious, and possibly what malicious category the app would pertain to.

Certain flows can be even more telling of malicious activity. In many of the categories, flows with a source of SmsManager:sendTextMessage were found to be malicious, which can be seen in Table 12. One app in our normal group that used this as a source stood out as well, com.bluecode.photo.space.effects.fx.apk. This app is stated to add space effects to one’s photos, and this flow may be construed as a way to send photos to friends via text messages. This assumption would be false though, as photos need to be sent via MMS using a built-in MMS app’s API. While we wanted to test this app further, it has since been removed from the Play Store, with AVG’s Threat Labs [5] reporting that this app contained adware.

3.4.3 Performance

BlueSeal is able to analyze and synthesize per-app Flow Permissions for all but the largest apps in under ten minutes. Only 163 apps require an analysis time greater than ten minutes (all of these 163 apps finish under 30 minutes). Fig. 12 shows the full performance results. However, Soot’s front-end Dex bytecode parser, Dexpler, has limitations and generates incorrect intermediate representations for 107 of the apps. These apps are all from the Google Play store. We are currently investigating the causes of the mis-translation of the remaining 107 apps.
<table>
<thead>
<tr>
<th>Category</th>
<th>Distinct Count</th>
<th>Total Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnserverBot</td>
<td>1</td>
<td>172</td>
</tr>
<tr>
<td>BeanBot</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>GoldDream</td>
<td>3</td>
<td>40</td>
</tr>
<tr>
<td>Normal</td>
<td>6</td>
<td>2198</td>
</tr>
<tr>
<td>DroidKungFu4</td>
<td>72</td>
<td>82</td>
</tr>
<tr>
<td>DroidKungFu3</td>
<td>188</td>
<td>236</td>
</tr>
</tbody>
</table>

Table 11: Distinct malicious and normal apps using a flow pair compared to distinct apps within a category.

<table>
<thead>
<tr>
<th>Category</th>
<th>Distinct Count</th>
<th>Total App Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bgserv</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CoinPirate</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DogWars</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DroidKungFu3</td>
<td>3</td>
<td>236</td>
</tr>
<tr>
<td>Endofday</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Geinimi</td>
<td>4</td>
<td>25</td>
</tr>
<tr>
<td>GamblerSMS</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>GPSSMSSpy</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>NickyBot</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Normal</td>
<td>20</td>
<td>2198</td>
</tr>
<tr>
<td>SMSReplicator</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Walkinwat</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 12: Distinct malicious and normal apps using the SmsManager:sendTextMessage sink.

To measure the performance of our cross-app analysis, we have tested the installation performance of BlueSeal’s augmented package installer with 44 random apps. In our experiment, we have installed each app on a Galaxy Nexus phone, one app at a time, without uninstalling previously-installed apps. We have measured the cross-app analysis time as well as the total time until the installer displays the installation screen. As Fig. 13 shows, it takes less than 2.1 seconds for all the apps to analyze cross-app permissions and display the installation screen. Synthesizing cross-app flows does not exceed 0.25 seconds for any of the app installs.

3.4.4 Manual Validation

False positives and false negatives are well-known limitations of static analysis, which also apply to BlueSeal. Thus, we manually validate BlueSeal in three ways to understand its limitations. First, we have compared against TaintDroid [25]—a custom Android OS that performs a dynamic taint analysis for identifying malicious flows. We have manually compared BlueSeal’s generated Flow Permissions to TaintDroid’s dynamically discovered taints on thirty apps. Each app was manually executed for 15 minutes and fed random key-presses. Unsurprisingly, the most common taints reported by TaintDroid mirrored our own findings and that of prior work. We have not discovered any taints reported by TaintDroid for which BlueSeal does not generate a corresponding Flow Permission.

Second, we have randomly chosen and inspected 100 apps that have exactly one BlueSeal-
reported flow. We have examined each app’s intermediate representation of the source and verified that BlueSeal detects actual flows in all 100 apps.

Lastly, we have vetted BlueSeal against DroidBench, an Android benchmark suite with 64 apps provided by FlowDroid [12]. BlueSeal can detect all the flows in DroidBench except implicit flows, since BlueSeal currently does not handle implicit flows. In addition, there are several apps for which BlueSeal reports false positives. This occurs for one of three reasons—flows in dead code, lack of context sensitivity, and flows in complex data structures.

BlueSeal reports flows in dead code since it does not perform any dead code analysis. BlueSeal simply relies on Soot to determine code reachability. Also, BlueSeal is not currently context sensitive; for example, if there is a flow from a source to a class variable, and another flow from the same class variable to a sink, BlueSeal reports that there is a flow, regardless of the relationship of the call sites in which those flows occur. Lastly, if an app has a flow from a source to a complex Java data structure such as HashMap, and a flow from the same data structure to a sink, BlueSeal reports that there is a flow. As a result, BlueSeal reports one flow in 12 apps in DroidBench that do not have any flow.

3.4.5 Flow Frequency in Malicious Apps

It is well-known that there are many different types of malware disguised as Android apps. In the MalGenome project, the authors have divided the 1,047 malicious apps they collected into different categories based on their malware types (e.g. DroidKungfu). In this section, we examine the flows frequently appearing in different categories classified in the MalGenome project.

Fig. 14 depicts the flow frequency of each category. For clarity of presentation, we use flow labels in Fig. 14 and show the label-flow mappings in Table. 13. In Fig. 14, the x-axis shows the categories, and the y-axis shows the flows. The frequency metric we use is defined by the number

![Figure 12: Scatter plot showing the time taken to analyze all apps in seconds.](image)

---

[85x41]0
[0x0]500
[0x0]1000
[0x0]1500
[0x0]2000
[0x0]2500
[0x0]3000
[0x0]3500
[0x0]4000

[333x16]AppId

Anaylsis Time(second)

Performance

Figure 12: Scatter plot showing the time taken to analyze all apps in seconds.
of apps within a category that have a particular flow, divided by the total number of apps in that category. The size of a dot indicates the frequency; the bigger the dot is, the more frequent the flow appears in that category. As a baseline for comparison, we have also analyzed normal apps and show the result in the left-most side of Fig. 14. We highlight one example in Fig. 14, where we use a horizontal line to show a flow, TelephonyManager: getDeviceId → Log:e; also, we use a vertical line to show a category (Coinpirate). The size of the dot indicates that the flow appears much more frequently in the Coinpirate category than in the normal apps.

Fig. 15 shows a restricted view, where we just show the flows that read a phone identifier. We show the label-flow mappings in Table 14. From this view, we can easily see that those flows appear highly frequently in each of the DroidKungfu categories. In contrast, it is not the case for normal apps.

We show another restricted view in Fig. 16. Similar to above, we show the label-flow mappings in in Table. 15. In this figure, we only show the flows that send a phone number or location information via SMS. As shown in the figure, this behavior appears heavily in the NickyBot category. Meanwhile, none of the normal apps exhibit this behavior. This result tells us that if an app sends private data via SMS, it is highly possible that the app is malicious.

From these figures, we can easily tell that apps in different categories contain different types of flows that involve private information. Thus, we anticipate that flow pattern analysis in these categories can yield good results in identifying malicious apps.

3.4.6 Comparison with FlowDroid over DroidBench

In this section, we compare BlueSeal to FlowDroid [12], a static data flow analysis tool for Android apps, using DroidBench. We have discussed the results of BlueSeal over DroidBench in Section 3.4.4. Here, we show the detailed results with actual flows in DroidBench and results from FlowDroid for comparison. There are total 64 apps in DroidBench, where 49 apps contain data
<table>
<thead>
<tr>
<th>Label</th>
<th>Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Location:getLatitude→Log:w</td>
</tr>
<tr>
<td>1</td>
<td>Location:getLatitude→Log:v</td>
</tr>
<tr>
<td>2</td>
<td>Location:getLatitude→Log:d</td>
</tr>
<tr>
<td>3</td>
<td>Location:getLongitude→Log:w</td>
</tr>
<tr>
<td>4</td>
<td>Location:getLongitude→Log:v</td>
</tr>
<tr>
<td>5</td>
<td>Location:getLongitude→Log:d</td>
</tr>
<tr>
<td>6</td>
<td>LocationManager:getLastKnownLocation→Log:w</td>
</tr>
<tr>
<td>7</td>
<td>LocationManager:getLastKnownLocation→Log:e</td>
</tr>
<tr>
<td>8</td>
<td>LocationManager:getLastKnownLocation→Log:d</td>
</tr>
<tr>
<td>9</td>
<td>TelephonyManager:getDeviceId→URL.openConnection</td>
</tr>
<tr>
<td>10</td>
<td>TelephonyManager:getDeviceId→PrintStream:println</td>
</tr>
<tr>
<td>11</td>
<td>TelephonyManager:getDeviceId→DataOutputStream:writeUTF</td>
</tr>
<tr>
<td>12</td>
<td>TelephonyManager:getDeviceId→DataOutputStream:writeShort</td>
</tr>
<tr>
<td>13</td>
<td>TelephonyManager:getDeviceId→DataOutputStream:writeLong</td>
</tr>
<tr>
<td>14</td>
<td>TelephonyManager:getDeviceId→DataOutputStream:writeBoolean</td>
</tr>
<tr>
<td>15</td>
<td>TelephonyManager:getDeviceId→DataOutputStream:write</td>
</tr>
<tr>
<td>16</td>
<td>TelephonyManager:getDeviceId→Log:w</td>
</tr>
<tr>
<td>17</td>
<td>TelephonyManager:getDeviceId→Log:v</td>
</tr>
<tr>
<td>18</td>
<td>TelephonyManager:getDeviceId→Log:i</td>
</tr>
<tr>
<td>19</td>
<td>TelephonyManager:getDeviceId→Log:e</td>
</tr>
</tbody>
</table>

Table 13: Label-Flow Mappings for Fig. 14

<table>
<thead>
<tr>
<th>Label</th>
<th>Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TelephonyManager:getCellLocation→Log:i</td>
</tr>
<tr>
<td>1</td>
<td>TelephonyManager:getDeviceId→Intent:putExtras</td>
</tr>
<tr>
<td>2</td>
<td>TelephonyManager:getDeviceId→OutputStream:write</td>
</tr>
<tr>
<td>3</td>
<td>TelephonyManager:getLine1Number→ByteArrayOutputStream:write</td>
</tr>
<tr>
<td>4</td>
<td>TelephonyManager:getLine1Number→OutputStream:write</td>
</tr>
<tr>
<td>5</td>
<td>TelephonyManager:getSubscriberId→HttpClient:execute</td>
</tr>
<tr>
<td>6</td>
<td>TelephonyManager:getSubscriberId→ByteArrayOutputStream:write</td>
</tr>
<tr>
<td>7</td>
<td>TelephonyManager:getSubscriberId→Log:w</td>
</tr>
<tr>
<td>8</td>
<td>TelephonyManager:getSubscriberId→Log:i</td>
</tr>
</tbody>
</table>

Table 14: Label-Flow Mappings for Fig. 15.

leakage (i.e., actual data flows) and 15 apps contain no data leakage.

In Table 16, we show the difference between FlowDroid and BlueSeal. The column named “DroidBench” shows the actual number of flows as the DroidBench documentation describes. The columns named “FlowDroid” and “BlueSeal” show the reported number of flows for each DroidBench app from FlowDroid and BlueSeal, respectively. Out of 64 DroidBench apps, FlowDroid and BlueSeal report different results in 23 apps, which we show in Table 16. As we mentioned in the previous section, BlueSeal does not currently handle implicit flows. The experiment results show that FlowDroid currently also cannot detect implicit flows as it is primarily geared toward detection of flows through Intents. Thus we focus our discussion on explicit flows in Table 16.
In Table 16, the first four rows indicate that FlowDroid and BlueSeal both report extra flows compared to the actual flows present in DroidBench applications. Compared to FlowDroid, BlueSeal reports fewer flows in these cases. This occurs because in BlueSeal, sources are defined API calls that actually read sensitive data. FlowDroid, on the other hand, uses a more liberal definition. For example, in app #3 (Callbacks_LocationLeak2), BlueSeal only considers the getLatitude and getLongitude APIs as sources, while FlowDroid reports another flow which treats the onLocationChanged API as a source. Neither BlueSeal nor FlowDroid misses any actual flows. It is important to note that BlueSeal can be configured to accept other APIs as sources (and sinks) depending on how strict the user would like to be.

There are six apps (app #6 to #11) in which BlueSeal reports the actual flows as defined by DroidBench while FlowDroid does not report any. In app #6 (ImplicitFlows_ImplicitFlow1), there are two actual flows in the app in total, one implicit flow and one normal flow. BlueSeal detects the normal flow but not the implicit flow. Also, BlueSeal reports an extra flow which is contained
in dead code. In contrast, FlowDroid does not report any flow for this app. Other five apps (app #7 to #11) fall into two cases which BlueSeal is able to (partially) cover: (1) flows that occur during static initialization of a class and (2) flows that occur through the use of reflection. FlowDroid does not support reflection, while BlueSeal provides partial support for reflection. Namely, BlueSeal can handle cases that can be statically determined through inspection of the source (i.e., class and method names can be disambiguated via string analysis). Currently, we do not support dynamically loaded reflection instances.

In the remaining twelve apps (app #12 to #23), BlueSeal reports false positives. The reasons for these false positives are the same as discussed in the previous Section 3.4.4. FlowDroid reports false positives in seven of these apps (app #17 to #23) and correctly handles the other five cases (app #12 to #16). Two of these five apps (apps #12 and #13) are related to field-sensitivity which is not handled in BlueSeal. In app #14 (AndroidSpecific_InactiveActivity), an Android Activity class has been disabled from AndroidManifest.xml, i.e., it is dead code and does not get executed at runtime. FlowDroid correctly handles this case while BlueSeal still analyzes all the Android Activity classes presented in the source code. Another case (app #15) is handling interface invokes. Currently, for each interface used in an app, BlueSeal enumerates all callback handlers that im-
<table>
<thead>
<tr>
<th>AppId</th>
<th>AppName</th>
<th>DroidBench</th>
<th>FlowDroid</th>
<th>BlueSeal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Callbacks_Button2</td>
<td>2</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Callbacks_LocationLeak1</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Callbacks_LocationLeak2</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Callbacks_LocationLeak3</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>GeneralJava_VirtualDispatch1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>ImplicitFlows_ImplicitFlow1</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>GeneralJava_StaticInitialization1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>Lifecycle_FragmentLifecycle1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>Reflection_Reflection2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>Reflection_Reflection3</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>Reflection_Reflection4</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>FieldAndObjectSensitivity_FieldSensitivity2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>FieldAndObjectSensitivity_FieldSensitivity4</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>AndroidSpecific_InactiveActivity</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>Callbacks_MultiHandlers1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>Callbacks_Ordering1</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>FieldAndObjectSensitivity_ObjectSensitivity2</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>Callbacks_Unregister1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>ArraysAndLists_ArrayAccess1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>ArraysAndLists_ArrayAccess1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>ArraysAndLists_HashMapAccess1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>ArraysAndLists_ListAccess1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>GeneralJava_Exceptions3</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 16: Comparison between BlueSeal and FlowDroid.

implement the interface, and associates all possible callback handlers to each interface invoke. Thus, there might be false positives reported if one interface is implemented by multiple handlers. The last case (app #16) needs to understand the execution order of the app, in order to disambiguate a flow. As we mentioned before, BlueSeal does not consider program execution orders currently.

