Usable Post-Classification Visualizations for Android Collusion Detection and Inspection

Daniel John Trevino-Barton

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

Master of Science
in
Computer Science and Applications

Danfeng Yao, Chair
Eli Tilevich
Christopher L. North

July 29, 2016
Blacksburg, Virginia

Keywords: Android Malware, Security, Visualization, App Collusion

Usable Post-Classification Visualizations for Android Collusion Detection and Inspection

Daniel John Trevino-Barton

(ABSTRACT)

Android malware collusion is a new threat model that occurs when multiple Android apps communicate in order to execute an attack. This threat model threatens all Android users' private information and system resource security. Although recent research has made advances in collusion detection and classification, security analysts still do not have robust tools which allow them to definitively identify colluding Android applications. Specifically, in order to determine whether an alert produced by a tool scanning for Android collusion is a true-positive or a false-positive, the analyst must perform manual analysis of the suspected apps, which is both time consuming and prone to human errors. In this thesis, we present a new approach to definitive Android collusion detection and confirmation by rendering inter-component communications between a set of potentially collusive Android applications. Inter-component communications (abbreviated to ICCs), are a feature of the Android framework that allows components from different applications to communicate with one another. Our approach allows Android security analysts to inspect all ICCs within a set of suspicious Android applications and subsequently identify collusive attacks which utilize ICCs. Furthermore, our approach also visualizes all potentially collusive data-flows within each component within a set of apps. This allows analysts to inspect, step-by-step, the the data-flows that are currently used by collusive attacks, or the data-flows that could be used for future collusive attacks. Our tool effectively visualizes the malicious and benign ICCs in sets of proof-of-concept and real-world colluding applications. We conducted a user study which revealed that our approach allows for accurate and efficient identification of true- and false-positive collusive ICCs while still maintaining usability.

This work received support from the Defense Advanced Research Projects Agency (DARPA) grant 450333.

Dedication

I would like to dedicate this work to my mother, Sylvia Trevino-Barton, and my father, Daniel Barton Jr., who have been the guiding light in my life. The sacrifices you have made enabled my brother and I to achieve feats we never thought possible. I will forever be indebted to both of you for all that you have done.

Acknowledgments

I would like to thank my adviser, Dr. Danfeng (Daphne) Yao for her continuing support, guidance, and insight throughout my graduate career here at Virginia Tech. Without her, this work would not have been possible. I would also like to thank my grandparents, Richard and Josie Trevino as well as Dan and Nancy Barton, for their encouragement and support. Furthermore, I would thank my mother, Sylvia, my father, Daniel, and my brother, Nathan, for all the countless visits, conversations, and reassurances throughout my graduate studies. Last but not least I would like to thank my wonderful girlfriend, Jasmyn Herburger, for accompanying me here to Virginia Tech. Thank you for all the stress-induced rants you endured and the constant love and support you provided. I could not imagine completing this journey without you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Current Security State of Android Mobile Devices

The popularity of the Android operating system attracts malware developers. This potentially increases the frequency of Android-based attacks. Malware may attempt to exfiltrate sensitive data (e.g. stealing personal credentials), abuse system resources (e.g. operating stealthy bots), or disrupt regular services (e.g. draining battery).

Malicious developers utilize various attack models when targeting the Android platform [45]. A new attack model, *Android malware collusion*, has recently emerged which subverts most pre-existing defense strategies [8, 31, 6]. *Android malware collusion* is defined as the collaboration of multiple applications to execute an attack on the Android platform, as opposed to an individual Android application executing an attack without assistance. Malware collusion, similar to the confused deputy attack, can result in a successful privilege escalation attack. In a confused deputy attack, a malicious app communicates with a vulnerable-yet-benign app via inter-application communications to perform an attack. While recent literature addresses the confused deputy attack [10, 29, 34], definitive identification of malware collusion remains an open problem.

Various approaches have been proposed in recent literature to combat malware collusion [8, 15, 26, 36]. However, they possess limitations, particularly *usability* related issues, which prevent either accurate or reliable identification of app collusion. For instance, XManDroid [8], a dynamic analysis-based monitoring tool, is proposed to inspect communication channels between Android apps at runtime. It identifies collusive apps through a course-grained classification system of communication flows between apps based on their respective sets of requested permissions. Because of its coarse-grained approach, XManDroid leads to a high number of false-positive classifications of communication flows. These false-positively classified flows need further human effort to justify.

To address the above problem, FUSE, a static analysis tool, has been proposed in [36]. Similar to XManDroid, it also employs a coarse-grained classification system, based on app permission sets,

to identify malicious single and multi-app communication flows revealed through static analysis. The static analysis framework in FUSE, applied with an inferior approach [14], is context and flow insensitive. Moreover, FUSE provides a node-link graph based visualization support post-classification investigation of communication flows. However, FUSE's visualization is admittedly overwhelming to users at times, which is remedied by integrating a filtering function. This function allows users to eliminate some specific subsets of communication flows. Nevertheless, it may lead to a situation, where users mistakenly configures the filter to hide a communication flow that potentially reveals app collusions. While recent approaches have used more precise static analysis techniques to identify collusive data-flows [27, 14], they do not provide a visualization of their results. This lack of visualization increases the difficulty of post-classification data-flow inspection.

Automatic detection methods can identify lists of suspicious application pairs that have high collusion risks. However, because there are benign apps that legitimately communicate via inter-app inter-component communication (ICC) calls, results from automatic detections have false positive ones (i.e., false alarms). A recent study [15] reports that out of 2,644 top-rated apps downloaded from Google Play Market, 84.4% of them (2,230) have external ICC communications with other apps. That means that the majority of benign apps routinely interact with built-in or third-party apps in order to send or receive data, or invoke or provide services. Such high degrees of inter-app interaction indicate that misclassifying benign inter-app ICC flows as malicious ones by automatic methods is quite likely. Thus, it is necessary to involve security analysts in the loop to check and justify these algorithmically identified, suspicious app pairs. This human-involved investigation serves two key purposes: 1) confirming true collusion cases (or true positives), and 2) ruling out benign cases (or false alerts).

To involve human in the loop for *Android malware collusions* analysis, we need present algorithmic results in a human readable, usable and actionable manner. This potentially requires the support of visualization, because it enables analysts to see algorithmically identified collusions from certain visual patterns, manipulate visual objects, and further make judgements about them. However, how to visualize collusions, particularly in the cyber security domain, is not systematic investigated in the literature, although it has been studied in other domains (e.g., intelligence analysis [19, 42]).

1.2 Challenge Statement

The problem we aim to handle is the difficulty associated with post-classification app collusion visualization and inspection, particularly and specifically, the design requirements of visual analysis tools to support human inspection for post-classified app collusion are yet to be defined.

The notion of app collusion classification is introduced in [8], and results in [13] which shows that collusion classification of Android apps is possible with a relatively high level of accuracy. However, these approaches do allow for the existence of false-positive and false-negative classifi-

cations. The former occur when a set of benign apps is classified as colluding, and the latter refer to that a set of colluding apps is classified as benign. To ensure accuracy when classifying a set of apps, analysts have to inspect the set after the classification has been made. Analyst inspection for security vulnerabilities, such as post-classification inspection for Android collusions, can be tedious, time consuming, and susceptible to human errors, but can be remedied with the support of analysis tools [43].

Little work has been done to investigate the design of these tools, particularly from a usability perspective, for the support of human inspection of post-classified collusions. In this paper, we aim to explore the potential design requirements to inform future design of such tools, which can potentially support human inspection of post-classification app collusion in a efficient, accurate and reliable way.

There are three key challenges, from three aspects, to design visual analysis tools for post-classified collusions analysis:

- From the *security* perspective: how can we accurately and reliably detect app collusion?
- From the *visualization* perspective: how can we display algorithmically identified app collusions from a large amount of data in a way that does not overwhelm the user?
- From the *interaction* perspective: how can we design interaction flows that enable users to retrieve necessary information to support their justification of algorithmic results?

1.3 Post-Classification Collusion Visualization

To address the challenges associated with cross-app communication flow visualization necessary for post-classified collusions analysis, this thesis proposes a novel design for post-classification Android collusion visualization. Our design aims to allow users to inspect a set of Android apps for collusive activity at various levels of granularity. The key insights that motivated our design are as follows:

- Context- and Flow- sensitive, Android-specific static analysis provides sufficient information to detect Android current and future Android collusion.
- A multi-layered approach to visualization increases usability and reduces cognitive load.
- Incrementally increasing the amount of collusion information presented to the user enables high collusion detection/inspection performance and accuracy.
- Graphical visualizations are more user friendly than raw text reports.

1.3.1 Design

We have attempted to congregate the strengths of existing work to form a new approach that is not hindered by their respective limitations. To address the threat of Android malware collusion, in this thesis present the design and implementation for an interactive, web-based visualization tool which utilizes fine-grained static analysis to render all explicit inter-component communications (ICCs) within a set of potentially collusive Android applications in a user friendly format. The tool consumes a set of Android applications as input and displays the communication flows between all communicating applications within the input set, as well as the intra-component data-flows pertaining the the ICCs. Communication flows can be viewed at three levels: the app communication level, the ICC level, and the component flow level. The app communication level allows the user to view the existence and direction of communication flows between input apps. The ICC level allows the user to view, and provides in-depth details of, all explicit ICCs between any two communicating apps. The component flow level provides a visualization of all collusive and potentially collusive data flows within a component of an input app that either sends or receives an ICC. Our design aims to allow analysts to quickly identify ICCs or data-flows that are likely collusive, and then quickly and accurately determine whether the data-flow or ICC is collusive.

Our design overcomes the challenge of accurately and reliably detecting communication flows by using a fine-grained, context- and flow- sensitive static analysis framework to generate communication flow information. This data is then visualized in a minimalistic and interactive layout that prevents users from hiding or blocking suspicious flows. Our tool implements the classification system introduced by Elish et al. [13], a highly accurate classification system for ICC communications between two Android apps. This allows the user to quickly identify seemingly collusive communication flows and specific suspicious ICCs, decreasing the probability of an undetected collusive attack while not overwhelming users with the amount of data presented to them.

1.3.2 Implementation

Our prototype implementation uses state-of-the-art static program analysis techniques to accurately and efficiently analyze a set of communicating Android applications [13]. The static analysis is conducted using the Soot [44] framework, an openly available and widely used static program analysis platform. The analysis itself is both context- and flow-sensitive, which provides a resulting analysis that is both performant and reasonably accurate. However, we designed our prototype to be modular, with the visualization and static analysis components in separate modules. Thus we can easily change the static program analysis to be either more or less sensitive, to decrease or increase performance.

We also use cutting-edge web technologies to create an interactive and scalable (in terms of the number of input applications) visualization. The web-application itself is implemented using the Spring framework [2], a widely used enterprise-level web-application framework. By using the Spring framework, our tool can be quickly ported for commercial use if necessary. The visu-

alization are generated using the D3 JavaScript library [7]. The D3 library allows for dynamic, highly-interactive views in our tools which enable quick and effortless collusion inspection. We designed our tool to be used by security analysts in a variety of organizations (government agencies, large corporations, app stores, etc.) to screen apps for collusion. Due to its web-based design, our tool could also be hosted by third-party groups for use by the general public to protect their Android devices from collusion attacks.

1.4 Research Contribution

The research contributions in this thesis are summarized as follows:

- **The design of a multi-layered visualization approach to Android collusion inspection**: Our design allows analysts to inspect input sets of apps for collusion at four levels. By separating the visualization into four hierarchical levels, we effectively overcome the existing challenges associated with post-classification inspection for Android collusion.
- **A prototype implementation of our design as a a scalable, web-based visualization tool**: Our tool utilizes a fine-grained static analysis method and an accurate classification system to effectively visualize the communication flows of potentially collusive sets of Android applications. The tool possesses a highly usable interface which allows for greater user interaction and deeper inspection than other state-of-the-art solutions.
- **A case study in which our tool is used to visualize several proof-of-concept sets of colluding applications**: Our tool accurately visualizes all collusive communications in an easily detectable manner for human analysts. Both information theft and privilege escalation attacks, normally associated with malware collusion, were detected in our case study. In addition, our tool also displays non-colluding applications in a format which allows the analyst to quickly deduce that collusion is not present. item **A user study which indicates the practical value to our target users**: We conducted a user study in which users were asked the use our prototype to inspect a set of apps for collusion. Results from our study indicate that our design may have practical use in post-classification Android collusion inspection.

1.5 Thesis Roadmap

The remainder of this thesis is organized as follows. Chapter 2 provides an overview of the technologies touched upon in our work. Chapter 3 provides a literature review of research publications closely related to our work. Chapter 4 defines the type of Android collusion our work addresses. Chapter 5 presents our approach to post-classification visualization of collusive information for analyst inspection, as well as potential use cases for our design. Chapter 6 presents the empirical

user study we conducted to evaluate our prototype implementation. Finally, Chapter 7 concludes and presents future work.

Chapter 2

Background

This chapter presents an overview of the Android operating system, the structure of Android applications, the Android inter-component- communication system, and static program analysis for security detection. While these summaries should not be considered comprehensive, they do ensure the reader can sufficiently comprehend the numerous concepts and technical terminology used in the remainder of this thesis.

2.1 Android Operating System Overview

The Android operating system (OS) was built upon a modified version of the Linux operating system kernel. The file system hierarchy in Android is very similar to those used by the most common Linux distributions (i.e. Ubuntu, Debian, Fedora, etc.). Furthermore, Android applications (shortened to apps) can be installed on the Android operating system. Android apps are archive files, based on the format of .zip files, with an extension of .apk. Each .apk archive contains executable Dalvik bytecode files, XML files, and binary resources such as image files. Android apps are written in Java, but are compiled down to Dalvik bytecode executable files, which are run on a custom virtual machine called the Dalvik VM which all Android operating systems run. XML files are used to define the layout as well as elements/widgets within the layout of a touch-screen graphical user interface (GUI) presented to the app's user. Each GUI element can be mapped back to a Java method, also known as callback, in the app.

At a high level, an Android app can be thought of as a collection of Java methods which are mapped to certain user interactions with the app's accompanying touch- screen GUI. For example, if a developer defines a button in an XML file for an app which defines a screen, and maps that button back to the Java method whenButtonClicked(), that method will be called by the Android OS whenever the user clicks the button. However, not all callback methods need to be mapped to a GUI element in order to be called. The Android OS will call certain pre-defined methods when certain events happen. For example, the onCreate() method in all apps will be called by

the Android OS whenever that app is opened by the user. Because the Android OS awaits user interaction to determine which action to take next, it has been termed an event-driven operating system.

Thus, an instance of the Android OS running on a mobile device can be thought of as an event-driven Linux distribution with a vastly different GUI. However, the security model employed by Android is also vastly different than the standard desktop security model. In the Android security model, all applications are treated as malicious. "Android applications are treated as mutually distrusting principals; they are isolated from each other and do not have access to each others' private data" [10]. Each app runs in its own process and is assigned a low-privilege user ID; they can only access their own files by default. This is done to secure each application's sensitive information.

In addition, the Android OS also protects sensitive system API calls from installed apps. By default, Android apps do not have access to sensitive system API calls. In order to access certain system calls, an app must request the permission associated with the desired system call. Each permission grants access to a set of related sensitive system API calls. At installation, the app will request all permissions it needs from the user in order to run properly. If the user does not agree to all the requested permissions, the app is not installed.

2.2 Android Application Structure

Android apps are composed of components which are logical application building blocks. Each component is a Java class which extends one of four predefined Android component classes. The specific class a component extends determines the type of the component. In order to be used by the Android OS, components must be declared in an app's manifest file, which is an XML configuration file that accompanies each application during their installation. The component can be either an activity, a service, a broadcast receiver, or a content provider.

2.2.1 Activities

An activity provides the user with a screen that he or she can interact with to accomplish various tasks. Basically, an activity represents a specific window within an app. Activities can be invoked by other components and can return data to them upon completion. An app is typically a set of activities which invoke one another. Activities are typically associated with a layout XML file located in the app's .apk file which defines the layout of the GUI widgets associated with the activity.

2.2.2 Services

A service performs long-running operations in the background and does not provide a user interface with which the user can interact. Other components can start services and they will continue to run in the background regardless of user interaction. An example of a service could be a component which will play music when it is started, so that the user can listen to music while he or she interacts with other apps.

2.2.3 Broadcast Receivers

A broadcast receiver receives messages which are transmitted by the Android OS to multiple applications. Each broadcast receiver is triggered by a specific message and runs in the background like a service to handle the event. Receivers are typically short-lived, as opposed to long-running like services, and usually send messages to activities or services within its app.

2.2.4 Content Providers

A content provider administers access to a structured set of data. They can be thought of as databases which are addressable by URIs. Not only do content providers contain the data, but they also define how that data can be accessed. They are typically used for both persistent internal data storage as well as an interface which allows information sharing with other apps.

2.3 Android Inter-Component Communication

Android provides a message passing system which allows components within an app to communicate amongst themselves or with components from other apps. This is often referred to as inter-component communication (ICC) or message passing in the ten papers this paper surveys. Communication is done by passing Intent objects between components. An intent is a Java object that declares its recipient and and may include various data fields. Intents are created by a source component and are handed off to the Android OS for delivery as parameters to a system call. There are two types of intents: explicit intents and implicit intents.

An explicit intent specifies the exact component within a specific app that it should be delivered to. On the other hand, implicit intents specify the general category of components they could be delivered to as opposed to the specific component. Implicit intents allow the Android OS to determine which exact component they should be delivered to. For both explicit and implicit intents, the target component could be within the same app as the source component that sends the intent or in a different app. Intents are used for practically all ICC related functions such as starting services and activities or sending system broadcast messages.

Components can either be exported or non-exported. Exported components, often called public components, can receive intents from components belonging to other applications. Intent filters are used by exported components to specify what type of intents can be delivered to it. Non-exported components, often referred to as private components, can only receive intents from components within the apps it belongs to. It is important to remember that intent filters are not a security mechanism for protecting components as they can easily be subverted by malicious Android apps.

2.4 Static Program Analysis in Android Security

Program analysis refers to techniques which produce "reliable information about the dynamic behavior of programs." [32]. There are two approaches to program analysis: static and dynamic program analysis. Dynamic program analysis entails executing the target programs while observing and/or analyzing their behaviors. Conversely, static program analysis does not require the execution of the target program, and usually involves analysis on its source or object code. Various literature proposes using both dynamic [16, 3, 41] and static [10, 29, 14] to detect and combat Android malware. Our approach presented in this thesis involves using static program analysis, more specifically, analysis on a Data Dependence Graph (DDG). A DDG is defined as a program analysis structure which represents the inter-procedural flow of data and control through a program [22, 18]. The actual DDG is a directed graph where nodes are program statements and directed edges represent flows of data or control from one statement to another. Specifically, we use the DDG construction techniques and analysis proposed by Elish et. al [13] due to their focus on Android collusion and its context- and flow-sensitives.

Chapter 3

Literature Review

This chapter presents a review of published scientific literature pertinent to the research presented in this thesis. We will discuss how our approach, and the problem we attempt to address, differs from the literature.

3.1 Android Vulnerability Detection Methods

This section presents literature that addresses the problem of vulnerability detection in Android applications. The goal of these literature presented in this section is to identify vulnerabilities in apps which could be targeted by Android malware, not to identify Android malware. These techniques focus on identifying vulnerabilities that result from misuse or abuse of Android's ICC system. Static analysis techniques are typically used to either identify or exact vulnerable entry/exit points in components or to identify certain data flows which constitute a vulnerability. While these methods do not directly combat Android collusion, the vulnerabilities they identify can be used by collusive sets of Android apps to execute an attack. Furthermore, the solutions this literature proposes does not enable analysts to identify Android collusion.

3.1.1 ICC Vulnerability Detection

The Android operating system possesses a rich inter-application message passing system which encourages app collaboration and reduces developer workload. However, this message passing system also allows an attack surface which can be exploited by malware developers. The content of messages can be sniffed, modified, stolen, or replaced. This allows for a scenario in which the Android user's privacy is compromised. "Analyzing Inter-Application Communication in Android" by Chin et al. [10] attempts to address this issue by providing a thorough overview of the Android operating system and its message passing system, defining the attack model of malicious

attacks which target the message passing system, and propose ComDroid, a tool which attempts to identify message passing vulnerabilities in Android apps.

