User-Intention Based Program Analysis for Android Security

Karim O. Elish

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science and Applications

Danfeng (Daphne) Yao, Chair
Barbara G. Ryder
Wenjing Lou
Eli Tilevich
Xuxian Jiang

July 1, 2015
Blacksburg, Virginia

Keywords: Android Malware, Application Collusion, User-Intention, Program Analysis, Inter-Component Communication (ICC), Mobile Security

Copyright 2015, Karim O. Elish
User-Intention Based Program Analysis for Android Security

Karim O. Elish

ABSTRACT

The number of mobile applications (i.e., apps) is rapidly growing, as the mobile computing becomes an integral part of the modern user experience. Malicious apps have infiltrated open marketplaces for mobile platforms. These malicious apps can exfiltrate user’s private data, abuse of system resources, or disrupting regular services. Despite the recent advances on mobile security, the problem of detecting vulnerable and malicious mobile apps with high detection accuracy remains an open problem.

In this thesis, we address the problem of Android security by presenting a new quantitative program analysis framework for security vetting of Android apps. We first introduce a highly accurate proactive detection solution for detecting individual malicious apps. Our approach enforces benign property as opposed of chasing malware signatures, and uses one complex feature rather than multi-feature as in the existing malware detection methods. In particular, we statically extract a data-flow feature on how user inputs trigger sensitive critical operations, a property referred to as the user-trigger dependence. This feature is extracted through nontrivial Android-specific static program analysis, which can be used in various quantitative analytical methods. Our evaluation on thousands of malicious apps and free popular apps gives a detection accuracy (2% false negative rate and false positive rate) that is better than, or at least competitive against, the state-of-the-art. Furthermore, our method discovers new malicious apps available in the Google Play store that have not been previously detected by anti-virus scanning tools.

Second, we present a new app collusion detection approach and algorithms to analyze pairs or groups of communicating apps. App collusion is a new technique utilized by the attackers to evade standard detection. It is a new threat where two or more apps, appearing benign, communicate to perform malicious task. Most of the existing solutions assume the attack model of a stand-alone malicious app, and hence cannot detect app collusion. We first demonstrate experimental evidence on the technical challenges associated with detecting app collusion. Then, we address these challenges by introducing a scalable and an in-depth cross-app static flow analysis approach to identify the risk level associated with communicating apps. Our approach statically analyzes the sensitivity and the context of each inter-app communica-
tion with low analysis complexity, and defines fine-grained security policies for the inter-app communication risk detection. Our evaluation results on thousands of free popular apps indicate that our technique is effective. It generates four times fewer false positives compared to the state-of-the-art collusion-detection solution, enhancing the detection capability. The advantages of our inter-app communication analysis approach are the analysis scalability with low complexity, and the substantially improved detection accuracy compared to the state-of-the-art solution. These types of proactive defenses solutions allow defenders to stay proactive when defending against constantly evolving malware threats.
Dedication

To my beloved family:

my parents (Omar and Fatma), my brothers (Mahmoud and Sameh), my sisters-in-law (Mariam and Manal), my nephews (Omar, Ali, and Adam), and my niece (Zeina).
Acknowledgments

I would like to thank my advisor Danfeng (Daphne) Yao for her continuous support, unlimited help, and valuable guidance and advice since we first met. She introduced me to the field of cyber security research and taught me technical writing skills. Moreover, she was always providing me with sincere beneficial advices regarding my career.

Also, I would like to thank our department head, Barbara Ryder for her very rich technical support and guidance. Her expertise in program analysis helps me a lot in my work. Despite her busy schedule, she always sets time for our weekly research meetings, and provides constructive feedbacks and valuable insight to make sure that our work is conducted at a high standard. I was lucky to have a lot of very fruitful discussions with her during the course of this work and I feel honored to have had the chance to work with her.

I would like to thank all the other committee members, Wenjing Lou, Eli Tilevich, and Xuxian Jiang (NC State University). Their valuable feedback and enlightening comments help me to improve my final dissertation.

Thanks to all colleagues and friends for their suggestions and encouragement during my graduate study at Virginia Tech.

Last but certainly not least, I would like to thank my family for their encouragement and continuous support. Indeed, without them, I would never be able to make it.
Contents

1 Introduction 1

1.1 Major Research Contributions 2

1.1.1 User-Trigger Dependence Analysis for Android Malware Detection 2

1.1.2 Characterization of Android Inter-Component Communication (ICC) 5

1.1.3 Inter-App ICC Analysis for App Collusion and Vulnerability Detection 6

1.2 Broader Impact 9

1.3 Dissertation Organization 9

2 Literature Review 10

2.1 Mobile App Classification 10

2.1.1 Classification with Static Features 10

2.1.2 Classification with Dynamic or Hybrid Features 12

2.1.3 Non-Classification Work 13

2.2 Mobile App Collusion and Vulnerability Analysis 14

2.2.1 App Collusion Analysis 14

2.2.2 App Vulnerability Analysis 15

2.2.3 General Purpose App Analysis 17

3 User-Trigger Dependence Analysis for Android Malware Detection 19
List of Figures

3.1  Partial abstract dependence graph for a legitimate app. sendTextMessage() has the required user dependence property. User triggers are shown in green nodes. ................................................................. 20

3.2  Partial abstract dependence graph for HippoSMS malware. There is no direct path showing a dependency between user triggers and sendTextMessage(). . 21

3.3  Workflow of our analysis. ................................................................. 22

3.4  Illustration of dependence paths and various metrics for a program having two distinct operations $c$ and $c'$. Each operation has two call sites $s_1$ and $s_2$ and $s'_1$ and $s'_2$, respectively. A solid line represents the existence of a dependence path from some user trigger to a call site. A dashed line represents that none of the call site’s dependence paths has a user trigger. ........................................ 24

3.5  Averaged DPVC vectors representing a fine-grained distribution of per-operation valid call sites for 1,433 malware apps (top) and 2,684 free popular apps (bottom). .................................................................................... 33

3.6  Distinct distributions of assurance scores ($V$) for known malicious apps and free popular apps. ................................................................. 37

4.1  Partial CAIMap for abc.ssd.TrafficInfoCheck application. ............... 56

4.2  Histogram showing the number of the 20 benign apps pairs that trigger alerts per XManDroid’s policy [13]. Some apps pairs trigger alerts for more than one policy. XManDroid’s policy can be found in [13]. ....................... 59
4.3 ICC involving non-sensitive data or request should not be alerted, despite of
the sensitive permission combination in manifest files (namely, ACCESS_FINE-
LOCATION and INTERNET). ......................................................... 61

4.4 We envision an in-depth data-flow analysis in both source and destination
apps that captures the context associated with the ICC. For classification,
the presence of a valid user trigger in the source app may serve as one of the
features that help rule out benign cases. .......................................... 62

5.1 Application collusion threat model via ICC. ............................................. 65
5.2 An example of permissions and operations being split between colluding ap-
plications. ......................................................................................... 66
5.3 A schematic drawing illustrating the complexity of the inter-app ICC mesh
between multiple apps. Individual apps have access to sensitive data and/or
perform critical operations. The apps collusion chain can be more than two
apps. ................................................................................................... 66
5.4 A schematic drawing illustrating how collusion setup can be used by botnet. 68
5.5 An example of colluding botnet scenario for injecting malicious JavaScript
code to leak private data. ................................................................. 69
5.6 Code snippet for the malicious code injection by the colluding bots. The
malicious app (on the top) and the PhotoManager app (on the bottom). ... 70
5.7 Code snippet for the PhotoUtils class. ................................................. 70
5.8 Code snippet for the injected malicious JavaScript. .............................. 70
5.9 Our inter-app ICC analysis workflow. The preprocessing step is performed
only once for each (source and target) app. ......................................... 71
5.10 Our four properties of the inter-app ICC flow used in our classification policies. 76
5.11 High level overview of inter-app ICC risk level according to our security policies. 78
5.12 Visualization of the explicit Intent inter-app ICC calls between the 175 apps pairs. Red nodes and edges represent high and medium risk inter-app ICC calls discovered using our security policies. Gray nodes and edges represent no risk inter-app ICC calls. 81

5.13 Histogram showing the number of the 175 apps pairs per Activity, Service, and Receiver ICC. Some apps pairs use more than one type of external explicit Intent ICC. 82

5.14 Histogram showing the number of external explicit Intent ICC channels for 175 apps pairs. Majority of the apps pairs have only one external explicit Intent ICC per apps pair. 83

5.15 Histogram showing a comparison between permission-based policies solution and our work on explicit Intent inter-app ICC classification. Majority of the apps pairs classified as collusion by permission-based policies solution do not have sensitive explicit Intent inter-app ICC calls (no sensitive data or critical operations involved). 86
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Comparison with related mobile app classification work.</td>
<td>18</td>
</tr>
<tr>
<td>3.1</td>
<td>Summary of classification results on 1,433 known malware apps. Rule 2 is applied to the apps that are classified as benign by Rule 1. The false negative (FN) rate refers to the portion of malware apps classified as benign by both rules and is 2.1%.</td>
<td>38</td>
</tr>
<tr>
<td>3.2</td>
<td>Assurance scores for the benign and malicious components in some repackaged malware apps.</td>
<td>39</td>
</tr>
<tr>
<td>3.3</td>
<td>Assurance scores of subset of selected benign apps including or excluding the ads/analytics libraries.</td>
<td>40</td>
</tr>
<tr>
<td>3.4</td>
<td>Summary of classification results after applying both rules on 2,684 free popular apps.</td>
<td>41</td>
</tr>
<tr>
<td>3.5</td>
<td>Average feature-extraction time for an app.</td>
<td>43</td>
</tr>
<tr>
<td>3.6</td>
<td>Malicious activities of a subset of new malware found by our method.</td>
<td>50</td>
</tr>
<tr>
<td>4.1</td>
<td>Summary of ICC analysis on 2,644 benign apps. Implicit and explicit Intent ICC are used for communication between apps.</td>
<td>63</td>
</tr>
<tr>
<td>5.1</td>
<td>Examples of sensitive data and critical operations in our study. We consider APIs that retrieve private data as sensitives data, and APIs that consume the private data or perform critical functions as critical operations.</td>
<td>74</td>
</tr>
<tr>
<td>5.2</td>
<td>Our risk classification policies for explicit Intent inter-app ICC.</td>
<td>79</td>
</tr>
</tbody>
</table>
5.3 Summary of explicit Intent inter-app ICC classification results on 175 apps pairs. .......................................................... 80

5.4 A comparison between permission-based policies and our policies on explicit Intent inter-app ICCs. Out of 175 apps pairs, 132 apps pairs classified as collusion by the permission-based policies solution, and 33 apps pairs classified as high and medium risk by our work. Our analysis has 4 times fewer flags than the permission-based policies solution. .................................................. 84

5.5 Summary of the number of the 175 apps pairs that trigger alerts per permission-based policies [13]. Some apps pairs trigger alerts for more than one policy. ................................................................. 85

5.6 Pros and cons of static flow-based analysis and dynamic permission-based analysis. ............................................................... 87

5.7 Summary of explicit Intent inter-app ICC risk reduction after applying and enforcing permissions on the target components in the target apps. .......................... 89
Chapter 1

Introduction

Malicious mobile apps and vulnerable mobile computing platforms threaten the confidentiality of personal and organization data and device integrity [21, 32, 37]. Malicious applications can exfiltrate sensitive data, abuse of system resources, and disrupt the normal usage of the device. With the increased connectivity to organizational networks, vulnerable smartphones increase the attack surface of organizations, threatening the security of systems and data at a grand scale. Recent studies show that there exist hundreds of thousands of unique Android malware samples belonging to over 300 malware families [42]. The existing malware detection solutions aim at detecting characteristic malware behaviors or signatures in binary code (e.g., commercial anti-virus scan tools). However, these solutions are reactive and limited to identify zero-day malware or new variants of malware. Because of the pervasive use of Android as a mobile operating system (over 50% market share in western and some Asian countries), proactive solutions for detecting malicious applications in the Android marketplace are urgently needed.

Malicious software (malware) has been constantly evolving to evade detection. This evolution nature of malware was first shown in the seminal paper by Cohen [18] to cause endless arms-races between defenders and attackers. The notion of app/malware collusion has been recently described in a few research papers [13, 62] as the next step that malware writers’ may evolve into. Collusion refers to the scenario where two or more applications – written by the same malware writer – interact with each other to perform malicious tasks. The danger of malware collusion is that each colluding malware only needs to request a minimal set of privileges, which may make it appear benign under conventional screening mechanisms.
Malware writers have strong incentives to write colluding malware. Virtually all the existing solutions assume the attack model of a *single malicious application*. The detection techniques are focused on inspecting apps individually, through building behavioral models with features obtained from various static and dynamic program analysis.

The Android system introduces inter-component communication (ICC) to facilitate the communication mechanism between two applications, or between two components of the same application. The attackers may utilize this type of communication to write colluding apps. Despite the recent advances on Android security, the problem of detecting app collusion remains an open problem.

In this dissertation, we present a new quantitative program analysis framework for security vetting of Android applications to address the above problems, namely, *i)* detecting individual (single) malware and *ii)* detecting application collusion. The novelty of our work is that we take the approach of anomaly detection (i.e., identifying deviations from normal patterns), as opposed to the conventional methods of identifying malware characteristics. The anomaly detection solutions are more robust and permanent than the conventional methods of detecting known malicious patterns.

In the remaining of this chapter, we briefly introduce the research parts of this dissertation along with the research challenges, and our major technical contributions and approaches.

### 1.1 Major Research Contributions

We describe below the main research parts and highlight the major contributions of this dissertation towards developing effective proactive defense solutions against Android malware.

#### 1.1.1 User-Trigger Dependence Analysis for Android Malware Detection

Classification solutions have been proposed to model and approximate the behaviors of Android apps and distinguish malicious apps from benign ones. Classification decisions are made by analyzing apps’ static (e.g., [49]) or dynamic (e.g., [8]) behavior features. Static features can be extracted from intermediate code representations obtained through decom-
piling Android Dalvik bytecode. Dynamic features are collected by observing the run-time behaviors of the program. Various types of features can be extracted from Android permission, code, or execution for app classification.

The detection accuracy of a classification method depends on the quality of the features, e.g., how specific the features are. The accuracy of existing Android classification solutions is still far from ideal. The state-of-the-art classification with pure static features gives a false negative rate (i.e., missed detection, FN) of 9% [49]. These features are extracted through data- and control-flow analyses. Hybrid features (i.e., a combination of static and dynamic features) extracted from programs give a better FN rate 4.2% [104] (e.g., dynamic features related to dynamic code loading and native code invocation). Most of the dynamic classification solutions give 10% or higher false positive rates (FP) while trying to maintain a reasonable FN rate, e.g., 10% FP in [77] and 15% FP in [8]. The false positive rate tells the percentage of benign apps wrongfully classified as malicious.

This work presents a high-precision Android app classification method based on one complex feature that leverages the dependence effects of program behaviors. Specifically, we extract the definition-and-use (i.e., def-use) data dependence properties related to sensitive operations and their user triggers in the app. Smartphone apps (Android, iOS, or Windows Phone) are unique in their user-centered and interaction-intensive design, in which operations typically require initiation by users’ specific actions (or triggers). Our classification leverages the dependence relations between user inputs/actions and sensitive API calls providing critical system functions. Our feature extracted from programs reflects the expected causal relations in the execution.

Our classification recognizes legitimate and desirable behavioral patterns in programs, as opposed to identifying malicious patterns. Those behaviors are commonly found in trustworthy programs, but not in malware. Our classification is based on whether or not a program possesses these benign properties.

Specifically, we analyze the def-use graph to extract a TriggerMetric feature for each API call. The TriggerMetric feature statically approximates whether or not the occurrences of the call (i.e., call sites) are triggered by the user. Specifically, the TriggerMetric value represents the number of valid call sites among all the call sites of a specific API. The validity of an API call is defined based on def-use semantics; a call is valid if at least one of the call’s arguments depends on some user input(s). In other words, the TriggerMetric values of an app reflect the
degree of sensitive operations that are triggered or intended by the user. The classification
decision is made based on TriggerMetric values (i.e., an app is classified as malware if it has
an overwhelming number of triggerless sensitive operations).

We summarize our contributions in this research part as follows.

- We present a new Android app classification method that uses one complex feature
  rather than multi-feature as in the existing malware detection methods which focus
  on the presence of simple features such as permission or API call. The TriggerMetric
  feature captures the static dependence relations between user inputs/actions and
  sensitive operations providing critical system functions in programs. This feature is
  extracted through nontrivial Android-specific static program analysis and is used in
  several quantitative analytical methods.

- We design and develop a static program analysis tool specific for Android to capture the
  dependence relations between user triggers and sensitive operations. Our tool builds
  flow- and context-sensitive data dependence graph through intra- and inter-procedural
  def-use analysis, and event-based information flow that includes the handling of data-
  flow through Android Intents and GUI related implicit method invocations.

- Our experimental evaluations on 2,684 free popular apps and 1,433 malicious apps sug-
  gest that our rule-based classification with the single feature of user-trigger dependence
  is very effective. It detects 97.9% of the malware apps with a low (2.0%) false positive
  rate.

- Our analysis reveals hundreds of malicious apps in the Google Play store, some of
  which were previously unreported and were not detected by any of the 48 VirusTotal [4]
  scanners.

The purpose of our work is not to advocate the use of fewer features in program classification.
Multiple classification tools and features should be utilized to paint a comprehensive picture
about a program.

Rather, our thesis in this mobile app classification work is to advocate the ap-
proach of benign property enforcement. Our analysis verifies whether or not a program
is in compliance with our benign-property standards. In the face of rapid malware evolu-
tion, this type of benign-property enforcement may yield a more proactive defense than the malware-oriented detection approaches.

1.1.2 Characterization of Android Inter-Component Communication (ICC)

In the Android system, Intent and inter-component communication (ICC) realize an encapsulated communication mechanism for passing messages between two applications, or between two internal components of the same application. Sending Intents through ICC channels is widely used in apps. For example, a restaurant search app may send an Intent to Google Maps app, so that Google Maps displays a map with the chosen restaurant’s location.

Malware collusion is a new malware generation attack that is very challenging to detect under the existing conventional screening techniques. A collusion attack occurs when malicious applications, likely written by the same adversary, collaborate to gain a set of permissions to perform malicious tasks. In malware collusion, each colluding malware only needs to perform a certain functionality, which may make it appear benign to evade the conventional detection tools. Hence, malware writers have strong motivation to write colluding malware to evade standard detection. The application collusion can be done through inter-component communication (ICC) which is standard communication channel in the Android system. Colluding applications may abuse system resources (e.g., sending spam SMS) or leak sensitive data.

Because of the wide usage of ICC calls in benign apps pairs, accurate classification is quite challenging. We argue that practical solutions for detecting Android malware collusion needs to satisfy several requirements:

- To be able to characterize the context associated with communication channels with fine granularity,
- To define security policies for classification that minimize false alerts,
- To be scalable to a large number of apps (e.g., tens of thousands of apps).

In this work, our goal is to characterize the ICC channels and investigate the technical challenges associated with distinguishing benign ICC flows from colluding ones. Inability
to solve this problem may result in a high number of false alerts, i.e., misclassifying benign ICCs as collusion.

We summarize our contributions in this research part as follows.

- We experimentally demonstrate the technical challenges associated with applications collusion detection. We design and develop a static analysis tool to model the Intent-based ICC of Android applications. We construct Cross-App ICC Map (CAIMap) to capture pairwise communicating channels. We analyze the ICC calls among 2,644 free popular applications from Google Play market. (These apps pass multiple screening tools and are considered benign.) 84.4% of these benign apps have external ICC calls. We apply a set of classification policies in the existing XManDroid [13] collusion detection solution to these apps. Our results show that these permission-based classification polices trigger a large number of false alerts in benign app pairs.

- To overcome the deficiencies in the existing work (namely, reducing the false alerts), we sketch several promising approaches that are based on in-depth static flow analysis. We give specific examples to show how to discover the context associated with benign ICC flows, and formulate more fine-grained policies.