In the last seven apps, both BlueSeal and FlowDroid report false positives. One case (app #17) is that the app writes sensitive information to a value and then overwrites the value later on. Finally, it sends this object via SMS. App #18 (Callbacks_Unregister1) and app #23 (GeneralJava_Exceptions3) fall into the dead-code category. The former unregistered the callback from the application which contains a flow, while the latter leaks data in a never used exception clause. The rest of the apps fall into the category of flows via complex data structures. This has been discussed in Section 3.4.4 as well.

To sum up, BlueSeal can detect true positives in all DroidBench apps except implicit flow ones; FlowDroid currently cannot handle implicit flow cases and reflection cases but correctly handle all the other cases. On the other hand, both BlueSeal and FlowDroid report false positives in some of the cases. BlueSeal reports false positives while FlowDroid does not in a few cases due to the lack of context- and field- sensitivity and specific dead-code detection.
3.4.7 Case Studies: Flow Permissions in the Real World

Based on our results gathered, we observe that the most common Flow Permissions generated for apps are: IMEI NUMBER → NETWORK, IMEI NUMBER → LOG, and LOCATION → NETWORK. In most cases these Flow Permissions correspond to the ad libraries leveraged by the apps. The IMEI number is typically used to uniquely identify the user being tracked. The location, established either by GPS or via networking data, is also frequently tracked. The most common cross-app Flow Permissions are those that leverage CPs, specifically CPs identified by URLs pertaining to contacts. The vast majority of the Flow Permissions generated for this CP (90%), use the CP as a source via the query method. The most common sink for these Flow Permissions is SMS.

- **Transmitting SIM Card and Phone Data** Currently in Android there is no easy way to anonymously identify a phone. Apps skirt this issue typically in two manners: 1) user login: the user must login to gain the benefit of leveraging their usage history within the app, and 2) Phone State or SIM Card data: the app accesses and transmits the low-level identification of the phone (IMEI number), the low-level identification of the SIM card (ICCID number), or the international mobile subscriber identity (IMSI number) of the user. Flow Permissions provide a mechanism to distinguish between these two methods as different Flow Permissions are generated in each case. For the former, IMEI → NETWORK, is generated by BlueSeal and corresponds to textual data input by the user (IME) being sent across the network. In the later cases the following Flow Permissions are generated: IMEI NUMBER → NETWORK, ICCID → NETWORK, or IMSI NUMBER → NETWORK. The IMSI number should be transmitted as rarely as possible and third party apps should not typically utilize this number. An app that is not provided by the user’s service provider that has this Flow Permission should be considered malicious. We note, however, that user logins are not a universal solution. As an example, consider PhoneLab at SUNY Buffalo [7], a state of the art smartphone testbed with over 200 users. Participants run the PhoneLab app in the background, which collects pertinent statistics on the data and telephony usage. Data is collected for scientific purposes and PhoneLab must be able to distinguish between devices.

- **Incorrect Usage of the Log** Android provides a logging service called logcat that apps leverage for debugging. However, we believe that no app should use logcat at run time as it is a form of permanent storage that malicious apps can potentially access [26]. Based on our Flow Permissions, we have discovered that most apps use logcat and the most commonly stored sources are location and IMEI. To address this problem, Google has recently changed its permission mechanisms to restrict the READ_LOG permission to vetted system apps.

3.4.8 User Study

To test the utility of Flow Permissions, we created a user survey and tested graduate and undergraduate students taking computer science courses. These students are mixed majors (CS and non-CS). Our survey results were obtained anonymously with 540 participants. The survey procedure is as follows. (1) The survey presented a description of an anonymized app and its requested permissions. (2) Students then responded how likely they were to install the app. (3) The same question was asked including our Flow Permissions synthesized for the app. (4) At the end of the survey, the anonymized app was revealed and the students were once again asked how likely they were to install the app. We repeated the survey for two apps, Twitter and Dropbox.

Table 17 presents the results of our survey. The percentages shown in the table show the likelihood of installation of the app. The first column presents results of the anonymized app
<table>
<thead>
<tr>
<th>App Name</th>
<th>Android Anonymized</th>
<th>Flow Permissions Anonymized</th>
<th>Android Named</th>
</tr>
</thead>
<tbody>
<tr>
<td>Twitter</td>
<td>21 %</td>
<td>15 %</td>
<td>40 %</td>
</tr>
<tr>
<td>DropBox</td>
<td>37 %</td>
<td>15 %</td>
<td>35 %</td>
</tr>
</tbody>
</table>

Table 17: User survey result showing how likely the user is to install the app.

with standard Android permissions. The second column shows results for our Flow Permission mechanism for the anonymized app. The last column shows how the answers change once the app name is revealed.

Our results indicate that Flow Permissions can significantly impact user’s decisions to install an app when the users are unbiased, i.e., when users do not have any preconceived notions about the app or the developer of the app. Flow Permissions have a minor impact on biased users. Although these results are preliminary, they do give a positive indication that Flow Permissions can be useful in a real-world setting, especially when users are not familiar with an app or its developer.

3.4.9 Threats to Validity and Discussion

We have used independent sources to obtain our testing apps, which helps us to validate BlueSeal without much selection bias. The first source is Mal Genome Project that provides 1,047 malicious apps. The second source is the Google Play Store, where we downloaded 2,992 free apps. These apps include top-rated apps from all categories and randomly-selected apps. Selection bias can potentially come from Mal Genome project’s choice of apps as well as Google’s categorization of apps. In addition, both CS and non-CS majors are represented in our user study, which also reduces the chance of selection bias.

4 BlueBench: Modern Android Malware Characterization

In this section, we discuss details about our systematic analysis over modern malware apps and how we build up BlueBench, our modern malware benchmark set. The only previously publicly available and maintained malware dataset is Mal Genome [70]. The Mal Genome team collected 1,260 Android malware samples in 49 different malware families in 2012 and provided a timeline analysis and characterize of the malware applications and families based on their behavior. While this dataset precisely characterizes Android Malware in 2012, it unfortunately does not cover all families and malware types found on Android today. In 2016 alone, 6 million new Android Malware applications have been reported [1]. Both modern benign and malicious apps have evolved rapidly in the past few years. New techniques (e.g., reflection, native code invocation and dynamic class loading) are deployed more frequently by malware apps to avoid detection. Also, due to various services, more and more modern benign apps access different types of sensitive data on device, which makes it more difficult to distinguish benign and malware apps. In the meanwhile, modern malicious apps are more complicated either to trick users for installation or bypass security scan mechanisms. As a result, the old collected malware apps are not representative of current malware apps. On the other hand, researchers have developed different ways for Android malware detection based on different features/metrics. However, most of these tools are evaluated on their own collections of modern apps. This makes it hard for readers to validate and compare these tools’ performance. It is commonly the case that a tool works very well on its own datasets and its performance drops a lot on others. In this case, a detailed survey/study on
modern malware apps are needed. And it requires a representative dataset of modern malware apps that can be leveraged by different tools based on different metrics in order to evaluate their performance for readers.

4.1 Survey Structure and Starting Assumptions

Our starting point is a set of modern malicious apps totaling 56,000 malware samples obtained from security operations in 2016 by a threat intelligence company operating in the United States and Europe. We are unable to disclose the company from which we obtained the malware samples. Each app from the 56K set has been scanned through multiple popular anti-virus tools including the VirusTotal suite [4]. Meta data is associated with each app including scan result of each anti-virus tool, time discovered, description of the app, malware family identification, etc. Specifically, for each app we have scan results from 55 anti-virus tools. If app is detected by a tool, it will be attached with malware family information as well as other static features, such as date of malware scanned, malware family and malware description url. In our dataset, all of the apps are confirmed to be malicious. Besides, we also perform studies on old malware apps from MalGenome project and modern benign apps from Google Play Store. In MalGenome project, it contains 1,260 apps, which is used to represent old malware apps. Our benign data set consisted of 14,500 apps downloaded from Google Play in August, 2016, representative of the top 200 most popular apps across all categories. These apps were submitted to VirusTotal for examination which indicated that 2,158 of these apps were in fact malware. A staggering 18% of the most popular apps available on Google Play in the summer of 2016 had some type of malicious code embedded within. This demonstrates that selection of the most current set of Google Play apps as a benign set is a risky choice as there is likely malware in the population. By constructing a benign set that is close to but not exactly current, we are able to leverage VirusTotal to filter out recently-discovered malicious apps. In modern malware apps, app code size ranges from 7 to 895,481; in MalGenome apps, app code size ranges from 89 to 105,957; in modern benign apps, app code size ranges from 3 to 1,567,368. As we can see, both modern benign and modern malware apps can be very complex or extremely simple. We manually examine the smallest 50 apps in modern malware apps and find out that 48 apps share the same pattern of doing dynamically class loading with a few normal or native method invocations as shown in Figure 17. The other two apps simply contain a random activity without doing anything.

4.1.1 Analysis Metrics

We explore a variety of statically extractable metrics that are commonly used by malware detection tools to analyze and understand our dataset. We leverage these metrics to pick a representative set of malicious apps to construct a benchmark for comparing modern malware detection techniques.
We discuss each of these metrics and our reasons for choosing them as a selection criteria.

- **Permissions.** Permissions are the fundamental mechanism Android leverages to protect users’ privacy. As such, many malware detection tools that leverage Permissions directly as features for classification [33, 45, 65]. By analyzing permissions, we can understand what types of sensitive data are accessed inside apps as well as where sensitive data can flow. As a result, understanding which permissions an app requests can give an estimation of possible behaviors the app can exhibit.

- **Android Specific Constructs.** There is a variety of Android specific constructs (e.g., ContentProvider, Intent, Android Storage). These constructs are very important for analyzing and understanding app behavior as they are the primary mechanisms for interaction between components, between applications, and exchanging of data. For example, Android malware are known to rely on actions of an Intent (e.g., a device booting event or a package installation) to trigger malicious behavior. Android ContentProvider and storage are closely related to sensitive data manipulation operations (e.g., read or write contacts data). These constructs are also leveraged by different malware detection tools [11, 66]. Our analysis results show that both modern apps and malicious apps leverage these features more heavily than before. Detailed discussion is in Section 4.5.

- **Reflection.** Reflection provides a mechanism to trigger code indirectly. Applications can use reflection to invoke Android APIs and leverage Android specific constructs [15, 44]. Similarly, many obfuscation techniques inject reflective calls to obscure code structures and intent to prevent accurate analysis as well as to protect against copying of code. Android malware also relies on reflection to modify itself at runtime [44] as a mechanism to avoid detection. Our analysis results show that modern malware apps will leverage reflection more frequently than old malware apps in order to bypass static detection mechanisms. Details are discussed in Section 4.3.

- **Native Code.** Native code invocation is another common feature in Android apps. Apps typically leverage native code to improve performance or to specialize code for specific devices. Native code invocation also provides attackers a mechanism to package exploits and hide malicious behavior [9].

- **Dynamic Execution.** Another emerging technique in Android malware is dynamic execution [54], which includes dynamic class loading and runtime execution. Dynamic class loading can either download and install a second app from third-party servers, which is the real malware app, or load another separate dex file, which contains malicious code. Runtime execution can execute a pre-written shell script to perform dangerous behavior (e.g. execute pm command to install new apps [23]).

- **Information Flows.** Information flow is an representation of sensitive data usage inside apps. It is normally in the format of (source, sink) pair, which means sensitive data is accessed at source and flows into sink. There are many tools for malware detection that leverage information flows directly [14] or indirectly [59] to determine if an app is malicious. Tracking information flows inside apps is very useful for identifying potential privacy leakage. However, as modern apps have grown in both complexity and variety, the number of information flows they contain has also grown. Our analysis results show that modern benign and malware apps are similar to each other regarding the raw information flows they contain.
• **Complex-Flows.** Complex-Flows is another way to represent sensitive data usage inside apps [59]. It analyzes structures and computation present within information flows, providing additional behavioral data along with the information flow. Our analysis results show that even though modern benign and malware apps may share the same set of information flows, their behavior along information flows will be different.

Besides, we compare and constrast these metrics between modern malware and old malware apps as well as between modern malware and modern benign apps. Details of comparison are discussed in each section.

4.1.2 Modern Malware Family Statistics

By categorizing malicious apps into different families, we can analyze and understand our data set more thoroughly and account for family bias in our benchmark. Family based malware analysis is an important technique for malware detection and analysis [34]. We extract family information based on scan results of different anti-virus tools. We have chosen Ikarus [2] as our canonical representation of malware families because it detects almost all malware apps (92% detection rate) in our datasets and provides a reasonable number of families (823 families in total). In other tools, it either provides a massive number of families (e.g., 18,572 families in McAfee) or cannot detect enough numbers of malware apps (e.g., 4,335 out of 56,000 in Symantec). Even though ESET-NOD32 detects the most number of malware apps (96.5% detection rate), 90% of these apps are categorized into one single family. In following sections, family information is based on Ikarus’s categorization. The total number of apps for each family ranges from 1 (e.g., SpamSold) to 9644 (Shedun). The average code size (LOC) of apps for each family ranges from 18 (e.g., BackDoor) to 656,911 (Youku).

4.2 Android Permission

One of Android’s main defense mechanisms against malicious apps is the permission-based model, which enforces restrictions on the specific operations that a particular process can perform. The permissions an app requests can, therefore, provide a static analysis tool with an approximation of which sensitive APIs an application can leverage. The presence of a requested permission does not mean that the sensitive APIs protected by the permission are actually present in the app’s bytecode. This, in turn does not mean the app does not leverage the APIs as it can still perform dynamic code loading to load bytecode (typically download from the web at runtime) that contains the sensitive APIs. In android.manifest.permission class, Google defines 153 permissions in total. Besides, developers are allowed to define their own permissions. In this case, the number of permissions in an app can be very big including both app-defined and Google-defined permissions.

First, we run a analysis on requested permission in apps’ Manifest.xml across our three datasets. In MalGenome, there are 115 distinct permissions requested in total; there are 4,285 distinct requested permissions in modern benign apps; the number in modern malware apps is 2,520. In MalGenome, 98 out of 115 permissions are Google defined permissions; there are 5 permissions are app defined permissions; the rest are device manufacturer defined permissions (e.g., motorola, LG). In both modern benign and modern malware apps, more than 90% of requested permissions are app or third-party defined ones. In following analysis, we focus on Google defined permissions since they are used to protect sensitive data on devices.

While analyzing permissions is largely a preliminary step that most static analysis tools perform, permissions used to be a distinguishing feature of malware [33, 45, 65, 70]. Comparing
common permissions requested between modern and early Android malware provides us with an insight as to how malware has evolved. We collect and show the top 10 used permission from the MalGenome data set as well as ours. Table 18 and Table 19 shows the top 10 permissions requested in MalGenome and BlueBench apps. These two datasets share the same top five permissions which are related to network, identifiers, and storage. This is not surprising as since many malware apps require network access to export sensitive information, to download code to load dynamically, to receive bot commands from remote server and so on. Another observation is that both old and modern malware apps leverage `RECEIVE_BOOT_COMPLETED` permission heavily, which monitors device boot event and can be used to trigger malicious code execution. This means modern and old malware apps still share certain characteristics. The remaining permissions, however, are different. We can see that older malware apps heavily favor simple behavior like sending/reading SMS. This is no longer the case in modern malware apps. Modern malware apps are more interested in many other permissions. For example, `INSTALL_SHORTCUT` permission allows an application to install a shortcut in Launcher and `WAKE_LOCK` allows using PowerManager WakeLocks to keep processor from sleeping or screen from dimming. Both of these two are very common permissions in modern benign apps. `SYSTEM_ALERT_WINDOW` allows an app to create windows shown on top of all other apps. However, very few apps should use this permission; these windows are intended for system-level interaction with the user. Another heavily favored permission in modern malware apps is `GET_TASKS`, which allows an app to get information about running tasks. Both `SYSTEM_ALERT_WINDOW` and `GET_TASKS` are rarely requested in modern benign apps. In general, modern malware apps are more and more similar to modern benign apps. However, there still are different behavior patterns exhibited in modern malware apps.