The authors begin by giving an overview of how Android's security model differs from desktop applications' security model. In an Android operating system, each app is treated as a mutually distrusted principal, which means they are isolated from one another and do not have access to each other's private data (similar to the SOP property commonly enforced in web browsers). However, the message passing system potentially allows apps to subvert this security policy. The authors' focus is on preventing apps from being exploited by one another through the message passing system.

The Android operating system allows apps to communicate with one another through intents. Intents are Java objects which declares a recipient and optionally includes data. Another way to think about an intent is that it is a self-contained object that specifies a remote procedure to invoke and includes the arguments for the procedure. Intents are used for both intra- and inter-application communication. There are two types of intents, implicit and explicit intents. Implicit intents simply specify which type of component they should be delivered to, where as explicit intents specify the exact component they should be delivered to. Intents are delivered to the proper component by the Android operating system.

The authors next detail the threat model they are attempting to address in this paper. Their primary focus is on: (1) intents sent to incorrect applications can result in user information leakage and (2) vulnerabilities related to the reception of external intents which are intents from other applications. It is important to note the authors only consider vulnerabilities in exported components which are components that can receive intents from other applications. Implicit intents can be sent to the wrong component or a malicious component could intercept an implicit intent. This can result in broadcast message theft, activity hijacking attacks, service hijacking attacks, and intent man-in-the-middle attacks. Malicious applications can send explicit intents to components which are not expecting them. This could result in malicious broadcast injection attacks, malicious activity launches, and malicious service launches. All of these attacks could result in the loss of user information privacy.

To address this issue, the authors present ComDroid, a tool which will detect potential message passing vulnerabilities. ComDroid takes Android apps as input and outputs a list of warnings which identify potential vulnerabilities. Intent creation and transmission are monitored by ComDroid throughout their relevant life-cycles for implicit intent vulnerabilities. To accomplish this, ComDroid statically analyzes the disassembled Android dex code. The intraprocedural, flow-sensitive static analysis is augmented with limited inter-procedural analysis that follows method invocations to a depth of one method call. ComDroid also performs component analysis on all of an app's components in order to determine if each component is susceptible to intent spoofing attacks. This is done by examining both the app's manifest as well as each component's Dalvik instructions. The determination is based mostly on whether the component is exported and the types of permissions required by each exported component.

The authors evaluated ComDroid by using it on 100 Google Play apps. Once ComDroid evaluated

all of the apps, the authors randomly selected twenty apps and performed manual analysis in-order to ascertain ComDroid's accuracy. Their manual analysis revealed that ComDroid detects 22.6 percent of implicit intent bugs/vulnerabilities and 38.6 percent of explicit intent bugs/vulnerabilities. Furthermore, their manual analysis revealed that ComDroid has a high rate of false-positive classifications. The authors conclude that their work shows that apps are vulnerable to the attacks they defined and that developers should take precautions to protect their apps.

3.1.2 Android Capability Leak Detection

In the Android operating system, a permission-based security model requires each app to explicitly request permissions up-front to access sensitive information and/or phone features. The permissions effectively define the capabilities the user grants to the app. Thus, it is essential for user information security that this model be strictly enforced by all Android platforms. However, Grace et al. [20] discovered that eight popular Android smartphones from leading manufacturers, including HTC, Motorola, and Samsung, do not perfectly enforce the above security model. Several sensitive permissions are exposed to third-party apps, which do not request the sensitive permissions, by pre-loaded apps for unchecked usage. This could lead to a variety of security breaches, including loss of privacy for user information as well as abuse of sensitive system features. To combat this threat, termed capability leaks, the authors developed Woodpecker, a tool which uses data flow analysis on pre-loaded apps which systematically analyzes each app to explore the reachability of a dangerous permission from an unprotected and public interface.

The authors focus on two types of capability leaks: explicit and implicit capability leaks. Explicit capability leaks "allow an app to successfully access certain permissions by exploiting some publicly-accessible interfaces or services without actually requesting these permissions by itself" [20]. Implicit capability leaks are similar, but instead permit an app to share the same permissions from another app with the same signing key. Explicit capability leaks subvert the Android permission-based security model while implicit capability leaks could allow for the misrepresentation of the true permissions of a given app.

Woodpecker identifies explicit capabilities leaks by performing two complementary analyses. First, Woodpecker locates the apps pre-loaded in the phone that possess each exploitable capability. Next, for each located app, the system identifies whether a public interface is exposed that can be used to gain access to the sensitive permission. At this point, the first of the two analyses is performed, whereby a control-flow graph is generated to identify possible paths from an entry point to some use of the capability. After the first analysis, the second analysis employs field- and path-sensitive inter-procedural dataflow analysis to determine which of the paths identified in the first analysis are feasible.

In order to detect implicit capability leaks, Woodpecker reports all occurrences of an app, which shares a user identifier, exercising an unrequested capability that has been requested by another app by the same author. The process for detecting implicit capability leaks is very similar to detecting explicit capability leaks, but instead assumes the pre-loaded app itself may be malicious.

Thus, instead of simply generating control-flow graphs from the entry points, the authors consider the app initialization site as well. Once Woodpecker identifies that an implicit capability leak exists, it then unions the permissions sets granted to each application with a shared user identifier, which yields a superset of all the permissions available to the apps. The tool reports any leaked permissions contained within the superset.

The authors conducted an evaluation of Woodpecker where they used it to analyze eight smartphones from four vendors (HTC, Motorola, Samsung, and Google). Their results showed that Woodpecker successfully found both explicit and implicit capability leaks, which were later confirmed by a few of the manufacturers. On average, Woodpecker required roughly fifty-one minutes to analyze a smartphone image. Given the off-line nature of their tool and the fact that it had not been optimized for speed, the authors reasoned that Woodpecker's performance was acceptable. The authors conclude that the results they discovered, that 11 of the 13 privileged permissions examined by Woodpecker were either explicitly or implicitly leaked, is worrisome.

3.1.3 Android App Vetting for ICC Vulnerabilities

As the popularity of the Android platform grows, the chances of poorly engineered apps which contain security vulnerabilities appearing on marketplaces increases. In order to properly vet apps for security vulnerabilities, app marketplaces require analysis tools which are both reasonably sound and scalable in order to accommodate the large number of apps available. Thus, Lu et al. [29] propose CHEX, a static app vetting tool for component hijacking vulnerabilities, which includes previously established vulnerabilities such as permission redelegation, intent spoofing, and private data leakage. Component hijacking vulnerabilities typically result in unauthorized read or write operations to/on sensitive resources. They address this class of vulnerabilities by transforming the detection problem into an equivalent data-flow problem that aims to identify hijacking-enabling flows in arbitrary apps.

In general, component hijacking attacks are defined as a class of attacks which seek to gain an unauthorized access to protected resources through exported components. A key feature of component hijacking attacks is that they are not caused by any insecurity intrinsic to the Android framework. Rather, component hijacking stems from issues that are hard to avoid in reality, such as undertrained developers. Given the trend that unskilled developers will flock to the popular Android platform, and also that attackers are struggling to craft new attack techniques for Android, the emerging attack vector of component hijacking is not likely to go un-exploited. The authors formally define component hijacking attacks as "an unauthorized app, issuing requests to one or more public components in a vulnerable app, seeks to: read sensitive data of the app, write to critical data regions inside the app, or perform a combination of the first two tasks" [29].

The analysis begins by following the authors' algorithm for app entry point identification. This algorithm first generates a call graph for the component and adds each overriding method to the results sent as well as all callbacks defined in the component's manifest. Once all entry points are identified, the analysis models each entry point's asynchronous invocations soundly by using

a novel technique called app code splitting. A split is defined as "a subset of the app code that is reachable from a particular entry point method" [29]. The analysis first generates app split data-flow summaries (SDS) for each split, followed by permutation data-flow summaries (PDS) by linking the SDS of each split in the sequence. Each PDS is then examined for data-flows that are indicative of component hijacking attacks. The PDS allows the analysis to track data-flows across splits and components.

The authors performed an in-depth empirical evaluation of CHEX in terms of its performance and accuracy. The authors used a corpus of 5,486 apps, which were collected in 2011, from both the Android market as well as alternative markets. The median processing time of an app was 37.02 seconds, however, 22 percent of the apps timed out (i.e. took more than 5 minutes to evaluate). The evaluation was performed on three computer clusters each equipped win an Intel Core i7-970 CPU as well as 12 GB of RAM. When CHEX flagged a component as potentially hijack-able, the authors performed manual analysis of the component to determine CHEX's accuracy. The authors discovered that CHEX has an accuracy rate of 81 percent (in other words, 19 out of 100 alerts were false positive alerts). However, the authors do explain that the number of false positives can be reduced by modifying CHEX to leverage domain knowledge about the partial orders in which Android component and their entry points can run or interleave. The authors conclude that CHEX possesses the scalability and performance necessary to automatically vet large numbers of apps which may be vulnerable to component hijacking attacks.

3.1.4 Effective ICC Mapping for Vulnerability Detection

While ComDroid by Chin et al. [10] was a good first step in combating attacks which exploit the Android message passing system, its methodology was imprecise and resulted in a large number of false positive alerts. However, in the paper by Octeau et al. [34], the authors attempt to formally recast the challenge of inter-component communication (i.e. the Android message passing system), or ICC for short, analysis to an IDE problem [38] in order to infer the locations as well as the values of all inter- and intra-application communications. Their approach provides a high-fidelity means to examine component interaction, which is necessary for comprehensive security analysis. Their approach involves formally specifying every ICC source and sink. When the exact value of an ICC cannot be statically inferred, their analysis deduces all possible ICC values, which results in a complete specification of how all ICCs can be used. The authors implemented their analysis in a tool called Epicc which is publicly available.

The authors began by outlining the goals their analysis hopes to accomplish. Their analysis must be sound, that is, it should generate all specifications for ICCs that may appear at runtime. This is to ensure that no ICC will go undetected, which would result in false negatives. Furthermore, their analysis must also be precise in the sense that it does not detect communications that do not occur in practice (i.e. limits the number of false positive alerts). With these goals in mind, the authors developed an analysis which is able to detect ICC vulnerabilities, detect attacks which target ICC vulnerabilities, and perform inter-component information flow analysis. Their analysis

aims to connect ICCs both within a single app and between different apps by, for each component in each app: (1) locating ICC entry points, (2) locating ICC exit points and inferring what types of components they can communicate with, (3) and matching exit points with possible entry points.

During this three step process, the authors recast the intent ICC problem to an Inter-procedural Distributive Environment (IDE) problem, which allows them to compute the intent values/possible values at each ICC exit and entry point. Reducing the challenge of mapping ICCs to an IDE problem, which can be solved efficiently, allows the analysis to be scalable, precise in the sense that it generates few false positive alerts, flow sensitive, and context sensitive. All of these properties combined result in a precise, performant, and sound analysis.

In order to reduce the intent ICC problem to an IDE problem and solve it, the authors needed to model four different Android objects: ComponentName objects, bundle objects, intent objects, and IntentFilter objects. The ComponentName model represents the value that a ComponentName object can have. The bundle model is generated for each potential bundle object by running two successive analyses. The first analysis identifies "placeholder" bundle objects and the second analysis determines the values possible at each placeholder. The intent model contains both ComponentName and bundle models and is generated in a two-step analysis similar to the one used to generate bundle models. IntentFilter models are similar to intent models, but do not involve object composition. This results in a single step analysis to generate IntentFilter models, which allow for the analysis of dynamic broadcast receivers. Once these objects have been modeled, environment transformers along with the algorithm presented in Sagiv et al. [38] allow the authors to solve the intent ICC problem.

The authors implemented their ICC security analysis in a tool called Epicc (Efficient and Precise ICC). They evaluated their tool through two separate experiments; the first presented raw statistical data for Epicc while the second compared Epicc to ComDroid. Epicc was able to generate unambiguous specifications for over 94 percent of the 58,989 ICC locations across 838 Google Play apps. The average time to complete an analysis for an app was about two and a half minutes. When compared to ComDroid, Epicc was able to generate all the warnings that ComDroid did, minus significantly less (32 percent less to be precise) false positives. Furthermore, Epicc was able to detect significantly more vulnerabilities, such as system broadcasts without action check, than ComDroid. ComDroid was not able to detect any vulnerabilities that Epicc missed. The authors conclude that their method for inferring ICC specifications is efficient, sound, and scalable.

3.2 Android Malware/Taint Detection

This section presents literature related to Android malware and/or taint detection and identification. The goal of the work presented by the literature is to take the next step from simply identifying and reporting vulnerabilities in Android to identifying actual malicious and/or harmful actions in Android applications. These approaches typically use some form of static-based taint analysis to identify malicious and/or harmful data flows or method call sequences in Android apps. Several of

these approaches are capable of detecting app collusion, such as IccTA [27], while FUSE [36] is the only work that directly attempts combat Android collusion. However, when compared to the approach presented in this thesis, FUSE is less accurate and usable than our approach. Furthermore, the authors of FUSE do not conduct a user study to evaluate the usability of their visualization.

3.2.1 Android Taint Analysis via Abstract Interpretation

Android's permission-based approach to security does not ensure the confidentiality of private information due to the fact that it does not trace the usage of private information. To address this, Kim et al. [25] present ScanDal, a static analyzer that detects privacy leaks in Android apps. A privacy leak is defined as a flow of information from a source of private information to a sink, which will transmit the data (i.e. sending it over a network, writing it to a file, etc.). ScanDal is sound in the sense that it will cover all possible states which may occur in an application, thus it will detect all possible privacy leaks.

The authors begin by outlining their definitions of sources and sinks. Sources are defined as API calls which return private/sensitive information. In particular, the authors focus on sources that grant access to device location information, phone identifiers (i.e. device ID and SIM card serial numbers), and audio/video recordings. Sinks are defined as API calls which transfer information to networks, files, or remote destinations via SMS messages. Every time private information leaves the Android device, ScanDal classifies it as a privacy leak.

ScanDal was designed by the authors in the abstract interpretation framework [11]. ScanDal begins its identification of privacy leaks by receiving a packaged Android app as input. Once the Android app's Dalvik bytecode has been extracted, ScanDal then translates the bytecode into Dalvik Core, an intermediate language designed by the authors to allow for simple and efficient analysis. However, it is important to note that any feature not represented in Dalvik Core (i.e. unrepresented Android APIs) but used by the app could lead to unsoundness in the analysis. During the translation, ScanDal builds a control-flow graph, adds hard-coded definitions of library calls, and creates the main entry point to the app according to the authors' model of the execution of the app. At this point, ScanDal proceeds with its abstract interpretation, where all potential privacy leaks are detected by considering every machine state which may occur when the application executes. ScanDal collects every value which could be created from information sources, and if they flow to a sink, ScanDal classifies it as an information leak. It is important to note that for the majority of the analysis (i.e. anything that is not during the initialization of the app), the analysis is flow-insensitive. ScanDal is also capable of tracking inter-component privacy leaks (i.e. privacy leaks utilizing ICCs).

The authors evaluate ScanDal by analyzing 90 Google Play apps as well as 8 known malicious apps. When used to analyze the Google Play apps, ScanDal was able to detect privacy leaks in 11 apps, which were manually verified by the authors. Most of the leaks appear to be information used by companies for advertising purposes, and thus not overly malicious. ScanDal was able to detect the privacy leaks in all 8 malicious applications. However, based on the authors' evaluation,

ScanDal does appear to suffer from high false-positive rates. This is mostly due to the flow-insensitive phase of their analysis. The authors conclude that ScanDal is a formal, sound, and automatic static analysis tool for privacy leak detection in Android apps.

3.2.2 Android Collusion Detection and Visualization

Due to the popularity of the Android operating system, the development of apps for it has skyrocketed, as well as the development of Android malware. To combat these malicious developers, researchers have developed security techniques which focus on identifying undesirable behaviors in individual applications, whether due to malicious intent or programmer error. In this paper, Ravitch et al. "presents a collection of tools that provide a static information flow analysis across a set of applications, showing a holistic view of all the applications destined for a particular Android device" [36]. Their goal is to identify undesirable and/or malicious communication patterns within a set of arbitrary apps. Their tool, named FUSE, graphically displays the directed communication graph to the user, where apps are nodes and edges are information flows between apps.

To begin analyzing a set of apps, FUSE begins by generating an extended manifest for each app in the set. An extended manifest is a data structure that builds on the standard Android manifest of each app to include all sources and sinks in the apps as well as information flows between them. Each sink records its type (i.e. sends SMS message, starts a new service, etc.) as well as all possible sensitive information sources that can flow to it.

In order to perform taint analysis, the authors use a pointer analysis, based on Andersen's analysis [4] which is field-sensitive but context- and flow-insensitive, to generate a call graph. The authors use the result of the pointer analysis to propagate taint labels from sources to sinks. Once all taint labels have been propagated to all sinks, FUSE then finds all reachable sinks by iterating over all the reachable call instructions in the app currently being analyzed. This list of reachable sinks, as well as the taint labels propagated to them, inform the user of all sensitive information that can flow to it. It is important to note that since the pointer analysis is context- and flow-insensitive, these labels do not mean that sensitive information does flow to this sink, only that it is possible that it does.

Once all the apps have been analyzed, and extended manifests have been produced for each, FUSE then generates a directed graph which characterizes the flow of information between all the apps. To construct the graph, a node is created for each permission granted to all the apps in the set, for each source, and for each sink. Once all the nodes are generated, FUSE then generates edges between sources and sinks that represent a flow of information. At this point, the full communication graph for the set of apps is complete.

However, the communication graph of a set of apps is often overwhelming and unusable. To address this, FUSE incorporates a system of information flow assertions that allow the user to specify information flows that are disallowed, expected, or only conditionally allowed. It is important to note that this particular feature of FUSE relies on the user's own knowledge of malicious data

flows; FUSE itself does not have any knowledge or notion of what is a benign flow or a malicious flow.

The authors evaluated only the single app analysis portion of FUSE, as the authors were not aware of any app evaluation suites which detect collusive behavior. They ran FUSE on the DroidBench benchmark suite and compared its results to those of FlowDroid [6]. FUSE's precision is lower than that of FlowDroid due to its lack of flow-sensitivity. However, FUSE's recall rate is equivalent to FlowDroid's recall rate. The authors conclude that FUSE and similar analysis tools are making steps to combat the growing threat of Android app collusion.

3.2.3 Cross-Component Taint Analysis

Up to this point, most privacy leak related research in Android focused on identifying privacy leak vulnerabilities or taint analysis within individual components. However, recent reports have documented instances of cross-component privacy leaks (i.e. the privacy leak starts in a component and ends in another) [46]. This new threat model will elude previous Android dataflow analysis tools, such as FlowDroid [6]. To combat this new threat mode of cross-component privacy leaks, Li et al. [27] present IccTA, an inter-component communication taint analysis tool, which uses a novel static taint analysis technique to find privacy leaks outside the app or device in a sound and precise fashion.

The authors begin by presenting an empirical study they conducted to ascertain how inter-component communications (ICCs) are used in Android apps and expose the difference of usage between malware and benign apps. The authors wanted to answer the following questions: (1) how often are ICCs used in Android apps?; (2) what kind of intents (implicit vs. explicit) are used for ICC in apps?; (3) and is the usage of ICC different between malware and benign apps? Their study revealed that nearly all the apps (96.4 percent of 2,046 apps) used some form of ICC during their executions. Furthermore, their study revealed that applications range in their rate of explicit intent usage vs implicit intent usage (of all intents used by an app, the percentage of explicit intents ranges from 51 to 40 percent). Finally, their study found that malicious applications manipulate significantly more intents and tend to use more explicit intents than benign apps.

IccTA detects ICC leaks across components by first transforming the Dalvik byte-code of a suspicious app into an intermediary representation. The tool then extracts the ICC links between components and stores them along with their associated metadata in a database. Based on the stored ICC links, the tool proceeds to modify the intermediary app representation to directly connect the linked components to enable cross component dataflow analysis. Using a modified version of FlowDroid [6], IccTA then builds a complete control flow graph for the input application on which the taint analysis is conducted. Any leaks that are discovered by the taint analysis are finally stored in the database.