Our work points out the urgent need for designing new and scalable deep static program analysis algorithms to screen for collusion, specifically on evaluating the sensitivity of the entire data-flow path across the apps that is beyond the ICC interface.

### 1.1.3 Inter-App ICC Analysis for App Collusion and Vulnerability Detection

In the realm of mobile application security, there exist a substantial amount of solutions for detecting malicious apps (e.g., [7, 11, 25, 29, 32, 40, 49, 72, 85, 86, 92, 95, 96]). Virtually all the existing solutions assume the attack model of a single malicious application. The detection techniques are focused on vetting apps individually.

The attackers always find new techniques for developing malware applications to evade detection. One example of this new emerging technique is the application collusion, where two
or more applications, appearing benign, communicate to perform malicious task. Unfortunately, existing literature does not provide satisfying solutions for detecting Android malware collusion. Virtually all existing ICC-based program analyses are for detecting vulnerable-yet-benign apps (e.g., due to inexperienced developers). For example, CHEX [60] identifies potentially vulnerable component interfaces that are exposed to the public without proper access restrictions in Android apps. ComDroid [17] and Epicc [66] identify application communication-based vulnerabilities and describe two main categories of abuses, i.e., Intent stealing and Intent spoofing. These works address the confused deputy attack, where a malicious app exploits vulnerable component interfaces of a benign app. However, malware collusion has a different attack model, where all colluding apps are written by malicious developers and share common attack goals. Furthermore, existing analyses on ICC (e.g., Apposcopy [40], IccTA [56], and Epicc [66]) do not provide complete cross-app ICC flow analysis, thus can not be directly applied for collusion detection.

XManDroid [13] is the first (runtime) solution for Android collusion detection. It specifies classification policies on inter-app communications. XManDroid’s policies are based on permissions that the source and destination apps request at the time of installation. This coarse-grained policy design has limitations in distinguishing benign ICC flows from colluding ones, resulting in a high number of false alerts.

**Collusion Detection Challenges.** Developing practical solution for Android malware collusion faces many challenges: (i) how to characterize the context associated with communication channels with fine granularity (ii) how to define security policies for classification that reduce the number of false alerts, and (iii) how to provide scalable solutions with minimum complexity to vet a large number of apps (e.g., in Google Play store) for possible collusion.

To overcome the above challenges, we present a scalable and fine-grained static program analysis algorithm (with a hash map data structure) to screen apps for collusion. Our approach statically analyzes the sensitivity and the context of each inter-app ICC-based flow between two communicating apps. Each inter-app ICC-based flow has a varying degree of sensitivity, depending on the type of data it carries or action it invokes/requests. Thus, we perform comprehensive static dataflow analysis across both apps that infers the inter-app ICC sensitivity, detailing how the data is created, modified, and consumed. Our detection policies are based on this in-depth analysis and are more fine-grained than permission-based policies, reducing the number of false alerts on benign inter-app ICC calls as demonstrated.
Karim O. Elish  Chapter 1. Introduction  8

by our experiments in Section 5.5.

The complexity of naive pairwise inter-app ICC analysis is \( O(n^2|V||E|m) \), where \( n \) is the number of apps, \( V \) and \( E \) are the number of nodes and edges respectively traversed on both apps graphical representation, and \( m \) is the maximum number of inter-app ICC calls between any pairs of apps. In comparison, our solution has a complexity of \( O(n^2m) \). We utilize the hash map data structure to store the information related to the context associated with each inter-app ICC-based flow, substantially reducing the complexity of a query.

We perform accurate static dataflow analysis across multiple apps to check the paths which led to the critical operations. Our approach is also applicable to perform precise pairwise vulnerability analysis. Additionally, by performing our pairwise inter-app ICC analysis, several attacks can be detected and prevented which include, but are not limited to, apps collusion, confused deputy attack, data leakage, and intent spoofing. Such analysis allows defenders to stay proactive in defending against constantly evolving malware threats.

Our contributions are summarized as follows.

- We design and develop Cross-App ICC Map (CAIMap), a static analysis tool to capture all the communication through ICC between parts of Android applications. This CAIMap is used as a first step in our approach to identify the pair/group of apps that communicate via ICC channels.

- We present a comprehensive cross-app static flow analysis technique to identify the risk level associated with the inter-app communication through the ICC channels. This risk level is used as an indication for apps collusion detection. Our fine-grained inter-app ICC behavioral analysis discovers the context associated with the inter-app ICC-based flows with low analysis complexity. Our method computes and stores the ICC entry and exit points of a given app only once to reduce the analysis complexity, and systematically matches them with entry and exit points of previously processed apps. We formulate more fine-grained security policies to identify the risk level associated with communicating apps.

- We perform our evaluation on 2,644 apps from Google Play store to identify the apps that communicate with each other. We found 175 apps pairs communicate using direct explicit Intent ICC from the corpus of 2,644 apps. We evaluate the effectiveness of our approach on these 175 real apps pairs. Our analysis results suggest that
our inter-app ICC risk classification method is effective and generates four times fewer false positives compared to the existing permission-based policies solution.

1.2 Broader Impact

We envision that our proactive defense solutions against Android malware, presented in this dissertation, will allow defenders to stay proactive in defending against constantly evolving malware threats. The specific expected potential applications of this research are as follows. First, this research can be used by the apps market administrator to automatically vet the apps before making them available to the users, and to perform massive screening of the apps to detect possible app collusion. Second, this research will enable the user to manage her/his mobile device by helping the user to make informed decision about whether to install an app or not, and to detect possible collusion with the pre-installed apps on her/his mobile device. Finally, these defense techniques can be used by an administrator to manage mobile devices connecting to an organizational network.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 reviews the related work on Android security with an emphasis on malware detection, app vulnerability analysis, and collusion detection. Chapter 3 describes the design, implementation and evaluation of our Android malware detection approach. Chapter 4 presents a characterization of Android ICC along with experimental evidence on the technical challenges associated with detecting colluding applications. In Chapter 5, we describe the design, implementation and evaluation of our inter-app ICC analysis for application collusion detection. Finally, Chapter 6 concludes the dissertation and presents the future work directions.
Chapter 2

Literature Review

2.1 Mobile App Classification

We categorize related Android app analysis work into \(i\) classification with static features, \(ii\) classification with dynamic or hybrid features, and \(iii\) non-classification work. Both static and dynamic approaches are necessary for evaluating app security, providing complementary behavioral profiles\(^1\). We compare some of the existing mobile app classification solutions in Table 2.1.

2.1.1 Classification with Static Features

Static program analysis has traditionally been applied to ensuring the data integrity and confidentiality of information flow, e.g., [22, 23, 58, 59, 63]. Using static program analysis for anomaly detection was first described by Wagner and Dean [82], then improved by [6, 12, 39, 45] with more accurate control flow analysis with calling context and call dependences. Our work is different in the sense that we customize the data dependence analysis to suit our goals, and perform Android Intent-based analysis tailored for Android OS.

In order to infer the trustworthiness of mobile applications, multiple approaches have been proposed to statically extract properties of a program from its code and/or its requested permissions (e.g., [69, 75]). One of the earliest such work is SCanDroid [43]. SCanDroid [43]

\(^1\)Not all related papers report both FP and FN rates.
proposed to extract security specifications from the app’s manifest and check whether the
data-flows through the app are consistent with the stated specifications. 2

The solution by [69] calculated risk scores from the permissions requested by Android apps and found the hierarchical mixture of naive Bayes to be the best classifier for the risk score based app classification. The work by [75] also extracted permission-usage based features, and evaluated several classifiers including random forests, naive Bayes, and Bayesian network. The false positive rate in [75] is higher than 11%.

DroidAPIMiner [5] extracted features related to API calls, and evaluated several machine learning classifiers including \( k \)-nearest neighbor (KNN), decision tree, and support vector machines. It achieves a 97.8% detection rate of the malware samples and a false positive rate of 2.2% with KNN. Drebin [9] analyzed AndroidManifest.xml and disassembled code to extract features on requested permissions and API calls, and used support vector machines (SVM) as a classifier. Drebin achieves 94% detection rate of the malware samples at a false positive rate of 1%. Both works used multiple sets of features as opposed to our work. A recent paper [84] on Android malware classification utilizes the assurance score feature and dozens of other manifest-based features. The solution by [84] achieves similar accuracy as ours. It utilizes a significant number of features more than our work. It employs machine learning techniques, as opposed to our simple rule-based classification.

In comparison to the above permission-based classification, features extracted from code analysis are more fine-grained and specific. We highlight several such solutions next. The security goal in AndroidLeaks [44], SCANDAL [53], and PiOS [28] for iOS is focused on detecting data leak vulnerabilities, specifically on information flow for confidentiality analysis. The methods label sensitive data/sources and potentially risky sinks (typically network API calls) and report when there are data-leaking dependence paths between them. PiOS reports a 13% false negative rate.

Although using dependence-path based analysis, our definitions for the path have different semantics. As a result, our analysis with a complete coverage of sensitive operations provides comprehensive app profiling, which offers more protection than data confidentiality. For example, our analysis also detects system-assurance-related operations such as unauthorized camera access or recording, which is out of the scope the data leak solutions.

Multiple features were utilized to make classification decisions in RiskRanker [49]. The

\(^2\)No experimental results were reported in SCanDroid.
classification is based on several types of suspicious behavior signatures extracted through control-flow and intra-method data-flow analyses. An example of such suspicious behaviors include accessing sensitive data in a dependence path that also contains decryption (usually for deobfuscation) and execution methods. RiskRanker reports a 9% false negative rate. In comparison, our method enforces benign properties of trustworthy programs (as opposed to detecting malicious properties). Our results also show better classification accuracy compared to the existing approaches.

DroidSIFT [97] is a recent Android malware classification system that is based on constructing dependence graphs to model the dependences between API calls. Its feature vector is extracted from the graphs. The work built graph databases for known benign and malicious Android apps, and performed graph similarity queries (based on graph edit distance) for unknown apps. Its approach correctly classifies 93% of known malware samples (with naive Bayes classifier). Their anomaly detector based on the benign graph database achieves a false negative rate of 2% and a false positive rate of 5.15%. The semantics of dependence properties in our work and DroidSIFT are different. Our work models the data dependency between user-input functions and sensitive APIs. Consequently, the classification mechanisms are different. Our solution – based on rules – does not rely on graph similarity computation, which might be expensive for large graphs.

2.1.2 Classification with Dynamic or Hybrid Features

Solutions in this category detect malware apps by their runtime execution patterns (i.e., dynamic features), sometimes together with statically extracted features. Andromaly [77] and [8] extract dynamic features including memory activity and CPU load to classify Android apps. They apply several classifiers including decision trees, naive Bayes, and Bayesian networks. The best classifier in Andromaly [77] achieves a 10.4% false positive rate. In [8] the false positive rate is over 15%. The work by [57] detected malicious behaviors on mobile devices by monitoring abnormal power consumption due to malware activities, and reports a false positive rate that ranges from 4.3% to 10%.

Crowdroid [14] performs $k$-means clustering algorithms on dynamic features collected from Android apps. The features are the frequencies of occurrences for system calls (e.g., \texttt{open()}, \texttt{kill()}) executed by an app. The proposed solution successfully identifies all of the author-created malware, while it reports a 20% false positive rate on the real-world repackaged
The features in DroidRanger [104] are hybrid. It statically extracts behavioral signatures of known malware samples. Examples of static features include sequences of APIs being called, package names, and class hierarchies. It also has a dynamic execution monitor that inspects the suspicious runtime behaviors of the app, such as loading dynamic code. The method reports a false negative rate of 4.2%.

These dynamic analyses provide useful information on runtime program behaviors and complements our static analysis work. Both approaches are necessary for app classification.

### 2.1.3 Non-Classification Work

Several validation and verification solutions have been proposed for mobile platforms to enhance the assurance of execution. These tools gather contextual information associated with sensitive operation invocations. This information is compared with models built through hybrid program analysis. For example, AppIntent [93] defines privacy leakage as user-unintended data transmission. It provides a security analyst the context information associated with the transmission. The human analyst then decides whether the transmission is legitimate or not. Pegasus [16] proposes a Permission Event Graph abstraction in order to detect sensitive operation invocations that are inconsistent with the UI events. It automatically verifies the app’s behaviors with respect to pre-defined app-specific policies.

DroidScope [90] presents a virtualization environment for the purposes of dynamic analysis and information tracking of Android apps. VetDroid [99] is a dynamic analysis platform for understanding Android apps behaviors associated with permission use. SmartDroid [100] presents a tool that uses static and dynamic analysis to gather the information on UI-based event trigger conditions in Android apps. ParanoidAndroid [71] performs runtime analysis of Android apps on a remote server in the cloud.

DroidMOSS [102] is an app similarity measurement system which uses fuzzy hashes to identify repackaged apps. DNADroid [20] detects cloned Android apps in the markets by comparing similarities of the program dependency graphs. AppInk [101] prevents apps repackaging with watermarking techniques.

Aurasium [89] repackages apps to add user-level sandbox and security policies so that the
app’s runtime behavior can be restricted. AppFence [50] modifies Android OS to protect private data from being leaked by providing and imposing fine-grained privacy controls on existing apps. TISSA [105] proposes a privacy mode in Android platform to provide fine-grained control over user privacy.

AdRisk [48] systematically studies a large number of popular ad libraries used in Android apps to evaluate the potential risks. AdDroid [68] and AdSplit [78] propose different approaches to separate the privileges between the ad library and its host app to eliminate the permissions requests done by the host app on behalf of its ad library.

User-driven access control gadget (ACG) was proposed in [74] to capture user authorization actions (keyboard shortcut or mouse movement) for assured resource access at runtime. Unlike ours, these solutions are not for malware classification, thus have different security goals and technical approaches from ours.

2.2 Mobile App Collusion and Vulnerability Analysis

Malware collusion is a new threat against Android application security that has not been systematically studied. We categorize the related work into i) app collusion analysis, ii) app vulnerability analysis, and iii) general purpose app analysis.

2.2.1 App Collusion Analysis

XManDroid [13] is a runtime monitoring solution of communication channels between apps. It defines communication classification policies based on certain permissions combinations of communicating apps. However, this approach has several drawbacks: (1) it does not scale, (2) it is costly and time consuming due to runtime interception and inspection of all IPC calls, and (3) it suffers from the high number of false alarms. In addition, this approach can be circumvented by using chain of three colluding apps or more to bypass the policies, e.g., app $X$ with READ_CONTACTS permission sends the contacts information to app $Y$ which has no permission. Then, app $Y$ sends the contacts information to app $Z$ which has INTERNET permission.

FUSE [73] presents a single-app and multi-app static information flow- and context in-
sensitive analysis. Similar to XManDroid, FUSE defines coarse-grained information flow assertions based on permissions combinations. In contrast, our analysis is flow- and context sensitive and our policies are fine-grained. FUSE is focused on evaluating single apps. DidFail [54] combines FlowDroid [10] and Epicc [66] to track data flows between Android components. DidFail currently focuses on ICC flows between Activities only, and hence it does not track data flows across other components such as Service and Broadcast Receiver. Marforio et al. [62] implements and measures the efficiency of different overt and covert channels for applications collusion. It does not provide any collusion defenses or classification policies.

Amandroid [83] presents a general inter-component data-flow analysis for vetting Android apps. The analysis is performed on individual apps, and does not provide any classification policies against the apps collusion attack.

### 2.2.2 App Vulnerability Analysis

Privilege escalation attack in the Android system was first demonstrated by Davi et al. [21]. However, they did not provide any defense techniques for this attack. Confused deputy is a special type of privilege escalation attack where a malicious app exploits a vulnerability of a trusted app to perform a critical operation.

IPC Inspection [38] addresses confused deputy attack and found that a number of pre-installed apps are vulnerable to this attack. The idea of IPC Inspection is to reduce the permissions of an app when it receives a message from another app with less privilege. This approach is somewhat strict because the apps can not receive messages from a less privileged app for legitimate purposes. In addition, reducing the app’s privileges can make the app malfunction or crash. Similarly, QUIRE [24] provides a lightweight provenance system to prevent the confused deputy attack. It tracks the call chain of ICC and denies the request if the caller app does not have the required permission. However, QUIRE is not designed for detecting the malicious colluding apps. Saint [67] presents policies for install-time permission granting and runtime inter-application communication based on their permissions. Their policies allow the application to control which applications can access it. Chan et al. [15] present a static analysis tool of Android apps to check if the apps can be exploited to launch privilege escalation attacks. However, this tool is a coarse-grained since it analyzes the manifest file only, not the source code.
ComDroid [17] and Epicc [66] identify application communication-based vulnerabilities. They analyze the Intent object used by ICC API calls to describe two main categories of attacks, i.e., Intent stealing and Intent spoofing. The focus of the analysis is on individual applications. In addition, Epicc [66] analyzes and collects information about ICC exit and entry points of the analyzed app, but no classification policy is provided to make a decision whether it is a malicious or benign inter-app communication. ComDroid and Epicc assume that any unprotected public component is vulnerable regardless if there is a path from the public component to the critical operations or not. However, this approach may increase the number of false alerts when there is no path from the public component to the critical operations. CHEX [60] identifies potentially vulnerable component interfaces that are exposed to the public without proper access restrictions in Android apps, using data-flow-based reachability analysis. CHEX is designed to detect vulnerable interfaces components within a single app and does not track data through app boundaries.

TaintDroid [32] presents dynamic taint tracking to track the flow of sensitive data in third-party applications. It helps to detect data leaks in the apps and found many potential information misuse in 20 apps. Similarly, SCanDroid [43] extracts security specifications from the app’s manifest and checks whether data flows through the app are consistent with the stated specifications. TaintDroid and SCanDroid focus on data flow. They could be generalized to support cross-app taint tracking for dynamic collusion detection. They could be generalized to support cross-app taint tracking for the purpose of dynamic collusion detection.

DroidChecker [70], IntentFuzzer [91], and Woodpecker [47] address the permission escalation attacks by identifying the permission/capability leaks vulnerabilities in Android applications. Wu et al. [87] perform several analyses such as permission usage and vulnerability analysis of the pre-installed Android apps. Kantola et al. [52] aim to reduce unnecessary exposed components surfaces in Android apps. AppSealer [98] proposes an approach to generate patches for preventing attacks related to exploit vulnerable components of Android apps.

Enck et al. [33] evaluates the security of Android apps by performing static analysis to detect the vulnerabilities and malicious components. Their analysis focuses on a single app, not on the communication between the apps.
2.2.3 General Purpose App Analysis

FlowDroid [10] and DroidSafe [46] present a general information flow analysis framework for Android applications to detect data leak. Ernst et al. [35] proposes an information flow verification model for Android applications to guarantee that the applications are free of malicious information flows. Apposcopy [40] presents a static analysis approach for detecting stand-alone malicious apps based on extracting Inter-Component Call Graph signatures.

Stowaway [36] identifies over-privileged Android apps by comparing the required and requested permissions based on mapping API calls used to permissions. RiskRanker [49] aims to detect Android malware using (i) a set of vulnerability specific-signatures and (ii) control-flow and intra-method data-flow analysis searching for suspicious behavior signatures. This work is for detecting stand-alone malicious apps.

Kirin [34] uses predefined rules which are a set of dangerous combinations of permissions that are not allowed for an app to have. Kirin focuses on individual apps, and colluding apps would not be detected.
Table 2.1: Comparison with related mobile app classification work.