We also examine usage of permissions in modern benign apps, shown in Table 20. As we can see, the top used permission set is very similar to BlueBench apps. Modern benign apps share the same set of top permissions with malware apps related to network, identifiers and storage. Besides, system event permission (`WAKE_LOCK`) is also heavily leveraged in benign apps. Even though `RECEIVE_BOOT_COMPLETED` permission is not on the top list, there are 20% of benign apps leverage this permission. The difference is that modern benign apps are very interested in device location information. This is not surprising since one of most important feature in modern benign apps is location-based service. The `VIBRATE` permission is to allow access to vibrator and there are half of modern malware apps request it. The `GET_ACCOUNT` permission allows access to the list of accounts in Android Accounts Service, which is also deployed by 12,356 modern malware apps. By examining usage of each permission in Table 19 against modern benign apps, we observed that all permissions in modern malware are heavily used in benign apps as well, while heavily used SMS permissions in MalGenome are leveraged by only 258 out of the modern benign apps we used in our comparison.

Next, we examine the usage pattern of Android permissions in different families. By analyzing detailed distribution of permissions requested based on each malware family, we observed that some families share similar count as well as types of permissions requested across apps within the family while other families do not share this type of behavior. Here, we show six families, that share similar patterns across apps as shown in Figure 18. Other families that do not share similar pattern are omitted here. The red lines are used to indicate the span of malware apps belonging to a family. Inside each family, apps are sorted based on app code size. We randomly select a number of apps from these families that share same permission count and examine these apps manually. We observe that 1) they request similar permission set; 2) they are variants to each other in most cases or share some common third party libraries in their source code; 3) malicious source code can be different among apps, but their behavior is quite similar. By manually examining requested permissions, we also observe that some malware apps contain repeated entries of same
Table 18: Top 10 Permissions in MalGenome

| 1 | android.permission.INTERNET  
| 2 | android.permission.ACCESS_NETWORK_STATE  
| 3 | android.permission.READ_PHONE_STATE  
| 4 | android.permission.WRITEEXTERNAL_STORAGE  
| 5 | android.permission.ACCESS_WIFI_STATE  
| 6 | android.permission.READ_SMS  
| 7 | android.permission.RECEIVE_BOOT_COMPLETED  
| 8 | android.permission.WRITE_SMS  
| 9 | android.permission.SEND_SMS  
| 10 | android.permission.RECEIVE_SMS

Table 19: Top 10 Permissions in BlueBench

| 1 | android.permission.INTERNET  
| 2 | android.permission.ACCESS_NETWORK_STATE  
| 3 | android.permission.READ_PHONE_STATE  
| 4 | android.permission.WRITE_EXTERNAL_STORAGE  
| 5 | android.permission.ACCESS_WIFI_STATE  
| 6 | android.permission.GET_TASKS  
| 7 | com.android.launcher.permission.INSTALL_SHORTCUT  
| 8 | android.permission.RECEIVE_BOOT_COMPLETED  
| 9 | android.permission.SYSTEM_ALERT_WINDOW  
| 10 | android.permission.WAKE_LOCK

4.3 Reflection analysis

Reflection is a common feature in Android apps. Reflections can be legitimately used to 1) implement generic functions, (e.g., developers can leverage reflection to initialize a Collection object at runtime); 2) reinforce app security, e.g., separating core functionality of a given app to an independent library, which can be loaded dynamically (via reflection) when a specific rule is reached; 3) access inaccessible APIs in android.jar. In Android, there are APIs that are shipped into running Android devices and not released to Android SDK. At runtime, Android framework.jar will replace Android.jar to support the execution of these APIs in Android apps. In this model, these inaccessible APIs can be accessed by third-party apps through reflection; 4) maintain backward compatibility, e.g., ensuring apps that are developed with latest features to be still executable on old devices, where the latest features are not yet available. For example, an app can leverage reflection to check whether the running Android device supports text-to-speech feature or not [44]. Reflection is also common in malware, either used to try to confound static analysis or injected via code obfuscation techniques. For the former, malware leverages reflection to deceive casual static code analysis that check for direct sensitive API invocations. A malicious developer can hide malicious code and dynamically run malicious code by leveraging reflection [54]. For example, a malware app can leverage reflective invocation of TelephonyManager.getSimNumber() to steal device SIM number.
Table 20: Top 10 Permissions in Play Store Apps

<table>
<thead>
<tr>
<th>Permission Category</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>android.permission.INTERNET</td>
<td>40</td>
</tr>
<tr>
<td>android.permission.ACCESS_NETWORK_STATE</td>
<td>30</td>
</tr>
<tr>
<td>android.permission.WRITE_EXTERNAL_STORAGE</td>
<td>20</td>
</tr>
<tr>
<td>android.permission.WAKE_LOCK</td>
<td>10</td>
</tr>
<tr>
<td>android.permission.ACCESS_WIFI_STATE</td>
<td>8</td>
</tr>
<tr>
<td>android.permission.READ_PHONE_STATE</td>
<td>6</td>
</tr>
<tr>
<td>android.permission.VIBRATE</td>
<td>5</td>
</tr>
<tr>
<td>android.permission.READ_EXTERNAL_STORAGE</td>
<td>4</td>
</tr>
<tr>
<td>android.permission.GET_ACCOUNTS</td>
<td>3</td>
</tr>
<tr>
<td>android.permission.ACCESS_FINE_LOCATION</td>
<td>2</td>
</tr>
<tr>
<td>android.permission.ACCESS_COARSE_LOCATION</td>
<td>1</td>
</tr>
</tbody>
</table>

Even worse, reflection can be combined with obfuscation/encryption of string that are passed in as arguments to reflective calls. The reflective target is decrypted at runtime. Figure 19 shows an example of a real-world modern malware app, which leverages reflection for code obfuscation. Line(4-8) in blue shows that the reflection invocation are obfuscated. The malware app decrypts the obfuscated string to retrieve real value and then makes invoke. Line(10) in red shows the usage of reflection. Figure 20 shows another example of a real-world malware app from MalGenome project. In this example, the malware app leverages reflection to access an inaccessible API `getITelephony` from Android platform. We randomly select 50 apps from MalGenome with reflection usage and all of them are using reflection in a similar way to leverage inaccessible APIs from different Android services (e.g., phone, power, wallpaper, view and so on). However, based on our manual validation, this is not the case in modern malware.

In our analysis, we find that 34,379 modern malware apps contain at least one reflective call; in MalGenome, 786 apps leverage reflection technique; in Play Store apps, there are 9,124 benign
private static void mtxpggnxu() {
    ...
    r6 = ttgj dusar();
    r4 = kkpeqrx
        ("@OU_yU_O2T02T1L2_bBtu_2>2BfxE5xE");
    staticinvoke wkpqowxp(r6, r4, r0);
    r4 = kkpeqrx
        ("@OU_yU_O2T02T1L2_bBtu_2>2BxqS5");
    ...
    r7.<reflect.Method: invoke>()(r6, r8);
}

Figure 19: Reflection Code Snippet in Modern Malware App.

Object getITelephony(Context context) {
    context = (TelephonyManager)context
        .getSystemService("phone");
    Method method = context.getClass()
        .getDeclaredMethod("getITelephony", new Class[0]);
    if (!method.isAccessible()) {
        method.setAccessible(true);
    }
    return method.invoke(context, new Object[0]);
}

Figure 20: Reflection Code Snippet in MalGenome Malware App.

apps that make invocation via reflection. Both benign and malware apps leverage reflection heav-
ily, which makes it even harder for detection tools to distinguish malicious behavior hidden by
reflection. Figure 21 shows reflection usage in modern malware apps sorted based on app code
size. It is interesting to find that medium sized apps contain most number of reflection usage while
small apps tend to contain only a very small number of reflection usage. We manually examine
some of these apps for a deeper analysis. First, we examine small apps with one single usage of
reflection. Most of these are pure malicious apps without third party library. They only leverage
reflection on their targeting API. This shows small malware apps mainly focus on achieving their
goal and try to bypass static analysis with minimum effort. Next, we manually examined apps
that contain the most reflection. All of these apps encrypt all method invocations with obfuscated
strings and use reflection to execute the malicious code.

We also examined the reflection usage distribution among families. Based on our observation,
there are two different patterns among family distribution. In most families, the distribution is ran-
don while there are families share similar patterns of reflection usage. Here, we select a number of
families to represent results as shown in Figure 22. Family names are indicated on x-axis. We
perform a manual validation on apps in SMSReg, Zdtad, Shedun (50 apps from each family) that
share same patterns and discovered that all applications in these families shared similar reflection
usage – namely they leverage a package that dynamically loads dex files and use reflection to do
method invocations on the loaded code. Even though different apps may leverage different pack-
ages, all of these packages share similar behavior of dynamic dex loading and method invocations
via reflection. In SendSMS, Ewind, the reflection usage are more random across apps. However,
most of these reflection usage are performed on obfuscated strings. Besides, all these apps include
encrypted classes and methods. An interesting finding is that there are apps in SendSMS family
contain no Android platform API invocation at all and all these API invocations are made by
reflection.

4.4 Dynamic Execution and Native Code Usage on Android

Dynamic loading of dex files is becoming a popular mechanism for malware to evade detection. In our malware dataset, more than 9,000 apps use dynamic loading; in MalGenome, there is only one app leverage dynamic dex loading; in modern benign apps, we find 348 apps that leverage
```java
protected Class findClass(String) {
$\text{1}$ $r1 := @\text{parameter0: String;}
$\text{2}$ $r2 = \text{com.upay.billing.utils.Util: regex.Pattern }()(\text{});
$\text{3}$ $r3 = r2.\text{regex.Pattern: matcher(CharSequence)}(r1);
$\text{4}$ $z0 = r3.\text{java.util.regex.Matcher: matches()}();
$\text{5}$ $r4 = r4.\text{StringBuilder: append()}("load: class=");
$\text{6}$ $r4 = r4.\text{StringBuilder: append()}(r1);
$\text{7}$ $r4 = r4.\text{StringBuilder: toString()}();
$\text{8}$ staticinvoke Log.i("Util", $r5);
$\text{9}$ $r6 = \text{class "com/upay/billing/utils/Util"};$
$\text{10}$ r7 = r6.\text{ClassLoader getClassLoader}();
$\text{11}$ r6 = r7.\text{ClassLoader loadClass()}(r1);
$\text{12}$ }
```

Figure 23: DexClassLoader Code Snippet from a Malware App.

this technique. Such apps typically contain a simple payload, which will load the real payload either from its resource folder or the internet. The real payload can be data leakage, sending SMS to premium number and so on. Figure 23 shows a real-world malicious app usage of dynamic dex bytecode loading. In this code snippet, the malware app first downloads a payload and then loads it dynamically at runtime using class loading. This makes it much more difficult to detect and it can evade traditional static analysis.

Regarding benign apps, researchers have pointed out there are legitimate reasons for leveraging dynamic class loading [54]. It can be leveraged to achieve 1) A/B testing and beta testing: It is common for software manufacturers to test modified versions of their applications with a small subset of their users in order to evaluate user satisfaction in comparison to the established version. This approach is usually referred to as A/B testing. For example, Facebook leverages APK installations to install updates on selected users’ devices; 2) Common frameworks: It is possible that multiple applications are based on the same framework, which is installed on the device as a separate application. All the applications can load code from the framework application dynamically instead of bundling it with every single application; 3) Framework updates: Many current applications bundle various frameworks for additional functionality. Well-known examples are advertisement frameworks, which display advertisements to the user and function largely independently from the rest of the application. To ensure that the latest version of these frameworks is used by all applications, dynamic class loading can be used to implement a self-update mechanism that is independent from updates of the containing app; 4) Loading add-ons: Some apps can be extended by add-ons that are installed as separate apps.

Since both malicious and benign apps leverage class loading, simply checking if an app leverages class loading is insufficient to differentiate between benign and malicious code. This is in sharp contrast to older malware, specifically there is only one app in the MalGenome project that does dynamic class loading.

Figure 24 shows dynamic class loading statistics of modern malware apps. Vertical axis shows the number of usage inside apps; horizontal axis indicates app ID sorted based on app code size. Based on our manual examination, most malicious apps only contain one code point for dynamic class loading, which is used to normally load a single dex file or a library class file. We also manually checked a selection of other apps that contain multiple instances of dynamic class loading. In these apps, there are two different usages: 1) apps load the same dex file in different program points. Apps try to trigger malicious behavior from different places; 2) apps load multiple dif-
different dex files. The reason for this might be that these apps hide different malicious behavior in different files. Next, we examine this against malware families. Since most of the apps contain only one usage. There are no clear different patterns in different families. Last but not least, based on our observation, there is no correlation between app code size and usage of dynamic class loading. App code size is more related to how many extra libraries are included inside apps.

Runtime .exec is another popular runtime technique that allows apps to execute arbitrary binaries (e.g. executing a shell script). In most static analysis tools, no security check will be performed on the executed binary. Thus, malicious behavior can be hidden. We examine statistics on runtime exec technique across modern malware apps and found that the usage of runtime exec is not correlated to code size. We observe that there are only two families (SMSReg, Masnu) that exhibit similar usage pattern across apps while the usage distribution in other families (e.g. SMS) is more random, shown in Figure 25. Through manual inspection, we discover that these two families leverage runtime execution technique to either acquire root access by running scripts or install new packages. In SMSReg family, apps share certain third-party libraries that run shell scripts from assets or command to access device data (e.g., “cat /sys/class/net/wlan0/address”). In Masnu family, we find that apps are variants to each other in most cases. Another observation is that there are apps in Masnu share some third-party libraries with apps in SMSReg. By analyzing this usage in MalGenome, modern malware and Google Play apps, results show that both beign and malware apps leverage this technique heavily. The usage statistics are 2,398 in Google Play apps, 588 in MalGenome apps and 16,857 in modern malware apps.

Android apps can also run native code via JNI to improve the performance of the app. It is commonly leveraged for shared native libraries as well as for games. Native code can be downloaded at runtime and need not to be packaged with the app. Since most of static analysis tools focus on DEX bytecode, native library becomes a good place to hide malicious behavior. For example, a malware app can save its remote server address in a native program [70]. Based on
our analysis, native code invocations are very popular in both benign and malware apps. In MalGenome, there are 636 apps that leverage this technique; in Play Store apps, there are 5,290 apps containing native invocations; in our modern malware dataset, the number is 26,916. Figure 26 shows native code usage distribution with sorted app code size in modern malware apps. Results show that bigger app code size does not necessarily indicate more native code usages. Based on our manual validation, many third-party libraries make native code invocations frequently and bigger apps can include more third-party libraries with embedded native calls. Figure 27 shows distribution of native code invocations across families. Here, we select and present results from seven families. There are two patterns: 1) apps in Agent, Dropper.Agent, SMSreg, Zdtad, Trojan.Agent, Masun share similar usage; 2) apps in PUA show a more random distribution. Based on manual validation, we find apps that share similar native code usage fall into three cases: 1) apps are variants to each other; 2) apps leverage same third-party libraries; 3) apps exhibit similar behavior that dynamically loads dex file and makes invocation via native code.