The main two technical contributions lie in the ICC links extraction process as well as the taint analysis. An ICC link is defined by authors as a pair consisting of an ICC method along with its

relevant metadata (i.e. name for an explicit intent or the action, category, mime-type, etc. for an implicit intent) and a target component. ICC links are identified by a three-step process that involves using Epicc [34] to obtain ICC methods and their parameters, parsing the app's manifest file for all possible target components, and finally matching ICC methods with their target components.

In order to accurately conduct the taint analysis across all components, the authors elected to instrument the intermediary representation of the app to handle ICC methods, lifecycle methods, and call back methods. The authors handle ICC methods by replacing an ICC method call with an instantiation of the target component with the accompanying intent. Finally, the authors incorporate lifecycle and callback methods into their analysis by generating a dummy main method for each component in which all lifecycle/callback methods in the app are included.

The authors evaluation of IccTA compared it to existing security tools for Android apps, detailed its performance on real world Android apps, and reported its runtime performance. Their evaluation not only revealed that IccTA outperforms the existing security tools, but that it also can find ICC leaks in a large set of the real-world apps. The authors conclude that their work effectively addresses the major challenge of performing data-flow analysis across multiple components for android apps.

3.2.4 Type-Analysis Based Inter-Component Taint Analysis

Static taint analysis for Android apps detects privacy leaks within the app without actually executing it. Existing approaches to taint analysis in Android, such as ComDroid [10] and FlowDroid [6], employ a points-to analysis as the basis for their taint analysis. However, Huang et al. [23] the authors propose a type-based taint analysis for Android apps, as opposed to a points-to based taint analysis. The authors' type-based taint analysis, termed DFlow, is lightweight and scalable which expensive points-to analyses typically prevent. DFlow is also precise because it is essentially a CFL-reachability computation, which is a highly precise analysis technique [37].

In the authors' type-based approach to taint analysis, each variable is typed via a type qualifier. A type qualifier can either be tainted, which means there is a flow from a sensitive source to the variable, or safe, which means there is not a flow from a sensitive source to the variable. Furthermore, context sensitivity is achieved by using a polymorphic type qualifier (shortened to poly) and viewpoint adaptation [12]. A poly is a type qualifier that is tainted in certain contexts and safe in others. Each poly's value is interpreted by the viewpoint adaptation operation.

The authors define their typing system over a syntax in which the results of field accesses, method calls, and instantiations are immediately stored in a variable. Given sets of sources and sinks, the authors' type inference derives an assignment from program variables to type qualifiers that type checks with their predefined set of typing rules. If the type inference succeeds, then there are no information leaks from sources to sinks, otherwise there has been a leak. However, even if a type inference succeeds, there still may be undiscovered flows from sources to sinks. To address this, the authors adapt method summary constraints, a technique which removes infeasible type

qualifiers, to help reveal undiscovered flows.

However, just because a type inference fails does not necessarily mean there is a leak from a source to a sink. The authors address this issue by mapping each type error (i.e. failed type inference) to one or more feasible source-sink paths. They do this by generating a dependency graph from the constraints for program statements which excludes parts of the program which are unaffected by sinks. Finally, the authors perform CFL-reachability reasoning on this graph. For each type error, the authors print feasible paths from the source at the type error to all reachable sinks. This approach provides a data-flow guarantee but not a control-flow guarantee (basically, all generated paths may never be executed in practice). Thus a conservative call graph is generated and then used to eliminate unreachable and/or unconfirmed paths.

This approach accounts for and handles ubiquitous Android libraries, Android callbacks, and inter-component communications, all of which are problematic Android constructs. The authors performed an empirical evaluation of their type-based taint analysis and compared it to similar points-to based taint analysis tools. The authors analysis outperformed both IBM AppScan Source and HP Fortify SCA, but performed comparably to FlowDroid [6]. However, the authors' type-based analysis required significantly less memory than FlowDroid. Furthermore, FlowDroid missed several "classical" flows (i.e. leaks to log files or leaks to a network) that the authors' tool identified. The authors found the FlowDroid was not able to identify network-based leaks even though it lists network based communications (i.e. HTTP requests) as sinks. The authors conclude that their analysis is effective, accurate, and scalable.

3.2.5 User-Trigger Profiling in Android Applications via Static Analysis

Recent studies have indicated that there could be hundreds of thousands of malicious Android applications (apps), belonging to over 300 malware families, which could exfiltrate sensitive data, abuse system resources, and disrupt the normal operation of Android devices. This creates a demand for an efficient and precise system which will classify an arbitrary Android app as benign or malicious. In the paper by Elish et al. [14], the authors present a quantitative program analysis approach for detecting malicious Android apps which proved more accurate than prior work in app classification. The authors's approach differs from previous work in that it recognizes legitimate and desirable behavioral patterns in programs instead of identifying malicious patterns. They chose to pursue this avenue because these behavioral patterns commonly occur in benign Android apps but rarely in malicious apps. The authors of this paper do not want to suggest using fewer features in program classification, but instead advocate the usage of benign property enforcement.

The authors begin by defining a few key terms uniquely used in their paper. First, an operation is defined as "an API call which refers to a function call providing system services such as network I/O, file I/O, or telephony services" [14]. A trigger is defined as a user's input or action/event in the app caused by user interaction. A valid dependence path is defined as "a directed dependence path between a trigger and an operation in the data dependence graph" [14] of the app being analyzed. A valid call site is defined as a call site of an operation that has a valid dependence path. The authors

keep track of the number of valid call sites for a certain operation as well as the total number of call sites for the same operation. This data is used to classify an app as benign or malicious.

Before an app can be classified as malicious or benign, the authors start by generating the data dependence graph (DDG) of the unclassified app. In the DDG, intra-procedural dependence edges are identified through local def-use chains while inter-procedural dependence edges are identified through a call-site context-sensitive call graph of the app generated by a points-to analysis. The authors augment this DDG to handle Android specific callbacks and inter-component communications (ICCs) by performing a control-flow analysis on the DDG to find bridges between disjoint graph components. This is necessary in order to capture data dependence relations between triggers and operations across multiple apps and components. This produces a "flow- and context-sensitive data-flow dependence graph with intra- and inter-procedural dependence analysis, and intra- and inter-application Intent based (ICC) analysis" [14]. A reachability analysis is then performed to remove "dead code" from the DDG, which increases the analysis's precision. At this point the DDG is fully constructed, and the authors proceed to identify paths between user triggers and operation calls. For each call site of an operation, the authors perform a backward trace from the call site on the DDG and search for any user triggers on all dependence paths. If the call site of the operation is valid, both the valid call site and total call site numbers are incremented for the operation, otherwise just the total call site number is incremented.

Once the analysis has extracted the two metrics for all the operations used in the app, it then classifies the app as benign or malicious based off these metrics and classification rules. The first rule states that if the total number of valid call sites divided by the total number of call sites is less than a predefined threshold, then the app is classified as malicious. The second rules states that if the distribution of the percentages of valid call sites for the app is above the similarity threshold when compared to the average malware distribution of the percentages of valid call sites, then the app is considered malicious. If both rules state that the app is benign, then the final classification of the app is also benign, otherwise the app is considered malicious.

The authors evaluate their classification tool by running it on 1,433 known Android malware apps as well as 2,684 real-world apps from the Google Play store. When evaluating the known malware, the authors' tool achieved an accuracy rate of 97.9 percent (i.e. a 2.1 percent false-negative rate). When evaluating the real-world Android apps, the authors found that their tool achieved a 2 percent false-positive rate. The average processing time for each app was roughly 158.01 seconds on a standard desktop PC. The authors' tool was able to detect new Android malware that went undetected by the popular VirusTotal security scanner. The authors conclude that their tool achieves high classification accuracy and that it allows the Android community to stay ahead of malicious Android developers.

Chapter 4

Android Malware Collusion

This chapter specifies the problem that the research in this thesis aims to address. This chapter defines Android malware collusion and the specific attack model our work aims to address. It is important to note that there are many variants of the Android malware collusion threat model that our model does not consider, such as implicit collusion. However, our static-analysis and visualization can be modified to address the other threat models, if desired.

4.1 Problem Definition

The problem we aim to address with our approach is the difficulty associated with post-classification Android app collusion inspection. While Bugiel et al. [8] introduced the notion of app collusion classification, Elish et al. [13] has shown that collusion classification of Android apps is possible with a relatively high level of accuracy. However, these approaches do allow for the existence of false-positive and false-negative classifications. A false-positive classification occurs when a set of benign apps is classified as colluding, and a false-negative classification occurs when a set of colluding apps is classified as benign. To ensure accuracy when classifying a set of apps, a human analyst must manually inspect the set after the classification has been made. Analyst inspection for security vulnerabilities, such as post-classification inspection for Android collusion, can be tedious, time consuming, and susceptible to human errors, but can be remedied by automated tools [43]. Little to no work has been done to develop solutions which remedy these issues associated with post-classification inspection for app collusion. Ravitch et al. [36] proposed the first post-classification visualization for Android collusion. However, the information they presented to analysts was limited due to their insensitive static analysis and their visualization was a single-layered graph which possessed known usability issues. **Our work aims to make post-classification app collusion inspection a more efficient, accurate, reliable, and user-friendly process for human analysts.**

4.2 Assumed Android Malware Collusion Attack Model

We define Android malware collusion as a set of Android applications which work together to execute a malicious attack on a user's Android device. Specifically, our approach attempts to combat the Android collusion attack model where a set of Android applications, utilizing explicit ICCs to distribute work, executes an attack to either steal sensitive data or abuse system resources. The Android framework features a data container, known as an intent, which allows Android components to communicate with one another. Components within a single app or in two separate apps can use intents to deliver messages or issue commands, such as the command to start a service. Intents are either implicit or explicit. Implicit intents state to which type of component they should be sent, where as explicit intents state the name of the component and app to which they should be delivered.

Android applications can use explicit intents to to directly communicate and perform colluding attacks [30]. Colluding apps which use explicit intents are not difficult to create. Figure 4.1 illustrates an example where two Android applications use an explicit intent (e.g., startService()) to steal a victim's device identification number. By utilizing ICCs, the pair of apps each request a single permission, thus decreasing their respective suspicion level. For example, a user might assume App A cannot divulge the device identification number, because it does not request the INTERNET, or SEND_SMS permissions. However, because App A sent an ICC to App B, it was able to gain access to all the permissions requested by App B. App A can indirectly use App B's permissions to send the identification number to a remote server. The attack model depicted by Figure 4.1, which steals the device ID, is the threat model our tool seeks to address.

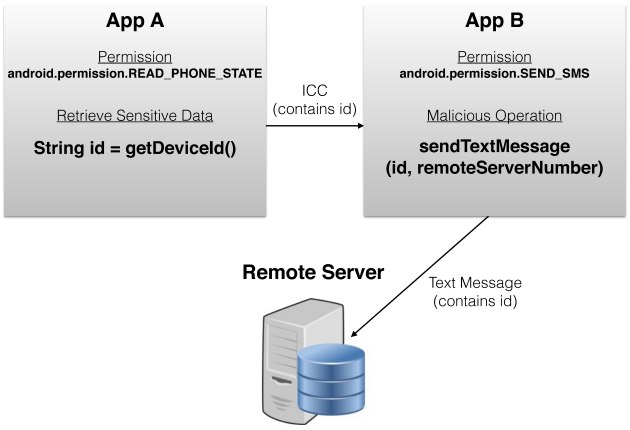

Figure 4.1: App collusion attack model we assume our adversaries use.

It is important to note that app collusion is possible through implicit intents. However this method, termed opportunistic collusion by Elish et al. [13], is unreliable and may not succeed depending on which apps the victim has installed on his or her device. Thus, we assume our adversaries will use explicit intents to achieve app collusion due to their higher probability of success. However, our work can easily be modified to include inspection for opportunistic collusion. Furthermore,

app collusion is also possible through more covert back channels [39]. However, these avenues of attack are excluded from our attack model, but can be incorporated with a more fine-grained static analysis.

Chapter 5

Usable Visualizations for Post-Classification Android Collusion Inspection

This chapter presents our work in developing usable visualizations for post-classification Android collusion inspection. Specifically, we including our design goals and requirements, an overview of the technologies used to create the prototype implementation of our visualization design, detailed descriptions of each view in our approach, and several use cases in which our approach is useful with regard to post-classification Android collusion inspection.

5.1 Goals and Requirements

In this section, we detail the goals and requirements of our visualization tool. The goals and requirements are divided in three categories: security goals, design requirements, and visualization requirements. These goals and requirements should not be viewed as specific features or security guarantees that our approach must incorporate, but rather, a set of guiding principles and high-level concepts that steer the direction of the development of our visualization. We believe this is sufficient as our work in developing multi-leveled Android collusion visualization is the first of its kind.

5.1.1 Security Goals

Our visualization design seeks to fulfill two main security goals: confidentiality, and authorization. It is important to note that the tool itself does not provide these security guarantees, an analyst properly using our tool provides them.

Confidentiality We strive to maintain the confidentiality of sensitive information (i.e. private user data, device information, etc.). Sensitive information can be sent between colluding apps via explicit intents, which could cause a loss of confidentiality. Our tool attempts to alert the user to instances of sensitive information transmission via explicit intents.

Authorization We aim to ensure apps do not exceed their stated permissions through explicit ICCs. Colluding apps could effectively utilize system resources they are not authorized to access by sending explicit ICCs to apps which do possess the necessary authorization. We ensure authorization by displaying all the permissions of the input apps and their respective component permissions, which in effect visualizes all occurrences of permission escalation through explicit ICCs.

5.1.2 Design Requirements

When designing the structure of our visualization tool, we identified a set of requirements our tool must fulfill in order to prevent undetected collusive attacks:

- **Rigor**: Our tool needs to be rigorous when identifying ICCs and presenting information to the user.
- **Relevance**: Our tool must only display information necessary to identify Android malware collusion. Due to the complexity of the collusion attack model, any extraneous information could prove overwhelming to the analyst.
- **Simplicity**: Our tool must simplify the complex network of ICCs between a set of collaborating apps. This is one of the distinguishing characteristics of our work, as prior visualizations of app communication networks have been overwhelming to their analysts.

5.1.3 Visualization Requirements

When designing the user interface of our tool, we identified a set of requirements our user interface must fulfill:

- **Minimalism**: Our tool must conform to a minimalistic style at all times. The interface should only display the least amount of information necessary at any given point, in order to avoid overwhelming the analyst.
- **Efficient Visual Differentiation**: Our tool must enable the analyst to visually differentiate between benign and malicious ICCs. This must be achieved, if possible, through mediums other than text. The analyst must be able to simultaneously evaluate large numbers of ICCs without becoming overwhelmed or mentally fatigued.

- **ICC Comprehensive**: Our tool must be able to display all ICCs between two communicating apps. If not, analysts cannot make a definitive classification of a set of communicating apps.

5.2 Usable Design for Cross-App ICC Visualization

In this section, we present and detail the design and features of our approach to post-classification collusion visualization. Our high-level approach is to divide the complex network of collusion information into several hierarchical levels. Each level presents the analyst with different types of collusion information. We developed a prototype implementation of our design as a client-server web application which takes a set of Android applications as input and displays directed graphs of the input's ICCs at four different levels; the application communication, the ICC, the component, and the intra-component flow levels. At each level, detailed information about collusive data flows are displayed to the user. We use a context- and flow-sensitive static analysis-based collusion detection framework to generate the data which our tool visualizes [15]. Our tool is designed to accommodate a varying number of input apps and their respective ICCs.

We illustrate our visualization interface through several proof-of-concept colluding app sets developed by our research group. We had to use these sets due to lack of real-world colluding application sets.

5.2.1 Prototype Architecture and Implementation

Our web application consists of three subsystems; the static analysis framework, the communication graph generation module, and the view resolver. Figure 5.1 illustrates the high level overview of our web-application's architecture. We used the Spring Framework [2] to implement our web application.

Static Analysis Framework

To begin the process of post-classification inspection, the analyst must upload the Java byte-code and Android manifest files for each Android app in the input set. Currently, the analyst must run the DARE tool and Apktool on each input app's apk archive client side to extract the necessary information and then package the output into a zip archive [1, 33]. However, our tool could be modified to perform the DARE and Apktool extraction server side, so the analyst need only upload a zip archive containing the apk archives of each app.

After the archive has been uploaded to the server, our tool performs static analysis-based collusion classification, based on the system presented by Elish et al. [13], to identify and assess the

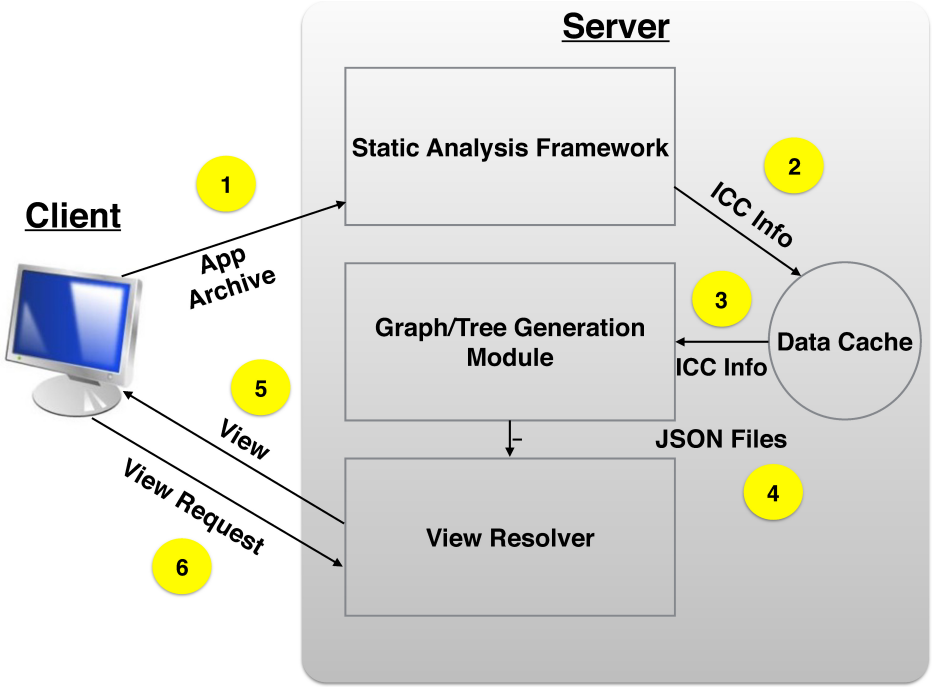

Figure 5.1: Overview of the architecture of our visualization tool.

threat level of each communication flow between input apps. We utilized this particular classification system due to its state-of-the-art static analysis techniques and fine-grained ICC classification system. Once our tool completes its static analysis and classification, it caches all relevant ICC information in a temporary data cache. This static analysis produces all the information that our visualization approach renders and allows the user to interact with.

Graph and Tree Generation Module

Once the ICC information is cached, we generate all ICC graphs between each of the input apps. Each ICC graph illustrates the flow of ICCs between the components of two communicating input apps. App components are rendered in node clusters, where each individual node represents an app component with a label containing the component's name. Nodes are clustered according to which app the component belongs. Individual ICCs are represented by directed edges from a node in one of the clusters to a node in the other cluster. We use the D3.js library to generate all visual graphs displayed by our tool. An ICC graph is generated for each instance of communicating input apps. For example, assume our input set contains apps A, B, and C. App A sends ICCs to Apps B and C, while app B sends ICCs to app C. Our tool would generate three ICC graphs for ICCs from A to B, A to C, and B to C.

After our tool generates a full set of ICC graphs, it then constructs a single app communication graph from the set. The app communication graph illustrates the communication flows between

the input applications. Nodes in the app communication graph represent input apps while directed edges between nodes represent flows of ICCs between the source and target apps. Using the previous example, the app communication graph would contain nodes labeled A, B, and C representing their respective input apps, with directed edges from A to both B and C and from B to C, as illustrated by Figure 5.2.

Once all ICC and app communication graphs are generated, our tool then generates an an exhaustive list of all components for each app in the input set. This is done to allow for exploration of the various collusive data flows within each component. Finally, the module generates the collusive data flow trees for each component. These trees detail all data flows that currently are and could be used for app collusion. A tree is generated for each component of each app. The root of each tree is the component itself, and each child of the root is a potentially collusive dataflow within the component.