<table>
<thead>
<tr>
<th>Solution</th>
<th>Aim</th>
<th>Feature Type</th>
<th># Features*</th>
<th>Feature Category**</th>
<th>Classification Policy/Algorithm</th>
<th>Evaluation Scale</th>
<th>Classification Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>AndroidLeaks [44]</td>
<td>Confidentiality</td>
<td>Static</td>
<td>S</td>
<td>DS</td>
<td>Rule: sensitive data used by risky APIs</td>
<td>24,350 apps</td>
<td>FP = 35%</td>
</tr>
<tr>
<td>Amos et al.[8]</td>
<td>Malware classification</td>
<td>Dynamic</td>
<td>M</td>
<td>DI</td>
<td>naive Bayes, Bayes nets, MLP, logistic regression, RF, DT</td>
<td>training (408 benign, 1,330 malware), testing (24 benign, 23 malware)</td>
<td>FP = 15%</td>
</tr>
<tr>
<td>PiOS [28]</td>
<td>Confidentiality</td>
<td>Static</td>
<td>S</td>
<td>DS</td>
<td>Rule: sensitive data used by risky APIs</td>
<td>1,407 apps</td>
<td>FN = 13%</td>
</tr>
<tr>
<td>Sanz et al.[75]</td>
<td>Malware classification</td>
<td>Static</td>
<td>M</td>
<td>DI</td>
<td>logistic regression, naive Bayes, Bayes nets, SVM, KNN, DT, RF</td>
<td>1,811 benign, 249 malware</td>
<td>FP = 11%</td>
</tr>
<tr>
<td>Andromaly [77]</td>
<td>Malware classification</td>
<td>Dynamic</td>
<td>M</td>
<td>DI</td>
<td>naive Bayes, Bayes nets, histograms, k-means, LR, DT</td>
<td>4 self-written malware, 40 benign</td>
<td>FP = 10%</td>
</tr>
<tr>
<td>Peng et al.[69]</td>
<td>Risk assessment</td>
<td>Static</td>
<td>M</td>
<td>DI</td>
<td>different probabilistic generative models</td>
<td>model generation &amp; testing (71,331 apps), validation (136,534 apps), 378 malware</td>
<td>FP = 4%</td>
</tr>
<tr>
<td>DroidAPIMiner [5]</td>
<td>Malware classification</td>
<td>Static</td>
<td>M</td>
<td>DI</td>
<td>KNN, DT, SVM</td>
<td>16,000 benign, 3,987 malware</td>
<td>FP = 2.2% FN = 2.2%</td>
</tr>
<tr>
<td>Drebin [9]</td>
<td>Malware classification</td>
<td>Static</td>
<td>M</td>
<td>DI</td>
<td>SVM</td>
<td>123,453 benign, 5,560 malware</td>
<td>FP = 1% FN = 6%</td>
</tr>
<tr>
<td>Ours</td>
<td>Identification of unauthorized API calls</td>
<td>Static</td>
<td>S</td>
<td>DS</td>
<td>Rule: trigger-based dependence for privileged API calls</td>
<td>2,684 benign, 1,433 malware</td>
<td>FP = 2% FN = 2.1%</td>
</tr>
</tbody>
</table>

*Number of features: single feature (S) or multiple features (M).

**Feature category: domain-specific (DS) or domain-independent (DI).
Chapter 3

User-Trigger Dependence Analysis for Android Malware Detection

In this chapter, we describe a highly accurate classification approach for detecting malicious Android apps. Our method statically extracts a data-flow feature on how user inputs trigger sensitive API invocations, a property referred to as the user-trigger dependence.

3.1 Overview and Definitions

Our classification methodology aims at exposing possible privileged actions of apps that are not intended by the user and lack proper dependences in the code. In this section, we give the description of how the trigger-based dependence feature is extracted from programs through static program analysis. We also discuss several metrics formed from our feature analysis.

3.1.1 Data Dependence Graph

A data dependence graph (DDG) is a common program analysis structure which represents inter-procedural flows of data through a program [51]. The DDG is a directed graph representing data dependence between program instructions, where a node represents a program instruction (e.g. assignment statement), and an edge represents the data dependence between two nodes. The data dependence edges are identified by data-flow analysis. A direct
edge from node $n_1$ to node $n_2$, which is denoted by $n_1 \rightarrow n_2$, means that $n_2$ uses the value of variable $x$ which is defined by $n_1$.

Formally, let $I$ be the set of instructions in a program $P$. The data dependence graph $G$ for program $P$ is denoted by $G = [I, E]$, where $E$ represents the directed edges in $G$, and a directed edge $I_i \rightarrow I_j \in E$ if there is a def-use path from instructions $I_i$ to $I_j$ with respect to a variable $x$ in $P$.

We show two DDG examples to motivate our data-flow analysis based on the dependence relations. The first example is a legitimate app for sending SMS messages. Figure 3.1 shows its partial def-use dependence graph. The graph indicates that the API call `sendTextMessage()` depends on the some inputs from the user, as one of its argument is entered by the user via text fields, through `getText()` API. There are direct dependence paths between user inputs (e.g., data and actions) and the `sendTextMessage()` API.

Another example is about a real-world Android malware `HippoSMS`, which affects Android smartphones by subscribing to premium SMS services. The malware sends SMS messages to a hard-coded premium-rated number without the user’s knowledge. Figure 3.2 shows a partial def-use dependence graph for HippoSMS. It shows the dependence relations associated with the arguments to a sensitive API call `sendTextMessage()`. Specifically, Figure 3.2 shows that `sendSMS(p0, p1, p2)` method is called with a hard-coded premium-rated number 1066156686 as its $p_0$ argument. The subsequent `sendSMS` method calls a sensitive API.
sendTextMessage() with the same hard-coded value p0 as its phoneNum argument. There is no direct dependence path between the sendTextMessage() API call and any user inputs (e.g., data and actions).

We accurately extract these types of dependence properties and quantify them for classification. Existing program analysis solutions cannot be directly applied to solve the problem, in part because of the lack of proper handling of Android-specific features such as Intents. In our work, we formalize the security problem of dependence-based app classification, and design efficient algorithms for parsing large specialized data-dependence graphs for extracting the trigger-based dependence feature. We refine our data-dependence graph with reachability analysis obtained from control-flow analysis. The reachability analysis prunes unused code for high program analysis accuracy. The workflow of our analysis is shown in Figure 3.3.

![Figure 3.2: Partial abstract dependence graph for HippoSMS malware. There is no direct path showing a dependency between user triggers and sendTextMessage().](image)

### 3.1.2 TriggerMetric Tuple Per Operation

In this section, we give the definitions for the terminology used in our classification, including operation, trigger, dependence path, and valid call site. For each operation in a program, we give our definition for the TriggerMetric tuple, which represents properties associated with call sites of the operation.

An operation is an API call which refers to a function call providing system service such as network I/O, file I/O, telephony services in the program. We focus on a subset of function
calls – the critical API calls that can be used for accessing private data and utilizing system resources.

Examples of the operations in our analysis are send/receive network traffic, create/read/write/delete operations for files, insert/update/delete operations in database and content provider, execute system commands using `java.lang.Runtime.exec`, access and return private information such as location information and phone identifiers, and send text messages in telephony services.

A **trigger** refers to a user’s input or action/event on the app. A trigger is a variable defined in the program. For example, the user’s input may be text entered via a text field, while the user’s action/event is any click on UI element, such as a button. Relevant API calls in UI objects that return a user’s input value or listen to user’s action/event are defined as triggers.

Our classification is based on analyzing unauthorized privileged operations that are not intended by the user. Because the analysis is automated (i.e., without any user participation), user-intention needs to be approximated. In our analysis user-intention is embodied in the trigger variables. We specify the names of functions corresponding to triggers and operations in the program analysis.

A **valid dependence path** is a (directed) dependence path between a trigger and an operation in a data dependence graph (DDG). In our static data-flow semantics, the path specifies a definition-and-consumption (def-use) relation, where a trigger is defined and later used as an argument to an operation. The existence of a valid dependence path means that the...
operation depends on a user trigger.

Figure 3.4 illustrates two different operations \( c \) and \( c' \) in a program, each having two call sites (i.e., each call occurs twice in the program), \( s_1 \) and \( s_2 \) for \( c \), \( s'_1 \) and \( s'_2 \) for \( c' \). Three dependence paths are valid, with proper user triggers on the paths, whereas a valid dependence path for call site \( s'_2 \) does not exist.

The trigger may be transformed before being used as an argument in the operation, thus the dependence path between them may be long. In Section 5.4 we present our detailed program analysis and graph algorithms.

A **valid call site** \( s \) of an operation \( c \) is a call site that has a valid user-trigger dependence path. A **call site** is the occurrence of an operation. An operation may have one or more call sites in a program.

**Definition 1** TriggerMetric feature is a two-item tuple \( < k, l > \) for an operation \( c \) in a program, where

- \( k \) is the number of valid call sites of operation \( c \), and
- \( l \) is the total number of call sites of operation \( c \).

For the example in Figure 3.4, the TriggerMetric values for operations \( c \) and \( c' \) are \( < 2, 2 > \) and \( < 1, 2 > \), respectively. For an app with \( n \) distinct operations, there are \( n \) TriggerMetric tuples associate with it, \( < k_1, l_1 >, \ldots, < k_n, l_n > \), one corresponding to each operation.

### 3.1.3 Aggregated Metrics

One can compute several useful values aggregated from the \( n \) TriggerMetric tuples of a program. These aggregated metrics provide a behavioral summary of the program. Intuitively, the **assurance score** \( V \) is a single value for an app representing the portion of call sites that are intended by the user across all operations in the app.

**Definition 2** Assurance score \( V \in [0\%, 100\%] \) of a program is the percentage of valid call sites out of the total number of call sites across all the operations. Given the \( n \) TriggerMetric tuples \( \{ < k_i, l_i > \} \) of a program, where \( k_i \) is the number of valid call sites and \( l_i \) is the
Figure 3.4: Illustration of dependence paths and various metrics for a program having two distinct operations $c$ and $c'$. Each operation has two call sites $s_1$ and $s_2$ and $s'_1$ and $s'_2$, respectively. A solid line represents the existence of a dependence path from some user trigger to a call site. A dashed line represents that none of the call site’s dependence paths has a user trigger.

\[
\text{TriggerMetric}(c) = \langle 2, 2 \rangle \\
\text{TriggerMetric}(c') = \langle 1, 2 \rangle
\]

### Assurance Score ($V$):
- 3 valid call sites / 4 call sites = 75%
- % of valid call site of $c$: 2 valid call sites / 2 call sites = 100%
- % of valid call site of $c'$: 1 valid call sites / 2 call sites = 50%

### Normalized DPVC Vector:

<table>
<thead>
<tr>
<th>%</th>
<th>0%</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
<th>50%</th>
<th>60%</th>
<th>70%</th>
<th>80%</th>
<th>90%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.5</td>
<td>0</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td>0.5</td>
</tr>
</tbody>
</table>

For the example in Figure 3.4, $V = \frac{3}{4}$, as there are total 4 call sites in the program, among which 3 are valid.

One can also compute the distribution associated with TriggerMetric values in a program, which provides useful insights into the program’s behaviors.

**Definition 3** DPVC Vector $W$ of a program is the normalized distribution of the percentages of valid call sites per operation. For operation $i$, the percentage of valid number of call sites is defined as $\frac{k_i}{l_i}$, where $k_i$ is the number of valid call sites and $l_i$ is the number of total call sites for the operation $i$. Let $n$ be the total number of distinct operations in the program.

Each percentage value determines the bin whose contents are augmented by one. After all percentage values are distributed, the value of each bin is divided by $n$, the total number of operation in the program. This yields a normalized distribution. Specifically, the distribution of the $n$ percentage values $\{\frac{k_1}{l_1}, \ldots, \frac{k_n}{l_n}\}$ is represented by the following 12 bins: 0%, (0%,
For the example in Figure 3.4 \((n = 2)\), the percentages of valid number of call sites for the two operations \((c \text{ and } c')\) are 100% \(\left(\frac{2}{2}\right)\) and 50% \(\left(\frac{1}{2}\right)\), respectively. Thus, most of the corresponding DPVC vector is 0, except for bins \([50\%, 60\%)\) and 100\%, i.e., one count in the \([50\%, 60\%)\) bin, and one count in the 100\% bin. After normalization, the entry for both the 100\% bin and \([50\%-60\%)\) bin is 0.5. Therefore, the final normalized distribution vector is \(\{0, 0, 0, 0, 0, 0, 0.5, 0, 0, 0, 0.5\}\), whose components are summed to 1.

The DPVC vector is computed from the TriggerMetric feature. Intuitively, it provides the in-depth statistics on the dependence-based validity of the calls in the program. The vector is used in our classification in Section 3.3, where we compare the DPVC vector of an unknown app with ones of known malware apps to infer their behavior similarities.

### 3.1.4 Program Analysis for Feature Extraction

The TriggerMetric feature is extracted from programs through static program analysis. In this section, we justify our use of data-flow analysis (as opposed to control-flow analysis) for this purpose. Our method tracks how a user’s input propagates throughout the program using data-flow analysis. Alternatively, one may attempt to capture how the user control action leads to a sensitive API call, which requires control-flow analysis.

For our trigger-based dependence analysis, data-flow analysis is more appropriate than control-flow. For example, control-flow analysis cannot be used to track the user’s input (data) that is used as arguments in sensitive API calls. However, data-flow analysis alone may overestimate the dependences due to the lack of the control analysis on branches (e.g., if). In this work, our feature is extracted from data-flow dependence analysis, which is coupled with event-specific control-flow dependence analysis. Our approach can be generalized to comprehensive control-flow analysis for improved accuracy.

Our dependence analysis tracks the propagation of triggers through events, including Android Intent. Intent is an event-based mechanism for communication between applications or components (Activity, Service, Receiver) in Android. For example, information entered by the user in one Activity may be passed through an Intent to another Activity or Service for processing. Therefore, the dependence graph needs to be augmented in order to obtain
the complete set of operations that depend on trigger variables through events. Without this expansion, the dependence analysis may underestimate the dependence relations (i.e., fail to report legitimate trigger-operation dependence relations). Because our focus is on dependences related to user activities, we perform Intent-specific control-flow analysis, as opposed to general control-flow analysis.

Next, we give a detailed description of the techniques used in our program analysis. The program analysis outputs TriggerMetric values for all the sensitive operations in the program. Then in Section 3.3, we present our classification method based on the TriggerMetric values. Our evaluation results are given in Section 3.4.

### 3.2 Feature Extraction Using Dependence Analysis

We present in detail our technique used for extracting the TriggerMetric feature from Android applications. To that end, we generate and analyze the data dependence graph, including

1) the general data-flow dependences,
2) the event-specific data dependence analysis for handling Android Intent and gathering comprehensive data dependence information,
3) reachability analysis for pruning unused code, and
4) backward depth-first search for finding dependence paths and computing a TriggerMetric for each operation.

Our program analysis takes as inputs the trigger set and the operation set, which are manually selected based on their semantics. The output of the program analysis is a set of TriggerMetric values \( \{ < k_c, l_c > \} \), one value for each sensitive operation \( c \), e.g., `sendTextMessage()`.

The pseudocode of our procedure for computing TriggerMetric values of a program is shown in Algorithm 1.
Algorithm 1 ComputeTriggerMetric

Input: $A \leftarrow \{\text{App code}\}$
$U \leftarrow \{\text{Set of user triggers of an app}\}$
$M \leftarrow \{\text{Set of operations call sites of an app}\}$
entryPoints of the app

Output: TriggerMetric set $S = \{< k_i, l_i >\}$, where $k_i$ is the number of valid call sites and $l_i$ is total number of call sites for operation $i$

1: $RUT \leftarrow \emptyset$ //a list of Reachable User Triggers (RUT)
2: $RCS \leftarrow \emptyset$ //a list of Reachable Call Sites (RCS)
3: $S \leftarrow \emptyset$ //a set of TriggerMetric values
4: parse AndroidManifest.xml file
5: $G \leftarrow \text{ConstructDataDependenceGraph}(A)$
6: $(RUT, RCS) \leftarrow \text{identifyReachableComponents}(U, M, \text{entryPoints})$
7: 
8: for each operation $i$ do
9: $C_i \leftarrow \{\text{set of call sites of operation } i\} \in RCS$
10: $k_i \leftarrow \text{checkPathExistence}(RUT, C_i, G)$
11: $l_i \leftarrow |C_i|$
12: $S \leftarrow S \cup < k_i, l_i >$
13: end for
14: return TriggerMetric set $S$
15:
16: procedure $\text{identifyReachableComponents}(U, M, \text{entryPoints})$
17: $G' \leftarrow \text{ConstructControlFlowGraph}(A)$
18: for each $u \in U$ do
19: perform DepthFirstSearch($u$, entryPoint, $G'$)
20: if a path $\in G'$ exists between $u$ and entryPoint then
21: $RUT \leftarrow RUT \cup \{u\}$ //Reachable User Triggers (RUT)
22: end if
23: end for
24: end procedure
25: for each $m \in M$ do
26: perform DepthFirstSearch($m$, entryPoint, $G'$)
27: if a path $\in G'$ exists between $m$ and entryPoint then
28: $RCS \leftarrow RCS \cup \{m\}$ //Reachable Call Sites (RCS)
29: end if
30: end for
31: return $RUT$ and $RCS$
32: end procedure
33:
34: procedure $\text{checkPathExistence}(RUT, C_i, G)$
35: $k_i \leftarrow 0$ //initialize $k_i$ for operation $i$
36: for each $c \in C_i$ do //for each call site of operation $i$
37: for each $u \in RUT$ do //for each user trigger
38: perform backward DepthFirstSearch($c$, $u$, $G$)
39: if a directed path $\in G$ exists between $c$ and $u$ then
40: $k_i ++$
41: break
42: end if
43: end for
44: end for
45: return $k_i$
46: end procedure
We first describe our construction of the dependence graph based on explicit def-use relations. The basic DDG graph is then augmented in order to capture def-use relations due to events.

### 3.2.1 General-Purpose Data-Flow Dependence

We use data-flow analysis to construct the data dependence graph (DDG) with intra- and inter-procedural call connectivity information to track the dependences between the definition and use of user-generated data in a given program. The intra-procedural dependence edges are identified based on local use-def chains. On the other hand, the inter-procedural dependence edges are identified based on constructing a call-site context-sensitive call graph supported by *points-to analysis* to build accurate call graphs. *Context-sensitive analysis* differentiates calling contexts of a function during analysis. *Context-insensitive analysis* analyzes a function summarizing over all calling contexts. Thus, a context-insensitive analysis may not provide as accurate a solution.

The above general-purpose data-flow analysis does not cover the data-flow associated with events, as Android event communications are usually implicit. To achieve a comprehensive dependence coverage, we describe our technique for the necessary event-specific dependence analysis next.

### 3.2.2 Augmentation with Event-Specific Data Dependence

Our augmented analysis handles two types of events – *i)* implicit method invocation (e.g., through listeners in GUI) and *ii)* Android-specific Intent-based inter-app or inter-component events. Our approach is to perform necessary control-flow analysis, which finds *bridges* between disjoint graph components, so that one can obtain the complete reachability of triggers. We describe our Android Intent-based dependence analysis that tracks the control-flow among Intent-sending methods in intra- and inter-application communication. This Intent-specific control-flow analysis is necessary for capturing data dependence relations between triggers and operations across multiple apps and their components.

Android Intent can declare a component name, an action and optionally includes data or extra data. For example, an Intent can be used to start a new activity by invoking the `startActivity(Intent i)` or `startActivityForResult (Intent i, ...)` methods.
Intent should be sent to a target component by matching the Intent’s fields with the declaration of the target component in the manifest. Android Intents can be used for explicit or implicit communication. An explicit Intent specifies that it should be delivered to a particular component specified by the Intent, whereas an implicit Intent requests the delivery to any component that supports a desired operation.

For explicit Intent, where the target component name is specified, we first identify the source component and the target component that are linked through an Intent object. This step pinpoints the Intent creation and sending methods (e.g., \texttt{startActivity(Intent i)} and \texttt{sendBroadcast(Intent i)}) to capture the control-flow dependences between the source and target components. In particular, we analyze the Intent object constructor to extract the name of the target component if it is provided. If it is not provided, we search the parameters in the \texttt{setClass()}, \texttt{setComponent()} or \texttt{setAction()} methods on the Intent object, which specify the target’s name to obtain the target component. Given this information, the dependence graph is augmented by adding a directed edge from the Intent-sending method of the source component to the entry point of the target component. This analysis is performed for all explicit Intents created in a given application.