4.5 Android Specific Constructs

In this section, we examine Android specific related metrics.

4.5.1 Inter Component Communication in Android

Inter-process and intra-process communication on Android is mainly performed through intents, passive data structures exchanged as asynchronous messages. Intents allow information about events to be shared between different components and applications. Researcher may collect intents listed in the manifest as part of a feature set, as malware often listens to specific intents. A typical example of an intent message involved in malware is BOOT_COMPLETED, which is used to trigger a malicious activity/service directly after booting a smartphone.

Here, we examine the usage of Android Intent in apps. Figure 28 shows intent usage against app code size in modern malware apps. In general, bigger app size may indicate more intent usage. And there are only a few apps that contain an extremely large number of intents. Another
interesting observation is that there is one app that defines more than 3,700 intent instances inside itself. After manually validation, this app contains hundreds of Android activities and use intents for communication. It also leverages intents to start these activities in different places. Besides, apps leverage intents heavily for inter-component communication.

In Play Store apps, we find one app (MiRutina), which is a fitness app, exhibits similar behavior with 29,038 intents used. Other modern benign and malware apps contain less than 2,00 intent usages; MalGenome apps contain at most 203 intent usages. This tells us modern benign and malware apps are more complicated regarding interactions among Android components than old malware apps.

Figure 29 shows app intent usage distribution against malware families. Results show that there are only four families (SMSreg, Ztdad, SMS.Agent, Masnu), in which apps share a similar pattern regarding intent usage, and all these apps contain a relatively small number of intent usages (<100). Based on manual examination, these apps are either variants to each other or there are not much inter-component communication inside apps with a limited number of Android components. Apps in other families leverage intent usage in a more random way, which may be due to different requirements of inter-component communication among Android components.

Android malware are known to track the action of an Intent (e.g., if a device has recently completed booting or whether a new package has been added) to choose when to perform a malicious behavior [70]. To receive and process Android intents of interest, an app component should declare separate intent filters for each unique job it can do in its AndroidManifest.xml.

We analyze intent filters declared in the manifest file for all three sets and present top five intents from each set in Table 21. Besides, we indicate the number of apps that track that intent. It is not surprising that almost all apps receive MAIN intent since it is the main entry point for Android Activity. Both old and modern malware favor heavily on BOOT_COMPLETED intent, which is commonly used to start malicious code execution. However, there are many benign apps
leverage it to start app services as well.

Another popular intent in benign apps is VIEW, which is also an Activity action used to interact with users. Other than that, there are no commonly used intents in benign apps. There are only less than 5% apps share SEND and SEARCH intents.

MalGenome apps are interested in SIG_STR intent, which monitors signal strength, and BATTERY_CHANGED_ACTION intent, which tracks battery information. However, modern malware apps are more interested in USER_PRESENT, which indicates if a user is actively present, PACKAGE_ADDED, which will be sent when a new package has been added, and PHONE_STATE, which indicates if the call state on the device is changed.

### 4.5.2 Content Provider in Android

Content Provider is another important Android construct. It manages access to a central repository of data. It handles inter-process communication and secures data access as well. Malware apps can also expose sensitive data from Content Providers. For example, malware apps can query Contacts Provider and leak user contacts [70]. Apps can create their own providers. Figure 30 shows Content Provider usage against app code size. As we can see, there is no clear correlation between Content Provider usage and app code size. It is more like random distribution among different apps. This is due to different purposes of apps and Content Provider is a very common data
structure leveraged by most apps. Figure 31 shows Content Provider usage distribution against malware families. Similarly, it is more of random distribution among different families. There are only two families contain a clear pattern. Based on manual validation, apps in these families are mainly targeting on leaking contacts from device or stealing MMS content. In MalGenome, there are 563 apps access Content Providers with a number of usage within range 1 to 23; in Play Store, there are 3,388 apps with range 1 to 235; in modern malware, there are 15,337 apps with a similar usage range as benign apps (from 1 to 225). This is because MalGenome apps simply target on specific Content Providers (e.g., Contacts) and steal data, while modern benign and malicious apps involve more complicated operations regarding Content Provider.
4.6 Simple Information Flow Analysis

Sensitive information leaks, as implemented by malicious or misused code (such as advertising libraries) in Android applications, constitute one of the most prominent security threats to the Android ecosystem. Even though Android currently supports a coarse-grain information security mechanism in which users grant application the right to access sensitive information. This mechanism has been less than successful at eliminating information leaks. In this case, analyzing information flows in malware apps is one very important factor when constructing malware benchmark dataset. Information flow analysis can reveal the usage of sensitive data in Android apps. Here,
we compare the information flow related statistics from both old and modern malware datasets.

First, we examine basic usage statistics of information flow between old malware and modern malware apps. Figure 32 and Figure 33 show information flow usage distribution for MalGenome and modern malware apps respectively. In these two figures, apps are sorted based on app code size. Based on comparison, we can see that many modern malware apps access sensitive data much more often than old malware apps. This is not surprising since modern malware apps are interested in different kinds of sensitive data on mobile device. In fact, modern benign apps access more and more sensitive data from device.

Besides, results show that the bigger the app size is, the more information flows it may contain.
Most of MalGenome apps contain less than 500 information flows, while many modern malware apps contain more than 1000 information flows. These statistics, again, prove that modern malware apps leverage much more information flows than old malware apps. In the figure, the red line shows the average number of information flows among test apps. The average number is 570 for modern malware apps, while it is 173 for MalGenome apps. And there are 16% of modern malware apps contain more than average number, while there are 23% of old malware apps contain more than its average. The yellow line shows the median number of information flows in each dataset. The median for modern malware apps is 125, while it is only 31 for MalGenome apps.

It is not surprising both average and median number of information flows are much bigger in modern malware apps than them in old malicious apps, since old malicious apps are more simple and focus on certain types of sensitive data only.

Next, we explore another statistic to show the relation between LOC and information flow usage in malware apps. Figure 34 and Figure 35 shows statistics on App LOC per information flow distribution in modern and old malware apps respectively. Similarly, apps in these figures are sorted based on app code size. From these figures, we can conclude that in modern malware apps, the bigger an app is, the more LOC it needs to leverage one information flow. However, in Figure 35, it shows there are no clear relation trend between information flows and app size. The reason is that modern malware apps are more complex and contain a large portion of benign code to pretend as benign. Another reason is that modern malware apps leverage more mechanisms to bypass security check, which makes modern malware apps more complex and larger in size. The red and yellow line show the average and median respectively. The number is 1,500 and 269 for modern malware apps, while it is 294 and 88 for old malware apps. This means in every 88 lines of code, there will be an information flow in old malware apps, while it will require analyzing 269 lines of code in average in order to find an information flow in modern malware apps. This proves that modern malware apps are more complex and similar to modern benign apps, which makes it more difficult to analyze and differentiate them from modern benign apps.

Next, we explore information flow usage in different families. Figure 36 shows information flow usage distribution among different families. As we can observe, there is no specific patterns
in families. Only one of these families shares a similar usage across apps. By manual validation, it turns out that these apps leverage the same adware library, which contains locations access, phone state reading and device information monitoring. Besides, given a family, there are apps contain an extremely large number of information flows while other apps contain only a small mount of information flow. In this case, we can conclude apps that belong to same family can be significantly different in leveraging information flows.

Last but not least, we compare top information flows in both old and modern malware apps, as well as modern benign apps. Table 22, Table 23 and Table 24 show the top information flows in MalGenome and modern malware as well as Play Store apps respectively. Based on results, we can tell that both old and modern malware apps are interested in phone identifier and location information as well as user input. However, old malware apps are more likely to write these information into leaking points, while modern malware apps are sending them to other components via Intents before leakage. This means modern malware are more sophisticated than old
malware apps. Besides, modern malware and modern benign apps almost share the same set of information flows. However, MalGenome apps are not interested in device location information, which are heavily used in modern benign and malware apps. In this case, we can conclude that modern malware apps are much more similar to modern benign apps and share more common behavior than old malware apps. Also, simple information flow analysis tools may not achieve a good performance on distinguishing modern malware and benign apps.

Next, we analyze top accessed sensitive data in apps. Table 26 and Table 25 show top used sources in modern malware and modern benign apps. These two lists are almost the same. The only difference is that benign apps heavily leverage app AccountManager to access user accounts. This is reasonable since many modern benign apps require user’s login information. In this case, we can conclude modern malware and benign apps share common behavior regarding accessing device sensitive information. This can be caused by two reasons: 1) modern benign apps access various sensitive information to provide more functionalities, 2) modern malware apps are more complicated and embed malicious code into benign code to trick users.
Table 24: Top Data Flows in Play Store Apps

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EditText:getText()</td>
</tr>
<tr>
<td>2</td>
<td>Location:getLongitude()</td>
</tr>
<tr>
<td>3</td>
<td>Location:getLatitude()</td>
</tr>
<tr>
<td>4</td>
<td>FileInputStream:read()</td>
</tr>
<tr>
<td>5</td>
<td>LocationManager:getLastKnownLocation()</td>
</tr>
<tr>
<td>6</td>
<td>TelephonyManager:getDeviceId()</td>
</tr>
<tr>
<td>7</td>
<td>AccountManager:getAccounts()</td>
</tr>
<tr>
<td>8</td>
<td>TelephonyManager:getCellLocation()</td>
</tr>
<tr>
<td>9</td>
<td>TelephonyManager:getSubscriberId()</td>
</tr>
<tr>
<td>10</td>
<td>TelephonyManager:getSimSerialNumber()</td>
</tr>
</tbody>
</table>

Table 25: Top 10 Sources in PlayStore Apps

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EditText:getText()</td>
</tr>
<tr>
<td>2</td>
<td>Location:getLongitude()</td>
</tr>
<tr>
<td>3</td>
<td>Location:getLatitude()</td>
</tr>
<tr>
<td>4</td>
<td>FileInputStream:read()</td>
</tr>
<tr>
<td>5</td>
<td>LocationManager:getLastKnownLocation()</td>
</tr>
<tr>
<td>6</td>
<td>TelephonyManager:getDeviceId()</td>
</tr>
<tr>
<td>7</td>
<td>AccountManager:getAccounts()</td>
</tr>
<tr>
<td>8</td>
<td>TelephonyManager:getCellLocation()</td>
</tr>
<tr>
<td>9</td>
<td>TelephonyManager:getSubscriberId()</td>
</tr>
<tr>
<td>10</td>
<td>TelephonyManager:getSimSerialNumber()</td>
</tr>
</tbody>
</table>

4.7 Complex-Flow Information Analysis

Mobile apps themselves have evolved in their sophistication and in the number of services they provide to the user. Along with this evolution, more and more sensitive data are required for different purposes. As a result, modern benign apps now access same sensitive data as malicious apps do. This makes it more difficult to distinguish malware from benign apps.

In this case, there is a need to look beyond simple flows in order to effectively leverage information flows analysis for malware detection. By analyzing recently collected malware, we show there has been an evolution in malware beyond simply collecting sensitive information and immediately exposing it. Modern malware performs complex computations before, during, and after collecting sensitive information and tends to aggregate data before exposing it. A simple (source, sink) view of information flow does not adequately capture such behavior. Even more, interaction of different information flows become more and more frequent in both modern benign and malicious apps.

Complex-Flows [59] is a new representation of information flow usage inside apps. It does not only reveal what types of sensitive data are leveraged by apps, but also reveals interactions between different information flows as well as what computation app performs along with information flows. To illustrate how Complex-Flows work, let us consider one benign and one malicious app that contain the same (source, sink) flows shown in Table 27. The benign app, com.kakapo.bingo.apk, is a popular bingo app available in Google Play. The malicious app masquerades as a video player, but it also starts a background service to send out premium messages and steals phone info including IMEI, IMSI. Both apps send out phone identifiers (IMEI, IMSI) over the Internet. Fig. 37 shows the Complex-Flow view of these two apps. In this case, we can
Table 26: Top 10 Sources in BlueBench

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>TelephonyManager:getDeviceId()</td>
<td>HttpClient:execute</td>
</tr>
<tr>
<td>TelephonyManager:getSubscriberId()</td>
<td>HttpClient:execute</td>
</tr>
</tbody>
</table>

Table 27: Information Flows in Both Benign and Malicious Apps

Distinguish these two apps by analyzing app behavior extracted in Complex-Flows, which cannot be captured in information flows with simple (source, sink) pairs.

In this case, by analyzing Complex-Flow in Android malware, we can better understand malicious apps in apps and it can be a very important security factor in analyzing malware apps for future tools. First, we provide Complex-Flow statistics from modern malware apps and show Complex-Flow is now leveraged by more and more malicious apps. Figure 38 shows statistics based on normalized app LOC size (per 1,000 lines of code). As shown in the figure, there are modern malware apps that contain more than 10 Complex-Flows with in 1,000 lines of code. This means modern malicious apps perform manipulation on sensitive data more and more often than before. Also, since each Complex-Flow can involve multiple sources/sinks. It indicates that dangerous operations in modern malware apps interact with each other more frequently. Next, we explore the relation between real Complex Flow Count and app LOC size. The result is shown in Figure 39. In this figure, the x-axis represents sorted app list based on app LOC from small to big. The result shows that there is no clear relation between number of Complex-Flows and app size. However, most small sized apps contain fewer than 100 Complex-Flows, while many large sized apps contain more than that. This can tell us that in modern malicious apps, the more lines of code an app contains, the more complicated dangerous behavior it can exhibit.

Besides the simple statistics above, we also provide more detailed complex-flow statistics similar to simple dataflows:

- **RQ1: what types of information are interested in apps?**
  Overall top 10 Multi-Flows in modern malware apps are shown in Table 30. We examine the presence of top Multi-Flows listed against MalGenome project. There is no MalGenome app contain any of these flows.

- **RQ2: what are the most commonly captured behavior in apps?**
  Top 10 sequences with size-1 and size-2 are shown in Table 28 and Table 29. This shows that modern malware apps are interested in various information from devices and intend to combine them together for malicious usage. And the top API sequences(grams) are mostly sensitive information related. We can conclude that malware apps try to access and aggregate multiple sensitive data sources.
Figure 37: App Behavior Comparison in Benign and Malware Apps

Figure 38: BlueBench Complex Flow Usage Normalized Distribution.

4.8 Dataset App Selection

To select the final benchmark apps out of the whole dataset, we consider metrics mentioned above and define some rules. By applying these rules, we build up a benchmark set with 1,000 apps. The selection process is based on following rules:

1. Apps should contain a manifest file with various components defined. This is because there are many app featured behavior defined in AndroidManifest.xml file (e.g., permissions and intent-filterers). These features are very important factors for malware detection.
2. An app should contain reflection usage. As we discussed, more and more malware leverage reflection with obfuscated string to dynamically load real payload. Modern malware detection tools should be aware of these apps.