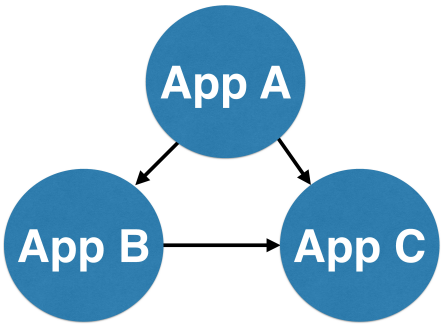

Figure 5.2: Example of our app communication graph.

View Resolver

Once graph and tree generation is complete, the view resolver presents the user with the App Communication view, detailed in Subsection 5.2.2. The view resolver allows the user to control the information that is displayed on the webpage. The user interacts with the view resolver in order to conduct post-classification inspection for Android collusion. At this point, our web-application has completed all static analysis and information cacheing required to allow an analyst to freely interact with the view resolver; no further computation, graph generation, or static analysis is required. Views are organized in a hierarchical format which aims to allow users to start at a broad, high-level overview of collusion information and refine their way down to more fine-grained inspection. Figure 5.3 features a flowchart that displays the direction in which users can explore views.

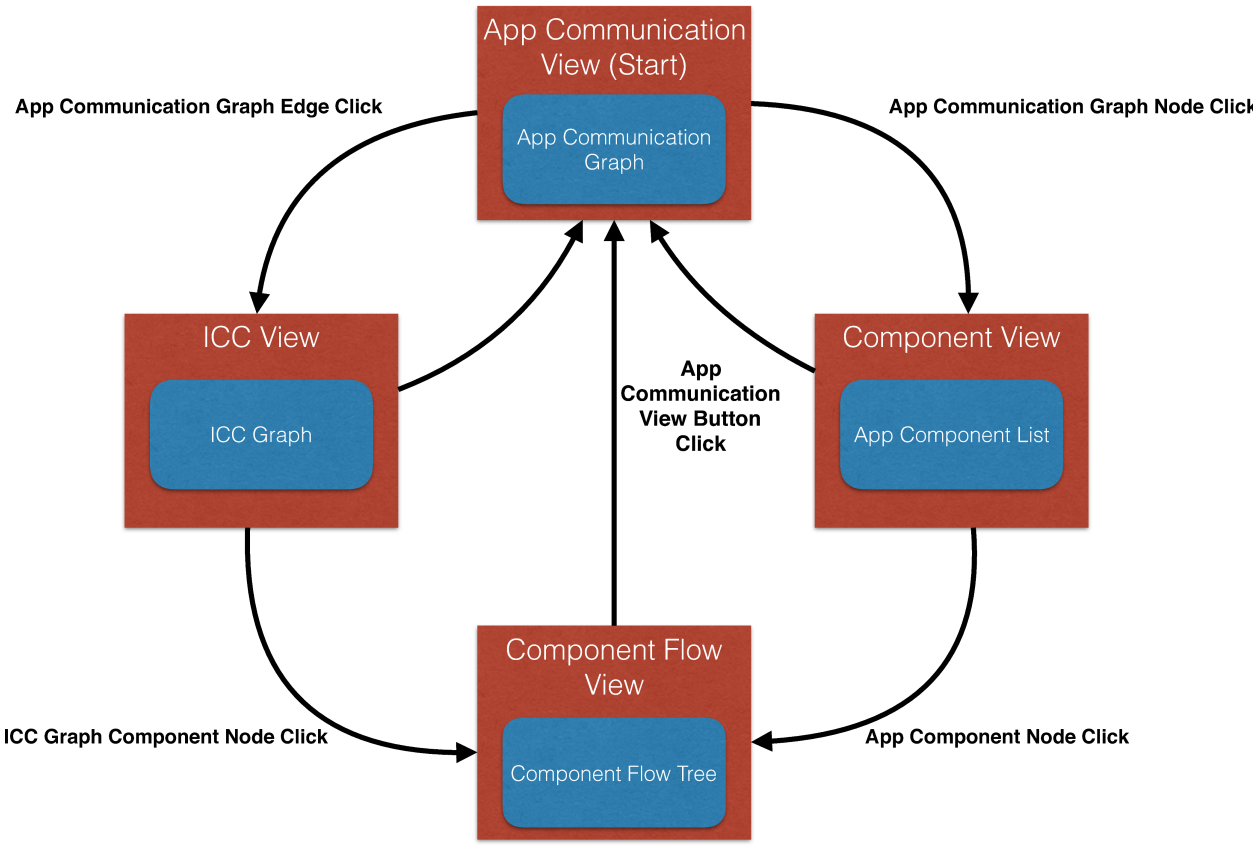

Figure 5.3: A flow chart that shows how users can navigate through the views in our design.

5.2.2 App Communication View

In this subsection, we detail the various features and information present in the app communication view. The goal of this view is to simplify the complex communication network which results from numerous ICCs passed between input applications. This view presents high level communication information between input apps through the app communication graph, first presented in Subsection 5.2.1, and an optional information box containing statistical data on a ICC flow edge. The information appears when a user hovers the mouse pointer over a communication edge. The app communication graph is a fully interactable web feature with which the analyst can control the data displayed on the web page. If the user finds the graph cluttered or unreadable, the nodes in the graph can be clicked and dragged to rearrange the graph to be more readable. Figure 5.4 provides an example of our app communication view where an input set, composed of two proof-of-concept apps and 3 real-world apps, executes a collusive attack.

In Figure 5.4, a set of five proof-of-concept and real-world apps have several medium risk communication flows, rendered as orange arrows, and one high risk flow, rendered as a red arrow. The graph visualizes the communication flows between the apps. While it is not displayed in this figure, the information box will appear when the user overs over a directed edge. Figure 5.5 provides

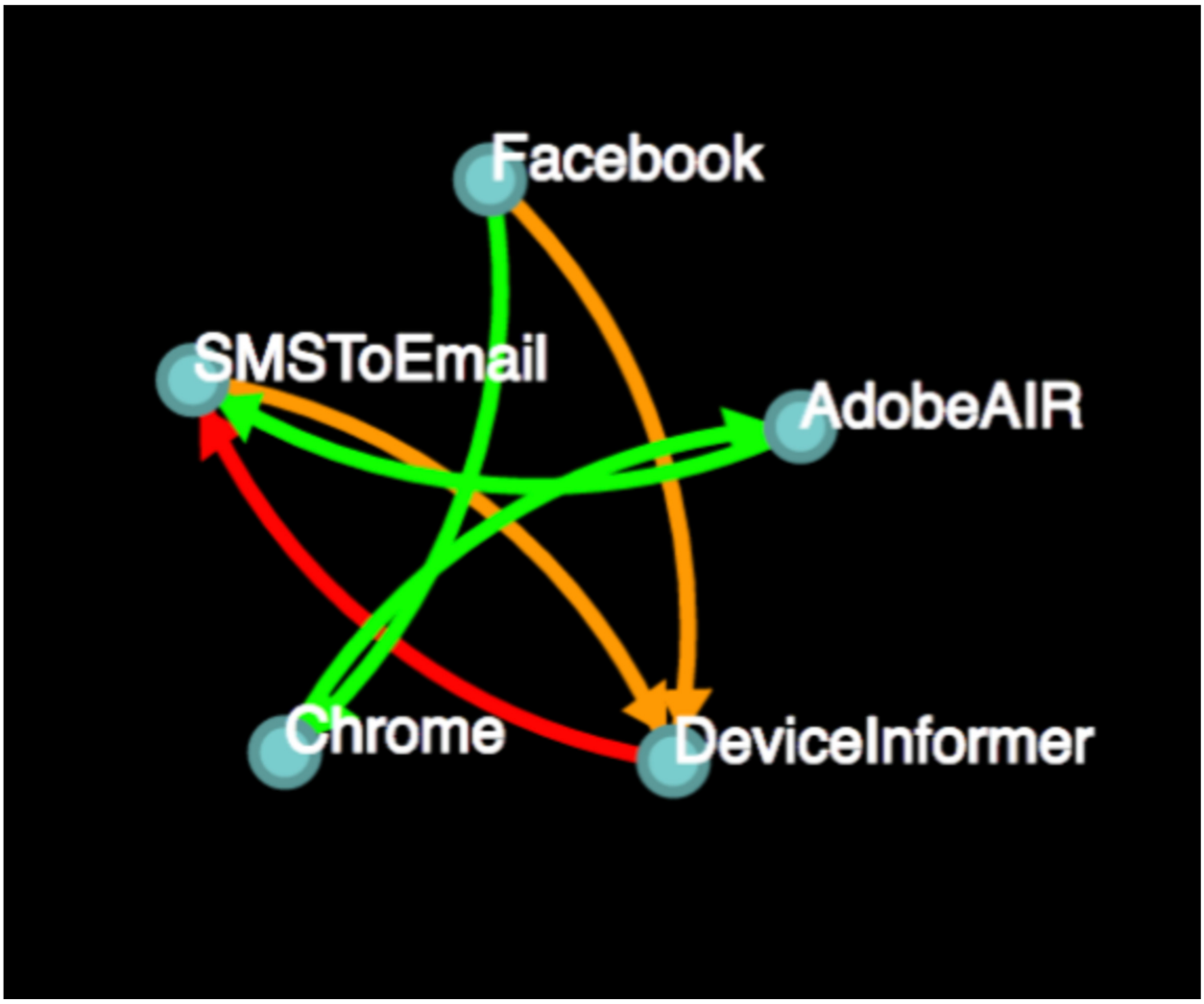

Figure 5.4: App communication view visualizing five colluding apps.

an example of the app communication view with a user hovering over a flow edge. It is important to note that whenever an edge is hovered over, all other edges disappear. This feature was included so that the user can easily verify which exact edge they are viewing information for.

While prior examples have contained a small number of apps, the app communication view is capable of visualizing a large number of communicating apps without becoming overwhelming or confusing to the user. Figure 5.6 provides an example of the app communication view which is visualizing a large number of communicating apps.

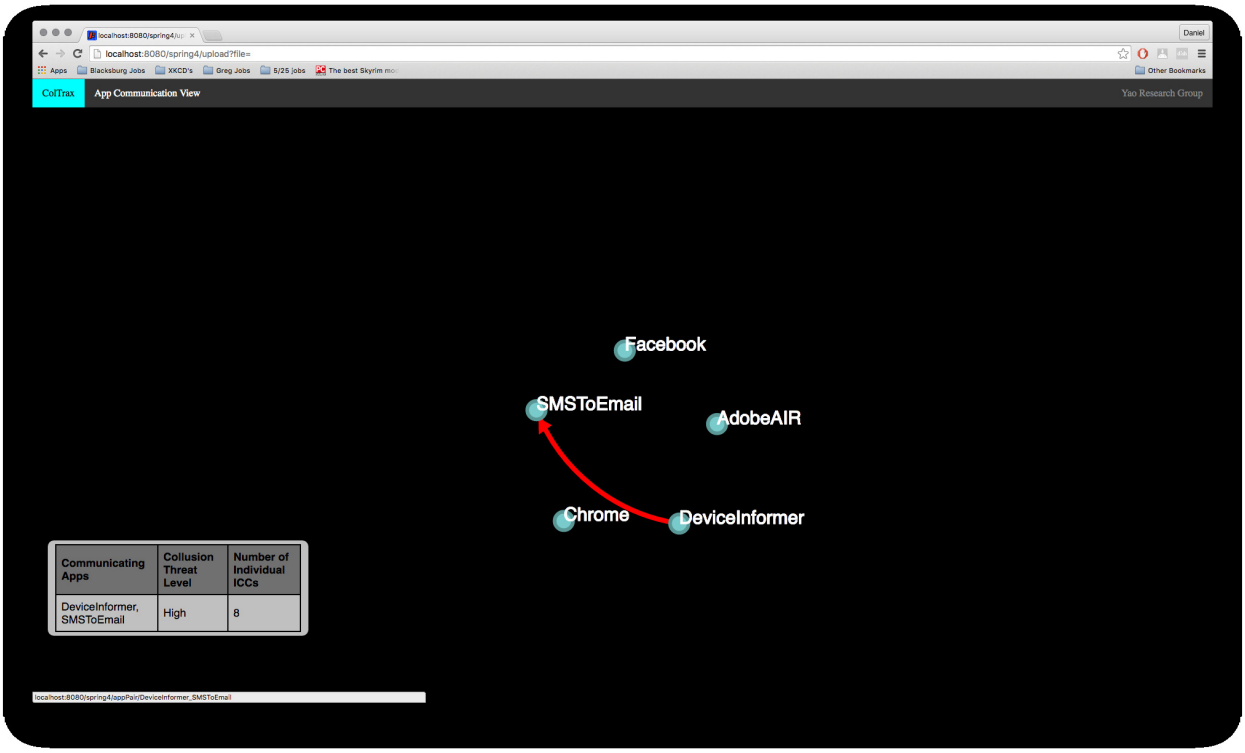

Figure 5.5: App communication view visualizing five colluding apps with the ICC flow information box visible.

App Communication Graph

Recent literature has shown that directed and undirected graphs are an effective medium when visualizing an instance of multiple communicating entities [17, 24, 28, 35, 48]. Because Android malware collusion involves at least two communicating apps, possibly many more (e.g. multiple communicating entities), we decided to visualize the communication flows as a directed graph. Visualizing inter-app communication flows in a layout which is rigorous and user-friendly was an established challenge due to its complex nature [36]. However, recent literature has also shown that incremental visualization is effective when visualizing increasingly complex environments [5, 40]. We solved the problem of inter-app communication flow visualization by separating the visualization into two levels: the app communication graph and the ICC graph.

The top level, the app communication graph is the focus of the app communication view. The graph has two visible features, nodes and edges. Figure 5.5 displays an instance of the app communication view featuring an app communication graph above an app communication information box.

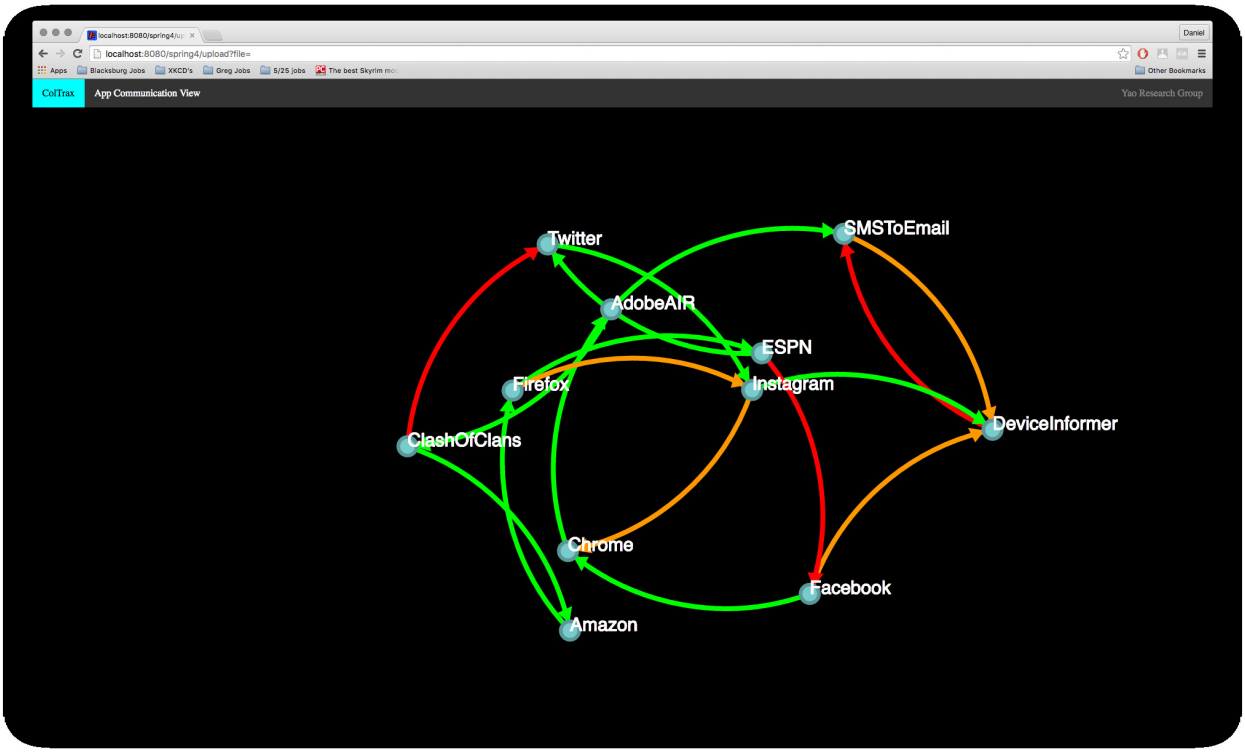

Figure 5.6: App communication view visualizing a large number of colluding apps.

Graph Nodes Each clickable, rectangular node in the app communication graph is labeled with the app's name which it represents. When the analyst hovers the mouse pointer over the node label, an information box appears containing the app's name, its permission set requested in the app's manifest, the number of ICCs it sends, and the number of ICCs it receives. Knowledge of each app's requested permission set is necessary when inspecting apps for collusion [8, 13]. Before a node is clicked, the app communication view only displays the app communication graph. An information box, detailed in 5.2.2 and visible in Figure 5.7, appears once the analyst clicks a node label. In Figure 5.7, the DeviceInformer app is being hovered over and the box contains information pertinent to it. Rendering each input app as a labeled node reduces the strain on the analyst while (s)he simultaneously views the entire set of input apps. Clicking on a node will transition the view to the App Component View.

Graph Edges Edges in the app communication graph appear as color-coded, directed arrows from a source node to a target node. The presence of an edge indicates that at least one ICC flows from a component in the source app to a component in the target app. Each edge is colored according to its threat level, which is determined by our own implementation of the classification system presented by Elish et al. [13]. We chose this system due to its low false-positive classification rate and fine-grained rule system.

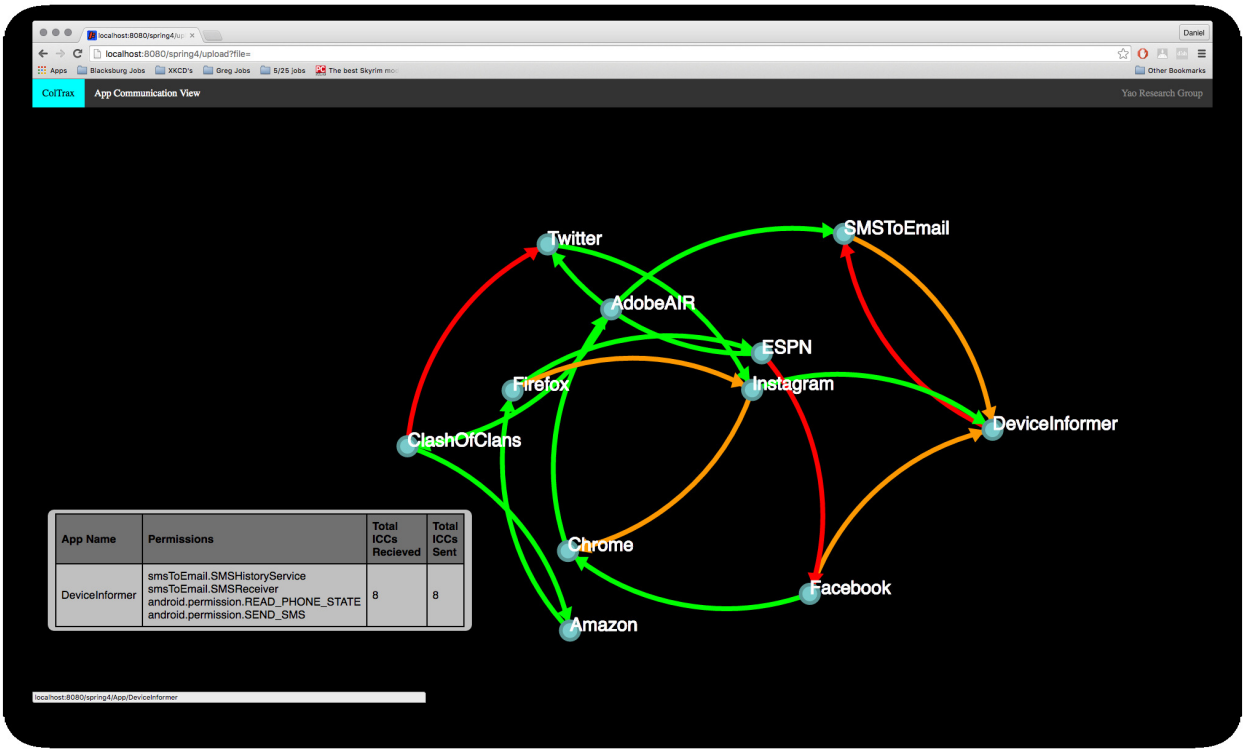

Figure 5.7: App communication view visualizing a large number of colluding apps along with an app information box.