For implicit Intent, the target component can be any component that declares its ability to handle a specified action. The target component is determined by the Android system based on the manifest file. We handle the implicit Intent by analyzing the AndroidManifest.xml file to extract a list of components with their actions to identify the target component. Implicit method invocation, such as those in the GUI, must be accounted for in the dependence graph. Our approach is to connect the dependent calls to the relevant API calls related to threads and listeners with their callee in the graph. For example, \texttt{Button.setOnClickListener()} is linked with an implicit call to its event handler implementation \texttt{onClick()}. We identified a list of all event handlers from Android developer documentation for our analysis. These methods effectively augment the general-purpose data dependence graph with the necessary Android event-specific data-flow information.

Obfuscation, Java reflection, and dynamic code loading cannot be analyzed statically. Dynamic analysis approaches (e.g., [64, 94]) are needed to extract related runtime behavioral features.
3.2.3 Reachability Analysis

The above operations produce a flow- and context-sensitive data-flow dependence graph with intra- and inter-procedural dependence analysis, and intra-and inter-application Intent-based dependence analysis. We then perform a reachability analysis for the app in order to remove unreachable code "dead code". Unreachable code is a portion of the program which contains classes/methods that are not executed. To that end, we construct an inter- and intra-procedural control-flow graph which shows all the possible execution paths. Given this control-flow graph and the list of user triggers and sensitive API calls, we perform reachability analysis to identify reachable user triggers and sensitive API calls from the entry points of the app. Specifically, we trace forward from the given entry point looking for the identified user triggers and sensitive API calls. For example, we perform reachability analysis to check whether a certain user trigger, e.g. click button, is reachable from the main activity. An activity is a visible portion of an application which handles user interaction.

There might be some user triggers inside other activities, but these activities never get executed or called from the main/parent activity. Hence, there is no reachable path from the entry point and these user triggers, and they can be safely ignored to increase the precision of our analysis. Similarly, some sensitive API calls may not be reachable from the entry points and never get executed. For example, a sensitive API getLastKnownLocation() in a tool app is unreachable from the apps entry points, and therefore will not be executed. Thus, we ignore and call it unreachable sensitive API call.

On the other hand, we call user trigger or sensitive API call reachable if there is a reachable path from the given entry point to this user trigger or sensitive API call. For example, assume that there is a sensitive API sendTextMessage() identified in a service component in app SendSMS. A service is an invisible portion of an application which performs background task. This service will be called from the main activity upon user clicks on a button. In this case, the sensitive API identified inside the service component will be executed. Thus, there is a reachable path from the main activity entry point to this sendTextMessage(), and hence we call it reachable sensitive API call.

As explained above, some user triggers and sensitive API calls may not be reachable and hence can be ignored in our analysis. Our subsequent dependence analysis will only be performed on reachable components. The reachability analysis increases the analysis precision by excluding unreachable code.
3.2.4 Finding User-Trigger Dependence Paths

Once the dependence graph is constructed, the next step is to identify paths between user trigger and sensitive API call pairs. We scan the graph for the occurrences of call sites of sensitive operations. In Algorithm 1, `checkPath Existence()` performs this task by performing backward depth-first traversal. For each call site \( s_i \) of an operation \( c \), we perform the backward tracing from \( s_i \) on the dependence graph searching for any user triggers on the dependence paths. For each \( c \), we record the valid number \( k_c \) of call sites, and the total number \( l_c \) of call sites. \(< k_c, l_c >\) is output as the TriggerMetric of the call \( c \), according to Definition 1.

Our implementation of the static analysis framework utilizes libraries in Soot, a static analysis toolkit for Java [1]. Our framework analyzes Java bytecode or source code.

Our DDG construction improves the def-use analyses provided by Soot \(^1\). Our prototype propagates def-use relations across the boundaries of methods. Our current prototype does not analyze native libraries. Yet, our approach can be generalized to analyze native code.

3.3 Classification Method

The classification decisions are based on the assurance score \( V \) and DPVC vector \( W \) of an app. An app is classified as either benign or malicious. These values are computed from the extracted TriggerMetric tuples \(< k_i, l_i >\) of the app, according to Definitions 2 and 3. Because of the simplicity of our feature, our classification is based on rules. In addition to classification decisions, our analysis also reports the names of operations with invalid call sites in the program.

Specifically, given the TriggerMetric values obtained from the program analysis, our classification has three steps: i) computing \( V \) and \( W \), ii) preliminary classification based on \( V \) with respect to a pre-defined threshold \( T \), and iii) further classification based on the weighted similarity analysis between vector \( W \) and those of known malware samples. In the next section, we present our two classification rules.

\(^1\)We augmented Soot libraries to support the inter-procedural call dependence analysis.
3.3.1 Our Classification Rules

Classification with assurance score. The threshold-based classification Rule 1 aims to detect apps that have low assurance scores, indicating the existence of a large portion of invalid call sites without proper user triggers.

**Rule 1** Given the assurance score $V$ of an Android app and an assurance threshold $T \in (0, 100\%)$, if $V < T$, then the app is classified as malware. Otherwise, it is classified as benign.

Clearly, the choice of $T$ affects the accuracy of the classification. In our experiments in Section 3.4, we found that a threshold of 75% gives a proper balance between the false positives (FP) and false negatives (FN). Probable malware needs to be further inspected.

For each app, we also applied the similarity-based classification rule.

Weighted similarity analysis on DPVC vector. This classification compares the DPVC vector of an app with the DPVC vectors of known malware samples. The purpose is to detect the apps who have similar distributions with malware in terms of the valid call sites. To that end, we first computed the DPVC vector $W^i$ for each malware $i \in [1, m]$ in a known malware sample set of size $m$. Then, we computed the average DPVC vector, which is denoted by $M$; that is, for each item $M_j$ in vector $M$, $M_j$ is computed as in Equation 3.2.

$$M_j = \frac{\sum_{i=1}^{m} W^i_j}{m} \quad (3.2)$$

Vector $M$ represents the average distribution of the percentage of valid call sites per operation among the known malware.

**Rule 2** Given the DPVC vector $W$ of an app, the average malware DPVC vector $M$, a similarity function $f$, and a threshold $T'$, if $f(W, M) \geq T'$, then the app is classified as malware. Otherwise, it is classified as benign.

Any similarity function may be used on DPVC vectors. In our experiments, we used a weighted cosine similarity function [81]. The function computes the cosine similarity between vectors $W$ and $M$, while applying weights to the ranges with smaller percentage values,
namely 0% and (0, 10%). The weights are computed based on an exponential function $2^x$ and then are normalized.

The reason for choosing the exponential weight function for this similarity measure is that we observed that the malware apps have a distinct distribution pattern from the legitimate apps towards the low percentage region, as shown in Figure 3.5. The weights amplify this distinction in the classification.

![Figure 3.5](image)

Figure 3.5: Averaged DPVC vectors representing a fine-grained distribution of per-operation valid call sites for 1,433 malware apps (top) and 2,684 free popular apps (bottom).

**Definition 4** A program is classified as benign if it is classified as benign by both Rule 1 and Rule 2. Otherwise, it is classified as malicious.

Our evaluation indicates the effectiveness of the above classification rules on the thousands of apps studied. We also painstakingly performed necessary manual inspections on some apps to validate our results and identified the causes of inaccuracies.
In the next section, we present a category of features derived from our TriggerMetric value which can be used for classification as well.

### 3.3.2 Variations of Classification Rules

Our classification rules are based on aggregated statistics on valid call sites of a program. One can define other classification rules using the TriggerMetric values \( \{< k, l >\} \) of a program. These rules may reflect different degrees of user-trigger dependence that is required in a trustworthy application.

To demonstrate the generality of the TriggerMetric feature, in this section we describe two examples of such classification rules, namely **All-Valid-Call-Sites Rule** and **Any-Valid-Call Site Rule**. Both rules defined below are based on the number of valid call sites \( k_i \) with respect to the total number of call sites \( l_i \) for an operation \( i \) in the program.

**Rule 3** All-Valid-Call-Sites Rule. *A program is classified as benign, if and only if all the call sites of all the sensitive operations are valid, i.e., having user-trigger dependence. If \( k_i = l_i \) \( \forall \) sensitive operation \( i \), then the program is benign. Otherwise, the program is classified as malicious.*

This above rule is equivalent to setting assurance threshold \( T \) to 100% in our classification Rule 1. In our experiments, there are 80.5% (2162) of apps that have 100% assurance scores. We conjecture that such a rule leads to low or zero missed detection, but many false positives.

A more relaxed classification rule can be defined below, which only requires *at least one* valid call site per sensitive operation.

**Rule 4** Any-Valid-Call-Site Rule. *A program is classified as benign, if for each sensitive operation there is at least one valid call site. If \( k_i \geq 1 \) \( \forall \) sensitive operation \( i \), then the program is classified as benign. Otherwise, the program is classified as malicious.*

For the example in Figure 3.4, this program is classified as malicious by Rule 3 and benign by Rule 4. In-depth comparison of the impact of these various classification rules and thresholds on Android security will be our future work.

In our experimental evaluation, the classification decisions are based on Rule 1 and Rule 2.
3.4 Experimental Evaluation

The objective of our evaluation is to answer the following questions:

1. Do the distributions of the assurance scores of malware and benign apps significantly differ?
2. What is the false negative (i.e., missed detection) rate when classifying known malware samples?
3. Can our method discover new malware apps that have not been previously reported?
4. What are the reasons for false positives?

3.4.1 Experiment Setup

We performed an evaluation with 1,433 Android malware apps collected by [103] and Virus Share [3]. The known Android malware apps perform malicious functionalities, such as sending unauthorized SMS messages (e.g., FakePlayer), subscribing to premium-rate messaging services automatically (e.g., RogueSPPush), listening to SMS-based commands to record and upload the victim’s current location (e.g., GPSSMSSpy), stealing users’ credentials (e.g., FakeNetflix), and granting unauthorized root privilege to some apps (e.g., Asroot and DroidDeluxe)\(^2\).

We also evaluated 2,684 free popular real-world Android apps from Google Play market, covering various application categories. These free apps include those with different levels of popularity as determined by the user rating scale. In particular, we used 1,039 high popularity apps, 713 intermediate popularity apps, and 932 low popularity apps. We assumed that the trustworthiness of these free apps is unknown and they may be malware or may contain malicious components. We converted Android app code (apk) from the .dex format to .class files using the Dare tool [65] and extracted features from the Java bytecode.

\textit{Averaged DPVC vector of known malware.} We computed the DPVC vector for each of the 1,433 malware samples, and then computed their average DPVC vector according to

\(^2\)The malware naming convention follows [103].
Equation 3.2. The average malware DPVC vector approximates the distribution of valid call sites in malicious apps. It was used for the similarity test of unknown apps in Rule 2.

**Thresholds for classification rules.** For our two classification rules (Section 3.3), we choose the assurance threshold $T$ to be 75% for Rule 1 and the similarity threshold $T'$ to be 0.8 for Rule 2. Empirical results showed that these values provide a high detection rate without producing excessive false alerts.

### 3.4.2 Known Malicious Apps

**Assurance Scores of Known Malware.** Most of the malware apps have low assurance scores, indicating that a significant number of sensitive API calls are made without proper user triggers. Invalid call sites that we observed include those for writing and sending information through the network, sending unauthorized SMS messages, executing system commands, and accessing user’s private data. E.g., *Asroot* and *BaseBridge* use `Runtime.exec()` to execute system commands without valid user triggers.

We found that 479 malware apps out of 1,433 apps have 0% assurance scores. The rest of the 954 apps have positive assurance scores. Among them, many malware apps are repackaged from benign apps\(^3\), e.g., *ADRD*, *DroidDream*, and *Geinimi*. Malware writers bundle malicious code with existing benign apps. Repackaging explains our observation that a significant number of malware apps (954 out of 1,433) have non-zero assurance scores. Positive assurance scores indicate that a portion of the sensitive operations in these malware apps exhibit the required dependences on user triggers.

**FakeNetflix** is the only malware app that has a 100% assurance score. **FakeNetflix** is a phishing app, which provides a fake user interface to trick the user to enter her or his Netflix credential. This type of phishing malware circumvents virtually all behavior-based detection approaches, including ours. App certification and user education are more effective defenses than program analysis for this type of social engineering malware.

The detailed distribution of the assurance scores for the known malicious apps can be found in Figure 3.6.

---

\(^3\)The problem of detecting repackage apps (e.g., [20]) has a more specific goal from our general app classification. It typically requires graph-based pair-wise app similarity analysis.
Figure 3.6: Distinct distributions of assurance scores ($V$) for known malicious apps and free popular apps.
Classification Results of Known Malware

The classification results on known malware apps are given in Table 3.1. Using assurance scores, Rule 1 labels most (92.5%) of the samples as malicious, as they have lower-than-75% \( V \) values. Rule 1 labels 108 apps (7.5%) as probably benign. Using DPVC vectors, Rule 2 labels malicious for 5.4% (77) apps out of the 108 probably benign cases, as these apps have low percentages for valid call sites per operation. Thus, we correctly detect 97.9% of the 1,433 malware samples. The false negative rate is 2.1%, i.e., 31 malware apps are misclassified as benign.

<table>
<thead>
<tr>
<th>Rule 1 (( V ))</th>
<th>Rule 2 (DPVC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malicious</td>
<td>Malicious</td>
</tr>
<tr>
<td>92.5%</td>
<td>5.4%</td>
</tr>
<tr>
<td>Benign</td>
<td>Benign (FN)</td>
</tr>
<tr>
<td>7.5%</td>
<td>2.1%</td>
</tr>
<tr>
<td></td>
<td>out of 7.5%</td>
</tr>
</tbody>
</table>

Table 3.1: Summary of classification results on 1,433 known malware apps. Rule 2 is applied to the apps that are classified as benign by Rule 1. The false negative (FN) rate refers to the portion of malware apps classified as benign by both rules and is 2.1%.

The main reason for misclassification is malware repackaged from existing benign code, resulting in malware with profiles similar to benign apps. For example, one of the 31 undetected malware apps is DroidKungFuSapp, which contains malicious code bundled with com.aijiaoyou.android.sipphone (an app for learning Chinese). As a result, this malware app has a high assurance score \( V \) of 85.7% and a low similarity value (0.015) with known malware.

There are two possible countermeasures to combat the misclassification of repackaged malware apps. The first countermeasure is to adjust the rules thresholds used for the classification. For example, we set a threshold for rule 1 (assurance score \( V \)) to 75% in our evaluation. One can raise this threshold to be 90% or more. In this case, the repackaged malware such as DroidKungFuSapp with assurance score \( V \) of 85.7% will be detected.

A more advanced countermeasure is to separate and identify the original benign portion of the app and the injected malicious code. In any repackaged app, the malicious components are highly communicated/connected together and loosely connected with other benign components. Hence, one possible way to identify this is to analyze the connectivity of the call
graph of a repackaged app to identify the loosely connected or disconnected graph components. Then, one can compute features separately for each graph components and observe the imbalance. Table 3.2 shows the results of our assurance scores $V$ for the benign and malicious components separately for some of the repackaged malware apps. The $V$ scores for the benign components are much higher than the malicious components which show the validity of our proposed feature.

Table 3.2: Assurance scores for the benign and malicious components in some repackaged malware apps.

<table>
<thead>
<tr>
<th>Repackaged Malware Name</th>
<th>Assurance Score of Benign Component</th>
<th>Assurance Score of Malicious Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.noisysounds</td>
<td>90%</td>
<td>26%</td>
</tr>
<tr>
<td>com.miniarmy.engine</td>
<td>100%</td>
<td>35%</td>
</tr>
<tr>
<td>com.chenyx.tiltmazs</td>
<td>78%</td>
<td>20%</td>
</tr>
<tr>
<td>com.craigsrace.headtoheadrcing</td>
<td>86%</td>
<td>28%</td>
</tr>
</tbody>
</table>

### 3.4.3 Free Popular Apps

Because the ground truth on trustworthiness of the free popular apps are not known, our analysis on them is more complex. Some of the classification decisions are validated through significant manual inspection of the code. We present our results on the i) assurance score computation, ii) classifications using two rules, and iii) new malware discovery.

**Assurance Scores of Free Apps**

Among the 2,684 free popular apps, 80.5% of them have 100% assurance scores, indicating that all the call sites of all the sensitive operations have valid user-trigger dependence. The detailed distribution of the assurance scores are shown in Figure 3.6. For the 80.5% of the apps that have 100% assurance scores, we utilized a signature-based malware scanning tool *VirusTotal* for additional validation. VirusTotal has 48 signature-based scanners (e.g., McAfee, NOD32, BitDefender). We found that only one scanner out of 48 scanners in *VirusTotal* triggers an adware alert for 13 free popular apps which have 100% assurance scores (true positives). The rest of the free popular apps with 100% assurance scores are benign (true negatives), none of them trigger any alert by *VirusTotal*. 
Through manual inspection, we find that the use of advertisement and analytics libraries is one main reason for sensitive operations to be called without proper user triggers. We selected several apps with less-than-100% $V$ scores and computed their assurance scores with and without the ad/analytics libraries. The $V$ scores are boosted significantly without the ad/analytics libraries. The results are shown in Table 3.3.

Table 3.3: Assurance scores of subset of selected benign apps including or excluding the ads/analytics libraries.

<table>
<thead>
<tr>
<th>App Name</th>
<th>Including Ads Libs</th>
<th>Excluding Ads Libs</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.canadadroid.fantasy</td>
<td>75.0%</td>
<td>100.0%</td>
</tr>
<tr>
<td>com.canadadroid.penguinskiing</td>
<td>79.2%</td>
<td>100.0%</td>
</tr>
<tr>
<td>com.CalcFinalProgress</td>
<td>85.2%</td>
<td>96.3%</td>
</tr>
<tr>
<td>AzureNightwalker.ContactList</td>
<td>89.7%</td>
<td>97.4%</td>
</tr>
</tbody>
</table>

We also found a few malicious apps with high enough assurance scores (e.g., $V$ is 89%) to pass our classification threshold (i.e., false negative), e.g., a spyware wallpaper app com.ysler.wps.d3d available on Google Play market.

Classification Results of Free Popular Apps

Our classification results are summarized in Table 3.4. Most of these free popular apps from Google Play market are classified as benign by both rules. Rule 1 labels 7.2% (193) of the 2,684 apps as malicious. We then applied Rule 2 to both categories of apps.

For apps classified as malicious by Rule 1. We applied Rule 2 to these 7.2% of the apps. Rule 2 labels 6.5% of the total (175 of 193) as malicious. The other 0.7% (18) are labeled benign.

For apps classified as benign by Rule 1. We applied Rule 2 to these 92.8% of the apps. Rule 2 labels 1.7% (47) of them as malicious, and classifies the rest 91.1% as benign.

There are 240 apps that are labeled as malicious by both or either one of the rules. Their popularity distribution is as follows, with higher concentrations of suspicious apps in medium and low popularity categories.

- High popularity category: 70 apps (29.2%)
Table 3.4: Summary of classification results after applying both rules on 2,684 free popular apps.

<table>
<thead>
<tr>
<th>Rule 1 (V)</th>
<th></th>
<th>Rule 2 (DPVC)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Malicious</td>
<td>7.2%</td>
<td>Malicious</td>
<td>6.5%</td>
</tr>
<tr>
<td>Benign</td>
<td>92.8%</td>
<td>Benign</td>
<td>0.7%</td>
</tr>
</tbody>
</table>

- Medium popularity category: 87 apps (36.3%)
- Low popularity category: 83 apps (34.5%)

To confirm the correctness of our results, we then performed various code inspection on them, the detail of which are described next.

**New Malicious Apps Found**

To confirm that the apps classified as malicious are truly malicious, manual code inspection was performed. We also utilized the VirusTotal tool for additional validation.