3. There should be apps that leverage native code invocation. Native code payload in malware apps is one of the most difficult behavior to detect, which is leveraged by modern malware apps heavily. It is essential for future tools to be able to distinguish them.

4. Apps with dynamic class loading and runtime execution usage should be included, since we have observed that many modern malware apps favor these techniques to hide malicious behavior.

5. Benchmark set should contain apps with information flow. Data leakage is one of most popular attacks in Android malware. Information flow is another important factor used to
Table 29: Top 10 Size-2 Sequences in BlueBench

<table>
<thead>
<tr>
<th>sink</th>
<th>source(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intent:putExtra</td>
<td>getDeviceId</td>
</tr>
<tr>
<td></td>
<td>getLine1Number</td>
</tr>
<tr>
<td></td>
<td>getSimSerialNumber</td>
</tr>
<tr>
<td></td>
<td>getSubscriberId</td>
</tr>
<tr>
<td>PrintStream:println</td>
<td>getDeviceId</td>
</tr>
<tr>
<td></td>
<td>getLine1Number</td>
</tr>
<tr>
<td></td>
<td>getSimSerialNumber</td>
</tr>
<tr>
<td></td>
<td>getSubscriberId</td>
</tr>
<tr>
<td>Intent:setDataAndType</td>
<td>getDeviceId</td>
</tr>
<tr>
<td></td>
<td>getLine1Number</td>
</tr>
<tr>
<td></td>
<td>getSimSerialNumber</td>
</tr>
<tr>
<td></td>
<td>getSubscriberId</td>
</tr>
<tr>
<td>HttpClient:execute</td>
<td>getDeviceId</td>
</tr>
<tr>
<td></td>
<td>getLine1Number</td>
</tr>
<tr>
<td></td>
<td>getSimSerialNumber</td>
</tr>
<tr>
<td></td>
<td>getSubscriberId</td>
</tr>
<tr>
<td>DefaultHttpClient:execute</td>
<td>getDeviceId</td>
</tr>
<tr>
<td></td>
<td>getLine1Number</td>
</tr>
<tr>
<td></td>
<td>getSimSerialNumber</td>
</tr>
<tr>
<td></td>
<td>getSubscriberId</td>
</tr>
</tbody>
</table>

Table 30: Top 10 MultiFlows in BlueBench

differentiate malware apps from benign apps. It can tell us how sensitive data are leveraged inside apps. However, simple information flow is not sufficient any more to detect modern malware apps. In this case, we use Complex-Flow instead, which will provide us more detailed app behavior along with information flows.

5 Complex-Flow for Malware Detection

In this section, we discuss our malware classification system using Complex-Flow. Complex-Flow is a new representation of data usage in Android apps. As we mentioned above, simple information flow itself will not be sufficient to distinguish malware apps from benign apps. To tackle this, Complex-Flow takes a step further to analyze app behavior along with information flows.
public static String getLmMobUID(Context context) {
    ...
    TelephonyManager tm = (TelephonyManager)
            context.getSystemService("phone");
    if (isPermission(context, "android.permission.READ_PHONE_STATE"))
        localStringBuffer.append(tm.getDeviceId());
    ...
}

public static String getImsi(Context context) {
    TelephonyManager tm = (TelephonyManager)
            context.getSystemService("phone");
    param = tm.getSubscriberId();
    ...
}

Figure 40: Data Access Code Snippet in Benign App

5.1 Motivation

We already illustrate how both modern benign and malicious apps can confound malware detectors leveraging information flows. Here, we use the same example apps discussed in Section 4.7. We know that even if we can detect the information flows in example apps, we cannot distinguish these two apps.

To combat this problem, many previous approaches would consider sending of phone identifiers as an indication of malicious intent [70]. This approach worked well for some time as this was often considered privileged information. However, we and others [14] [68] have noticed that sending this information is becoming more common in benign apps, usually as a secondary authentication token for banking apps, or in the case of our bingo app and many other games, as a way to uniquely identify a user. In general, it has become more common that benign apps require additional information to provide in-app functionality. Many ad engines collect this kind of information as well [50]. Thus, it is difficult to tell which apps are benign and which are malicious by examining source and sink pairs alone. More information is required to differentiate these two apps.

Let us examine how both our example apps access sensitive data, to see if we can differentiate between them. We present the bingo app and the malicious app in the form of decompiled DEX bytecode (Android’s bytecode format) in code snippets 40 and 41, respectively. We observe that the benign bingo app accesses the sensitive data it requires in lines 6, and 12, whereas the malicious app collects the sensitive data in aggregate in a single method in lines 3-4. The malicious app also bundles the data in lines 5-8 and sends the aggregated data over the network in line 10. In contrast, our bingo app does not send data immediately after collecting it. As shown in this example, the two apps contain the same information flows, but the structure of these flows is quite different.

The difference becomes even more profound if we examine the computation the apps perform along the code path of the information flow. Previous studies [39, 28] have shown that system call sequences effectively capture the computations done in a program; thus, we examine the API call sequences occurring along the flows in both benign and malicious apps, and compare them.

37 shows the information flow view of these two apps. In particular, we use the flow `TelephonyManager:getSubscriberId → HttpClient:execute` as an example to illustrate the differences in benign and malicious apps. 42 and 43 show the API call sequences occurring along the flow. The lines in black show the same behavior of the two apps, with both preparing to fetch the IMSI. The difference between the apps is highlighted in red. The malicious app fetches another phone
private void execTask()
{
    ...
    this.imei = localObject2.getDeviceId();
    this.imsi = localObject2.getSubscriberId();
    str2 = "http://" + Base64.encodebook("2maodb3ialke8mdeme3gkos9g1icaofm", 6, 3) + "/mm.do?imei=" + this.imei;
    localStr2 = str2 + "&imsi=" + this.imsi;
    ...
    paramString1 = ((HttpClient)localObject).execute(localStr2);
    ...
}

Figure 41: Data Access Code Snippet in Malware App

<table>
<thead>
<tr>
<th>Context: getSystemService(String)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Context: getSystemService(String)&gt;</td>
</tr>
<tr>
<td>&lt;TelephonyManager: getSubscriberId() &gt;</td>
</tr>
<tr>
<td>&lt;TelephonyManager: getDeviceId() &gt;</td>
</tr>
<tr>
<td>&lt;BasicNameValuePair: &lt;init&gt;(String,String)&gt;</td>
</tr>
<tr>
<td>&lt;URLEncodedUtils: format(List,String)&gt;</td>
</tr>
<tr>
<td>&lt;XmlServerConnector: byte[] zip(byte[]) &gt;</td>
</tr>
<tr>
<td>&lt;HttpGet: void &lt;init&gt;(String)&gt;</td>
</tr>
<tr>
<td>&lt;DefaultHttpClient: void &lt;init&gt;() &gt;</td>
</tr>
<tr>
<td>&lt;HttpClient: getParams()&gt;</td>
</tr>
<tr>
<td>&lt;HttpParams: setParameter(String, Object)&gt;</td>
</tr>
<tr>
<td>&lt;HttpClient: getParams()&gt;</td>
</tr>
<tr>
<td>&lt;HttpPost: setParameter(String, Object)&gt;</td>
</tr>
<tr>
<td>&lt;HttpClient: execute(HttpUriRequest)&gt;</td>
</tr>
</tbody>
</table>

Figure 42: API Call Sequence in Malware App

identifier(IMEI) (line 3) right after fetching IMSI, then couples this data (line 5) and compresses it (line 6). The benign app, on the other hand, simply checks and uses the network (lines 3-5).

This example shows that by comparing the API sequences we can infer that these two apps differ in behavior even though they share the same information flows. Traditional data flow analysis fails to differentiate malicious app behavior from benign one if they both leverage the same set of sensitive data, since it misses the relation of different information flows and the different behavior of these two apps. To show how realistic this behavior difference is in real-world apps, we examine both benign and malware test dataset apps leveraging same APIs with encoding and compression as shown in the above example. It turns out that there are 188 malware apps that contain the same behavior while there are only 3 benign apps that exhibit such behavior. In our approach, we leverage this insight and represent a set of related simple flows as a Complex-Flow, and develop a machine learning technique to discover which behavior along information flows and Complex-Flows are indicative of malicious code. We further describe this in the following section.

5.2 Complex-Flows

The analysis of our example apps revealed that it is common for multiple data flows to access sensitive resource data. However, the intent, purpose, and net effect of these operations often differ between malicious and benign code. In this section we propose the concept of a Complex-Flow, a mechanism that captures the usage of sensitive mobile resources, but also reveals the
structure of this usage as well as the relation between different uses.

5.2.1 Multi-Flows

To compute Complex-Flows, we must first discover the relationships between simple flows. We call simple flows which are computationally related to one another, either by data flow or control flow, Multi-Flows. Abstractly, a Multi-Flow is composed of multiple simple flows, such that any two simple flows in the Multi-Flow share a subset of their computation.

Let $SRC$ be the data source an app accesses. Let $SNK$ be the sink point the data flows into. Let $S_n$ be an intermediate statement in the program where the source data or data derived from the source data is used (i.e. a data flow).

**Definition 1** A simple flow, $SRC \rightarrow SNK$, is composed of a sequence of statements $\tilde{S}$, which includes $SRC$ and $SNK$: $\tilde{S} = SRC \leadsto S_1 \leadsto S_2 \leadsto \ldots \leadsto S_{n-1} \leadsto S_n \leadsto SNK$. We say that a sequence $\tilde{S}$ is a subsequence of a flow $F$, written as $\tilde{S} \subseteq F$, if $\tilde{S}$ is contained within $F$.

**Definition 2** A Multi-Flow represents multiple simple flows that share common computation within a program. Let $\bar{F}$ be a set of all simple flows in a program. A Multi-Flow for a sequence $\tilde{S}$, $\bar{F}(\tilde{S})$, is a set of simple flows in $\bar{F}$ that share $\tilde{S}$ as a common subsequence. It is defined as:

$$\bar{F}(\tilde{S}) = \{F_i | F_i \in \bar{F} \text{ and } \tilde{S} \subseteq F_i\}$$

Thus, the simplest Multi-Flow occurs when two simple flows share the same source or sink. It is important to distinguish that by source and sink we not only mean a given API call, but where that API occur within the program. 5.1 provides a real-world Multi-Flow example with multiple device identifiers collected at once and sent out over the network. Here, the data is sent out not only just over the same sink, but also over the same control flow path.

5.2.2 Complex Flows

Information flow analysis focuses on discovering the start and end points of data flows, whether they be simple flows or Multi-Flows. Analysis of the computations captured by Complex-Flows is required to gain understanding of the behavior of the Multi-Flow. Specifically we focus on the discovery of the interactions between an app and platform framework: if an app wants to send DeviceId over network, it must leverage the public network APIs of the platform framework to complete this operation. Or if the app wants to write DeviceId via the logging system, it must

Figure 43: API Call Sequence in Benign app
invoke the APIs of the Android provided \texttt{android.uti.Log} package. Even if the app does nothing but simply display sensitive information on screen, it still must do so through the framework GUI APIs. Below, we provide a formal definition of Complex-Flows.

**Definition 3** Let $\hat{S}$ be a simple flow $\text{SRC} \rightarrow S_1 \rightarrow \ldots \rightarrow S_n \rightarrow \text{SNK}$, where \text{SRC} is a source, \text{SNK} is a sink, and $S_i$ is a program statement. We define an API sequence of $\hat{S}$ as a filtered sequence over $\hat{S}$ that only contains API call statements. Note that both the source and sink are API calls by definition.

For a formal definition of an API sequence, we write $S \in \overline{\text{API}}$, if the statement $S$ is a call to an API function. Then an API sequence of $\hat{S}$ is produced by filtering $\hat{S}$ recursively using the following three rules, which essentially removes all non-API calls from a simple flow (below, $S$ is a single statement, and $\hat{S}'$ is a sequence of statements):

\begin{align*}
\text{Rule 1} : \overline{\text{filter}}(S \rightarrow \hat{S}') &= S \rightarrow \overline{\text{filter}}(\hat{S}') \quad \text{if } S \in \overline{\text{API}} \quad (1) \\
\text{Rule 2} : \overline{\text{filter}}(S \rightarrow \hat{S}') &= \overline{\text{filter}}(\hat{S}') \quad \text{if } S \notin \overline{\text{API}} \quad (2) \\
\text{Rule 3} : \overline{\text{filter}}(\emptyset) &= \emptyset \quad (3)
\end{align*}

**Definition 4** We define a Complex Flow $\text{CF}$ in terms of a Multi-Flow, $F(\hat{S})$ as the set of filtered sequences (i.e., API sequences - $\overline{\text{AS}}$) for each flow in the Multi-Flow:

$$\text{CF} = \{\overline{\text{AS}}|\overline{\text{AS}} = \overline{\text{filter}}(F), F \in F(\hat{S})\}.$$ 

**Definition 5** An N-gram API set is a set of API sequences of size $N$ derived from an API sequence. Formally, a set of N-grams over a filtered sequence is defined as follows, where $|\hat{S}'|$ denotes the size of the filtered sequence $\hat{S}'$:

$$\text{N-gram}(\hat{S}) = \{\hat{S}'|\hat{S}' \subseteq \hat{S}, |\hat{S}'| = n\}$$

**Definition 6** We define all N-grams for a Complex Flow $\text{CF}$ as a set of N-gram API sets, one derived from each filtered sequence $\overline{\text{AS}}$ contained in the Complex Flow:

$$\{\overline{\text{NG}}|\overline{\text{NG}} = \text{N-gram}(\overline{\text{AS}}), \overline{\text{AS}} \in \text{CF}\}.$$ 

We extract the app’s framework API call sequences to capture the computations performed over sensitive data. We only include those sequences present within Complex-Flows. A Complex-Flow, represented as a set of sequences of APIs, including the source and sink pairs of all simple flows present in the Multi-Flow.

### 5.3 System Design

We have built an automated malware detection system that classifies apps as malicious or benign via analyzing the N-gram representation of Complex Flows described in Sections 5 and 5.2. This classification system is integrated into our BlueSeal compiler 3, a static information flow analysis engine originally developed to extract information flows from Android apps. It also can handle information flows triggered by UI events and sensor events. BlueSeal is context sensitive but is not path sensitive. It takes as input the Dalvik Executable (DEX) bytecode for an app, bypassing the need for an app’s source. BlueSeal is built on top of the Soot Java Optimization Framework [62] and leverages both intraprocedural and interprocedural data flow analysis. In addition, BlueSeal is able to resolve different Android specific constructs and reflection.

Our implementation extends BlueSeal to discover Complex-Flows in addition to its native capability to detect simple information flows. The automated classification component performs the following four analysis phases to generate features and perform classification of apps as malicious or benign: (1) Multi-Flow discovery, (2) API call sequence extraction, (3) N-gram feature generation, and (4) Classification. Our tool is open-source and available on the BlueSeal website.
5.3.1 Multi-Flow Discovery

Traditional information flow analysis mainly focuses on the discovery of a flow from a single source to a single sink. We have extended BlueSeal to extract Multi-Flows, where individual single source to a single sink flows are aggregated and connected. We leverage data flow analysis techniques to extract paths contained within each simple flow. If two information flows share a subpath then these two information flows belong to the same Multi-Flow. Each Multi-Flow can contain multiple information flows, which means it can contain multiple sources and multiple sinks. We then analyze these Multi-Flows to extract API sequences present within the Multi-Flow to create Complex-Flows.