A flow's threat level is determined by its riskiest ICC. For example, a flow between two apps containing low, medium, and high risk ICCs would be classified as high risk. *Low risk* flows appear as *green arrows*, *medium risk* flows appear as *orange arrows*, and *high risk* flows appear as *red arrows*.

Like graph nodes, each edge also causes an information box to appear which contains the the name of the source and target apps, the edge's threat level as determined by the classification system, and the total number of ICCs sent from source to target. Clicking an edge transitions the view from the app communication view to the ICC view, detailed in Section 5.2.3. Our usage of directed edges allows the analyst to simultaneously view all communication flows between input apps. Furthermore, the threat level classification system allows the analyst to quickly identify which individual flows in the complex communication network may be collusive via edge color.

The app communication graph effectively encapsulates large groups of communicating app components and ICCs into individual nodes and edges. The layout and features of the graph enable the analyst to concurrently view the communication network between all input apps and identify areas where collusion could occur. Because all communication flows are rendered and color coded, the risk of an undetected collusive attack decreases.

App Communication and ICC Flow Information Boxes

The information box in the app communication view becomes visible beneath the app communication graph when the analyst overs over a node or a directed edge. The box contains information regarding the app/ICC flow which the hovered node/edge represents. Table 5.1 details the information fields present in the app communication information box while Table 5.2 details the information fields present in the data flow information box. Figure 5.7 illustrates the app communication view when the app information box is visible beneath the app communication graph while Figure 5.5 illustrates the app communication view when the ICC flow information box is visible beneath the app communication graph.

Table 5.1: App communication information box fields and descriptions.

Info. Box Field	Description
App Name	The name of the app
Permissions	Set of requested permissions by the app
Total ICCs Received	Total number of ICCs received by the app from all other apps
Total ICCs Sent	Total number of ICCs sent by the app to all other apps

Table 5.2: ICC flow information box fields and descriptions.

Info. Box Field	Description
Communicating Apps	The name of the source and target apps, separated by a comma
Collusion Threat Level	The collusive threat level of the specific flow
Number of Individual ICCs	Total number of ICCs in this particular flow

The statistical information in the box, along with edge tooltips, allows the analyst to compare the number of ICCs exchanged between a pair of communicating apps to the total number of ICCs sent or received by the selected app. This allows the analyst to identify which apps collaborate more frequently. Furthermore, by hovering the mouse pointer over a source app and then the target app, the analyst can compare permission sets of the communicating apps. This allows the analyst to identify communication flows which enable the escalation of app permissions. By hiding the information box until the analyst hovers over a node or edge, the app communication view ensures the analyst is not overwhelmed by superfluous information.

5.2.3 Inter-Component Communication (ICC) View

In this subsection, we detail the various features and information present in the ICC view, which can only be reached by clicking an edge in the app communication graph. The goal of the ICC view is to provide the analyst with a visualization of all ICCs between the source and target apps of the

selected app communication edge. The ICC view provides in-depth information for specific ICCs by displaying the ICC graph of the source and target apps along with two information boxes. The information boxes in the ICC view are static while the ICC graph is a fully interactable web feature that provides control over the displayed data, much like the app communication view. Figure 5.8 provides an example of our ICC view where the DeviceInformer and SMSToEmail apps exchange low, medium, and high risk ICCs. In this figure, only the ICC graph is visible.

Figure 5.8: ICC view with a medium number of ICCs being exchanged between two apps.

ICC Graph

The ICC graph is the bottom of the two visualization levels introduced in 5.2.2 which combine to overcome the challenge of simplifying the complex network of ICCs between a set of communicating apps. The ICC graph, which is the focus of the ICC view, contains node clusters composed of individual nodes and edges which connect nodes from different clusters.

Graph Nodes and Node Clusters ICC graph nodes, clustered according to which app they compose, represent individual app components. The app name for a cluster of nodes appears above all other (darker) nodes. Hovering over an app node reveals the app's permission set in an information box, as shown in Figure 5.9 where the DeviceInformer app is being hovered over. Component

nodes are darker than app nodes and are labeled with the component's name. They also display information boxes which display the component's requested permission set. Component permission sets specify which permission(s) an incoming ICC's source app must possess, or else the target component ignores the communication. Unprotected components can indicate app collusion [13], thus we include component permission sets in node tooltips. Unlike the app communication graph, ICC graph nodes are not clickable. Rendering app components as nodes allows the analyst to simultaneously view all components which may collude from two communicating apps in a layout, which reduces cognitive load (e.g. is not overwhelming). Figure 5.10 features an example of the ICC view where a permission protected component is being hovered over while Figure 5.11 features an example of the ICC view where an unprotected component is being hovered over.

Figure 5.9: ICC view where an app node is being hovered over.

Graph Edges Similar to the app communication graph edges, ICC graph edges also appear as color-coded edges from a source node within a node cluster to a target node in a different cluster. An edge appears for each individual ICC sent and its threat level is determined by our implementation of the ICC classification system introduced by Elish et al. [13]. ICC graph edges use the same color scheme detailed in Subsection 5.2.2 to color low, medium, and high threat ICCs.

ICC graphs may contain many edges which can result in multiple edge intersections. This increases graph clutter and may cause an analyst to erroneously select an edge. Minimizing the number of edge intersections is a known NP-hard problem [9] and is not the focus of this paper. To address

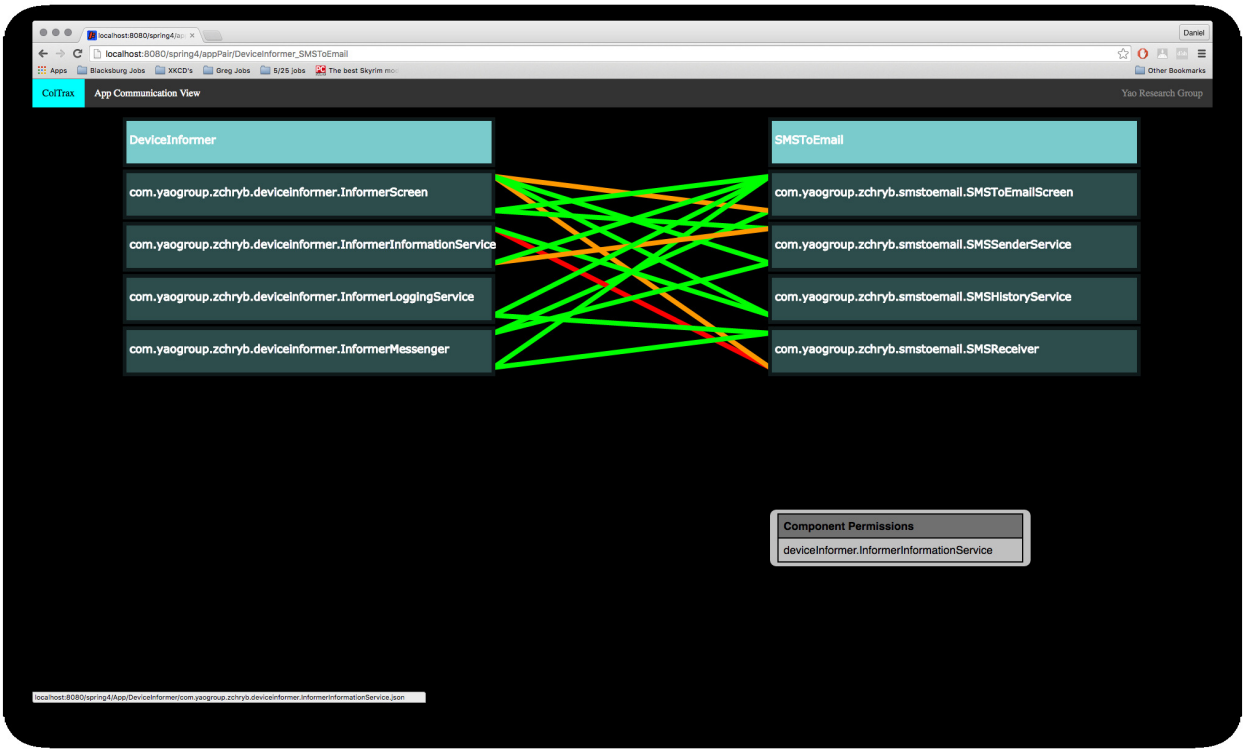

Figure 5.10: ICC view where a protected component is being hovered over.

this issue, we hide all edges that are not being hovered over. Hovering over an edge edge selects the ICC it represents for inspection, and reveals or updates the ICC information box, but does not transition the view. Figure 5.8 features an ICC graph before an edge is selected for inspection while Figure 5.12 features an ICC graph after and edge has been hovered over. Our ICC graph allows the analyst to simultaneously view all ICCs between two communicating input apps.

The threat level classification system coupled with graph edges allows the analyst to near instantaneously identify suspicious ICCs via color, which is a defining aspect of our approach from prior work in collusion visualization. The ICC graph visualizes the complex communication network of ICCs transmitted between two communicating apps. While it is possible that specific instances of our ICC graph may contain many ICC edges resulting in an increase of graph complexity and clutter, the clustering system, tooltips, and color scheme enables the analyst to efficiently, accurately, and reliably identify suspicious ICCs. Figure 5.8 features an ICC graph with a moderate number of ICC edges where as Figure 5.13 features an ICC graph with high number of ICC edges.

ICC and Permissions Information Boxes

Two information boxes can appear beneath the ICC graph; the ICC information box and the permissions information box. The ICC box becomes visible after a ICC edge is hovered over and

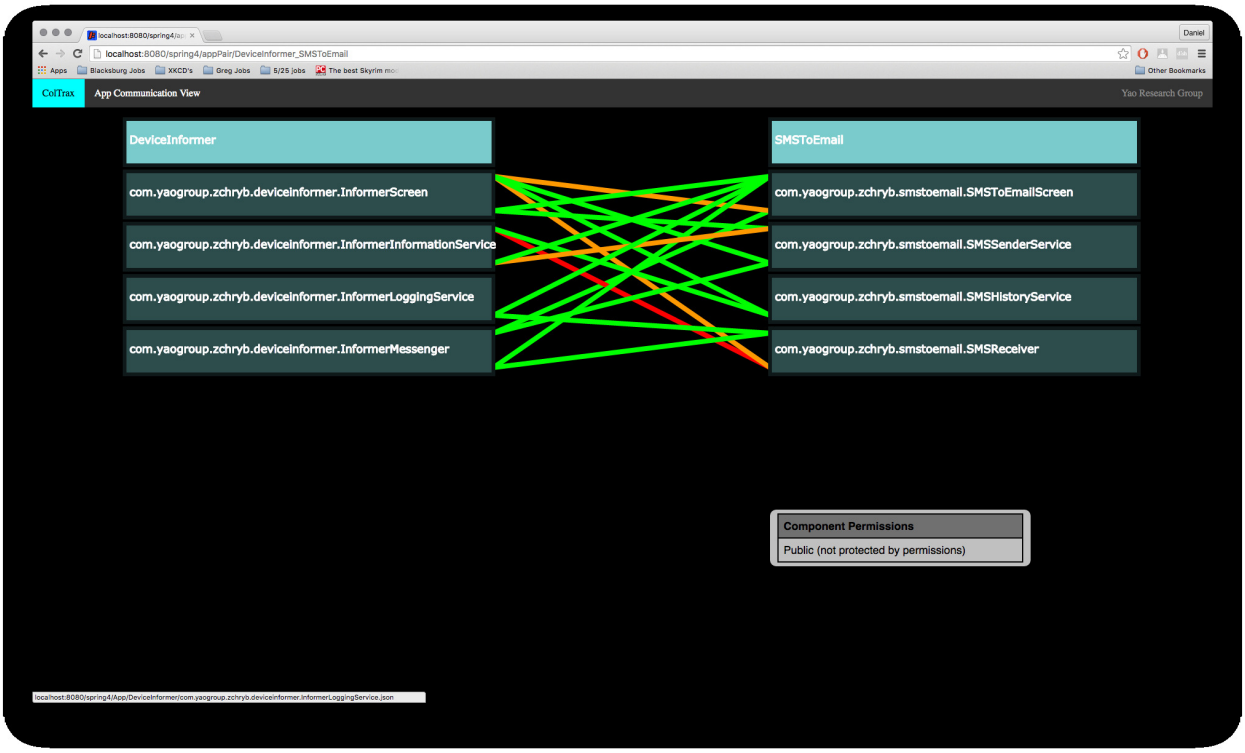

Figure 5.11: ICC view where an unprotected component is being hovered over.

contains in-depth details about the ICC. The permissions box is visible in the ICC view when an app or component node is hovered over and contains the requested permission set of the app or component.

Permissions Information Boxes The permissions information box contains a single field, which is the set of requested permissions. These box allows the analyst to perform a side-by-side comparison of the requested permission sets of the two communicating apps or components, which allows the analyst to identify possible escalations of privileges through ICCs. Without this information box, the user would have to return to the app communication view to ascertain whether the two apps in the ICC graph could escalate their privileges, which reduces overall usability.

ICC Information Box The ICC information box contains vital collusion-related information on ICCs selected by the user. The information box contains five fields; the Sensitive Data, Sensitive Operation, User Trigger, Exit Point, and Entry Point fields. This box allows the analyst to definitively classify ICC communications as collusive or benign as well as identify false-positive and false-negative classifications. The box is hidden until an analyst selects an ICC edge for inspection in order to adhere to the relevance, simplicity, and minimalism requirements.

Entries in the ICC information box fields adhere to the following format: [class path]: [return type]

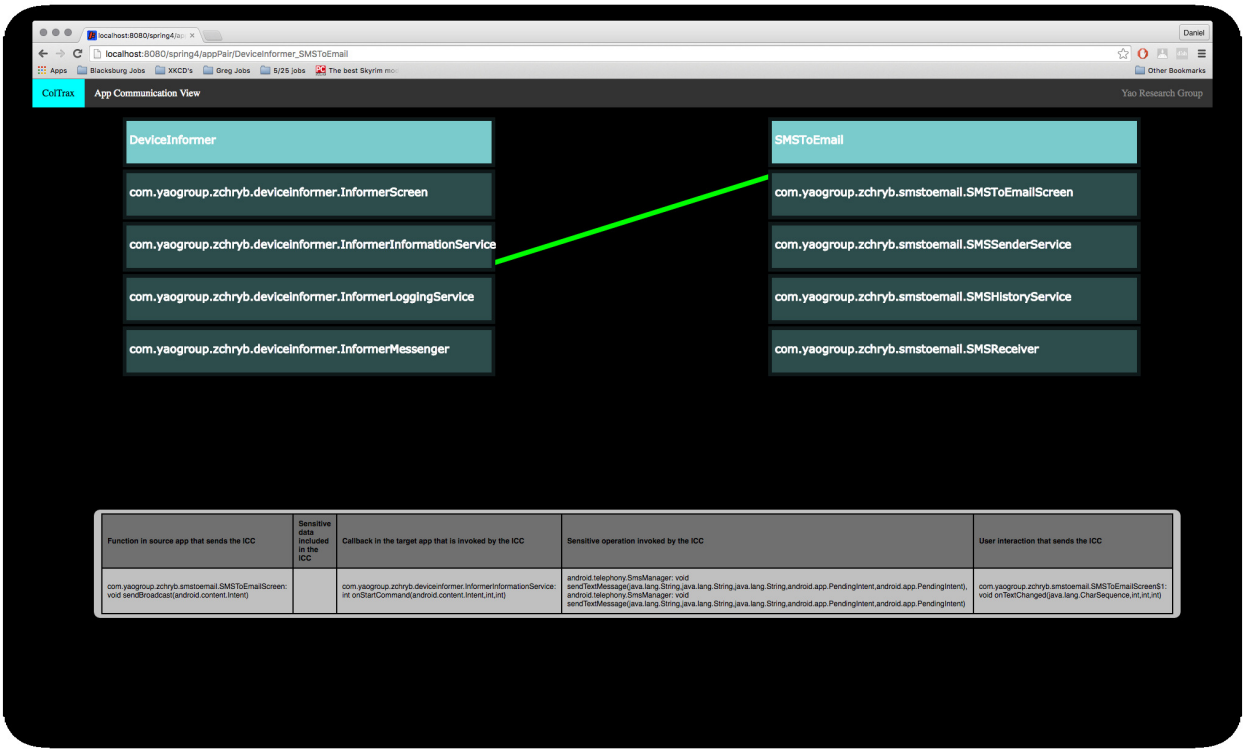

Figure 5.12: ICC view after an edge has been hovered over.

[method name]([method parameters]). The class path is the package and class name in which the method specified by the method name appears. The return type is the type of object the method returns when called and the method parameters are the parameters the method takes when it is called. Figure 5.14 displays an ICC information box, located beneath the ICC graph, with entries in all fields except the user trigger field.

Sensitive Data The sensitive data field presents the analyst with information regarding sensitive information contained in the ICC from the source component to the target component. Each comma-separated entry represents a sensitive API call which was used by the source component to access sensitive information included in the ICC. Multiple entries in the field indicates that several pieces of sensitive information were included in the ICC. This allows the analyst to identify exactly which pieces of information the target component receives from the source component, which can be used to identify app collusion. Figure 5.14 provides an example where a source component has sent the device ID, accessed through the getDeviceId() method, in an ICC to a target component.

Sensitive Operation The sensitive operation field presents the analyst with information related to sensitive system calls used by the target component. Each comma-separated entry represents a sensitive system call used by the target component after it receives the ICC. Furthermore, data from

Figure 5.13: ICC view with a high number of ICC edges.

the ICC, including any sensitive information within it, can flow into the sensitive system operation. Multiple entries in the field indicates that several sensitive system calls were made upon the arrival of the ICC. This field informs the analyst which sensitive system calls are triggered by and receives information from the ICC, which can also be used to identify app collusion. Figure 5.14 provides an example where a target component calls sendTextMessage() twice when it receives the selected ICC.

User Trigger The user trigger field presents the analyst with information related to user triggers which initiate the transmission of a selected ICC. Recent literature indicates that Android events which are not user triggered have a high probability of possessing malicious intent [14, 47]. Each user trigger for an ICC is displayed as a comma-separated entry in the user trigger field. If the user trigger field is blank, the transmission of the ICC is not caused by a user trigger, thus increasing the probability of app collusion. Figure 5.14 provides an example of an ICC which is not user triggered.