Our method discovered many new malicious Android apps that cannot be detected by the VirusTotal tool. These new malware apps did not trigger any alerts in VirusTotal. A subset of these new malicious apps is shown in Table 3.6 with examples of their sensitive function calls that lack of valid user-trigger dependence. All of them are confirmed by our manual analysis to have malicious functionalities. In Table 3.6, each column is a category of malicious action, e.g., unjustified dynamic code loading and unnecessary accessing of user information. Names of call sites without valid user-trigger dependence are given. All the apps shown in this table fail both of our classification rules, yet do not trigger any alerts in VirusTotal.

We highlight a few of the new malware that we discovered in the free popular apps. Our method detects a malicious app *Time Machine*, which is repackaged from an ebook app. The

---

4 Out of the 240 apps, 137 apps triggers at least one alert in VirusTotal.
malware invokes many sensitive APIs (in Jslibs library) to perform unjustified operations, such as recording sound, retrieving phone state, and exfiltrating geolocation information. We find that an organizer app com.via3apps.usobesit618 is bundled with a piece of malware collecting private information, such as device ID, email address, latitude and longitude, phone number, and username, and it uploads the details to a remote server. Another malware app is a game-guide app com.bfrs.krokr, which is bundled with adware AndroidApperhand (aka Android.Counterclank). AndroidApperhand is a piece of aggressive adware. It attempts to modify the browser’s home page, copy bookmarks on the device, shortcuts, push notifications, and steal build information (brand, device, manufacturer, model). This adware also attempts to connect to a remote host.

For the apps that are labeled as malicious by only one rule (2.4% out of 2,684 apps), we have confirmed that most of the apps (2.2% of 2.4%) contain aggressive advertisement libraries, such as Mobclix, Tapjoy, and Waps. These libraries invoke sensitive operations without any user triggers. Unlike regular ad libraries, these aggressive ad libraries contain an overwhelming amount of invalid call sites. Most of them have a large number (> 50%) of sensitive operations with zero valid call sites, which is consistent with known malware. Other researchers have also confirmed the potential security issues raised by these aggressive ad libraries [48].

**False Positive Rate (FPR)**

FPR is computed as \( \frac{FP}{FP+TN} \), where TN stands for true negative (benign apps). 240 apps are classified as malicious by our method. VirusTotal scanning confirms 137 of them are malicious. For the rest of 103 apps, we randomly selected 21 apps out of these 103 apps and perform a thorough manually code inspection. We found that 11 of the 21 apps have definitive malicious or aggressive code behaviors that threaten the system assurance and data confidentiality in Android (described in Section 3.4.3 and Table 3.6). These behaviors were found in either the main components or adware. In the other 10 apps we did not find any threats, thus concluded that they are benign (false positives). The total false positives are estimated at 103 * \( \frac{10}{21} \) = 49. Since the trustworthiness of the free popular apps is unknown, we used VirusTotal to check all the free popular apps classified as benign by our method (true negatives). We found that only one scanner out of 48 scanners in VirusTotal triggers an adware alert for 27 apps (true positives). The true negatives (TN)
are $2684 - 240 + 49 - 27 = 2466$, yielding a 2.0% FPR.

### 3.4.4 Performance Evaluation

The experiments were conducted on a computer which has 3.0GHz Intel Core 2 Duo CPU E8400 processor and 3GB of RAM. We measure the time for parsing the AndroidManifest.xml file, Soot execution for constructing the dependence graph, the reachability analysis, and finding the dependence paths by traversing the graph. The average processing time for an app is about 158.01 seconds. This processing time does not include the time required to convert the dex format to jar. Table 3.5 shows the average time required by each analysis phase.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Average Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reachability Analysis</td>
<td>14.17</td>
</tr>
<tr>
<td>Finding Dependence Paths</td>
<td>54.30</td>
</tr>
<tr>
<td>AndroidManifest.xml Parsing</td>
<td>0.01</td>
</tr>
<tr>
<td>Graph Construction using Soot</td>
<td>89.53</td>
</tr>
<tr>
<td><strong>Total Time</strong></td>
<td><strong>158.01</strong></td>
</tr>
</tbody>
</table>

### 3.4.5 Summary of Experimental Findings

These experimental results suggest that our rule-based classification with a single complex feature is quite effective. We summarize our major experimental findings.

1. There are an overwhelming number of malware apps with zero or low assurance scores, indicating that a large portion of sensitive call sites in these programs are invalid. The DPVC vectors (representing a fine-grained distribution of per-operation valid call sites) of malware and benign apps have significantly different distributions (shown in Figure 3.5). Malware has a high concentration of zero or low per-operation valid call sites.

2. We obtained a low false negative (i.e., missed detection) rate of 2.1% when classifying
1,433 known malware samples based on their assurance scores and DPVC vectors, suggesting the effectiveness of our detection.

3. Our method identified 240 free popular apps (8.9%) as suspicious from Google Play market\(^5\). These malware excessively access device resources and personal information without any user knowledge. Our program analysis method effectively pinpoints these problematic call sites. Our method detects many malware that cannot be detected by VirusTotal scanning. Some of them are shown in Table 3.6. We confirmed them by manual code inspection. Our false positive rate \((\frac{FP}{FP+TN})\) is estimated at 2.0%.

Our method identified more suspicious apps from the medium and low popularity categories than the high popularity category.

4. We observed several types of triggerless operations that are benign. Sensitive operations during \(i\) app launching activities (e.g., default_app_set.main.ver1), \(ii\) background service components (e.g., com.monotype.android.font.dev.comic), or \(iii\) benign ad/analytical libraries (e.g., rappsd.v1) are typically automatically completed without user triggers. These factors result in lower assurance scores and skewed DPVC vectors, which may cause false positives. The classification accuracy is also affected by the accuracy of \textit{Dare} in translating Dalvik bytecode to Java bytecode.

\section{3.5 Discussion}

In this section, we discuss the security guarantees provided by our app classification work, and sources of inaccuracy in our program analysis. We also describe possible extensions to the feature definitions.

\subsection{3.5.1 Security Analysis}

Our app classification can be used to detect malware that invokes sensitive operations. Sensitive operations typically involve accessing system resources and sensitive data. Inferring their user-intention dependences enables the detection of potential data confidentiality and

\footnote{Google later took some malware apps off the Play market, e.g., Us-Obesity-And-You-Teenagers.}
authorization issues. Examples of malicious patterns that can be detected by our analysis include:

- **Resource access:** executing sensitive operations without proper user triggers, such as sending unauthorized SMS messages, subscribing to premium-rate services automatically, or granting unauthorized root privilege to apps.

- **Data access:** accessing sensitive data items without proper user triggers, such as recording and uploading the victim’s current location. Our static analysis does not track sensitive data variables. Instead, the function calls that may be used to access sensitive data are labeled (as operations) and analyzed.

In our model, the accuracy of the analysis is closely related to the accuracy of the data dependence analysis. Intra-procedural analysis captures fine-grained def-use relations within a function. The intra-procedural def-use relations can prevent a superfluous user input attack, thusly. One possible attack scenario is where the malware may require superfluous user inputs (before making function calls to conduct unauthorized activities) attempting to satisfy the dependence, but the user inputs are not consumed by the calls. For example, the user enters a phone number and a message to send SMS. The phone number entered by the user can be ignored and replaced with other number inside `sendTextMessage()` function. This type of data flow can be detected by tracking the dependence between the user inputs entered and the sensitive API calls, thus the superfluous user inputs can be identified.

Social engineering app is an application that provides fake user interface to look legitimate in order to circumvent the user and perform malicious activities (e.g., stealing money). Social engineering apps may demonstrate proper trigger-operation dependences, because of the seemingly conforming dependence paths between user triggers and sensitive operations. Therefore, due to the intrinsic nature of our user-intention analysis, it is not suitable for detecting social engineering apps. Possible solutions for this could be using app certification and user education.

The legitimate apps which require few user interactions may raise false alarm. For example, a calendar app can send an automatic reminder email message of a calendar event that previously scheduled by the user. Hence, the sensitive API that sends the email message may raise an alarm according to our security model since it is not explicitly triggered by the user. For example, the user has previously entered this event into the calendar. This
action can be used as a trigger that justifies the operation of sending reminder emails. Our approach can be extended to address this problem by expanding and generalizing the definition of user triggers. The analysis for this calendar problem will be more complicated than our current solution. The reason is that the information entered by the user is stored in a data structure or file to be read back when it is needed. Hence, there is no direct dependence between sending reminder email operation and the original user triggers used to store the information. One needs to expand and include this type of indirect dependence relation.

For the rule-based method, it is easy for the malware writers to game with the analysis than the machine learning-based classification. This is because the machine learning techniques utilize a large number of features compared to the rule-based method. So, it is harder for the attacker to compromise since she/he has to deal with many features in order to circumvent the security solution. On the other hand, the rule-based method might be easy for the attacker to compromise since she/he has to deal with a one/fewer number of features.

Precisely modeling a program’s semantics and intention is in general challenging and open problem. In the seminal work on computer virus [18], Cohen described the seminal impossibility result on malware analysis. The defense is still an open problem and similar arm-race issue exists in virtually all security solutions.

3.5.2 Sources of Inaccuracy in Feature Extraction

Overestimation of trigger-operation dependence may cause false negatives in the analysis report (i.e., failing to detect potentially malicious operations in the app). Certain dependence paths may only exist under specific data or control conditions. These branch conditions may not be statically predictable, resulting in overestimation. Some data dependence overestimation may be mitigated by identifying the specific conditions for certain dependence paths to be valid (e.g., by symbolic execution).

Conversely, underestimation of triggers may cause false positives. For instance, legitimate API calls can be triggered by runtime events such as clock-driven events from the calendar (e.g., the calendar app sends a reminder email message of a calendar event), or triggered by incoming network events. These runtime events may not be explicitly triggered by the user and thus lack the proper dependence according to our security model. One mitigation to the
problem is to generalize and expand our definitions of triggers to include other legitimate triggering events. However, because triggers may be generated at runtime, static analysis alone may not be sufficient for feature extraction. Hybrid features extracted from both static and dynamic analyses are needed for complete dependence properties in a program. Its realization remains an interesting open problem.

Static program analysis has difficulty in performing the analysis on programs that employ obfuscation or encryption techniques. Obfuscation is mainly used to make the programs code difficult to understand.

Some Android apps use obfuscation to protect intellectual property [33]. ProGuard is a recommended obfuscation tool by Google to protect against readability and does not obfuscate control flow. Hence, its impact is limited on static program analysis.

As indicated by [33], it is easy to recognize some forms of the obfuscated code in Android apps. In particular, class, method, variable, and Java filename names are converted to single letters (e.g., a.java). However, several ads and analytics libraries are obfuscated to protect their intellectual property [33]. To obtain a rough estimate of the number of apps whose main code is obfuscated not the ads or analytics libraries, we used the same approach proposed in [33] to search for a single letter Java filename (e.g., a.java) within a file path of the package name. This heuristic is used to obtain insight for finding obfuscation code in apps, but it is not a solid characterization. We found only 40 malware apps (2.8%) out of the 1433 apps have this code obfuscation. Moreover, we found 250 free popular apps (9.3%) out of the 2684 apps have this code obfuscation in part of their main code. We applied our analysis on the reversed engineering Java bytecode using Dare tool to translate Dalvik bytecode to Java bytecode. The accuracy of our analysis is constrained by the accuracy of the reverse engineering tools.

There are several obfuscation techniques:

- **Renaming technique**: it renames classes, variables, and methods using meaningless names. This type of technique can not affect our approach since it just renames classes, variables, and methods without changing the content or the control flow structure.

- **String encryption technique**: it encrypts the string data.

• Control flow obfuscation technique: it reorders the code and inserts additional code statements while preserving the code semantic.

The latter two techniques can affect our approach since they change the data and the structure of the program. On possible solution is to use dynamic analysis [64, 94] to provide insights about the programs runtime execution. As a future work, we plan to utilize the dynamic analysis with our user trigger dependence approach to get insights on which sensitive APIs are triggered by user inputs/actions. One way to do this is to label the user inputs/actions and to interpose the sensitive APIs in .apk file and insert monitoring code to get the sensitive API call logs during the app execution.

3.6 Comprehensive Behavior Profiling

Machine learning techniques have been widely adopted in the computer security literature since the work by Lee et al. [55]. Equipped with domain knowledge, the methods extract domain specific features based on empirical observations of malicious programs or traffic patterns. For example, solutions described by Cova et al. [19] use binary classification techniques to identify malicious Javascript code on the web. The features they extracted from malicious code include the presence of redirection and obfuscation. Xie et al. [88] used a Bayesian network to infer abnormal network traffic patterns. Besides classifying programs and network traffic, learning-based security research also includes database intrusion detection [79] and SMS/social network spam detection [80].

We present another technique [84] of screening for malicious Android applications that combines two different types of features about Android applications. The first one is the percentage of valid call sites (PVCS) metric (described in Section 3.1.3). The second one is the permissions that the application requests when it is installed. While each kind of feature is useful on its own, when combining them, we are able to achieve a significant improvement compared to the state of the art.

The features are used with machine learning algorithms to classify previously unseen applications as malicious or benign with a high degree of accuracy. We used the same datasets described in Section 3.4 and compared five different classifiers: support vector machines (SVMs), random forests, naive Bayes, k-nearest neighbors (KNN), and boosted decision
trees (J48 with Adaboost). Our technique with boosted decision trees classifier outperforms the previous state of the art by a significant margin, with particularly low false positive rates (1%) as shown in [84]. Furthermore, the classifier evaluation is performed on malware families that were not used in the training phase, simulating the accuracy of the classifier on malware yet to be developed. We found that our PVCS metric and the SEND_SMS permission are the specific pieces of information that are most useful to the classifier.

3.7 Summary

We demonstrated the high classification accuracy achieved by using a single well-prepared feature on Java programs. What differs our feature from those used in existing work is that our classification enforces carefully-chosen benign properties in programs. These benign properties are observed in trustworthy programs, but not in malware. Our method computes the percentage of sensitive API calls that depend on some form of user inputs or actions through def-use analysis of the code. Our analysis successfully discovered new malware apps available in Google Play store. Our approach can be applied to general user-centered programs and applications beyond the specific Android environment studied.
Table 3.6: Malicious activities of a subset of new malware found by our method.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>com.pougamefree.cheats†</td>
<td>getDeviceId() -</td>
<td></td>
<td>getLongitude() -</td>
<td>getLatitude()</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>getLatitude()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getAllBookmarks() -</td>
<td>getIPAddress()</td>
<td>getLatitude() -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getLatitude()</td>
<td>getIpAddress()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>com.canny.FishHunter</td>
<td>getDeviceId() -</td>
<td></td>
<td>getLongitude() -</td>
<td>getLatitude()</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>getLatitude()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getSimSerialNumber() -</td>
<td>getLine1Number()</td>
<td>getLatitude() -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getLatitude()</td>
<td>getIpAddress()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>com.hd.peliculasbd†</td>
<td>getDeviceId() -</td>
<td></td>
<td>getLongitude() -</td>
<td>getLatitude()</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>getLatitude()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getLine1Number() -</td>
<td>getSimSerialNumber()</td>
<td>getLatitude() -</td>
<td>getLongitude()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getLatitude()</td>
<td>getSimSerialNumber()</td>
<td>getLongitude()</td>
<td>getIpAddress()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>oms.cj.kobodl</td>
<td>getDeviceId() -</td>
<td></td>
<td>getLongitude() -</td>
<td>getLatitude()</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>getLatitude()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getSimSerialNumber() -</td>
<td>getLine1Number()</td>
<td>getLatitude() -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>com.canny.RankSwap</td>
<td>getDeviceId() -</td>
<td></td>
<td>getLongitude() -</td>
<td>getLatitude()</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>getLatitude()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getLine1Number() -</td>
<td>getSimSerialNumber()</td>
<td>getLatitude() -</td>
<td>getLongitude()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getLatitude()</td>
<td>getSimSerialNumber()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>com.via3apps.relation958</td>
<td>getDeviceId() -</td>
<td></td>
<td>getLongitude() -</td>
<td>getLatitude()</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>getLatitude()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getSimSerialNumber() -</td>
<td>getLine1Number()</td>
<td>getLatitude() -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getLatitude()</td>
<td>getLine1Number()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>com.berobo.android.scanner</td>
<td>getDeviceId() -</td>
<td></td>
<td>getLongitude() -</td>
<td>getIpAddress() -</td>
<td></td>
<td>loadClass()</td>
<td>sendTextMessage()</td>
</tr>
<tr>
<td></td>
<td>getLatitude()</td>
<td></td>
<td></td>
<td>getIpAddress() -</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getAccuracy()</td>
<td></td>
<td>getLine1Number() -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getAccuracy()</td>
<td></td>
<td>getSimSerialNumber()</td>
<td>getLine1Number()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getAccuracy()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

†App has been removed from Google play market by 12/05/2013.
Chapter 4

Characterization of Android Inter-Component Communication (ICC)

In this chapter, we characterize and analyze the Intent-based ICC channels which are standard communication channels in Android. Moreover, we experimentally demonstrate the difficulties and technical challenges associated with distinguishing benign ICC flows from colluding ones.

4.1 Android ICC

Android applications are component based which use Intents to communicate within the application’s components or with other applications’ components. Intents can be delivered to the following application components: Activities which provide user interfaces, Services which run in the background and do not interact with the user, and Broadcast Receivers which receive Intent that is sent to multiple applications. Android Intent can declare a component name, an action and optionally includes data. For example, an Intent can be used to start a new activity by invoking the \texttt{startActivity(Intent i)} or \texttt{startActivityForResult (Intent i, ...)} methods. An Intent should be sent to a target component by matching the Intent’s fields with the declaration of the target component in the manifest. Android
Intents can be used for \textit{explicit} or \textit{implicit} communication. An \textit{explicit Intent} specifies its particular target component by name, whereas an \textit{implicit Intent} requests the delivery to any component that supports a desired action.

### 4.2 ICC Channel Characterization

The purpose of our analysis is to study the ICC in Android and investigate whether it is commonly used by the apps or not. To achieve this, we define \textit{CAIMap} to identify which apps are communicating with a given app by analyzing the ICC-based communication.

#### 4.2.1 CAIMap Definition and Construction

The objective of the \textit{CAIMap} is to capture all types of communication (internal and external) of an app. Specifically, we need to identify the intra- and inter-application communications of the app via Intent-based ICC APIs such as \texttt{startActivity(Intent i)}, \texttt{startService(Intent i)}, and \texttt{sendBroadcast(Intent i)}.

\textbf{Definition 5} \textit{CAIMap} is a directed cyclic graph $G(V, E)$ for app $A$. Each node $v \in V$ in $G$ represents a component or action name, and each edge $e \in E$ in $G$ represents an ICC API. There are two types of communication:

- \textit{Internal communication}: component $X$ is communicating with component $Y$, where both $X$ and $Y$ are internal components in app $A$.

- \textit{External communication}: component $X$ is communicating with component $Y$, where component $X$ is in app $A$, and component $Y$ is in app $B$.

To determine if the app has an external communication, we get the list of components and their actions defined in the manifest of the app. Then, for each target component or action we found during our analysis, we match this target component with this list. If this target component is not defined as one of the internal components, we label it as external component and infer that this application has external ICC.
For each app, we store the CAIMap information as a set $S$ consisting of multiple four-item tuples $<ICCName_k, sourceComponent_k, targetComponent_k, typeOfCommunication_k>$, where

- **ICCName** is the API name of ICC$_k$, such as `startActivity()` and `startService()`, and

- **sourceComponent** is the name of the component which initiates the ICC$_k$ (exit point). It is a subclass of `Activity`, `Service`, `BroadcastReceiver`, and

- **targetComponent** is the name of the component which receives the ICC$_k$ (entry point). It is a subclass of `Activity`, `Service`, `BroadcastReceiver`, and

- **typeOfCommunication** is the type of ICC$_k$ communication, internal or external.