The goal of the Multi-Flow detection algorithm is to: (1) create a global graph of complete information flow paths for an app, and (2) detect the intersection between individual information flow paths that represent Multi-Flows. Here, the intersection of two information flow paths simply means two information flow paths share at least one node in the global graph. The Multi-Flow detection algorithm itself works by taking as input BlueSeal’s natively detected individual information flow paths, which track simple flows with a single source and single sink. To generate Multi-Flows, we augment BlueSeal as follows. 1) Whenever we encounter a statement containing sensitive API invocation (which accesses a device’s sensitive data), we add the invocation as a node in the global graph. This is considered the starting point of a data flow path. 2) Next, we check each program statement to see if there is a data flow from the current statement to the initial, detected statement. If so, we build an intermediate source node in the global data flow graph, adding an edge from the node for the initial statement. This step is recursive and if there is a data flow from another program statement to the intermediate source node, we create a new intermediate source node as above. These intermediate nodes are critical as they connect together single flows to create Multi-Flows. 3) The data flow’s path ends when we find a sink point. These three types of points (i.e., source, intermediate, and sink) are able to capture the whole data flow path for a simple information flow while simultaneously outputting a global graph that includes all, potentially interconnected, data flow paths. 4) Multi-Flows are detected by iterating through this global graph, finding simple data flows as well as Multi-Flows. 5) We extract API call sequences for all Multi-Flows. While doing so, we analyze control-flow paths in each Multi-Flow to extract API call sequences. We discuss this further next.

As mentioned in [60], we note that our BlueSeal engine will address Android specific constructs. Android Intent will be treated as a sink since it’s a potential point to leak data outside. Also, reflection will be resolved if it can be statically determined. Readers can refer to [60] for details.

5.3.2 Complex-Flow Extraction with API Sequences

Although the previous phase gives us the global graph for an app with all Multi-Flows, it does not provide the exact API call sequences occurring along the Multi-Flows, i.e., Complex-Flows. Analyzing Complex-Flows requires us to consider control paths with branches and loops, since they produce separate code paths. For example, if there is an if-else block in-between a source and a sink, there can be two separate API sequences that start with the same source and end with the same sink. Thus, we develop a mechanism to examine all code paths along the Multi-Flows detected by the previous phase, and extract the API call sequences.

Technically, this can be done within the previous phase, as the original BlueSeal implementation already considers control paths when analyzing data flows. However, we implement our API sequence extraction as a separate phase for clean separation of our new logic.
private void PhoneInfo(){
    imei = Object2.getDeviceId();
    mobile = Object2.getLine1Number();
    imsi = Object2.getSubscriberId();
    iccid = Object2.getSimSerialNumber();
    url = "http://"+str1+.xml?sim="+imei+
    "&tel="+mobile+"&imsi="+imsi+"&iccid="+iccid;
    Object2 = getStringByURL(Object2);
    if ((Object2 != null) && (!"".equals(Object2))){
        sendSMS(this.destMobile, "imei:" + this.imei);
    }else{
        writeRecordLog(url);
    }
}

private void sendSMS(String str1, String str2){
    SmsManager.getDefault().sendTextMessage(str1, null, str2,null,null,0);
}

private void writeRecordLog(String param){
    Log.i("phoneinfo", param);
}

public String getStringByURL(String paramString){
    HttpURLConnection conn = (HttpURLConnection)new URL(paramString).openConnection();
    conn.setDoInput(true);
    conn.connect();
    return null;
}

Figure 44: API Call Sequence Extraction Example

Figure 45: Data Flow Structure of Example Code Snippet

We illustrate this process with an example. 44 is a code snippet extracted from a known malicious app. For simplicity, we remove other pieces of code not pertinent to our discussion. The general code’s data flow structure is shown in 45 and the corresponding control-flow graph is shown in 46. 44 and 46 show that there are two execution paths that must be extracted from the larger, singular Multi-Flow structure shown in 45. Thus, we output one API call sequence for each single path. The final output of the example code snippet is shown in 31.

In order to extract such API sequences, we analyze each control flow path, statement by statement, in the execution order to extract all platform APIs invoked along with Multi-Flows. As
mentioned earlier, we consider different branches separately, which means that for each branch point, we create two separate branch paths. For a loop, we consider the execution of its body once if an API is invoked inside a loop. This is due to the fact that precise handling of loops itself is a challenging problem and an active area of research, which requires loop bound analysis followed by unrolling each loop for \(N\) times where \(N\) is the analyzed bound for the given loop. Previous work proposes a mechanism to precisely handle loops in Android apps [32]; it is our future work to incorporate it. It is worth mentioning here that we have an opportunity to reduce the complexity of precise loop analysis, since our N-gram analysis described next has a maximum bound for an API call sequence, i.e., we are only interested in an API call sequence of size \(N\). This means that we only need to unroll a loop enough times to get an API call sequence of size \(N\), which reduces the complexity of handling loops. However, we leave the full investigation of this as our future work.
Next, our system uses the API call sequences extracted in the previous step to generate features for classification purposes. As mentioned above, the API sequences are the interaction between app and platform, and they represent app behavior regarding sensitive data usage. We use the N-grams technique to generate these features from the API call sequences as N-grams. Traditionally, the N-grams technique uses byte sequences as input. In our approach, we generate N-grams using API call sequences as input to reveal app behavior. We consider each gram to be a subsequence of a given API call sequence. Sequence N-grams are overlapping substrings, collected in a sliding-window fashion where the windows of a fixed size slides one API call at a time. Sequence N-grams not only capture the statistics of sub-sequences of API calls of length $n$ but implicitly represent frequencies of longer call sequences as well. A simple example of an API sequence and its corresponding N-grams is shown in Table 32. In detail, the first 2-gram indicates that the app accesses the IMEI and phone number at once; while the second 2-gram indicates that the app accesses the phone number and IMSI at once.

### 5.3.4 Classification

The last step of our classification tool is leveraging classification techniques on top of N-grams features obtained from the Complex-Flow representation. The goal of this classification to identify discriminative behavior between malicious and benign apps. In this process, our system is trained using both benign and malicious apps in order to capture and distinguish behavior patterns between them. In addition, our malware dataset contains malware apps from different families, which captures behavior patterns exhibited in different types of malicious families.

We generate N-grams for each app analyzed and then use every N-gram in any app as a feature to form a global feature space. Based on this feature space, we generate a feature vector for each app, taking the count of each gram feature into consideration. For example, if a gram feature appears three times in an app, the corresponding value of this gram feature in app’s feature vector will be three. Finally, we feed app feature vectors into the classifier. We use two-class SVM classification to determine whether an app is malicious or benign. The SVM model is a popular supervised learning model for classification and also leveraged by other systems to perform malicious app detection [14].

We note that determining whether an app is malicious or not is rather a complex problem [48], as demonstrated by the multi-billion USD antivirus industry. Furthermore, we note that the malware dataset we use for training is obtained with a non-disclosure agreement from a threat intelligence company operating in the United States and Europe. Thus, the initial detection of malicious apps, constituting the ground-truth in our study, falls out of the main scope of this work. Nevertheless, to briefly summarize the malware collection process, it follows the industry’s best practices—it utilizes dynamic and static analysis (against predefined sets of rules similar to the

<table>
<thead>
<tr>
<th>API Sequence</th>
<th>2-grams</th>
</tr>
</thead>
<tbody>
<tr>
<td>TelephonyManager:getDeviceId()</td>
<td>TelephonyManager:getDeviceId()</td>
</tr>
<tr>
<td>TelephonyManager:getLine1Number()</td>
<td>TelephonyManager:getLine1Number()</td>
</tr>
<tr>
<td>TelephonyManager:getSubscriberId()</td>
<td>TelephonyManager:getSubscriberId()</td>
</tr>
</tbody>
</table>

Table 32: Example of API Sequence and its 2-grams

### 5.3.3 N-gram Feature Generation

5.3.3 N-gram Feature Generation

Next, our system uses the API call sequences extracted in the previous step to generate features for classification purposes. As mentioned above, the API sequences are the interaction between app and platform, and they represent app behavior regarding sensitive data usage. We use the N-grams technique to generate these features from the API call sequences as N-grams. Traditionally, the N-grams technique uses byte sequences as input. In our approach, we generate N-grams using API call sequences as input to reveal app behavior. We consider each gram to be a subsequence of a given API call sequence. Sequence N-grams are overlapping substrings, collected in a sliding-window fashion where the windows of a fixed size slides one API call at a time. Sequence N-grams not only capture the statistics of sub-sequences of API calls of length $n$ but implicitly represent frequencies of longer call sequences as well. A simple example of an API sequence and its corresponding N-grams is shown in Table 32. In detail, the first 2-gram indicates that the app accesses the IMEI and phone number at once; while the second 2-gram indicates that the app accesses the phone number and IMSI at once.
rules employed by major antivirus scanners incorporated in portals such as VirusTotal), as well as contexts (users’ reports) to determine whether an app is malicious or benign. The majority of the apps are determined to be malicious against VirusTotal detection, which utilizes a combination of the above static/dynamic techniques as well as permission use and abuse.

Since we rely on our malware dataset for training, the maliciousness of an app is essentially determined by whether or not the app shows behavior patterns similar to the patterns exhibited in our malware dataset. Here, the behavior patterns are represented by Complex-Flows. Likewise, the benignness of an app is determined by whether or not the app shows behavior patterns more similar to the patterns exhibited in our benign training dataset.

We note that our benign dataset is diverse, and covers popular apps as well as apps from different categories. As those apps do not show a positive detection in public tools, such as VirusTotal, and are not indicated by the Play Store as such, we have assumed them being benign (more details are in Section 5.4.9, when addressing false detection analysis).

Our malware dataset is also diverse, containing hundreds of different families reported by different detection tools. Despite the variety of behavior exhibited in the different families, our system is shown to learn how different types of malware leverage information flows and what types of behavior it contains. We discuss this in more details in Section 5.4.

5.4 Evaluation

Correct selection of training data in classification is very important. There are cases where classifiers work extremely well on one set of data but fail on other sets of data due to over-training [55]. We use four sets of apps with different characteristics to better evaluate our tool against different benign and malicious app set combinations and avoid the over-training pitfall. Two sets consist of benign apps and two sets consist of malicious apps. All datasets and scripts for evaluation are available. Please visit http://blueseal.cse.buffalo.edu/ for details.

Benign apps. The benign apps are free apps downloaded from Google Play and include two sub-sets. One contains the top 100 most popular free apps across all categories (i.e. Art & Design, Beauty, Books & Reference, etc.) from January, 2014 and the other contains random free apps across multiple categories from Oct, 2016. We have used 3,899 apps in total from the set of apps downloaded, excluding the apps that either have no flow reported by our tool or exceed the execution time limit set for processing the app (60 minutes). This execution time limit is needed because some apps take hours to finish while more than 90% of the apps can be analyzed in well under an hour.

Malicious apps. We use two malware data sets. The first set is from the MalGenome project [70]. We leverage it as a comparison point and to aid in reproducibility as previous studies rely heavily upon it. The other malicious apps are from a dataset of over 70,000 malware samples obtained from security operations over a month by a threat intelligence company operating in the United States and Europe. Due to a non-disclosure agreement this set is not publicly available. Each app from the 70K set has been scanned through multiple popular anti-virus tools. Meta data is associated with each app including scan results from each anti-virus tool, time discovered, description of the app and so on. Out of the entire set, we have randomly selected 3,899 apps that contain information flows to match up the number of benign apps. Analogous to the benign apps, MalGenome apps represent older, outdated apps while the other set represents new, modern malware apps.

We label each of the four datasets as follows:

- **Play 2014**: Apps collected from Google Play in Jan, 2014. The total number of apps is 800.
- **Play 2016**: Apps collected from Google Play in Oct, 2016. The total number of apps is 3,099.
• MalGenome: Malware apps collected from MalGenome project. The total number of apps is 800.

• Malware: Another set of malware apps collected from intelligence company. The total number of apps is 3,899.

5.4.1 Evaluation Methodology and Metrics

We have used different combinations of these four sets in our experiments to evaluate our classification system. The evaluation process is as follows:

• We use the 10-fold cross-validation technique to divide apps into a training set and a testing set. We trained the classifier on the feature vectors from a random 90% of both benign and malicious apps. The remaining 10% form the testing dataset. Then we rotate on the training and testing dataset. The classification process will be repeated ten times in total and we calculate the results average. This is a commonly used statistical analysis technique.

• The training set is based on both benign and malicious apps. N-grams generated from these apps are used to form the global feature space. For each app, a feature vector is built based on N-gram features.

• Then feature vectors of apps of the training set are used to train a two-class SVM classifier.

• Lastly, after training, we use the testing set of mixed benign and malicious apps for classification. The classifier then provides a decision on an app, based on its N-grams feature vector, as either ”malicious” or ”benign”.

Upon completion, we collect statistics based on the classification results. We use the following four metrics for our evaluation:

TP True positive rate—the rate of benign apps recognized correctly as benign.

TN True negative rate—the rate of malware recognized correctly as malicious.

FP False positive rate—the rate of malware recognized incorrectly as benign.

FN False negative rate—the rate of benign apps recognized incorrectly as malicious.

The rest of this section details the results.

5.4.2 Runtime Performance

As mentioned earlier, we set an execution time limit on extracting Complex-Flows. Here, we collect statistics on system performance of app execution time of modern malware. All tests are running on machines with twelve Intel(R) Xeon(R) CPU E5645(2.40GHz, 12M Cache). For each app, we assign 6G memory to JAVA Virtual Machine. Our system is able to analyze and extract Complex-Flows for 90% of testing apps. There are a few apps (<130 apps) that exceed this execution limit. Among these apps, we increase the time limit to two hours and half of these apps (86 apps) can be analyzed. Only a few apps throw out of memory exception. By increasing JVM memory to 16G and time limit to 24 hours, all apps are able to be analyzed. One of the apps takes around eight hours to finish and all others are able to be done under four hours. Fig. 47 shows the full performance results for apps analyzed under an hour limit. As we can tell from this result, our system can analyze and extract Complex-Flows for 90% of the apps under ten minutes. Only 10% of the apps requires an analysis time greater than that, but can be analyzed in an hour.
5.4.3 Play 2014 Apps versus MalGenome Apps

We first show our results with the older benign apps (Play 2014) and the older malicious apps (MalGenome). Figure 47 shows that when the gram size is small, it is sufficient to differentiate the MalGenome apps from the benign apps. For the gram size of 1, we achieve a true positive rate of 97.5% and the true negative rate of 85.2%.

A manual examination of single API usage in both benign and malware apps shows there are 4,457 distinct APIs in benign apps while there are only 813 for malware apps. The overlap of these two sets is 771. An examination of these 771 shared API calls shows that most of them are sensitive APIs. This indicates that the Malgenome contains malware that is heavily reliant on a specific set of APIs when compared with the benign apps. We also looked at APIs that are exclusive in malware apps, and found them to be primarily APIs related to device WiFi status and database permissions. These APIs are also commonly used in benign apps as well. However, they are not captured along with information flow paths. From this, we can conclude that the usage of single API calls between the benign apps and the MalGenome apps is quite different. By examining the sensitive APIs involved in information flows in these apps, the most common flows indicates that data is often used inside the app for benign apps, while in malicious apps data mainly flows to the network and storage. As we increase the gram size, we gain more precision for classifying malicious apps while losing precision for classifying benign apps. This is anticipated, since benign apps are more diverse than malicious apps. Increasing the gram size causes a loss of common behavior pattern in benign apps. To this end, we conclude that MalGenome apps are less complicated than benign apps – a result confirmed by our manual inspection. Another conclusion we can make is that MalGenome apps are more interested in a certain set of single APIs heavily compared to benign apps. This can be captured from the fact that the gram size of 1 is enough to differentiate these malware apps from benign apps.