Exit and Entry Points While our tool provides a reliable and comprehensive visualization of app collusion via static analysis, some analysts prefer manual inspection [43]. If an analyst wishes to manually inspect an app's source code for collusion, our tool provides the analyst with the exact

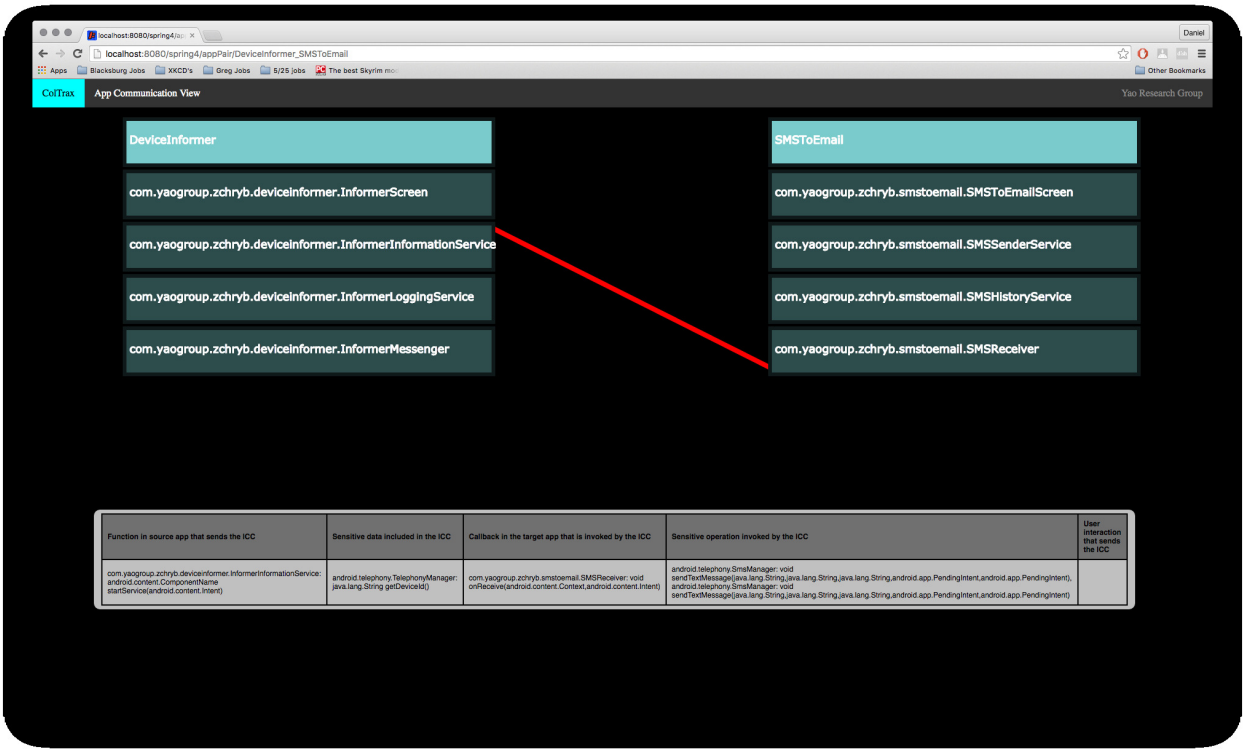

Figure 5.14: ICC view with a high threat ICC selected for inspection.

method signatures, in the exit point and entry point fields, from which the ICC is sent and received in the source code. This allows the analyst to pinpoint precisely which segments of source code, in the source and target apps, is executed by the ICC. Figure 5.14 provides an example where an ICC is sent from the startService() method in the source component to the onReceive() method in the target component. Furthermore, these two fields also allow the analyst to identify which component sends the ICC, and which component receives the ICC.

The app communication and ICC views together visualize a complex communication network between a set of apps in a layout which achieves all design and visualization requirements. The visualization allows an analyst to track explicit ICCs across a variably-sized chain of apps, inspect individual communications, gather information about sensitive information and system resource usage, and determine whether a collusive attack is present in app communication flows. Furthermore, the ICC view also allows the analyst to identify false-positive classifications. Figure 5.15 visualizes an instance where the classification system deemed the ICC a medium risk, but in actuality, it is benign.

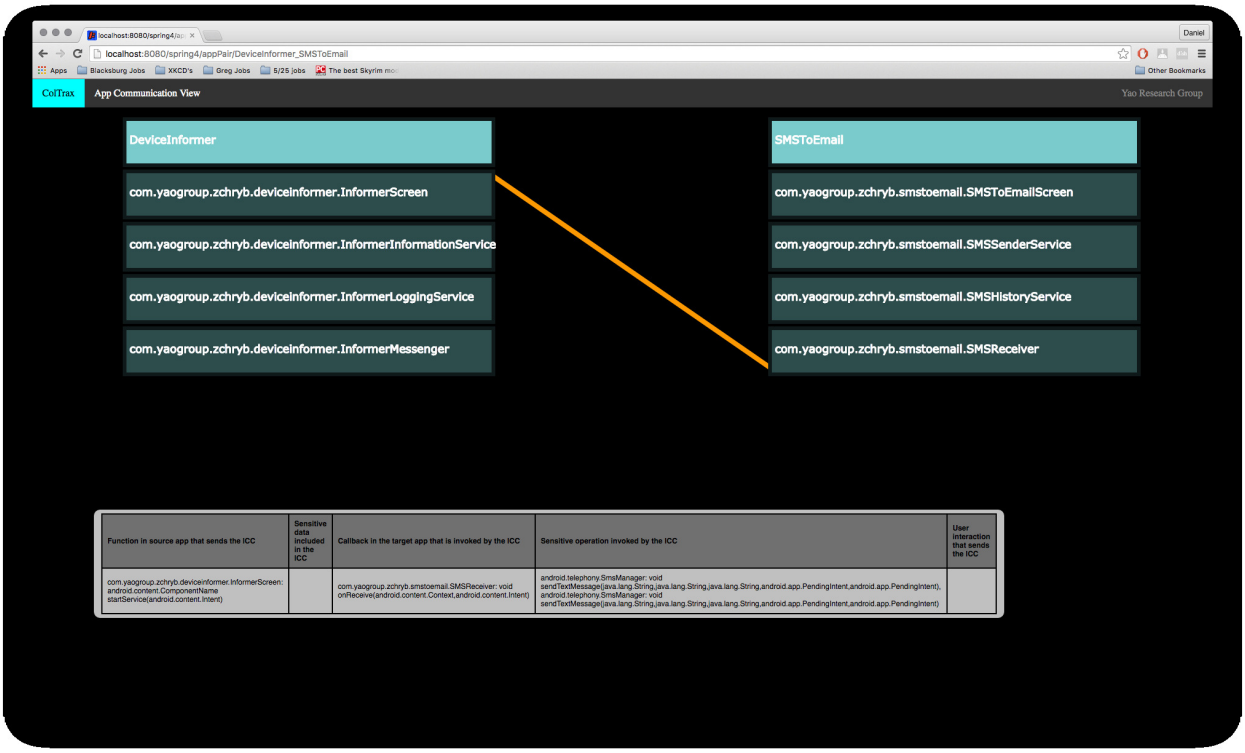

Figure 5.15: ICC view with a false-positive ICC misclassification selected.

5.2.4 App Component View

In this subsection, we detail the various features and information present in the App Component view, which can only be reached by clicking a node in the app communication graph. The goal of the App Component view is to provide the analyst with a visualization of all components within a given app, and allow them to navigate to the specific component they wish to view collusive data-flows in. Each component is rendered as a square box, and the app itself is identified by a box at the top of the view which is a lighter color than the component boxes. Each component and app box is labeled with the name of the app or component. Figure 5.16 provides an example of the App Communication view for the SMSToEmail colluding app.

Each component box is clickable, and will allow the analyst to explore collusive data flows within each component. The reason for including the App Component view is the same reason for the division between the App Communication and ICC views; to simplify a complex visualization through a multi-layered design. Each component within an app could have many collusive data flows, if all data flows were presented to the analyst for all components of an app, the user would be quickly overwhelmed by the amount of information on the screen. Clicking on a component box transitions the view to the Component Data-Flow view, which allows for detailed inspection of collusive data-flows. Thus the app component view allows the user to select specific components which he or she wishes to view data flow information for without being overwhelmed by excessive

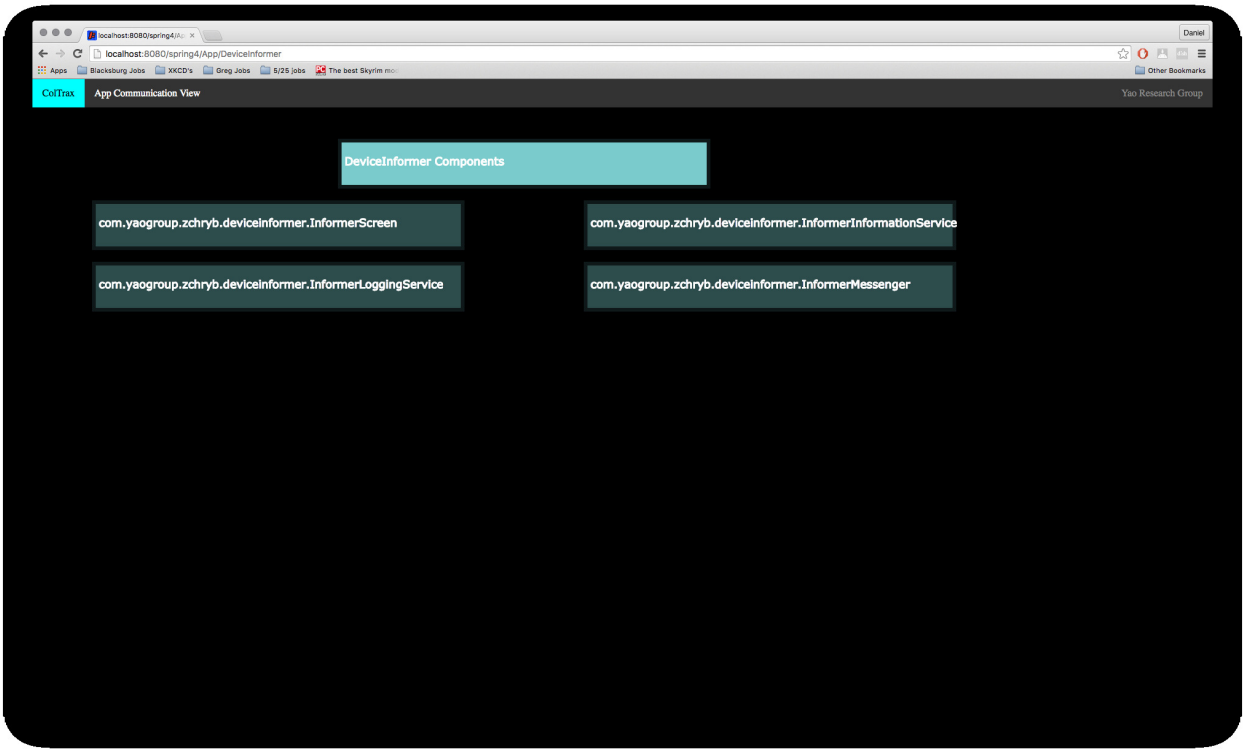

Figure 5.16: Example of the App Communication view.

amounts of information.

Through our user study, we discovered the current features and information in the App component view is limited, and can be expanded upon. We will discuss new features planned for the App Component view in Chapter 6.

5.2.5 Component Data-Flow View

In this subsection, we detail the various features and information present in the Component Data-Flow view, which can be reached by clicking a component box in the App Component view or a node in the ICC view. The goal of the Component Data-Flow view is to provide the analyst with a visualization of all potentially collusive data-flows within a specific component. The main feature of this view is the data-flow tree, which is a horizontal tree where the component name being the root node, and all potentially collusive data-flows are rendered as children of the root node. Each child of the root node is a summary node with high level information of the potentially collusive data-flow it represents. It is important to note that this view included potentially collusive data-flows that may not be used by ICCs in the app set. This is done so analysts may inspect for future app collusion that may not occur in the current app set. Figure 5.17 provides an example of the Component Data-Flow view for the InformerInformationService within the DeviceInformer app.

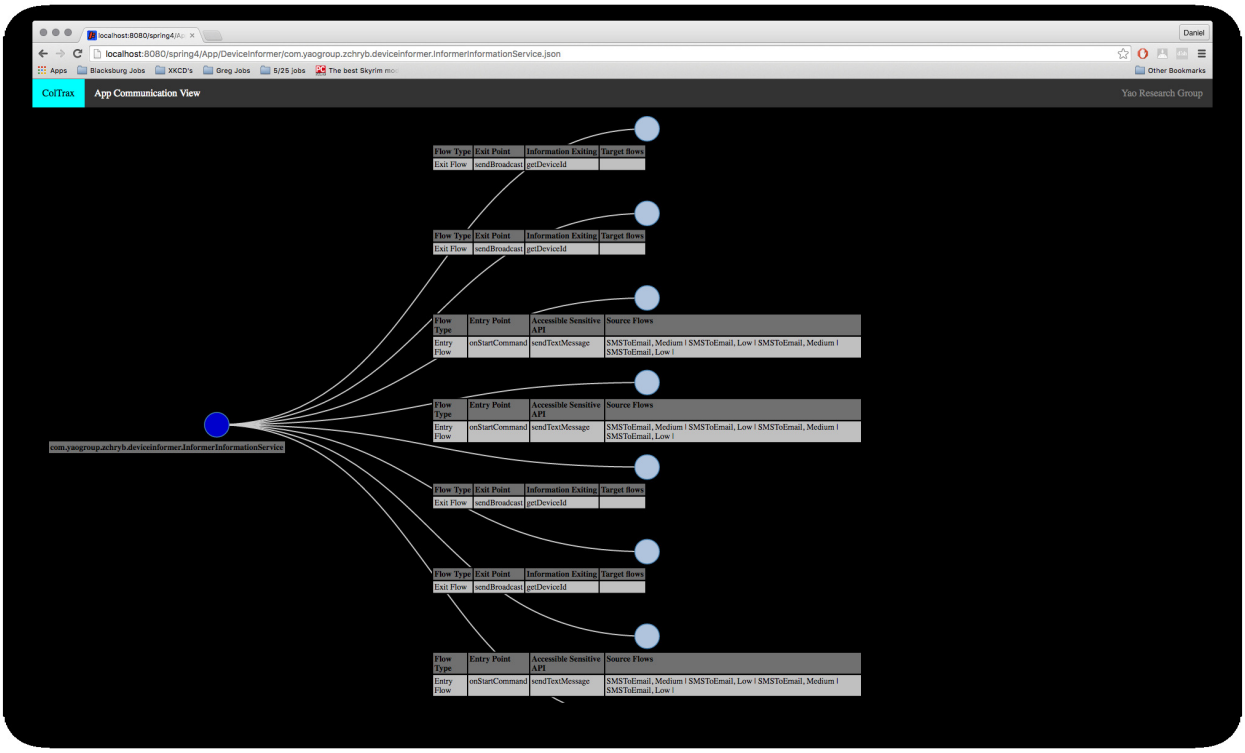

Figure 5.17: Example of the Component Data-Flow view.

Data-Flow Tree Each summary node in the Data-Flow Tree possesses an information box beneath it which possess four fields; the Flow Type, Entry/Exit Point, Accessible Sensitive API, and the Source/Target Flows fields. The Flow Type field can possess one of two values; Entry Flow or Exit Flow. An entry flow value means that this data flow allows an ICC to be received by the component and allows data to flow from the ICC to a sensitive API call. A exit flow value means that in the data flow sensitive information is retrieved through a certain API call and eventually flows into an ICC that leaves the component. The Entry/Exit point field provides the exact method signature that allows an ICC to enter/exit the component. The Accessible Sensitive API field provides the signature of the sensitive API call that an Entry/Exit flow uses to provide access to/retrieve sensitive information. Finally, the Source/Target Flows field provides a bar separated list of apps and the associated collusion threat level of the ICC that send ICCs to or receive ICCs from this particular flow. This allows the analyst ascertain which data-flows are used and how likely they are to be collusive. It is important to note that all potentially collusive data-flows are rendered, even if no app in the current app set being analyzed uses it. This is done so that analyst may inspect data-flows that currently are not collusive, buy may be in the future by apps that are not present in the uploaded set.

Clicking on a summary node expands the flow for inspection by rendering the first step in the data flow. Each subsequent click to a flow node expands the data flow by one node until it reaches the end of the flow. Double clicking a node will expand the entire flow to the end, thus eliminating the

need to click each node in the flow chain. Nodes that are light blue still possess children, where as dark blue nodes do not possess any children. A user knows he or she is a the end of a data flow when the last node in the chain is dark blue. Furthermore, each link between nodes possess and arrow indicating the direction of the flow of data between program statements. Figure 5.18 features an example of the Component Data-Flow view where an Exit and Entry flow have each been expanded by one. Figure 5.19 features an example of the Component Data-Flow view where an Exit flow has been fully expanded while Figure 5.20 features the view where an Entry flow has been fully expanded.

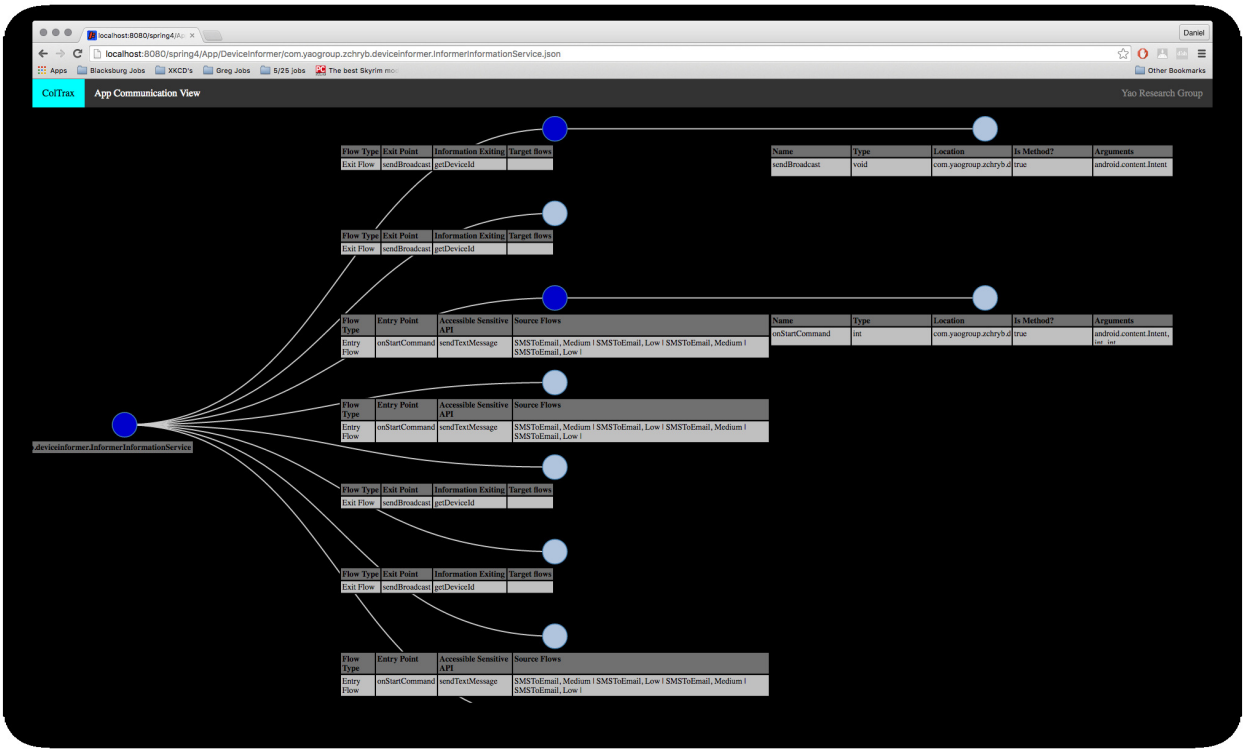

Figure 5.18: Component Data-Flow view where an Exit and Entry flow have been expanded by one.

Each step in a data-flow possesses an information box with five fields: the name, type, location, is method, and arguments fields. The name field provides the name of the method of variable that is represented by this node in the data-flow chain. The type field provides the type of the variable or the return type of the method represented by the node. The "is method" field informs the user whether the node represents a variable or a method call. Finally, the arguments field provides the arguments as a comma separated string of types that are passed to the node, if it is a method. These five fields provide an analyst with the most important information necessary at each step in the data-flow chain to comprehend how data flows through a component.

Figure 5.19: Component Data-Flow view with a fully expanded Exit flow.

5.3 Use Cases

In the following section, we present three use cases in which our visualization tool reduces the challenges associated with post-classification analyst inspection for malware collusion. In particular, we show how the visualization provides vital information to the analyst that (s)he needs to determine if does/does not collusion occur. Because a real-wold instance of malware collusion through ICCs has yet to be discovered, we evaluate our tool using proof-of-concept sets of colluding apps. Our set of collusive apps attempt to resemble real malicious apps as closely as possible.

Escalation of Privileges Android malware collusion can be used to escalate the privileges of a set of colluding apps. To evaluate the effectiveness of our tool, we developed a set of poof-of-concept apps which collude, through explicit ICCs, to achieve a set of permissions higher than each app's stated permissions.