The goal is to find all Intents used in the ICC APIs to identify the source and target components linked by the Intents. We utilize the data-flow analysis in Soot framework [1] to find the Intent object used in the ICC API. Specifically, we use data-flow analysis to construct the data dependence graph (DDG) of intra- and inter-procedural dependences to track the dependences between the definition and use of Intents in a given app. The DDG is a common program analysis structure which represents inter-procedural flows of data through a program [51].

For an explicit Intent, where the target component name is specified, we first identify the source component and the target component that are linked through an Intent object. In particular, we analyze the Intent object constructor to extract the name of the target component if it is provided. If it is not provided, we search the parameters in the `setClass()`, `setComponent()` or `setAction()` methods on the Intent object, which specify the target’s name to obtain the target component.

For an implicit Intent, the target component can be any component that is declared to handle a specific action. The target component is determined by the Android system based on the manifest file. We handle the implicit Intent by analyzing the manifest.xml file to extract a list of components with their actions to identify the target component. The source component is obtained by identifying the class/component where the ICC API is defined. Once we identify the source and target components, we add a link between them.
Our CAIMap construction takes as input the source or bytecode of the app and its manifest. The output is a set of CAIMap information \{<ICCName_k, sourceComponent_k, target Component_k, typeofCommunication_k>\} for each ICC_k. The pseudocode of our procedure is shown in Algorithm 2.
Algorithm 2 CAIMap Construction

Input: $A \leftarrow \{\text{app code}\}$ $B \leftarrow \{\text{app manifest.xml}\}$
Output: CAIMap set $S = \{<\ ICCName_k, \ sourceComponent_k, \ targetComponent_k, \ typeOfCommunication_k >\}$

1: $ICCLlist \leftarrow \emptyset$ //a list of ICC APIs in an app
2: $InternalCompList \leftarrow \emptyset$ //a list of identified components in an app
3: $InternalCompList \leftarrow \text{parse}(B)$
4: $ICCLlist \leftarrow \text{scan app’s code for ICC APIs}$
5: $G \leftarrow \text{ConstructDataDependenceGraph}(A)$
6: $S \leftarrow \text{constructCAIMap}(ICCLlist, G)$
7: return $S$

procedure constructCAIMap(ICCLlist, $G$)

11: for each $icc \in ICCLlist$ do
12: $SourceComp \leftarrow \text{component name where }icc\text{ is defined}$
13: $TargetComp \leftarrow \text{getTargetComponentName}(icc, G)$
14: if $TargetComp$ is not in $InternalCompList$ then
15: $typeOfCommunication \leftarrow \text{external}$
16: else
17: $typeOfCommunication \leftarrow \text{internal}$
18: end if
19: $S \leftarrow S \cup \{<icc, SourceComp, TargetComp, typeOfCommunication>\}$
20: end for
21: return $S$
end procedure

procedure getTargetComponentName(icc, $G$)

25: perform backward depth-first traversal from $icc$ to the Intent object used in $icc$ on $G$
26: $TargetComp \leftarrow \text{extract the target component name from the Intent object}$
27: return $TargetComp$
end procedure
4.2.2 Example of CAIMap

Figure 4.1 shows an example of partial CAIMap for abc.ssd.TrafficInfoCheck app. It is an app to display information about the Japanese railway service. In this example, there are five ICC APIs. This map is constructed based on the ICCMap information set which has five tuples for this example. For instance, <startActivity(), abc.ssd.TrafficInfoCheck. TrafficInfoActivity, android.intent.action.VIEW, external> is one tuple in the CAIMap information set S. This tuple can be interpreted as there is an ICC call startActivity() from the source component (abc.ssd.TrafficInfoCheck.TrafficInfoActivity) to a target component which declares to handle this action android.intent.action.VIEW. This target component is an external component since none of the internal components in this app can handle this action. Only other components in other apps, which are declared to handle this action, will receive the request. Hence, we infer that this app is communicating with another app through this action.

![Partial CAIMap for abc.ssd.TrafficInfoCheck application.](image)

4.3 Experimental Evaluation

The purpose of this experiment is to show the difficulties and challenges in classifying benign Android ICCs from colluding ones. In particular, the objective of this experiment is to answer the following questions:

1. How often do benign apps perform inter-app communications with other apps?
2. How effective is the existing collusion detection solution (namely XManDroid [13]) in terms of false positive rate?
We implemented a static analysis tool in Java to construct the CAIMap for a given Android app. We performed our ICC characterization study on 2,644 free popular real-world Android apps from Google Play market. These apps covering various application categories and different levels of popularity as determined by the user rating scale. We checked these apps using different tools such as VirusTotal\footnote{https://www.virustotal.com} and Elish et al. [29]. All these tools indicate that these apps are benign.

4.3.1 ICC Analysis on Benign Apps

For each app, we construct the CAIMap to capture all types of communication that the app performs. In particular, we need to find whether the app is interacting with other apps or not. We found that 2,230 apps (84.4\%) out of 2,644 apps perform external ICC with other third party or built-in apps by either using implicit or explicit Intent. Table 4.1 summarizes our ICC analysis on 2,644 free popular apps.

For the apps with the external ICC (84.4\%), 298 apps (11.3\%) use explicit Intent ICC. In particular, these apps send external ICC to other third party apps by specifying explicitly the name of the target component. On the other hand, 1,932 apps (73.1\%) use implicit Intent ICC (not specifying the target component) for communicating with other third party or built-in apps.

Actions used in implicit Intent. Based on our ICC analysis of 2,644 apps, we found that the most two frequent actions used for external implicit ICC are android.intent.action.VIEW and android.intent.action.Send. In particular, 1,870 apps (70.7\%) use android.intent.action.VIEW and 943 apps (35.7\%) use android.intent.action.Send. The android.intent.action.VIEW is a generic standard Android action used to display the data to the user. For example, it can be used with a URL to view a webpage on the browser and can be used with a telephone number to display on the dialer. The android.intent.action.Send is a standard Android action used to send data to an entity. For example, it can be used to send data to an email app installed on the device. These standard actions are mostly used to communicate with the built-in apps. Table 4.1 shows the list of standard Android actions, we found during our analysis, used for the implicit Intent ICC.

Possible attacks through implicit Intent. Permissions are used to restrict access to compo-
nents and they are declared in the manifest file. If a component is protected by a permission, it can be only accessed by applications that have acquired that permission. The permission checking is enforced by the OS at runtime. However, if a component is not protected by a permission, then it is vulnerable to be accessed by any applications. For instance, Android built-in apps such as the browser app are not protected by a permission and any app can communicate with it. This gives an opportunity for the attacker to write a malicious app that can collude with the Android built-in app to perform malicious attack without this built-in app knows that it is colluding with malicious app. This can be done by sending an implicit Intent to the built-in app. For example, a malicious app can send an implicit Intent using this action `android.media.action.IMAGE_CAPTURE`, which is a standard action to have the built-in camera app captures a picture and returns it to the caller app. Based on our ICC analysis on 2,644 apps, we found 231 apps send this type of action. Another example is to have a malicious app communicate with the built-in phone app by sending the action `android.intent.action.CALL` with a specified number to make premium phone calls and generate revenue.

### 4.3.2 ICC Classification Using Existing Tool

XManDroid [13] is a runtime monitoring of communication links between apps, and it defines communication classification policies based on certain permissions combinations. XManDroid suffers from high false positives as it is acknowledged by its authors. We confirmed this property by evaluating a set of XManDroid’s policies on randomly selected 20 benign apps pairs. We found these 20 benign apps pairs using our CAIMap analysis. Each pair uses direct explicit Intent ICC channels for the communication between the apps. We found that 11 out of 20 benign pairs of apps are misclassified as collusion according to XManDroid’s policies, a very high false positive rate (55%). Figure 4.2 shows the number of the 20 benign apps pairs that trigger alerts per XManDroid’s policy.

The empirical results indicate that many Android apps are interacting with other third party or built-in apps through ICC channels. This ICC-based communication provides a potential opportunity for the attackers to develop malicious colluding apps by utilizing the ICC APIs in a similar way as in the communication between the benign apps. Therefore, it is challenging to develop a detection technique that can differentiate well between non-malicious ICC and malicious ICC. The reason is that most of the benign apps are interacting
with other apps according to our empirical results, which makes it challenging to develop a detection mechanism.

![Bar chart showing the number of apps pairs that trigger alerts per XManDroid’s policy.](image)

Figure 4.2: Histogram showing the number of the 20 benign apps pairs that trigger alerts per XManDroid’s policy [13]. Some apps pairs trigger alerts for more than one policy. XManDroid’s policy can be found in [13].

### 4.3.3 Summary of Experimental Findings

The results of this study indicate that many Android apps are interacting with other third party or built-in apps through ICC. We summarize our major analysis findings.

1. This study describes precisely whether the app has inter-application or intra-application communication. We observed that 414 apps (15.6%) have intra-application communication, and 2,230 apps (84.4%) have inter-application communication by either using implicit (1,932 apps (73.1%)) or explicit (298 apps (11.3%)) Intent.

2. We observed that most of the benign apps (84.3%) interact with third party or built-in apps via ICC APIs. This provides a potential opportunity for the attackers to develop malicious colluding apps by utilizing the ICC APIs in similar way as in the communication between the benign apps.

3. We evaluated the XManDroid’s policies on randomly selected 20 benign pairs of communicating apps. We found that these polices are strict to enforce and they produce...
many false alarms (false positive 55%).

4. It is challenging to develop a detection technique that can differentiate between non-malicious ICC and malicious ICC. The reason is that most of the benign apps are interacting with other apps according to our experimental results, which makes it challenging to come up with a detection mechanism.

4.4 Lessons Learned

The above experiments show the challenges in distinguishing benign inter-app ICC from colluding inter-app ICC in practice, since most of the apps perform external communications. In this work, we proposed CAIMap to statically characterize the inter-app ICC channels among the Android apps. This CAIMap does not provide a solution for apps collusion detection but it helps to identify pair or group of communicating apps. Thus, one can utilize this CAIMap to first identify the communicating apps. Then, a set of security policies can be defined and applied to differentiate between benign communicating apps and colluding ones.

We argue the need for in-depth static flow analysis (e.g., definition-use relations) in both source and destination apps for collusion detection. The key is the discovery of the context associated with benign ICC flows. The discovery requires new static program analysis algorithms and data structures. Static analysis-based solution provides complete analysis coverage, and scalable to analyze large number of apps. Admittedly, static analysis-based solution in general is affected by obfuscated code and can not handle dynamic code loading. However, we argue that any successful collusion detection technique should be comprehensive, scalable, and not limited by the device’s resources constraints. Hence, we advocate the use of static analysis-based solution for collusion detection because of the scalability and completeness provided by the static analysis.

4.4.1 Non-Sensitive Inter-App ICC

One category of false alerts is due to the non-sensitive ICC, specifically the data or action involved in a benign inter-app ICC call does not pose any security concerns. For example,
the current time and date information (\texttt{getTime()} and \texttt{getDate()}) involved in ICC call makes it non-sensitive ICC.

There may be multiple ICC calls between two apps. The sensitivity of each ICC call may differ substantially and needs to be analyzed case-by-case. Permission-based policies cannot achieve this granularity. We illustrate this point in Figure 4.3. Inferring the ICC sensitivity requires in-depth data-flow analysis in both apps, detailing how the data is created, modified, and consumed. Policies based on such an analysis will be more fine-grained than permission-based policies, reducing false alerts on benign ICCs.

![Figure 4.3: ICC involving non-sensitive data or request should not be alerted, despite of the sensitive permission combination in manifest files (namely, ACCESS_FINE_, LOCATION and INTERNET).](image)

4.4.2 Benign Inter-App ICC with User Trigger

All ICC flows that involve sensitive data need to be inspected. Existing success in single-app malware detection can be borrowed to design policies for collusion detection, in particular how data-flow and control-flow behavioral features are chosen. Predictive features are unique in either benign or malicious apps.

For example, new policies may be based on how much the app actively involves user in implicitly authorizing sensitive API invocations, including inter-app ICC calls. Existing work on single-app classification showed that statically extracted features on user-trigger dependence, i.e., the degree of sensitive API calls having def-use dependence relations on user inputs\(^2\), is very effective [29, 30, 84]. Benign single apps have a high degree of user-trigger dependence, whereas malware – often performing activities under stealth mode – does

\(^2\)User inputs may be onClick(), onItemClick(), etc.
not. It is conceivable to write similar user-trigger dependence policies for collusion detection, as illustrated in Figure 4.4. Such a policy is under the hypothesis that benign ICC flows are intended and initiated by the users (at the source app), whereas colluding ones are not.

![Diagram](image)

Figure 4.4: We envision an in-depth data-flow analysis in both source and destination apps that captures the context associated with the ICC. For classification, the presence of a valid user trigger in the source app may serve as one of the features that help rule out benign cases.

Admittedly, all policies have their limitations and may be bypassed by sophisticated malware writers. Yet, the advantages of fine-grained policies are twofold: making the tool more usable by reducing false alerts, and creating bigger obstacles for malware to bypass.

### 4.5 Summary

In this work, we argue that it is very challenging to detect malware collusion. We demonstrate the challenges through experimental evidence. The experimental results indicate that many Android apps communicate with other apps through ICC channels, which makes it challenging to develop a detection mechanism without generating many false alerts as in XManDroid [13]. None of the existing solutions provides a complete solution against app collusion attack. An effective solution should be scalable to a large number of apps, and define policies for classification that minimize false alerts.
Table 4.1: Summary of ICC analysis on 2,644 benign apps. Implicit and explicit Intent ICC are used for communication between apps.

<table>
<thead>
<tr>
<th>Action Used in External Implicit Intent ICC</th>
<th># of Apps (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>android.intent.action.VIEW</td>
<td>1870 (70.7%)</td>
</tr>
<tr>
<td>android.intent.action.SEND</td>
<td>943 (35.7%)</td>
</tr>
<tr>
<td>android.intent.action.DIAL</td>
<td>399 (15.1%)</td>
</tr>
<tr>
<td>android.intent.action.GET_CONTENT</td>
<td>275 (10.4%)</td>
</tr>
<tr>
<td>android.media.action.IMAGE_CAPTURE</td>
<td>231 (8.7%)</td>
</tr>
<tr>
<td>android.intent.action.CALL</td>
<td>158 (6.0%)</td>
</tr>
<tr>
<td>android.intent.action.PICK</td>
<td>139 (5.3%)</td>
</tr>
<tr>
<td>android.intent.action.SENDTO</td>
<td>122 (4.6%)</td>
</tr>
<tr>
<td>android.media.action.VIDEO_CAPTURE</td>
<td>62 (2.3%)</td>
</tr>
<tr>
<td>android.intent.action.DELETE</td>
<td>53 (2.0%)</td>
</tr>
<tr>
<td>android.intent.action.EDIT</td>
<td>48 (1.8%)</td>
</tr>
<tr>
<td>android.speech.action.RECOGNIZE_SPEECH</td>
<td>45 (1.7%)</td>
</tr>
<tr>
<td>android.intent.action.MEDIA_MOUNTED</td>
<td>42 (1.6%)</td>
</tr>
<tr>
<td>android.intent.action.INSERT</td>
<td>33 (1.2%)</td>
</tr>
<tr>
<td>android.intent.action.SEARCH</td>
<td>20 (0.8%)</td>
</tr>
<tr>
<td>android.intent.action.RINGTONE_PICKER</td>
<td>19 (0.7%)</td>
</tr>
<tr>
<td>android.intent.action.WEB_SEARCH</td>
<td>12 (0.5%)</td>
</tr>
<tr>
<td>android.intent.action.SYNC</td>
<td>3 (0.1%)</td>
</tr>
<tr>
<td>android.intent.action.ANSWER</td>
<td>2 (0.1%)</td>
</tr>
</tbody>
</table>

# of apps with external implicit Intent ICC 1932 (73.1%)

# of apps with external explicit Intent ICC 298 (11.3%)

Total # of apps with external ICC 2230 (84.4%)

Total # of apps with Internal ICC 414 (15.6%)
Chapter 5

Inter-App ICC Analysis for App Collusion and Vulnerability Detection

In the previous chapter, we experimentally demonstrate the technical challenges in distinguishing benign inter-app ICC from colluding inter-app ICC in practice. In this chapter, we present our approach to address these challenges. Specifically, we introduce a scalable and deep cross-app static flow analysis approach to statically analyze the sensitivity and the context of each inter-app ICC. In addition, we define more fine-grained security policies for the inter-app ICC risk classification.

5.1 Malware Evolution and Attack Model

Malware collusion is a new malware generation attack that is very challenging to detect under the existing conventional screening techniques. A collusion attack occurs when malicious applications, likely written by the same adversary, collaborate to gain a set of permissions to perform malicious tasks. In malware collusion, each colluding malware only needs to perform a certain functionality, which may make it appear benign to evade the conventional detection tools. Hence, malware writers have strong motivation to write colluding malware to evade standard detection. Figure 5.1 depicts the application collusion threat model through Intent-based inter-component communication (ICC). There are two main categories of malware abuse through colluding applications:
• **Collusion for data leak**: app X has the access to some sensitive data (e.g., contact information), but app X does not request the capability to access the network, in order to appear innocuous to the conventional stand-alone app screening. App Y (written by the same malware author) has the permission to access the network, but not the access to sensitive data. App X asks app Y to send the sensitive data to the remote attacker through ICC.

• **Collusion for system abuse**: apps X and Y work together to send spam SMS messages. App X prepares the spam SMS messages, and requests app Y to send them. App X does not need to request permission for sending SMS.

![Application collusion threat model via ICC.](image)

Figure 5.1: Application collusion threat model via ICC.

Figure 5.2 illustrates that with malware collusion, privileged Android APIs necessary for completing attacks can be distributed into multiple applications. The colluding apps communicate directly or indirectly with each other in order to combine their privileges and perform malicious tasks that threaten the data privacy or system integrity.

Computation in many existing app scoring systems is based on properties of how privileged operations are used in a single app, e.g., [5, 9, 29, 49, 69]. Distributing privileges across multiple apps results in apps appearing less risky, substantially weakening the power of these detection systems.

The chain of the colluding apps can be more than two apps as depicted in Figure 5.3. Colluding apps also may communicate indirectly, e.g., through shared files, or through covert channels as demonstrated in [62]. Marforio et al. [62] thoroughly measured the efficiency of
Figure 5.2: An example of permissions and operations being split between colluding applications.

different overt and covert channels for apps collusion, but their work does not provide any concrete defense mechanisms. Detecting covert-channel based malware collusion remains an open problem.

Figure 5.3: A schematic drawing illustrating the complexity of the inter-app ICC mesh between multiple apps. Individual apps have access to sensitive data and/or perform critical operations. The apps collusion chain can be more than two apps.

**Opportunistic Collusion.** In this work, we focus on analyzing inter-app communication through *explicit* Intent ICC where the target component/app is explicitly specified. In principle, apps collusion can also be achieved through inter-app *implicit* Intent ICC. However, the collusion may not always succeed since it depends on the other apps installed on the
mobile device and what are the actions they can handle. We define opportunistic collusion as two or more apps communicating with each other through implicit Intent ICC channels to perform malicious tasks. A recent study shows that around 73% of 2,644 Android apps communicate with other apps using implicit Intent ICC [31]. The attacker may use this opportunity to develop colluding malware apps which communicate using implicit Intent ICC channels. However, the collusion may not always work in this case since there might be another app which is declared to handle the same action string android.action.xxx. Thus, there is a chance that this implicit Intent will be wrongly delivered to a different app not the one intended by the attacker in her/his collusion scenario. We plan to investigate inter-app implicit Intent ICC in future work.

5.2 Applications of Inter-App ICC Analysis

Our inter-app ICC analysis can be used for a number of security analyses. In particular, we discuss how our inter-app ICC analysis can be useful for applications collusion analysis and applications vulnerability analysis.

Applications Collusion Analysis. The apps collusion attack occurs when malicious apps, likely written by the same adversary, collaborate to gain a set of permissions to perform malicious operations. Soundcomber [76] is an example of colluding malware, where one app records the audio during the call, while the second app has access to the Internet to leak the audio records to the attacker. The focus of this work is to show how our inter-app ICC analysis is used to detect sensitive inter-app ICC-based flows to identify risky apps pairs.