We also evaluated classification on combined gram features by aggregating different gram size features together as our global feature space. The result is shown as the last 4 rows in Figure 47. By aggregating different gram features, we can achieve high accuracy rates in classifying both benign apps and malicious apps. We can also see that by aggregating different gram features,
Table 33: Gram Based Classification Results of Play_2014 and MalGenome Apps

<table>
<thead>
<tr>
<th>gram size</th>
<th>TP</th>
<th>TN</th>
<th>FP</th>
<th>FN</th>
<th>accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.975</td>
<td>0.852</td>
<td>0.148</td>
<td>0.025</td>
<td>0.913</td>
</tr>
<tr>
<td>2</td>
<td>0.950</td>
<td>0.699</td>
<td>0.301</td>
<td>0.050</td>
<td>0.822</td>
</tr>
<tr>
<td>3</td>
<td>0.980</td>
<td>0.688</td>
<td>0.312</td>
<td>0.020</td>
<td>0.831</td>
</tr>
<tr>
<td>4</td>
<td>0.549</td>
<td>0.948</td>
<td>0.052</td>
<td>0.451</td>
<td>0.753</td>
</tr>
<tr>
<td>5</td>
<td>0.485</td>
<td>0.952</td>
<td>0.048</td>
<td>0.515</td>
<td>0.725</td>
</tr>
<tr>
<td>1,2</td>
<td>0.81</td>
<td>0.92</td>
<td>0.08</td>
<td>0.19</td>
<td>0.868</td>
</tr>
<tr>
<td>1,2,3</td>
<td>0.886</td>
<td>0.84</td>
<td>0.16</td>
<td>0.114</td>
<td>0.863</td>
</tr>
<tr>
<td>1,2,3,4</td>
<td>0.759</td>
<td>0.889</td>
<td>0.111</td>
<td>0.241</td>
<td>0.825</td>
</tr>
<tr>
<td>1,2,3,4,5</td>
<td>0.696</td>
<td>0.938</td>
<td>0.062</td>
<td>0.304</td>
<td>0.819</td>
</tr>
</tbody>
</table>

we can achieve better precision than using a single gram. However, we also degrade classifier performance by adding too much information. This is captured by the fact that gram-1,2,3,4,5 has worse precision when compared to other combinations.

5.4.4 Google Play Apps versus Modern Malware Apps

In this section, we designed different experiments to evaluate our system thoroughly based on benign apps and modern malware apps. First, we examine the old and new Google Play apps against modern malicious apps individually; then, we run analysis on all benign apps and malicious apps. To do this, we divide modern malicious apps randomly into two subsets to match up with old and new benign apps accordingly and label them as Malware_1 and Malware_2. The detailed results are discussed below.

1. 2014 Google Play Apps vs. Modern Malware Apps

First, we show the result with the older benign apps (Play_2014) and the newer malicious apps (Malware_1). The detailed results are shown in 34. Interestingly, our classification with 1-grams does not perform well in distinguishing malicious apps from benign apps. This is quite different from the result with MalGenome apps, which gives us a high precision using 1-grams. However, benign app classification still shows a high precision, since the true positive rate is 98.7%. Our classification on single gram size works best with 2-grams with the true positive rate of 95% and the true negative rate of 84.8%. In this case, we can conclude that recent malicious apps are more similar to benign apps regarding the usage of single APIs than MalGenome apps. However, the computational differences between benign and malicious apps is captured by the fact that we can still achieve very good accuracy in classification using different gram sizes.

The last 4 rows in 34 show the result or using combined gram sizes. Similar to our previous result with the MalGenome apps, the true negative rate increases along with the increase in gram size, while the true positive rate decreases. The best performance is provided by combining gram sizes of 1, 2, and 3. It has a false positive rate of only 14.8% and a false negative rate of 10.5%. We can also conclude that the aggregated feature space improves the performance more than the single gram size feature space.

2. 2016 Google Play Apps vs. Modern Malware Apps

Next, we evaluate our approach on different sets of apps. In this experiment, we have used the most recent Google Play apps, labeled as Play_2016, as our benign set. We have then chosen a different set of malicious apps, labeled as Malware_2. The result is shown in 35. The results show similar behavior as we increase the gram size. We still can achieve highly precise classification on this new set of apps, while keeping the
false positive rates low. Similar to previous results, smaller gram sizes give us better accuracy for both benign apps and malicious apps.

Additionally, we ran our classification on the new Google Play apps versus the other set of malicious apps (Malware_1). The evaluation results are shown in Table 36. As we can see, the results are very similar. These results also support our conclusion above that unlike MalGenome apps, modern malware apps are more similar to benign apps regarding the use of single APIs. However, they are still very different from benign apps from the app behavior perspective. This behavioral difference information can be leveraged to distinguish malicious apps from benign apps. Lastly, these results shows that our approach is effective across all our apps included for evaluation.

3. Simple Information Flow Based Classification

For comparison purpose, we run experiments on simple information flow (source, sink) pair based classification. The evaluation process are exactly the same as described in 5.4.1. The only difference is that we use (source, sink) pairs as features instead of examining their structure. We run two experiments over four datasets mentioned above (1,2) and show the results in Table 37. As we see from the table, the simple flow based classification does not perform well on classifying benign and malicious apps in both experiments. This is highlighted by the fact of low true negative rate(61.9%) in Play_2014 vs Malware_1 and low true positive rate(58.7%) in Play_2016 vs Malware_2. The implication is, simple information flows are insufficient to classify benign and malicious apps.

<table>
<thead>
<tr>
<th>gram size</th>
<th>TP</th>
<th>TN</th>
<th>FP</th>
<th>FN</th>
<th>accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.987</td>
<td>0.71</td>
<td>0.29</td>
<td>0.013</td>
<td>0.796</td>
</tr>
<tr>
<td>2</td>
<td>0.95</td>
<td>0.848</td>
<td>0.152</td>
<td>0.05</td>
<td>0.882</td>
</tr>
<tr>
<td>3</td>
<td>0.59</td>
<td>0.924</td>
<td>0.076</td>
<td>0.41</td>
<td>0.809</td>
</tr>
<tr>
<td>4</td>
<td>0.478</td>
<td>0.939</td>
<td>0.061</td>
<td>0.522</td>
<td>0.765</td>
</tr>
<tr>
<td>5</td>
<td>0.333</td>
<td>0.953</td>
<td>0.047</td>
<td>0.667</td>
<td>0.726</td>
</tr>
<tr>
<td>1,2</td>
<td>0.967</td>
<td>0.813</td>
<td>0.187</td>
<td>0.033</td>
<td>0.864</td>
</tr>
<tr>
<td>1,2,3</td>
<td>0.895</td>
<td>0.852</td>
<td>0.148</td>
<td>0.105</td>
<td>0.865</td>
</tr>
<tr>
<td>1,2,3,4</td>
<td>0.624</td>
<td>0.923</td>
<td>0.077</td>
<td>0.376</td>
<td>0.822</td>
</tr>
<tr>
<td>1,2,3,4,5</td>
<td>0.596</td>
<td>0.923</td>
<td>0.077</td>
<td>0.404</td>
<td>0.812</td>
</tr>
</tbody>
</table>

Table 34: Gram Based Classification Results of Play_2014 and Malware_1 Apps

<table>
<thead>
<tr>
<th>gram size</th>
<th>TP</th>
<th>TN</th>
<th>FP</th>
<th>FN</th>
<th>accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.921</td>
<td>0.767</td>
<td>0.233</td>
<td>0.079</td>
<td>0.838</td>
</tr>
<tr>
<td>2</td>
<td>0.841</td>
<td>0.863</td>
<td>0.137</td>
<td>0.159</td>
<td>0.853</td>
</tr>
<tr>
<td>3</td>
<td>0.619</td>
<td>0.863</td>
<td>0.137</td>
<td>0.381</td>
<td>0.75</td>
</tr>
<tr>
<td>4</td>
<td>0.475</td>
<td>0.948</td>
<td>0.052</td>
<td>0.525</td>
<td>0.743</td>
</tr>
<tr>
<td>5</td>
<td>0.424</td>
<td>0.948</td>
<td>0.052</td>
<td>0.576</td>
<td>0.721</td>
</tr>
<tr>
<td>1,2</td>
<td>0.968</td>
<td>0.849</td>
<td>0.151</td>
<td>0.032</td>
<td>0.904</td>
</tr>
<tr>
<td>1,2,3</td>
<td>0.857</td>
<td>0.877</td>
<td>0.123</td>
<td>0.143</td>
<td>0.868</td>
</tr>
<tr>
<td>1,2,3,4</td>
<td>0.683</td>
<td>0.877</td>
<td>0.123</td>
<td>0.317</td>
<td>0.787</td>
</tr>
<tr>
<td>1,2,3,4,5</td>
<td>0.667</td>
<td>0.89</td>
<td>0.11</td>
<td>0.333</td>
<td>0.787</td>
</tr>
</tbody>
</table>

Table 35: Classification Results on Play_2016 vs Malware_2 Apps
<table>
<thead>
<tr>
<th>gram size</th>
<th>TP</th>
<th>TN</th>
<th>FP</th>
<th>FN</th>
<th>accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.937</td>
<td>0.795</td>
<td>0.205</td>
<td>0.063</td>
<td>0.86</td>
</tr>
<tr>
<td>2</td>
<td>0.937</td>
<td>0.781</td>
<td>0.219</td>
<td>0.063</td>
<td>0.853</td>
</tr>
<tr>
<td>3</td>
<td>0.841</td>
<td>0.904</td>
<td>0.096</td>
<td>0.159</td>
<td>0.875</td>
</tr>
<tr>
<td>4</td>
<td>0.441</td>
<td>0.883</td>
<td>0.117</td>
<td>0.559</td>
<td>0.691</td>
</tr>
<tr>
<td>5</td>
<td>0.39</td>
<td>0.909</td>
<td>0.091</td>
<td>0.61</td>
<td>0.684</td>
</tr>
<tr>
<td>1,2</td>
<td>0.937</td>
<td>0.836</td>
<td>0.164</td>
<td>0.063</td>
<td>0.882</td>
</tr>
<tr>
<td>1,2,3</td>
<td>0.825</td>
<td>0.89</td>
<td>0.11</td>
<td>0.175</td>
<td>0.86</td>
</tr>
<tr>
<td>1,2,3,4</td>
<td>0.703</td>
<td>0.909</td>
<td>0.091</td>
<td>0.297</td>
<td>0.849</td>
</tr>
<tr>
<td>1,2,3,4,5</td>
<td>0.688</td>
<td>0.909</td>
<td>0.091</td>
<td>0.313</td>
<td>0.844</td>
</tr>
</tbody>
</table>

Table 36: Classification Results on Play_2016 vs Malware_1 Apps

<table>
<thead>
<tr>
<th>appset</th>
<th>TP</th>
<th>TN</th>
<th>FP</th>
<th>FN</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Play_2014vsMalware_1</td>
<td>0.869</td>
<td>0.619</td>
<td>0.381</td>
<td>0.131</td>
<td>0.744</td>
</tr>
<tr>
<td>Play_2016vsMalware_2</td>
<td>0.587</td>
<td>0.821</td>
<td>0.178</td>
<td>0.413</td>
<td>0.713</td>
</tr>
</tbody>
</table>

Table 37: Simple Information Flow Based Classification Results

5.4.5 General Google Play Apps vs. Modern Malware Apps

Lastly, in order to evaluate the effectiveness of our approach, we run our analysis over a mixed set of both benign and malicious APKs, which contains all 3,899 Google Play apps and 3,899 modern malicious apps. The detailed results are shown in 38. We have run classification analysis on single size grams as well as combined grams. As shown in the table, the results are quite similar to our previous results. Single size grams do not perform well in distinguishing malicious apps from benign apps, while combined grams work better than single grams. Our classification on combined grams works best with combination of 1-gram, 2-gram, and 3-gram with the true positive rate of 97.6% and the true negative rate of 91.0%. Again, we conclude that recent malicious apps are more similar to benign apps regarding the usage of single APIs than MalGenome apps. There might be two reasons for this. First, many modern malicious apps are repackaged apps from legitimate apps; secondly, many modern malicious apps attempt to trick people into installing their apps by delivering desired functionality using benign code. However, the fact that our classification system can still achieve very good accuracy using different gram sizes means that computational differences between benign and malicious apps play a significant role in the data sets.

Due to different complexity of apps we evenly divide both benign and malware APK set into three different categories based on app size, to verify our approach against complexity bias. The big size set contains the top 30% of the APKs based on size; the medium set contains the middle 30% of the apps; finally, the small set contains the rest of the apps. We also run the classification analysis based on these different sizes. 39, 40 and 41 shows the results for big, medium and small set respectively. As shown in 39 and 40, the results are similar. The single gram classification does not perform well, but the combined gram classification can achieve high precision. The interesting part is in classification on the small-size sets. As shown in 41, this result shows similar behavior with MalGenome apps, i.e., high precision with 1-gram classification. Indeed, these apps are similar in nature to MalGenome apps that were collected over 5 years ago—they are small in size and less complex, and exhibit simple malicious behavior.