Our source app, named MySocialMedia, represents an app which claims to congregate all information received by the various social media apps installed on the user's device. MySocialMedia only requests the playerInformation permission because it only requires the permissions of the apps from which it collects information, which is SpeedRacer in this case, in order to not appear suspicious. Our target app, named SpeedRacer, represents a game app which allows a user to

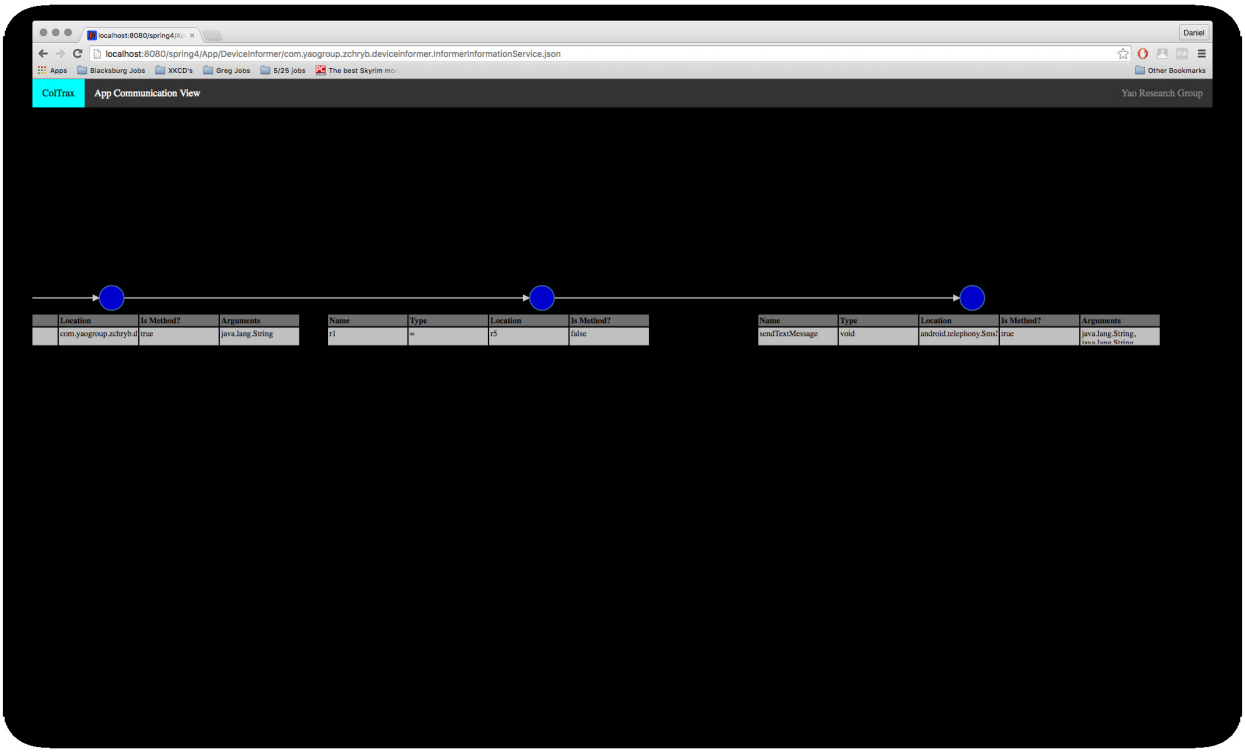

Figure 5.20: Component Data-Flow view with a fully expanded Entry flow.

play a racing game and record the score in a local file. SpeedRacer requests the sendToReceiver, postMessage, and WRITE_EXTERNAL_STORAGE permissions, which are not suspicious given its stated purpose. MySocialMedia sends a malicious ICC to SpeedRacer instructing it to delete a sensitive file, which MySocialMedia cannot accomplish with its native permission set. Our tool assigned the malicious ICC a high threat level and, once the ICC was selected, showed that the source app instructed the target app to delete a file via the ICC information box, thus successfully identifying collusion. Figure 5.21 depicts the ICC view before the malicious edge was selected in the ICC graph while Figure 5.22 features the ICC view after the selection.

Information Theft Theft of sensitive and/or private information can also occur through Android malware collusion. We developed another set of colluding apps which commit private information theft via explicit ICCs. This set of colluding apps steals information by escalating privileges, although privilege escalation is not a requirement for information theft.

Our source app, named SystemMonitor, represents an app which claims to monitor Android system activity. System monitor requests the READ_PHONE_STATE permission which is not suspicious given SystemMonitor's stated purposes. However, READ_PHONE_STATE also provides the SystemMonitor app with access to the victim's device identification number. Our target app, named HealthHelper, represents an app which claims to transmit the user's personal health information to

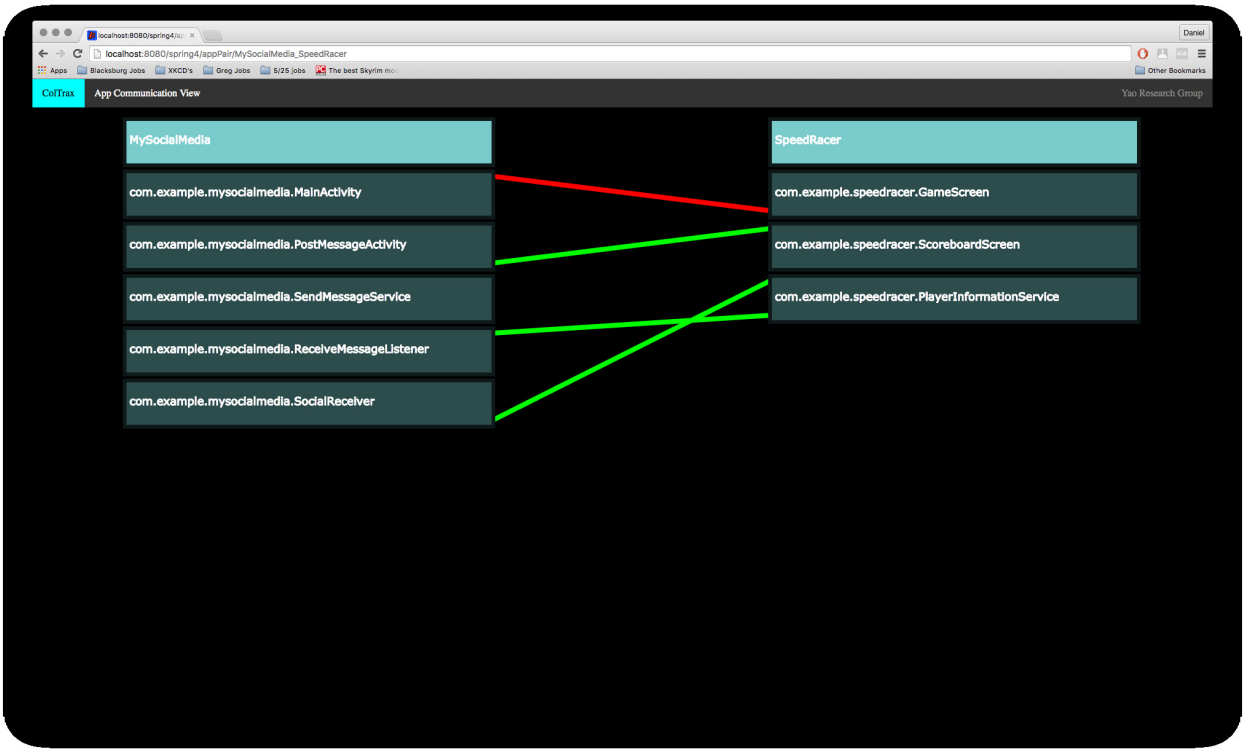

Figure 5.21: Visualization of an ICC achieving an escalation of privileges before the high risk edge has been selected.

specific web servers. HealthHelper requests the ACCESS_NETWORK_STATE and INTERNET permissions, which would not be suspicious for an app which claims to transmit health information via the internet. SystemMonitor sends explicit ICCs containing the device ID to HealthHelper which in turn transmits the ID to a remote server. Our tool classified each collusive ICC as high risk. Each edge was selected, revealing that the device ID was transmitted and subsequently sent to a remote server. Figure 5.23 illustrates the ICC view before a high risk ICC from SystemMonitor was selected while Figure 5.24 illustrates the ICC view after a high risk ICC was selected.

False-Positive/Negative Classifications While our tool effectively visualizes collusive attacks seeking to escalate privileges or commit information theft, it can also can be used to identify false-positive classifications of ICCs which do not execute collusive attacks. To demonstrate, we developed a set of collaborating apps which are classified as collusive, but are actually benign. The source app, named PixelGame, represents a mobile game which claims to transmit a user's score via text message. The target app, named FriendTexter, claims to allow installed apps to send text messages to a list of predefined numbers. The source app sends the user's score in an ICC to the target app, which in turn transmits it via text message. The classification system labeled the ICC as medium risk, however a human analyst understands that this communication is benign due to knowledge of the purpose of the two apps. By selecting the suspicious communication,

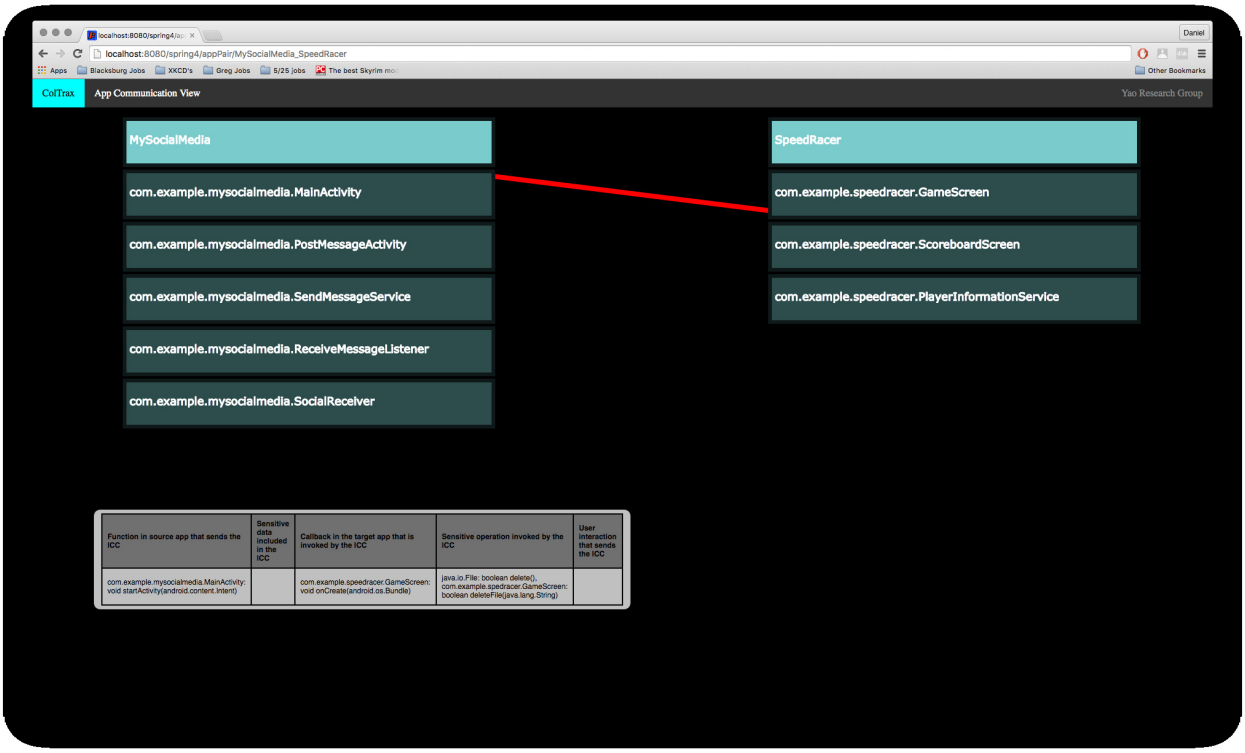

Figure 5.22: Visualization of an ICC achieving an escalation of privileges after the high risk edge has been selected.

the analyst is able to view the contents of the ICC which reveals that no sensitive information is communicated and sensitive system resources are used properly. Figure 5.25 illustrates the ICC view which renders the information contained in the suspicious ICC between PixelGame and FriendTexter. Our tool enabled the analyst to determine that the classification of the medium risk ICC was a false-positive. This approach can also be used to identify false-negative classifications of low risk ICCs.

An analyst can guarantee the security goals of authority and confidentiality as stated in 5.1.1, which escalation of privilege and information theft collusion attacks compromise, as well as identify false-positive/negative classifications of ICCs through proper use of our visualization tool.

Inspection for Future Collusion Attacks Through the Component Flow View, our approach allows analysts to inspect for collusive attacks that could happen to apps present in the app set. As this view visualizes all data-flows within components that could be used for collusion, regardless of wether they are used by the current app set, an analyst could examine each data-flow for their potential for collusion. This way, apps which may be used by future apps for collusion can be identified before they have a chance to execute an attack.

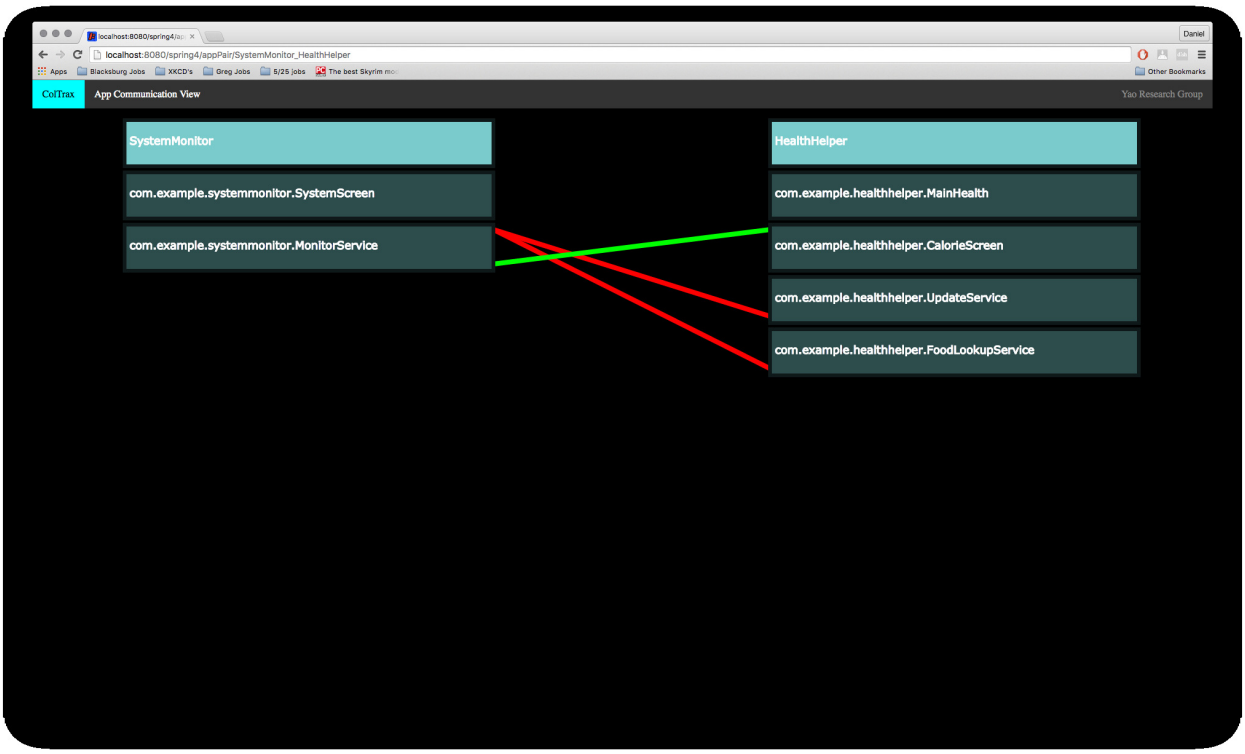

Figure 5.23: Visualization of an ICC committing information theft before an ICC was selected.

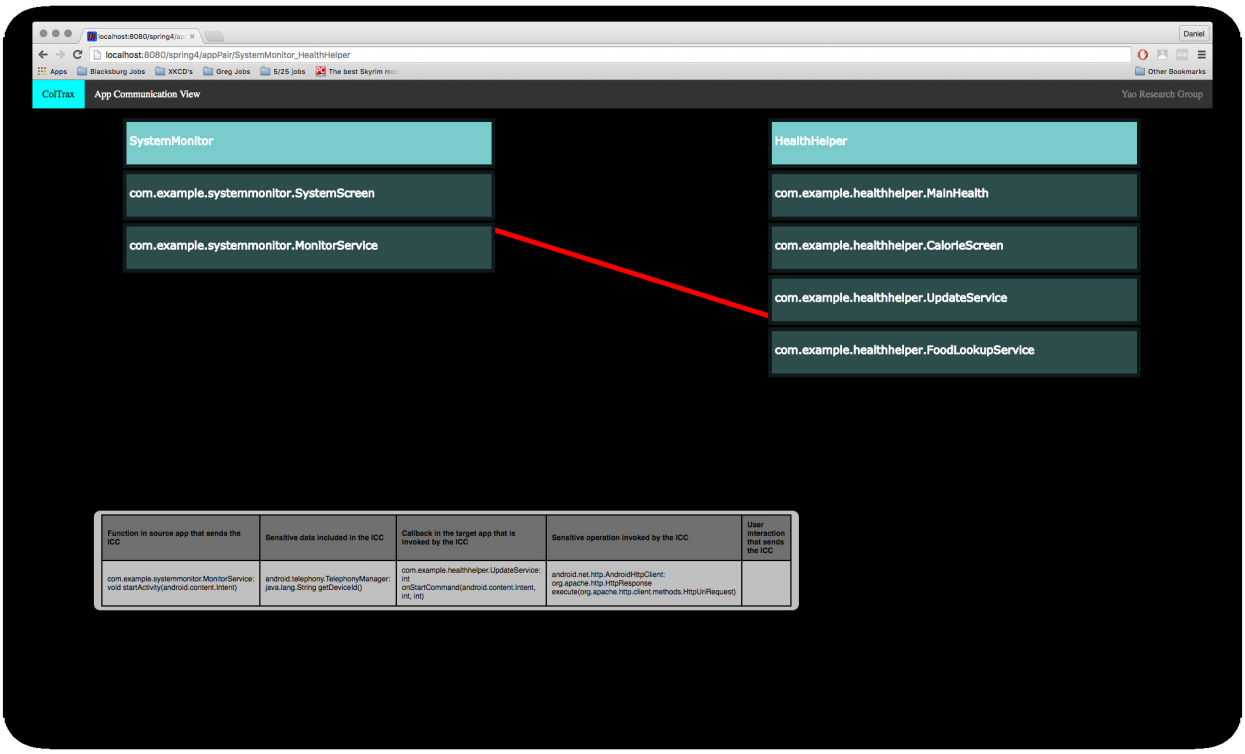

Figure 5.24: Visualization of an ICC committing information theft where the top most high risk edge has been selected.