Applications Vulnerability Analysis. In the confused deputy attack, a malicious app exploits vulnerable component interfaces of benign applications. Recent studies show that the vulnerabilities exist in third party applications and Android built-in applications [38]. Our inter-app ICC analysis provides more precise pairwise vulnerability analysis than individual state-of-the-art solutions, i.e., ComDroid [17] and Epicc [66]. These solutions assume that any unprotected public component is vulnerable regardless if there is a path from the public component to the critical operations or not. However, we perform in-depth dataflow analysis across multiple apps and inside the public vulnerable components to check for a path’s existence of such a path. Hence, this will reduce the false alarms when no path exists.
5.3 Colluding Bot Scenarios

Several Android botnets were recently discovered and reported, e.g., MisoSMS, SpamSoldier, and Android.Bmaster. (These botnets are not collusion-based.) We demonstrate how applications collusion setup can be leveraged by Android botnets to abuse system resources and leak private data.

Figure 5.4 illustrates the colluding bot architecture. The malicious app \( X \) is responsible for communicating with the bot master, whereas other apps \((Y \text{ and } Z)\) are for executing the commands and launching attacks. For example, app \( X \) receives a command from the command and control server (C&C) to send spam SMS to all contacts on the user’s phone. As \( X \) itself does not have the capability to send SMS, it forwards this command to another app \( Y \), which has the necessary capability. The request is sent through an ICC channel as an Android Intent.

Similarly, app \( X \) may receive another command from C&C server to leak the user’s photos on her/his device. \( X \) forwards this command to another app \( Z \), which is a PhotoManager app to leak the photos.

**Figure 5.4:** A schematic drawing illustrating how collusion setup can be used by botnet.

**WebView-based Colluding Bot Scenario.** We show a specific WebView-based attack that allows colluding bots to leak private data. Specifically, app \( X \) sends malicious URL to another app (PhotoManager) to display malicious web page which inserts malicious JavaScript code to leak the user’s photos. The purpose of demonstrating WebView-based bots is to show the versatility in colluding bot’s capabilities, in particular the ease of accessing dynamic code from the Internet, as well as uploading stolen data.

WebView is a component used to display web pages. It allows Android apps to embed a
web browser to display web contents and interact with web servers. The WebView component is used pervasively; 86% of the studied top 20 most downloaded Android apps use WebView component as part of the application [61]. WebView provides a mechanism to enable the JavaScript code to invoke Android app’s Java code. This is done by the `addJavascriptInterface` API which allows to register Java objects to WebView and all the objects’ public methods can be called by the JavaScript code.

The objective of attacker is to compromise an app which has WebView component by injecting malicious JavaScript code. To achieve this, the attacker needs to send malicious URL to that app to load malicious web page into the app, and then launch the attack. Figure 5.5 depicts an overview of this attack model. The code snippet for this code injection collusion is shown in Figure 5.6.

In this example, the app X receives a command from C&C server to leak the user’s photos (step 1). Then, it creates an implicit Intent with `android.intent.action.VIEW` and includes malicious URL as a data in the Intent. After that, it makes an ICC call `startActivity()` which will be handled by the `WebViewActivity` component in PhotoManager app (step 2), since this component can handle `android.intent.action.VIEW`. The `WebViewActivity` has a `WebView` component which has one Java object registered ”PUtils”, its public methods are shown in Figure 5.7.

`WebViewActivity` component in PhotoManager app receives the malicious URL from the malicious app, the `WebView` will load the URL which has the malicious JavaScript code (step 3). This JavaScript code can use PUtils object to call the PhotoUtils class’s methods. Specifically, the malicious JavaScript code in the malicious web page can be executed to upload the user’s photos to the malicious server through PUtils (step 4) as shown in Figure 5.8.
// malicious app
...
// create an intent with action view
Intent i = new Intent("android.intent.action.VIEW");
// add malicious URL
i.putExtra("url", "http://maliciouswebpage.html");
// start the intent
startActivity(i);

// WebViewActivity class in PhotoManager app
...
wv.getSettings().setJavaScriptEnabled(true);
wv.addJavascriptInterface(new PhotoUtils(this), "PUtils");
// get url from the intent
Intent i = getIntent();
String url = i.getStringExtra("url");
// load the webpage
wv.loadUrl(url);

Figure 5.6: Code snippet for the malicious code injection by the colluding bots. The malicious app (on the top) and the PhotoManager app (on the bottom).

// The PhotoUtils class has the following methods:
public void uploadPhoto(String path, String url) {
    // upload photo to the server
    ... 
    HttpPost httppost = new HttpPost(url);
    File file = new File(path);
    FileBody fBody = new FileBody(file);
    MultipartEntity reqEntity = new MultipartEntity(HttpMultipartMode.BROWSER_COMPATIBLE);
    reqEntity.addPart("photo", fBody);
    httppost.setEntity(reqEntity);
    HttpResponse res = httpclient.execute(httppost);
    ... 
}
public void deletePhoto(String path) {}
public void sharePhoto(String path) {}
... 

Figure 5.7: Code snippet for the PhotoUtils class.

<script type="text/javascript">
    PUtils.uploadPhoto("/sdcard/DCIM/* .jpg", "http://attackerserver");
</script>

Figure 5.8: Code snippet for the injected malicious JavaScript.
5.4 Cross-App Dataflow Analysis

In this section, we present our technique used to perform cross-app flow analysis. Figure 5.9 illustrates our inter-app ICC analysis workflow. The preprocessing steps follow standard data and Android-specific control-flow analysis as in [29]. Our new main contribution is to introduce and perform CAIMap analysis to identify possible target apps that communicate with a given app. Moreover, we match ICC across the apps by connecting the ICC exit points in the source app with the ICC entry points in the target apps based on CAIMap.

We next give a detailed description of our approach. The pseudocode of our procedure is shown in Algorithm 3.

Figure 5.9: Our inter-app ICC analysis workflow. The preprocessing step is performed only once for each (source and target) app.
Algorithm 3 Inter-App ICC Risk Analysis

Input: $A \leftarrow \{\text{source app's bytecode and Manifest.xml}\}$

$B \leftarrow \{\text{destination app's bytecode and Manifest.xml}\}$

CAIMap set $S = \{<\text{ICCName}_k, \text{sourceComponent}_k, \text{targetComponent}_k, \text{typeOfCommunication}_k>\}$

Output: risk \_level as high, medium, low

1: $\text{Src}_\text{App}_\text{HashMap} \leftarrow \emptyset$ // where key represents source component name, and value represents a list $\{\text{ICC name, sensitive data, UI trigger}\}$

2: $\text{Dest}_\text{App}_\text{HashMap} \leftarrow \emptyset$ // where key represents target component name, and value represents a list $\{\text{entry point, comProtection, critical operations}\}$

3: $\text{ICCList} \leftarrow \text{scan source app's bytecode for ICC APIs}$

4: $G_{\text{Src}} \leftarrow \text{ConstructDataDependenceGraph}(A)$

5: $\text{srcAppAnalysis}(\text{ICCList}, G_{\text{Src}}, \text{Src}_\text{App}_\text{HashMap})$

6: $\text{TargetComps} \leftarrow \text{extract components name from destination app's Manifest.xml}$

7: $G_{\text{Dest}} \leftarrow \text{ConstructDataDependenceGraph}(B)$

8: $\text{destAppAnalysis}(\text{TargetComps}, G_{\text{Dest}}, \text{Dest}_\text{App}_\text{HashMap})$

9: $\text{risk}_\text{level} \leftarrow \text{pairAnalysis}(S, \text{Src}_\text{App}_\text{HashMap}, \text{Dest}_\text{App}_\text{HashMap})$

10: return $\text{risk}_\text{level}$

11: procedure $\text{pairAnalysis}(S, \text{Src}_\text{App}_\text{HashMap}, \text{Dest}_\text{App}_\text{HashMap})$

12: for each $\text{mapEdge} \in S$ do

13: $\text{srcComp} \leftarrow \text{get srcComp from mapEdge}$

14: $\text{targetComp} \leftarrow \text{get targetComp from mapEdge}$

15: $\text{ICC}_\text{exit} \leftarrow \text{search Src}_\text{App}_\text{HashMap by srcComp}$

16: $\text{ICC}_\text{entry} \leftarrow \text{search Dest}_\text{App}_\text{HashMap by targetComp}$

17: connect $\text{ICC}_\text{exit}$ in $G_{\text{Src}}$ with $\text{ICC}_\text{entry}$ in $G_{\text{Dest}}$

18: end for

19: $\text{risk}_\text{level} \leftarrow \text{check our security policy}$

20: return $\text{risk}_\text{level}$

21: end procedure

22: procedure $\text{srcAppAnalysis}(\text{ICCList}, G_{\text{Src}}, \text{Src}_\text{App}_\text{HashMap})$

23: for each $\text{icc} \in \text{ICCList}$ do

24: perform backward depth-first traversal from $\text{icc}$ to find any sensitive data and/or UI trigger on $G_{\text{Src}}$

25: $\text{Src}_\text{App}_\text{HashMap} \leftarrow \text{Src}_\text{App}_\text{HashMap} \cup \{<\text{srcCompName}, \{\text{icc, sensitiveData, UITrigger}\}>\}$

26: end for

27: return $\text{Src}_\text{App}_\text{HashMap}$

28: end procedure

29: procedure $\text{destAppAnalysis}(\text{TargetComps}, G_{\text{Dest}}, \text{Dest}_\text{App}_\text{HashMap})$

30: for each $\text{comp} \in \text{TargetComps}$ do

31: perform forward depth-first traversal from comp's entry point to find any critical operations on $G_{\text{Dest}}$

32: $\text{comProtection} \leftarrow \text{check comp protection}$

33: $\text{Dest}_\text{App}_\text{HashMap} \leftarrow \text{Dest}_\text{App}_\text{HashMap} \cup \{<\text{comp}, \{\text{entryPoint, comProtection, criticalOperations}\}>\}$

34: end for

35: return $\text{Dest}_\text{App}_\text{HashMap}$

36: end procedure
5.4.1 Extract ICC Exit and Entry Points

**Hash map data structure.** We use hash map data structure to store some information related to ICC exit and entry points. In particular, we use two different hash maps one for exit points and one for entry points, namely, SourceAppICCExitHashMap and TargetAppICCEntryHashMap. Each hash map has multiple entries. Each entry consists of ComponentName as a key, and a list of values. We next describe how ICC exit and entry points with their related information are extracted and stored in our hash maps.

**ICC exit points.** We first scan the source app’s bytecode to identify the ICC exit points, user trigger, and sensitive data. The ICC exit points include all Intent-based ICC APIs such as startActivity(Intent i), startService(Intent i), and sendBroadcast(Intent i). User trigger refers to a user’s input or action/event on the app. For example, the user’s input may be text entered via a text field, while the user’s action/event is any click on UI element, such as a button. Relevant API calls in UI objects that return a user’s input value or listen to user’s action/event are defined as triggers. The sensitive data refers to the APIs that retrieve private data, such as getAccounts() and getPassword(...) as shown in Table 5.1.

Then, we construct the data dependence graph (DDG) for the source app. We design our dataflow analysis, which is augmented with Android specific control-flow analysis for handling Android Intent to construct the data dependence graph of intra- and inter-procedural dependences.

Once the data dependence graph is constructed for the source app, we perform backward depth-first traversal on DDG from each ICC exit point to check if it involves sensitive data or user trigger and to store this information in SourceAppICCExitHashMap.

This hash map has multiple entries. Each entry consists of SrcComponentName as a key, and a list {ICCExitName, SensitiveData, UserTrigger} as a value. For example, <compX, {startService(Intent i), getDeviceID(), onClick()}> represents one entry in the SourceAppICCExitHashMap, where compX is the component name that initiates the inter-app ICC call startService(Intent i) with sensitive data device ID included as part of the Intent of this call and onClick() as the user event to trigger this call.

**ICC entry points.** We scan the target app’s bytecode to identify the ICC entry points and critical operations. The ICC entry points include all components’ entry points such as onStart() and onCreate(...). Critical operation is an API call which refers to a function
call providing system service such as network operations, file operations, telephony services in the app (examples are shown in Table 5.1). Additionally, we analyze the target app’s manifest.xml file to get the information about the protection of each component, i.e., the permission(s) defined to access the component.

We construct the data dependence graph (DDG) for the target app and perform \textit{forward depth-first traversal} on its DDG from each ICC entry point to find any critical operations and store this information in \texttt{TargetAppICCEntryHashMap}.

This hash map has multiple entries and each entry consists of \texttt{TrgComponentName} as a key, and a list \{\texttt{ICCEntryName}, \texttt{CompProtection}, \texttt{CriticalOperations}\} as a value. For example, \texttt{<compY, \{onStart(), No, java.io.FileOutputStream.write(...)\}>} represents one entry in the \texttt{TargetAppICCEntryHashMap}, where \texttt{compY} is the component name that receives the inter-app ICC call, and \texttt{onStart()} is the entry point of \texttt{compY} which is not protected (\texttt{No}) and has critical operation \texttt{java.io.FileOutputStream.write(...)}.

Table 5.1: Examples of sensitive data and critical operations in our study. We consider APIs that retrieve private data as sensitives data, and APIs that consume the private data or perform critical functions as critical operations.

<table>
<thead>
<tr>
<th>Category</th>
<th>Examples of APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sensitive Data</strong></td>
<td></td>
</tr>
<tr>
<td>User Input</td>
<td>android.widget.EditText.getText()</td>
</tr>
<tr>
<td>User’s Account Info</td>
<td>android.accounts.AccountManager.getAccounts(), getPassword(...)</td>
</tr>
<tr>
<td>Location Info</td>
<td>android.location.Location.getLatitude(), getLongitude()</td>
</tr>
<tr>
<td>Phone Info</td>
<td>android.telephony.TelephonyManager.getDeviceId(), getSimSerialNumber()</td>
</tr>
<tr>
<td>Network Info</td>
<td>android.net.wifi.WifiInfo.getMacAddress(), getIpAddress()</td>
</tr>
<tr>
<td>Web Browser Info</td>
<td>android.provider.Browser.getAllBookmarks(...), getAllVisitedUrls(...)</td>
</tr>
<tr>
<td>File Data</td>
<td>java.io.FileInputStream.read(), java.io.BufferedReader.readLine()</td>
</tr>
<tr>
<td>Content Provider Data</td>
<td>android.content.ContentResolver.query(...)</td>
</tr>
<tr>
<td><strong>Critical Operations</strong></td>
<td></td>
</tr>
<tr>
<td>Network</td>
<td>java.net.DatagramSocket.send(...), android.net.http.AndroidHttpClient.execute(...)</td>
</tr>
<tr>
<td>File</td>
<td>java.io.FileOutputStream.write(...), java.io.File.delete()</td>
</tr>
<tr>
<td>Content Provider</td>
<td>android.content.ContentResolver.delete(...), update(...)</td>
</tr>
<tr>
<td>System Commands</td>
<td>java.lang.Runtime.exec(...)</td>
</tr>
<tr>
<td>Telephony Services</td>
<td>android.telephony.SmsManager.sendTextMessage(...), sendDataMessage(...)</td>
</tr>
<tr>
<td>Dynamic Code Loading</td>
<td>java.lang.ClassLoader.loadClass(...)</td>
</tr>
</tbody>
</table>
5.4.2 Matching Inter-App ICC Calls

Given the CAIMap information constructed in step 1, and the two hash maps (SourceAppICC ExitHashMap and TargetAppICCEntryHashMap), we connect the inter-app ICC calls as follows. First, for each edge $e$ in the CAIMap, we obtain the names of its source and target components. Then, we search in SourceAppICCExitHashMap for the source component name. The search results return its information (ICCExitName, SensitiveData, UserTrigger). We search the same with the target component in TargetAppICCEntryHashMap instead to get its information (ICCEntryName, CompProtection, CriticalOperations). After that, we connect the ICC exit point in the source component with its corresponding ICC entry point in the target component. This operation provides the complete path of the ICC calls from the source to the destination across multiple apps. Next, we describe our security policies used to classify the sensitivity of the inter-app ICC calls.

5.4.3 Risk Classification Methodology

Our classification decisions are made based on inspecting the sensitivity of each inter-app ICC call. We define our policies based on four properties of the ICC flow that impact the inter-app ICC risk level, as illustrated in Figure 5.10.

- **User trigger validation.** One of the properties is based on how much the app actively involves user in implicitly authorizing inter-app ICC calls. Existing work on single-app classification showed that statically extracted features on user-trigger dependence, i.e., the degree of sensitive API calls having def-use dependence relations on user inputs, was shown effective (e.g., [29, 84]). User inputs may be onClick(), onItemClick(), etc. Benign single apps have a high degree of user-trigger dependence, whereas malware – often performing activities under stealth mode – does not. Hence, one of our fine-grained policies is that benign inter-app ICC calls are intended and initiated by the users (at the source app), whereas colluding ones are not.

- **Access to sensitive data.** Another property is based on whether the inter-app ICC call involves transmitting any sensitive data (e.g., getPassword(…)).

- **Permission checking.** We check if the target component (at the target app) is protected by permission(s) or not. This analysis is important because the attacker
may exploit vulnerable target app (no protection) as part of her/his apps collusion attack.

• **Critical operation.** The last property is based on whether the inter-app ICC call involves critical operation (e.g., network operations, file operations) in the target component.

In the next section, we give a detailed description of our risk classification policies for the inter-app ICC call and our experimental evaluation.

![Diagram](image)

**Figure 5.10:** Our four properties of the inter-app ICC flow used in our classification policies.

### 5.5 Experimental Evaluation

This section presents the experimental results that characterize the effectiveness of our approach in classifying external explicit Intent ICCs. In particular, the objective of our evaluation is to answer the following questions:

1. How many communicating apps pairs under study have risky inter-app communication? (Section 5.5.3)

2. How effective is the existing permission-based policy solution in terms of false positive rate? (Section 5.5.4)

3. Can our approach reduce the number of false alerts generated by the permission-based policy solution? (Section 5.5.4)
5.5.1 Experiment Setup

We performed an evaluation with 2,644 free popular real-world Android apps from Google Play store. These apps covering various application categories and different levels of popularity as determined by the user rating scale. We checked these apps using tools such as VirusTotal\(^1\) and Elish et al. [29], and these tools label the apps as benign.

We implemented a static analysis tool in Java to construct the CAIMap for the Android apps in order to identify the communicating apps pairs. Once we identified the apps pair, we performed our cross-app dataflow analysis. Our analysis can run on Java bytecode or source code. We used Dare tool [65] to translate Dalvik bytecode to Java bytecode. The manifest file is extracted using apktool [2].

5.5.2 Risk Classification Policies

Our classification is based on rules which identify the risk level associated with the inter-app ICC call. Figure 5.11 illustrates high level overview of our security policies which can be defined as follows:

Rule 1 Suppose component \(C_1\) in app \(P_1\) calls component \(C_2\) in app \(P_2\), i.e., \(C_1 \rightarrow C_2\). If the ICC exit point in \(C_1\) does not have a valid user trigger and the target component \(C_2\) is not protected by permission checking and has critical operation, then this ICC channel is classified as a high risk inter-app ICC channel.

Rule 2 Suppose component \(C_1\) in app \(P_1\) calls component \(C_2\) in app \(P_2\), i.e., \(C_1 \rightarrow C_2\). If the ICC exit point in \(C_1\) has a valid user trigger and the target component \(C_2\) is not protected by permission checking and has critical operation, then this ICC channel is classified as a medium risk inter-app ICC channel.

Rule 3 Suppose component \(C_1\) in app \(P_1\) calls component \(C_2\) in app \(P_2\), i.e., \(C_1 \rightarrow C_2\). If the ICC exit point in \(C_1\) does not have a valid user trigger and the target component \(C_2\) is protected by permission checking and has critical operation, then this ICC channel is classified as a medium risk inter-app ICC channel.