In general, analyzing all Google play and modern malicious apps via our classification system
Table 38: Gram Based Classification Results of Google Play and Malware Apps

<table>
<thead>
<tr>
<th>gram size</th>
<th>TP</th>
<th>TN</th>
<th>FP</th>
<th>FN</th>
<th>accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.967</td>
<td>0.788</td>
<td>0.212</td>
<td>0.033</td>
<td>0.858</td>
</tr>
<tr>
<td>2</td>
<td>0.961</td>
<td>0.802</td>
<td>0.198</td>
<td>0.039</td>
<td>0.865</td>
</tr>
<tr>
<td>3</td>
<td>0.988</td>
<td>0.659</td>
<td>0.341</td>
<td>0.012</td>
<td>0.833</td>
</tr>
<tr>
<td>4</td>
<td>0.974</td>
<td>0.540</td>
<td>0.460</td>
<td>0.025</td>
<td>0.758</td>
</tr>
<tr>
<td>5</td>
<td>0.976</td>
<td>0.528</td>
<td>0.472</td>
<td>0.024</td>
<td>0.768</td>
</tr>
<tr>
<td>1,2</td>
<td>0.980</td>
<td>0.865</td>
<td>0.135</td>
<td>0.020</td>
<td>0.926</td>
</tr>
<tr>
<td>1,2,3</td>
<td>0.976</td>
<td>0.910</td>
<td>0.090</td>
<td>0.024</td>
<td>0.945</td>
</tr>
<tr>
<td>1,2,3,4</td>
<td>0.976</td>
<td>0.757</td>
<td>0.243</td>
<td>0.024</td>
<td>0.874</td>
</tr>
<tr>
<td>1,2,3,4,5</td>
<td>0.949</td>
<td>0.716</td>
<td>0.284</td>
<td>0.051</td>
<td>0.840</td>
</tr>
</tbody>
</table>

Table 39: Gram Based Classification Results of Google Play and Malware Apps with Big Size Set

<table>
<thead>
<tr>
<th>gram size</th>
<th>TP</th>
<th>TN</th>
<th>FP</th>
<th>FN</th>
<th>accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.958</td>
<td>0.685</td>
<td>0.315</td>
<td>0.042</td>
<td>0.806</td>
</tr>
<tr>
<td>2</td>
<td>0.92</td>
<td>0.762</td>
<td>0.238</td>
<td>0.08</td>
<td>0.836</td>
</tr>
<tr>
<td>3</td>
<td>0.8</td>
<td>0.75</td>
<td>0.25</td>
<td>0.2</td>
<td>0.774</td>
</tr>
<tr>
<td>4</td>
<td>0.838</td>
<td>0.797</td>
<td>0.203</td>
<td>0.162</td>
<td>0.818</td>
</tr>
<tr>
<td>5</td>
<td>0.701</td>
<td>0.763</td>
<td>0.238</td>
<td>0.299</td>
<td>0.732</td>
</tr>
<tr>
<td>1,2</td>
<td>0.972</td>
<td>0.775</td>
<td>0.225</td>
<td>0.028</td>
<td>0.863</td>
</tr>
<tr>
<td>1,2,3</td>
<td>0.915</td>
<td>0.820</td>
<td>0.180</td>
<td>0.085</td>
<td>0.863</td>
</tr>
<tr>
<td>1,2,3,4</td>
<td>0.831</td>
<td>0.798</td>
<td>0.202</td>
<td>0.169</td>
<td>0.813</td>
</tr>
<tr>
<td>1,2,3,4,5</td>
<td>0.817</td>
<td>0.798</td>
<td>0.202</td>
<td>0.183</td>
<td>0.806</td>
</tr>
</tbody>
</table>

proves that behavior analysis with Complex-flows and N-grams can achieve a good performance at distinguishing malicious apps from benign apps. However, even though the performance of the above results is good, there is one issue that the TN precision varies a lot while the TP precision remains high. We have hypothesized one possible reason for this, which is the imbalanced code size for benign and malicious apps. That is, the code size for benign apps could be much larger than malicious apps, which might affect our results.

To account for this potential size bias, we have designed another experiment that balances the code size of benign and malicious apps instead of balancing the number of apps. This set contains 876 benign apps and 1,352 modern malicious apps, but the total code size of both sets are similar. The detailed results are shown in 42. The results show slightly different behavior than our previous results in 38, i.e., the precision on malware classification increases with gram size while the precision on benign apps decreases. Nevertheless, our system can still achieve high precision distinguishing malicious apps from benign ones, achieving 96.8% on true positive and 84.9% on true negative.

5.4.6 Comparison to MudFlow

The closest related research to ours is MudFlow [14], which directly leverages information flows as features for classification and leverages machine learning techniques to classify apps in order to detect malware. Even though the main goal of MudFlow is to detect abnormal usage of information flows, authors also provide strategy to identify malware based on its flow of sensitive data.
A similar comparison using MudFlow to detect malware has been made by other researchers [34]. While MudFlow focuses on classifying apps based on information flows only, we take a step further and examine Complex-Flows. As such, MudFlow is a good comparison point for evaluating the benefit of using Complex-Flows. Thus, we evaluate whether Complex-Flows is a better factor than simple information flows in malware classification. It is not our goal to compare the performance between these two tools. We believe these two systems can be complementary to each other.

We have obtained MudFlow from the authors and run MudFlow on our evaluation datasets. However, some of the apps were not successfully processed by MudFlow. The reason for this is either that the MudFlow execution time exceeded the one-hour time limit (which we also use for our tool) or that there was simply no output generated by MudFlow. In such instances we opted to discard these apps from the dataset. Thus, we have used 605 benign apps and 876 malicious apps that are processed correctly by MudFlow. We note that the MalGenome apps are excluded from this comparison as our tests successfully reproduced the results reported in [14]. We would like to thank the authors of MudFlow for providing their tool publicly to facilitate the comparison.

MudFlow provides two different strategies for classification: one-class SVM and two-class SVM. One class SVM is trained only using benign apps, while two-class SVM is trained using both benign and malicious apps. In addition, there are two different settings in two-class SVM. For our evaluation, we have used all three settings.
Table 42: Classification Results on Google Play vs Malware Apps with Balanced Code Size

<table>
<thead>
<tr>
<th>gram size</th>
<th>TP</th>
<th>TN</th>
<th>FP</th>
<th>FN</th>
<th>accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.921</td>
<td>0.767</td>
<td>0.233</td>
<td>0.079</td>
<td>0.838</td>
</tr>
<tr>
<td>2</td>
<td>0.841</td>
<td>0.863</td>
<td>0.137</td>
<td>0.159</td>
<td>0.853</td>
</tr>
<tr>
<td>3</td>
<td>0.619</td>
<td>0.863</td>
<td>0.137</td>
<td>0.381</td>
<td>0.75</td>
</tr>
<tr>
<td>4</td>
<td>0.475</td>
<td>0.948</td>
<td>0.052</td>
<td>0.525</td>
<td>0.743</td>
</tr>
<tr>
<td>5</td>
<td>0.424</td>
<td>0.948</td>
<td>0.052</td>
<td>0.576</td>
<td>0.721</td>
</tr>
<tr>
<td>1,2</td>
<td>0.968</td>
<td>0.849</td>
<td>0.151</td>
<td>0.032</td>
<td>0.904</td>
</tr>
<tr>
<td>1,2,3</td>
<td>0.857</td>
<td>0.877</td>
<td>0.123</td>
<td>0.143</td>
<td>0.868</td>
</tr>
<tr>
<td>1,2,3,4</td>
<td>0.683</td>
<td>0.877</td>
<td>0.123</td>
<td>0.317</td>
<td>0.787</td>
</tr>
<tr>
<td>1,2,3,4,5</td>
<td>0.667</td>
<td>0.89</td>
<td>0.11</td>
<td>0.333</td>
<td>0.787</td>
</tr>
</tbody>
</table>

Table 43: MudFlow Results on Evaluation Apps

<table>
<thead>
<tr>
<th>run</th>
<th>tnr</th>
<th>tpr</th>
<th>accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>one-class</td>
<td>0.678</td>
<td>0.673</td>
<td>0.676</td>
</tr>
<tr>
<td>two-class-1</td>
<td>0.665</td>
<td>0.675</td>
<td>0.669</td>
</tr>
<tr>
<td>two-class-2</td>
<td>0.695</td>
<td>0.715</td>
<td>0.712</td>
</tr>
</tbody>
</table>

43 and 44 show the results of MudFlow and our approach. As shown, MudFlow can achieve the true negative rate of 69.5% and the true positive rate of 71.5%. In our approach, there is a significant tradeoff between true positive and true negative rates when single gram sizes are used from 1-gram to 5-gram. However, when we combine different gram sizes, we achieve much better accuracy on classification. We get the best performance result when we combine gram sizes of 1, 2, 3, and 4; the true positive rate is 92.2% and the true negative rate is 75%. We can achieve a better true negative rate (5.5% higher) and a much higher (20%) true positive rate than MudFlow. Recall that MudFlow leverages simple information flows as a feature and our approach leverage N-gram features from Multi-Flow structure for classification. As mentioned in 5.1, simple (source, sink) pair cannot distinguish two apps that contain the same flow while app behavior features extracted from Multi-Flow structure can provide us more information to distinguish malicious apps from benign apps. The results clearly prove this point. Our approach collects more information based on app behavior to achieve a better solution, while MudFlow ignores this kind of information.

Due to the number of apps evaluated, the complete effectiveness of both tools is difficult to discern. However, we believe the number of apps evaluated is enough to illustrate their effectiveness of our approach. And evaluation results also show that app behavior features captured by Complex-Flows can be a very important and effective factor to classify malware from benign apps. It is important to note that even though both MudFlow and our approach internally leverage information flows to detect malware, fundamentally we are using different feature sets for classification. Again, we believe these two approaches are complementary to each other.

5.4.7 Classification Precision and Recall Rates

To understand performance of our classification system on different malicious and benign apps, we summarize experimental results and show how our system performs. 45 shows the Precision-Recall rate based on different experiments we mentioned above. Here, we show results from Play_2014 against MalGenome to represent old malicious and benign apps, and results from Play_2016
against Malware_1 apps to represent modern malicious and benign apps. Finally, we show results from app size related experiments, which include big, medium and small sized apps. For each of them, we calculate precision and recall statistics based on different gram strategy we leverage on each dataset and present our best system performance here. Gram size column indicates the best gram strategy for each dataset. Plus, we present precision/recall curve to show an overview of our classification system performance for these datasets over all gram strategies, as shown in Figure. 48. We also highlight the best performance gram strategy as indicated with arrows in this figure. This, again, shows that our system can achieve a good performance on classification malware from benign apps by applying different gram strategies on different datasets.

As we can see, old malicious apps are easier to detect since our system can distinguish them from benign apps based on size 1 gram with a high precision and recall rate. While both modern malicious and benign apps are more complicated, single API usage analysis cannot differentiate malware from benign apps. However, different behavior patterns in malicious and benign apps can still be captured when leveraging complex combination of grams. In this dataset, we can achieve a good detection performance by combining 1-gram and 2-gram. Last but not least, modern big and medium size malicious apps are complicated and more difficult to detect than small size ones. This is shown in 45 as it requires complex strategy to achieve a good precision and recall rate from big and medium size apps while it is enough to leverage single API usage analysis for small ones.

### 5.4.8 Classification based on SVM with Balanced Training

In order to evaluate the efficiency of our classification system on imbalanced training classifier, we design another experiment with more malware apps. In this section, we managed to run another 12,000 malware apps and run our classification system over them against previous 3,899 Play Store apps. The results are shown in Table 46. Similar to other evaluation settings, we have run classifi-
classification analysis on single size grams as well as combined grams. As shown in the table, the results, show that single size gram does not perform well in distinguishing malicious apps from benign apps, while combined grams work better than single grams. Our classification on combined grams works best with combination of 1-gram, 2-gram and 3-gram with the true positive rate of 87.1% and the true negative rate of 89.2%. Interestingly, unlike the previous results, the true positive rate never achieves a high precision with single size gram classification. We believe this is caused by the imbalanced number of benign and malware apps. This also tells us that modern malware apps are more similar to benign apps. Even though there is a slightly drop on classification performance, our classification system can still achieve good accuracy using different gram sizes. We can conclude that computational differences between benign and malware apps is a very important factor in distinguishing malicious apps from benign apps.

### 5.4.9 Manual Validation

False positives and false negatives are well-known limitations of static analysis and apply to our system as well. Since benign apps are directly downloaded from Play Store, it is possible that this set contains undiscovered malicious apps. We manually validate our system based on our classifi-
cation results to account for this possibility. We choose fifty benign apps categorized as malicious by our system for manual examination. Most of these apps include multiple third-party libraries such as ad libraries, which likely contribute to the malicious rating. These libraries are complex and lack detailed documentation, making it difficult for us to determine the maliciousness of each and every apps. However, we do find some interesting cases, which we outline below.

Consider, photo.android.hd.camera.apk, which claims it is a camera usage app. We found during our manual inspection that this app includes a third party library known as umeng, which is known as a high risk adware library. In particular, this library has a capability to download and request installation of new apps. It also monitors running apps on device and sends this process list to a remote location. It also sends device information, such as IMEI, location and network info to remote servers. Another interesting case is the benign app com.necta.aircall.accept.free.apk, which has over a million downloads. This app is categorized as benign by our classification system. However, based on one of the results of an online malware detection services, it is reported as malicious in our data set. We manually examined the source code of this app to understand this discrepancy. We found it to be a phone call app, which also monitors incoming and outgoing calls on the device. It can also receive and send SMS. Based on our observation and manually analyzed purpose of the app, we do not find any inherent malicious behavior though it is understandable why some tools may classify it as malware based solely on its type of activities.

5.4.10 Discussion

Information flows themselves may not provide enough information to distinguish malware apps (misclassified malware from MudFlow). Detailed app behavior, captured by N-grams, is an important feature that can provide critical information used to distinguish malicious apps from benign apps. The detailed app behavior collected by Complex-Flow provides more evidence of the maliciousness of an app (higher true negative rate of our approach). For example, consider the following observation identified by the research. Similar, long API call sequence are less common across benign apps, indicating that benign apps vary greatly in app behavior. However, long API call sequence are common across malware apps and can improve the detection rate of malicious apps, indicating malware shares common behavior patterns. Different sizes of N-grams indicate different complexities of app behavior. Many MalGenome apps can be classified separately from benign apps based on gram-1 features alone, meaning these apps show significant difference of app behavior on single API versus benign apps. In contrast, classification of other modern malware apps requires more than gram-1 feature. This means these malware are more similar with benign apps than the MalGenome ones. However, they can still be differentiated from benign apps by analyzing detailed app behaviors represented by different gram features.

5.5 Threats to Validity and Limitations

Our classification system leverages static analysis to generate Multi-Flows and thus suffers from the classic limitations of this approach. As new techniques are developed to improve precision of static analysis, specifically static analysis techniques of Android, our tool will be able to leverage these improvements. We currently do not handle analysis of native code in Android apps. Our approach cannot detect malicious behavior that is present in native code. We observe that current statistics show that around 5% to 30% of apps make use of native code [71] [9]. Unfortunately, there are a number of know classes of Android malware whose threat vector is primarily located in native code. Our implementation currently does not consider these cases. Other classification schemes use the presence of native code as a feature itself [61] [58].
Another limitation of our approach is that we consider all the apps we downloaded from Google Play Store as benign, but we cannot be completely certain that there are no malicious apps among them (§5.3.4 and §5.4.9). The malicious apps we used in this paper stem from collections of malware where each apps has been identified as malicious at some point. We do not know its main attack type and its malicious code features and our identification scheme is currently agnostic to this information. Lastly, we only consider continuous sub-sequence of API calls in this paper. N-grams features of non continuous sequence of API calls may also be great features for classification purposes.

6 Conclusion

In this thesis, we present a flow based extension to the Android permission mechanism, called Flow Permissions. We detailed a comprehensive primer on Android specific mechanisms and libraries in our description of BlueSeal, an automated infrastructure for synthesizing Flow Permissions. We provided a comprehensive evaluation of Flow Permissions in a wide variety of Android apps as well as a preliminary user study indicating the utility of Flow Permissions on users’ decision to install apps.

Next, we present a systematical study on modern Android malware apps. We analyze 56,000 detected malware apps and perform comparison against modern benign apps as well as old malware apps. We examine each app based various metrics including permissions, structural analysis, security related features and so on. Based on our analysis, results indicate that modern malware apps contain more complicated behavior and leverage more sophisticated features than older malware apps. We also study recent malware detection tools and collect popular features used in these tools. Eventually, we perform a selection phase to build up modern malware benchmark set for future researchers.

Last but not least, we proposed a new concept of Complex Flows to derive app behavior on device sensitive data. We also present an automated classification system that leverages app behavior along with app information flows for classifying benign and malicious Android apps. We have detailed our approach to discover Complex Flows in an app, extract app behavior features, and apply a classification procedure. We show the effectiveness of our classification system by presenting evaluation results on Google Play Store apps and known malicious apps.
References