---
\(^1\)https://www.virustotal.com/
Rule 4 Suppose component $C_1$ in app $P_1$ calls component $C_2$ in app $P_2$, i.e., $C_1 \to C_2$. If the ICC exit point in $C_1$ has a valid user trigger and the target component $C_2$ is protected by permission checking and has critical operation, then this ICC channel is classified as a low (no) risk inter-app ICC channel.

![Diagram of inter-app ICC risk level]

Figure 5.11: High level overview of inter-app ICC risk level according to our security policies.

In addition to the user trigger and component protection checking, we inspect the inter-app ICC call to determine if it involves sensitive data or critical operations. Therefore, we extend our classification policies to have 16 fine-grained policies summarized in Table 5.2.

Definition 6 The final classification decision of the apps pair is determined as follows:

- The apps pair is classified as high risk pair if there is one high risk inter-app ICC channel among the apps.
- The apps pair is classified as medium risk pair if there is one medium risk inter-app ICC channel and no high risk inter-app ICC channels among the apps.
- The apps pair is classified as no risk pair if none of the inter-app ICC channels among the apps is classified as high or medium risk.
Admittedly, all policies have their limitations and may be bypassed by sophisticated malware writers. Yet, the advantages of fine-grained policies are twofold: making the tool more usable by reducing the number of false alerts, and creating bigger obstacles for malware to bypass.

Table 5.2: Our risk classification policies for explicit Intent inter-app ICC.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Source Component / App</th>
<th>Destination Component / App</th>
<th>Risk Level</th>
<th>Possible Attack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>User Trigger</td>
<td>Access to Sensitive Data</td>
<td>Exposed Component*</td>
<td>Critical Operation</td>
</tr>
<tr>
<td>1</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>8</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>9</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>10</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>11</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>12</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>13</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>14</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>15</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>16</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

*Component is public and not protected by permission(s).

**Confused deputy: attacker may exploit vulnerable target app as part of her/his apps collusion attack.

5.5.3 Evaluation of Inter-App ICC Classification

We constructed the CAIMap for each of 2,644 apps. 175 of apps pairs communicate using direct explicit Intent ICC channels. We performed our analysis on these 175 apps pairs to evaluate the effectiveness of our explicit inter-app ICC risk classification policies.
Table 5.3 presents our classification results on 175 apps pairs according to our external explicit Intent ICC policies defined in Table 5.2. 31 apps pairs (17.7%) are classified as high risk by policy #3 defined in Table 5.2, i.e., they have high risk ICC channels. High risk ICC channel means that the external explicit Intent ICC is not initiated by the user trigger in the source app, and the target component in the target app is exposed, is not protected by permission(s) and has critical operation(s).

Moreover, 2 apps pairs (1.1%) have medium risk ICC channels according to our policy #13 defined in Table 5.2. Each pair in these 2 medium risk pairs has explicit inter-app ICCs which are initiated by the user triggers but the target components are not protected. Also, no sensitive data is involved in the inter-app ICCs among them. For 18.8% apps pairs classified as high and medium risk, we found that the explicit inter-app ICC calls initiate critical operations only in the target apps and no sensitive data involved as shown in Table 5.4.

Additionally, we found that the explicit inter-app ICCs between 142 apps pairs (81.2%) do not pose any security risk according to our policies. This means that either (i) the external explicit Intent ICCs are initiated by user triggers and target components are protected or (ii) no sensitive data or critical operations are involved in the external explicit Intent ICCs. Figure 5.12 illustrates visualization of the risk level associated with the explicit Intent inter-app ICC calls among the 175 apps pairs.

Table 5.3: Summary of explicit Intent inter-app ICC classification results on 175 apps pairs.

<table>
<thead>
<tr>
<th>Risk Level</th>
<th># of Apps Pairs (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>High Risk</td>
<td>31 (17.7%)</td>
</tr>
<tr>
<td>Medium Risk</td>
<td>2 (1.1%)</td>
</tr>
<tr>
<td>Low/No Risk</td>
<td>142 (81.2%)</td>
</tr>
<tr>
<td>Total # of Apps Pairs</td>
<td>175 (100%)</td>
</tr>
</tbody>
</table>

Case Study on a High Risk Apps Pair. We present a case study on pair of communicating apps that is classified as high risk pair because of the existence of high risk inter-app ICC channels among the apps. In this pair, the first app is called com.pdanet which allows the user to share the mobile device’s Internet connection with any device through USB, Bluetooth or WiFi Hotspot. The second app is called com.foxfi.addon which enables free WiFi Tether on the mobile devices. Two components, namely, com.foxfi.al and com.foxfi.aq in the com.pdanet app communicate with the Receiver component
Figure 5.12: Visualization of the explicit Intent inter-app ICC calls between the 175 apps pairs. Red nodes and edges represent high and medium risk inter-app ICC calls discovered using our security policies. Gray nodes and edges represent no risk inter-app ICC calls.
com.foxfi.addon.AddOnEvents in the second app com.foxfi.addon by sending explicit inter-app ICC calls using sendBroadcast(android.content.Intent). This inter-app ICC calls do not carry any sensitive data. However, the destination component com.foxfi.addon.AddOnEvents has many critical operations that are executed in response to the inter-app ICC call. The critical operations include java.io.InputStream.read(...) and java.io.OutputStream.write(...). The inter-app ICC calls are considered to be high risk according to our policies because they are initiated without user awareness (no user trigger) in the source app com.pdanet. Furthermore, the destination component com.foxfi.addon.AddOnEvents in the target app com.foxfi.addon is exposed and not protected by the permission(s). This example shows a risky apps pair that executes critical operations in a stealthy mode.

Figure 5.13 shows the types of explicit inter-app ICC for the 175 apps pairs. Most of the explicit inter-app ICC calls are related to Activity component, i.e, 166 out of 175 apps pairs use Activity-based inter-app ICC channels (e.g., startActivity(...)). 22 apps pairs use Service-based inter-app ICC channels, and 7 apps pairs use Receiver-based inter-app ICC channels. Some apps pairs use more than one type of explicit inter-app ICC call. Service and Receiver components perform operations in the background without the user awareness as opposed to the Activity component. Thus, we expect the malware developers to utilize Service and Receiver-based inter-app ICC channels for the apps collusion to hide the communication.

![Figure 5.13: Histogram showing the number of the 175 apps pairs per Activity, Service, and Receiver ICC. Some apps pairs use more than one type of external explicit Intent ICC.](image)

We present some statistics on the number of external explicit Intent ICC channels per apps
pair as shown in Figure 5.14. We found the majority of the apps pairs (50.3%) use only one explicit inter-app ICC channel for communication.

![Histogram showing the number of external explicit Intent ICC channels for 175 apps pairs. Majority of the apps pairs have only one external explicit Intent ICC per apps pair.](image)

**Figure 5.14**: Histogram showing the number of external explicit Intent ICC channels for 175 apps pairs. Majority of the apps pairs have only one external explicit Intent ICC per apps pair.

### 5.5.4 Comparison with Existing Solution

The purpose of this experiment is to demonstrate that it is feasible to design flow-based policies for accurate classification. We compare our classification policies with a well-known permission-based collusion detection solution (XManDroid) \[13\] for Android. XManDroid\(^2\) is a run-time monitoring of communication channels between the apps, and it defines communication classification policies based on certain permissions combinations. For example, an app with permission `READ_CONTACTS` must not communicate with an app that has permission `INTERNET`. Their proposed analysis is performed on the device in which one should consider strong adversary model.

We evaluated a set of permission-based policies \[13\] on our 175 apps pairs. Table 5.4 presents the classification results using permission-based policies \[13\] on 175 pairs of communicating apps. The permission policies classify 132 out of 175 apps pairs are classified as collusion according to permission-based policies. In other words, 75.4% of the apps pairs are classified 75.4% of the apps pairs are classified

\(^2\) XManDroid may suffer from a high number of false positives as acknowledged by its authors.
Table 5.4: A comparison between permission-based policies and our policies on explicit Intent inter-app ICCs. Out of 175 apps pairs, 132 apps pairs classified as collusion by the permission-based policies solution, and 33 apps pairs classified as high and medium risk by our work. Our analysis has 4 times fewer flags than the permission-based policies solution.

<table>
<thead>
<tr>
<th>Permission-based Policies</th>
<th>Our Work</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Apps Pairs</td>
<td># of Apps Pairs</td>
</tr>
<tr>
<td>Inter-App ICC with Sensitive Data Only</td>
<td>9</td>
</tr>
<tr>
<td>Inter-App ICC with Critical Operations Only</td>
<td>27</td>
</tr>
<tr>
<td>Inter-App ICC without Sensitive Data or Critical Operations</td>
<td>96</td>
</tr>
<tr>
<td>Inter-App ICC with Sensitive Data and Critical Operations</td>
<td>0</td>
</tr>
<tr>
<td>Total # of Apps Pairs</td>
<td>132</td>
</tr>
</tbody>
</table>

as collusion. Table 5.5 shows the number of the 175 apps pairs that trigger alerts per permission-based policies [13]. Some apps pairs trigger multiple alerts.

In order to investigate the correctness of the flags raised by permission-based policies and estimate their false positives, we analyzed each inter-app ICC channel in 132 apps pairs that are classified as collusion by XManDroid. We analyzed each inter-app ICC channel to check if it involves sensitive data, critical operations, or both.

For the 132 apps pairs classified as collusion by permission-based policies, we found that 9 apps pairs (5.1%) have inter-app ICC calls with sensitive data, and 27 apps pairs (15.4%) perform critical operations due to inter-app ICC calls. 96 apps pairs (54.9%) do not have sensitive inter-app ICC channels, i.e., no sensitive data or critical operations involved as shown in Table 5.4. Hence, we can infer that permission-based policies solution produces large number of false positives, i.e., 96 (54.9%) out of 132 apps pairs are misclassified as collusion. The main category of false positives is due to the non-sensitive inter-app ICCs, specifically no sensitive data or critical operations involved in a benign explicit inter-app ICC call which does not pose any security concerns.

Our classification rules are fine-grained. Our analysis produces 4 times fewer flags compared to the permission-based policies as shown in Table 5.4 and Figure 5.15.

Case Study on a Permission-based Policy False Positive. We present a case study on
Table 5.5: Summary of the number of the 175 apps pairs that trigger alerts per permission-based policies [13]. Some apps pairs trigger alerts for more than one policy.

<table>
<thead>
<tr>
<th>Permission-based Policy [13]</th>
<th># of Apps Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(8) A third party application with permission ACCESS_FINE_LOCATION must not communicate to a third party application that has permission INTERNET</td>
<td>126</td>
</tr>
<tr>
<td>(9) A third party application that has permission READ_CONTACTS must not communicate to a third party application that has permission INTERNET</td>
<td>122</td>
</tr>
<tr>
<td>(10) A third party application that has permission READ_SMS must not communicate to a third party application that has permission INTERNET</td>
<td>113</td>
</tr>
<tr>
<td>(11) A third party application that has permissions RECORD_AUDIO and PHONE_STATE or PROCESS_OUTGOING_CALLS must not directly or indirectly communicate to a third party application with permission INTERNET</td>
<td>115</td>
</tr>
</tbody>
</table>

pair of communicating apps that is misclassified as colluding apps according to permission-based policies solution. In this case study, one app `com.projectx.android.ScouterLite` is used for fun to measure the power level of the people and save pictures. The other app `com.cooliris.media.Gallery` is a gallery app used to show pictures taken from the camera app. These two apps communicate using an explicit inter-app ICC call `startActivity(android.content.Intent)`. We checked the sensitivity of the inter-app ICC call and found that no sensitive data or critical operations are involved. However, this apps pair is classified as collusion by permission-based policies because it violates policy 8 in XManDroid, i.e., the app (`com.cooliris.media.Gallery`) with permission ACCESS_FINE_LOCATION must not communicate to the app (`com.projectx.android.ScouterLite`) that has permission INTERNET.

5.5.5 Summary and Discussion

We summarize our major experimental findings.

1. We observed that the state-of-the-art permission-based policies solution classifies 75.4% of the 175 apps pairs as collusion. 96 of apps pairs classified as collusion by the state-of-the-art do not have a sensitive inter-app ICC call (no sensitive data or critical operations involved). This suggests that permission-based collusion-detection policies
generate high number of false alerts.

Static flow-based analysis solution (e.g., our work) and dynamic permission-based analy-

2. 18.8% of the 175 apps pairs found to have risky inter-app ICC calls. We checked the

3. Our method significantly reduces the number of false alerts of the state-of-the-art solu-

Limitations and Sources of Inaccuracy. In this work, we focus our analysis on the direct

standard communication channel between Android apps, i.e., inter-component commu-
Table 5.6: Pros and cons of static flow-based analysis and dynamic permission-based analysis.

<table>
<thead>
<tr>
<th>Analysis Type</th>
<th>Pros &amp; Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Permission-based</td>
<td>+ Handle explicit and implicit Intent</td>
</tr>
<tr>
<td></td>
<td>- Suffer from high false positives</td>
</tr>
<tr>
<td>Static Flow-based</td>
<td>+ Relatively fewer false positives</td>
</tr>
<tr>
<td></td>
<td>- Restricted to explicit Intent</td>
</tr>
</tbody>
</table>

APIs to files. There are two possible scenarios: (i) both apps read/write to the same hard-coded files, and (ii) both apps read/write to files, however, the filenames are dynamically generated using string operations. The first scenario can be addressed by matching file names in both apps. The second scenario is more complicated to identify. One possible solution is to design heuristics that makes decision based on the proceeding function calls and context associated with file read and write, e.g., suspicious string operations proceeding file open.

ICC can be used for communication between the app’s components and the internal ads library’s components within the app. The ad-related ICC calls can increase the false positives as some ICC calls to the ad’s components may involve sensitive data or critical operations. Since our analysis is on the external inter-app communication not the internal communication, this does not affect our classification accuracy. If one needs to analyze the risk associated with the internal ICC calls, then this issue needs to be addressed carefully as it might affect the classification accuracy.

Our approach identifies statically the predicted risk level associated with the inter-app ICC calls, but it does not confirm the existence of the apps collusion.

Similar to any static analysis based solution, our analysis has difficulty in performing the analysis on programs that employ obfuscation techniques, dynamic code loading, or use of reflection. Moreover, we performed our analysis on the reversed engineering Java bytecode using Dare tool to translate Dalvik bytecode to Java bytecode. The accuracy of our analysis is constrained by the accuracy of the reverse engineering tool.

5.5.6 Recommendations for Secure ICC

Android developers should pay close attention when developing their own apps especially the communications. We suggest that developers check our configuration policies presented
in Table 5.2 when utilizing inter-app ICCs. Below we provide general guidelines for secure ICCs across multiple apps.

When possible, always request a user trigger for the invocation of any ICC call that is part of an inter-app communication. For example, the user should be aware to which app the data will be sent.

Developers should avoid exporting components without permission(s) protection. If the component is exported (public) without any protection, then the component is vulnerable to attack. Permissions are used to restrict access to components and they are declared in the manifest file. If a component is protected by a permission, it can be only accessed by applications that have acquired that permission. The permission checking is enforced by the OS at runtime. However, if a component is not protected by a permission, then it is vulnerable to be accessed by any application. For instance, Android built-in apps such as the browser app are not protected by a permission and any app can communicate with it. This gives an opportunity to the attacker to write a malicious app that can collude with the Android built-in app to perform malicious attack without the built-in app’s knowledge. This can be done by sending an explicit/implicit Intent to the built-in app. For instance, a malicious app can send an implicit Intent using this action `android.media.action.IMAGE_CAPTURE`, which is a standard action to have the built-in camera app capture a picture and returns it to the caller app.

In order to make components more secure and to limit the component’s exposure, developers can enforce certain permissions requirements for the sensitive components in the application manifest which can be checked using dynamic permission check `checkPermission(...)`. We performed an evaluation on how to reduce the explicit inter-app ICC risk level if the developers enforce permissions on the components. Specifically, we measured how many apps pairs classified as high and medium risk can be reduced to lower risk level by making the target component protected. If the developers enforce the permissions on the unprotected components in the target apps, all apps pairs with high inter-app ICC risk would be reduced to medium risk level as shown in Table 5.7. Similarly, all apps pairs with medium inter-app ICC risk would be reduced to no risk level.
Table 5.7: Summary of explicit Intent inter-app ICC risk reduction after applying and enforcing permissions on the target components in the target apps.

<table>
<thead>
<tr>
<th># of Apps Pairs (%)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>High -&gt; Medium</td>
<td>31 (100%)</td>
</tr>
<tr>
<td>Medium -&gt; Low (No Risk)</td>
<td>2 (100%)</td>
</tr>
</tbody>
</table>

5.6 Summary

We presented precise and scalable deep cross-app static flow analysis approach to identify the risk level associated with communicating applications. Our approach with our classification security policies are able to significantly reduce the number of false alerts generated by the state-of-the-art permission-based policies solution. Our cross-app ICC analysis can be used for many useful security analyses which include, but are not limited to, applications collusion analysis and vulnerability analysis. Such analysis will allow defenders to stay proactive in defending against evolving malware threats.
Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this dissertation, we have presented novel proactive solutions to the problem of Android malware detection by leveraging advanced program analysis techniques. In the aspect of detecting stand-alone (single) malicious app, we presented a high-precision detection approach based on one complex feature, referred to as the \textit{user-trigger dependence}, that leverages the dependence effects of program behaviors. This feature is extracted through new Android-specific static program analysis algorithm. The uniqueness of our approach is that we enforce benign properties in programs and not relying on malware signatures. To evaluate the effectiveness of this approach, we conducted a large-scale experiments and analyses. The evaluation results suggest that our approach achieves high detection accuracy (2\% false negative rate and false positive rate) which is better than, or at least competitive against, the state-of-the-art. Additionally, our analysis successfully discovered new undetected malicious apps available in Google Play store.

In the aspect of detecting application collusion, we first illustrated the technical challenges associated with detecting app collusion through several experimental studies. We then addressed these challenges by presenting new scalable and deep cross-app static program analysis algorithm, and defining new fine-grained security policies to detect the risk level associated with inter-app communication. The proposed approach statically captures the sensitivity and the context of the inter-app communication with low analysis complexity. Extensive
experimental evaluations showed the effectiveness and practicality of our approach in identifying the risk level of the inter-app communication, when compared with the state-of-the-art collusion-detection solution. Our method substantially reduces the number of false alerts generated by the existing solution, enhancing the detection capability.

The proactive defenses solutions presented in this dissertation allow defenders to stay ahead of the game in the eternal arms race between attack and defense. Furthermore, they can be applied to general user-centered programs and applications beyond the specific Android environment studied.

6.2 Future Work

We envision to extend the work presented in this dissertation in several directions. First, capturing and enforcing the dependency between critical operations and user actions is a novel and promising approach for defending against malware. We envision that our approach can be applied to other mobile OS platforms beyond the specific Android OS studied, such as iOS and Windows Phone platforms. Moreover, we plan to generalize the dependence definitions to include non-user triggers (e.g., automatic system triggers) and indirect dependence relation.

Our second future direction involves using advanced program analysis techniques such as blended analysis [26, 27, 41] to capture a more refined and accurate program behaviors. Using both static and dynamic analyses can provide additional assurance to the users. The dynamic analysis approach can compensate for the inability of static analysis to reason about dynamically loaded code and code obfuscation. The dynamic analysis can be used with our user-trigger dependence approach to get insights on which critical operations are triggered by user actions during runtime execution.

The third future direction is to further extend our inter-app analysis to approximate implicit Intent inter-app ICC and define more security policies to further improve the classification accuracy. We also plan to investigate the indirect and covert communication channels between the applications for malware collusion detection. Our current inter-app analysis focuses on standard communication channel (inter-component communication) in Android.

Finally, for the deployment perspective, we plan to provide and present informative and
intuitive interpretation of the multiple dimensional analysis results from various tools to users. Additionally, this will help the security analysts to get more insights about the program and reason about the false alerts.