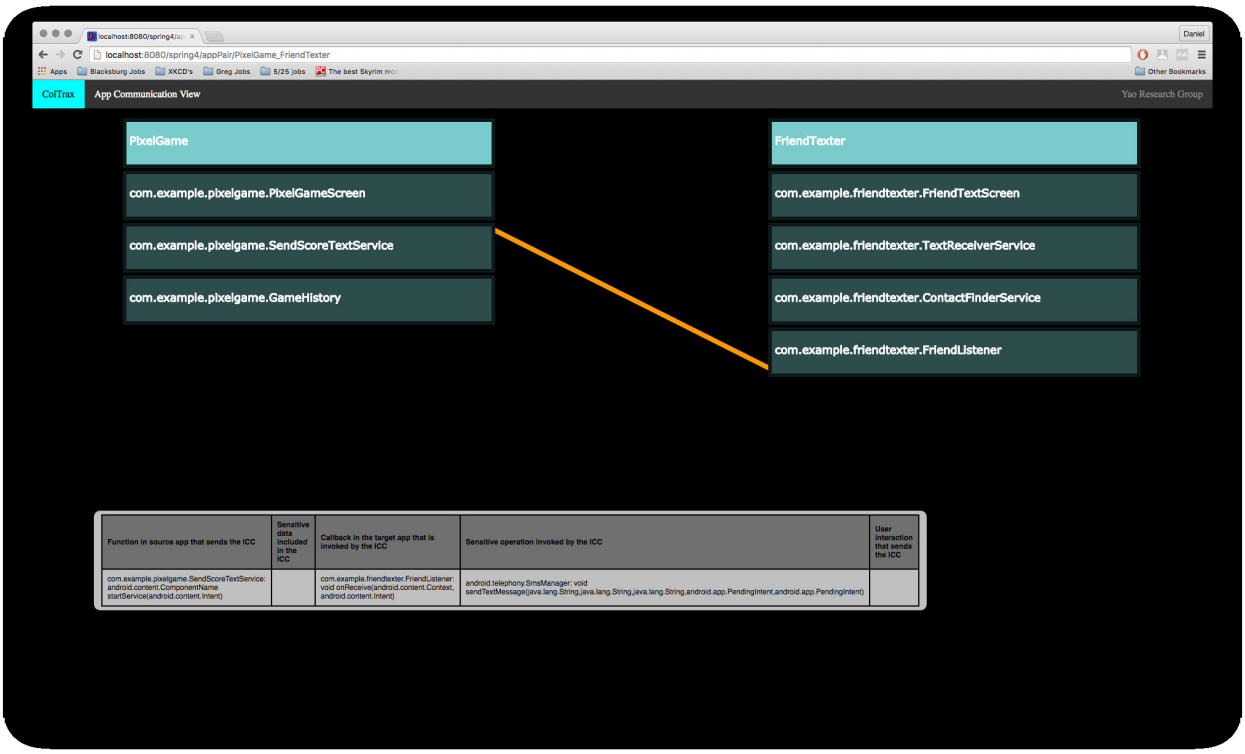

Figure 5.25: Visualization of a false-positive ICC, which appears as the orange edge.

Chapter 6

Evaluation

In this chapter, we present an evaluation of our approach to post-classification visualization for human inspection through an empirical user study. As our principle contribution is the design of a visualization layout, our user study aims to evaluate the usability, user accuracy, and user performance of the participants when they interact with our prototype. We asked the participants to complete a series of tasks that are typically done by security analysts when attempting to detect Android collusion in a set of Android apps. We quantified usability, user accuracy, and user performance through a combination of investigator recorded user metrics (such as the time a user required to complete a task) and various questionaries answered by study participants. The results from our user study indicate that our approach does not possess any glaring usability issues and can be accurately and efficiently used to detect Android malware collusion as well as false-positive collusion alerts.

6.1 User Study Goals

Our user study was designed to achieve two main goals: (1) determine the usability of our design, (2) evaluate user performance and precision while using our prototype.

Usability We wanted to ascertain the overall usability of each view as well as the entire visualization layout. Developing a usable design was a key feature of our research, as prior post-classification visualizations possess high usability. We measured usability by providing the user with two separate questionnaires. The first questionnaire was designed to evaluate the usability of each individual view in our prototype while the second questionnaire evaluates the overall usability of the prototype and all its views. Questions on the questionnaires were either measured on an agreeance scale from 0 (Strongly Disagree) to 4 (Strongly Agree) or were open ended.

User Performance and Precision In addition to usability, we wanted to measure user precision and performance when performing post-classification inspection for Android Malware collusion with our prototype. Ensuring that analysts can quickly and accurately identify Android collusion was a key design requirement when developing our layout. We measured user performance and precision by recording the time needed by users to identify potentially collusive information flows, ICCs, and intra-component data flows. Furthermore, we also recorded the users accuracy when they attempted to identify the collusive and false-positive data flows.

6.2 User Study Design

In this section, we detail the study participants and how they were selected, the methodology we adopted for conducting the user study, as well as the design and approach of the questionaries and efficiency and accuracy metrics.

6.2.1 Study Participants

As our design for post-classification inspection for Android malware collusion is designed for security analysts, we deemed it necessary to be selective in our search for study participants. We chose participants that possessed, what we believe to be, knowledge and/or skills that all security analysts would possess at minimum. It should be noted that we are not claiming that our participants accurately represent our target security analyst users, simply that they possess skills and knowledge that all security analysts must know. We selected participants that had basic knowledge of the Android platform, general security attack model knowledge, and rudimentary data-flow concepts within programs. Adequate knowledge in these three areas is sufficient to properly use our prototype to detect and inspect Android malware collusion, as the visualization eliminates the need for any more expertise. We ensured the users possessed adequate knowledge in these three areas by performing informal interviews before scheduling a user study session.

Android Platform Knowledge We required that our study participants possess a proficient understanding of Android application development as well as the inter-component communication feature of the Android operating system. Without this knowledge, the participants would not understand what is presented to them in the various views, and thus would not be able to properly identify Android malware collusion. All security analysts using our tool would possess this level of understand of the Android platform.

General Security Attack Model Knowledge We required that our study participants possess a basic understanding of attack models developed by malware developers. Many common attack models can be adopted by malware writers, and distributed amongst a set of apps to achieve a

collusive attack. Knowledge of these attack models are necessary to spot a collusive attack. Our target user group, security analysts, will also possess knowledge of basic attack models.

Collusion Data-Flow Concepts We required that our study participants possess a rudimentary understanding of what data-flow within a program is. This is necessary when using our tool to detect collusion, as many of the views visualize data flows and directions. While security analysts may not possess in-depth data flow concept knowledge, they must have at least a rudimentary understanding of it to conduct manual inspection for Android collusion. However, we could not expect our study participants to properly identify Android collusion with the above listed knowledge alone. Thus we provided the participants with the criteria necessary to identify a collusive data flow from a benign data flow. We assume that actual analysts will possess enough contextual and background knowledge to properly identify collusive flows with the help of our prototype.

6.2.2 Study Methodology

In this subsection, we present and detail the methodology we adopted to conduct our user study. This includes the participant recruitment process, participant introduction, task selection and instruction, performance metrics, and user input gathering.

Recruitment Process In order to recruit participants that possessed the proper knowledge detailed in Subsection 6.2.1, we conducted informal interview is potential participants who we knew likely had the expertise needed. If they exhibited the requisite knowledge and expertise, they were then asked if they would like to participate in the user study. They were told that they would be payed twenty USD upon completion of the study. Through this process, we were able to recruit ten qualified individuals to participate in our study. While this is a relatively low number of participants, due to the highly specialized nature of our design, we are content with it for the claims we wanted to support.

Participant Introduction Before we asked users to complete tasks using our prototype, we presented them with an overview of the Android platform, Android collusion, and the various views and features of our prototype. This was done so that the users would have some familiarity with what we were asking them to complete in the task list, which in turn helps provide more accurate results. However, it is important to note that this introduction did not in any way influence the users decision in identifying Android collusion. It essentially formalized several features that would be used in the task list. In addition, we also provided the criteria which would qualify an ICC as a collusive ICC, or a benign ICC.

User Tasks After the introduction, we asked our participants to complete a series of tasks with our prototype which visualized two different sets of colluding Android apps. The tasks involved

the requisite steps for identifying Android collusion or false-positive classifications using our prototype. Table 6.1 contains the tasks, along with the view they were performed in, we asked our study participants complete. Each task was carried out twice per participant, one on a smaller set of colluding apps, and one on a larger set of colluding apps. Table 6.2 provides detailed information on the two apps sets the study was conducted using. We used two different sets to determine how well our visualization scales to accommodate collusion inspection in a large app set when compared to a smaller app set. After each timed/accuracy measured task was completed, we recorded the time and accuracy for each user performing each task on each app set. If the user instantaneously (or near it) answered the question, we recorded the time as one second. All tasks were completed in the current view before the user was asked to navigate to the next view. The views were explored in the order of: App Communication view, ICC view, App Component view, and Component Flow view. After each view was completed, the user was asked to fill out a questionnaire about the the view designed to evaluate the usability of that particular view. Finally, when all tasks were complete, the participant was asked to complete an overall questionnaire designed to evaluate the overall usability of the prototype.

6.2.3 Questionnaires and Performance Metrics

In this subsection, we detail the design of our usability questionaries as well as a description of how we recorded performance metrics.

Questionnaires We presented our participants with two questionnaire: one designed to evaluate the usability of each individual view and one to evaluate the overall usability of the prototype. Both questionnaires were based off usability questionnaires compiled by Hartson et al. [21]; we selected the questions from their questionaries that were pertinent to what we were attempting to evaluate, such as overall system usability and ease of learning, and tailored them to our tool. Figure 6.1 provides an example of the questionnaire provided to users after they completed a specific view while Figures 6.2, 6.3, and 6.4 provide examples of the overall usability questionnaire.

The overall usability questionnaire was divided into subsections dependent on what the questions within them pertained to. All questions, minus the open-ended questions, allowed the user to answer on a scale from 0 (Strongly Disagree) to 4 (Strongly Agree).

Performance Metrics Time and accuracy of task performance by the participant was recorded by an investigator with a stopwatch. If the user instantaneously replied to the task question by the investigator, we recorded the time as a second. Accuracy was determined by asking the user to identify specific types of data flows, collusive or benign ICCs, or suspicious intra-component data flows. If the user answered correctly, the accuracy was recorded as 100 percent, otherwise it was recorded as 0.

6.3 Results

We present the results from our user study in this section. First we present the performance metrics recorded by an investigator followed by the results from questionnaires completed by the study participants. Finally, we present a series of changes to our design for post-classification inspection for Android collusion we derived from the results.

6.3.1 User Performance and Evaluations

In this subsection, we present the results we collected from conducting our user study with our prototype implementation.

User Performance Metrics We measured the the time and accuracy of users when completing metric-specified tasks listed in Table 6.1. The time to complete a task was measured by an investigator with a stop watch while accuracy was determined by asking a user to complete a task and the investigator evaluates their answer. Because the recording was done by an investigator with a stop watch, the time values are estimates and should not be considered precise measurements. However, for the usability purposes we wish to support, this estimation approach. Table 6.3 presents the average time and accuracy, from all ten study participants, when identifying various high, medium, and low risk communication flows, individual ICCs, and suspicions data-flows in the App Communication, ICC view, and Component Flow views, respectively.

For the tasks listed in Table 6.3, the study participant was asked to identify (verbally, visually, or with the mouse pointer) the specific edge indicated by the specific type of flow listed in the "Task" column. The total column represents the total number of edges the averages were calculated with. For the App Communication and ICC views, every task was completed with **100 percent accuracy**, as well as **all averages for time required being less than five seconds**. Thus, we claim that our data shows that **our visualization allows for highly accurate and efficient identification of collusive data flows as well as individual ICCs by human analysts**. Furthermore, while not one-hundred percent accurate like the App Communication and ICC views, our Component Flow visualization also allows for **relatively efficient and accurate identification of intra-component data flows**. The loss of accuracy was due to the user being unaware that there were more flows present to be considered. From our results in Table 6.3, we were able to conclude that **our visualizations may allow the user to accurately and efficiently identify the types of data flows and individual ICCs that indicate collusion from those that are benign**, thus increasing the reliability of the collusion classification. We are not claiming that this is true for all cases, due to the limited sample size, only that our design may have merit for when attempting to differentiate potentially collusive data flows and ICCs from seemingly benign cases in a performant and accurate manor. In addition to collecting time/accuracy metrics when identifying, we also timed and recorded accuracy for user identification of false-positive and true-positive individual ICCs, as displayed in Table 6.4.

For all 80 false-positive ICCs the users were asked to classify, the were able to correctly classify them with **100 precent accuracy in under thirty seconds**. From the data presented in Table 6.4, we can conclude that **our ICC view allows users to quickly and accurately differentiate between true-positive and false-positive ICC classifications**. We believe that the reason for the accuracy not being one-hundred percent when identifying true-positive ICCs was due to the fact that our participants were not security analysts. With the relatively high accuracy participants in our study exhibited when identifying true-positives, it is reasonable to believe that security analysts will have a higher accuracy when using our tool than our participants, who largely had only a basic understanding of the requisite knowledge.

Questionnaire Results After our participants completed all the tasks in each of the views, we asked them to fill out the survey in Figure 6.1. Table 6.5 presents the average of the answers from our participants for each survey for each view. The "Q 1-9" columns contain the average answer from participants for the corresponding question in Figure 6.1.

The results in Table 6.5 indicate that the following statements are widely agreed upon by our study participants:

- Each view was easy to use.
- Each view required the fewest steps possible to accomplish what the study participant wanted to do with it.
- Interacting with the visualization in the view required little effort.
- The participants did not notice any inconsistencies in any of the views.
- Participants could recover from mistakes quickly and easily.
- Features of the views are related to the tasks in all views.
- The users did not have to memorize any information when using the views.
- Each view helped simplify the tasks which were performed in them.

Users agreeing upon the above statements indicate that **our design for each view shows promise for being a usable visualization for post-classification Android collusion inspection**. The statements listed represent eight of the nine questions present in the survey. However, for question four for the ICC and Component Flow views, we recognize that these are more complex views than the others. The results for these two view indicate that we should explore adding some written instructions to augment the current design of the views. In addition to the individual view surveys, we also distributed a survey, depicted in Figures 6.2, 6.3, and 6.4, after our participants finished all the tasks in our study. This survey was designed the evaluate the system usability, usefulness, ease

of use, ease of learning, and user satisfaction with our view as a whole. The aggregate results from this survey appear in Table 6.6.

This data indicates that the following statements about our overall design may have merit according to our study participants:

- The users would like to use the system frequently when performing collusion inspection.
- The system was not unnecessarily complex.
- The system was easy to use as a whole.
- The sequence of views was logical and/or optimal.
- The support of a technical person is not necessary when using this system.
- The various features in the system were well integrated.
- The users believe most people would learn to use the system quickly.
- The users did not need to learn a lot before they could start using the system.
- The system makes accomplishing the tasks easier.
- The system does everything the users would expect for post-classification collusion inspection.
- The system requires the fewest steps possible to accomplish post-classification collusion inspection.
- There were not any inconsistencies.
- The users could recover from mistakes quickly and easily at any point in the system.
- The users learned to use the prototype quickly.
- The users easily remembered how to use the system.
- The users were satisfied with the various visualizations and how they related to one another.
- The system operates the way the user wants it to.

As the data shows, the users answered the questions, on average, favorably for our prototype. Thus with the answers to this surveys we can conclude that **our design for post-classification inspection shows promise in-terms of usability and user satisfaction**. Thus, it shows promise in overcoming the established challenges in post-classification inspection for Android collusion.

Open-Ended Question Results Our questionnaires also included several open-ended questions, such as the ones features in Figure 6.4. We received a variety of answers to these questions, but a few responses were received more than others which included:

- Too much clutter in the App Communication View.
- Interacting with the Component Flow View was not as intuitive as the other views.
- The color scheme was helpful, and the visualizations were intuitive.
- Multi-layered approach helped simplify the overall visualization.

6.3.2 Improvements from User Responses

Our user study allowed us to compile a list of improvements to our prototype that will increase its overall usability. They include:

- Disallowing overlap with node names in the App Communication View.
- Allowing users to pin nodes in the App Communication View.
- Hiding all links that are not connected to a node when it is hovered over.
- Add a field at the top navigation bar to inform the user which view they are in.
- Thickness of lines in App Communication View increases with number of ICCs sent in the flow.
- Option to display legends to help with understanding of views.
- As the design relies heavily on color-coding, incorporate color-blind options.
- Scale visualization to use the whole screen and eliminate blank space.
- Add summary information boxes to nodes in the App Component view.

When interacting with the App Communication view, users frequently had trouble selecting specific flows and/or nodes due to overlap with other nodes and node names. Disallowing the overlap would help increase usability of the App Communication view. Currently, the App Communication view does not allow nodes to be pinned; the D3.js library automatically repositions nodes after a user drags and drops a node. Several users stated that they would like the ability to pin certain nodes in specific locations. Furthermore, in the App Communication view, study participants also expressed difficulty when attempting to identify which flows belonged to a specific node. They expressed a desire for all other flows not associated with a node to disappear when the pointer is

hovered over it (the node). When navigating through the various views, study participants often expressed confusion as to which view they were currently in. A field in to top navigation bar could rectify this problem. Presently, flow edges in the App Communication view are all a constant thickness and the user must hover over them to learn the number of ICCs they represent. Allowing the thickness of the edges scale with the number of ICCs in the flow would help decrease the time needed to determine the number of ICCs present in each flow relative to one another. The remaining improvements are self explanatory. This is not an exhaustive list, and many more improvements can be learned from further analysis of our results and larger scale user studies.

6.3.3 Summary of Results

To summarize our user study, we found that our design and accompanying implementation: (1) indicate that our design of each individual view and overall approach to post-classification collusion visualization may have merit in the pursuit of usable visualizations; (2) may allow human analysts quickly identify collusive communications via color; (3) may enable human analysts to quickly and accurately identify true- and false-positive collusion ICC classifications; (4) our design and implementation satisfies all our design requirements and security goals. It is important to note that due to the limited size of our participants (only 10 in total), we cannot make any definitive claims about our design or implementation. However, the small sample size does enable us to claim that our approach at lease shows promise for the above statements. Had there been any glaring usability issues, or that our design failed to meet its goals, it would have appeared in the results despite the small sample size.

Table 6.1: Tasks completed by study participants.

Task	Task Number	View	Metrics Collected
Identify a high risk ICC communication flow	1	App Communication	Yes
Identify all high risk ICC communication flow	2	App Communication	Yes
Identify a medium risk ICC communication flow	3	App Communication	Yes
Identify all medium risk ICC communication flow	4	App Communication	Yes
Identify a low risk ICC communication flow	5	App Communication	Yes
Identify all low risk ICC communication flow	6	App Communication	Yes
Identify app and component permissions	7	ICC	No
Identify a high risk ICC communication	8	ICC	Yes
Identify all high risk ICC communications	9	ICC	Yes
Identify a medium risk ICC communication	10	ICC	Yes
Identify all medium risk ICC communications	11	ICC	Yes
Identify a low risk ICC communication	12	ICC	Yes
Identify all low risk ICC communications	13	ICC	Yes
Inspect each high/medium risk ICC and determine if collusive	14	ICC	Yes
Determine app and component permissions	15	App Component	No
Identify the flows within a component that are currently most suspect	16	Component Flow	Yes
Expand and collapse various flows	17	Component Flow	No

Table 6.2: App sets evaluated by user study participants.

Set Designation	No. Apps	No. ICC Data Flows	No. Collusive ICCs	No. False-Positive ICCs
Small Set	5	6	1	3
Large Set	11	17	4	5

ColTrax User-study Tasks Questionnaire

(View Name Here) View

1. This view was easy to use.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

2. It requires the fewest steps possible to accomplish what I want to do with it.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

3. Interacting with the visualizations requires little effort.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

4. I can use it without written instructions.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

5. I do not notice any inconsistencies as I use it.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

6. I can recover from mistakes quickly and easily.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

7. Features of this view are related to the tasks.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

8. I did not have to memorize much when using this view

   Strongly Disagree Disagree Neutral Agree Strongly Agree

9. The view helped to simplify the tasks.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

Additional Comments:

Figure 6.1: Example of an individual view questionnaire.

ColTrax User-study Questionnaire

System Usability Scale

Please circle the agreeance level beneath each statement which most appropriately reflects your impressions of ColTrax.

1. I think that I would like to use this system frequently when performing collusion inspection.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

2. I did not feel this system was unnecessarily complex.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

3. I thought this system was easy to use.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

4. I thought the sequence of views was logical and/or optimal.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

5. I think that I would need the support of a technical person to be able to use this system.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

6. I found the various functions in this system were well integrated.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

7. I would imagine that most people would learn to use this system very quickly.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

8. I need to learn a lot of things before I could get going with this system.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

Usefulness

1. It makes accomplishing the tasks easier.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

Figure 6.2: Example of the overall prototype questionnaire. Page 1.

1. It does everything I would expect it to do when inspecting for collusion.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

Ease of Use

1. It requires the fewest steps possible to accomplish collusion detection.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

2. I do not notice any inconsistencies as I use it.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

3. I can recover from mistakes quickly and easily.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

Ease of Learning

1. I learned to use ColTrax quickly.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

2. I easily remember how to use it.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

3. Each view was easy to learn.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

Satisfaction

1. I am satisfied with the visualizations.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

2. It works the way I want it to work.

   Strongly Disagree Disagree Neutral Agree Strongly Agree

Figure 6.3: Example of the overall prototype questionnaire. Page 2.

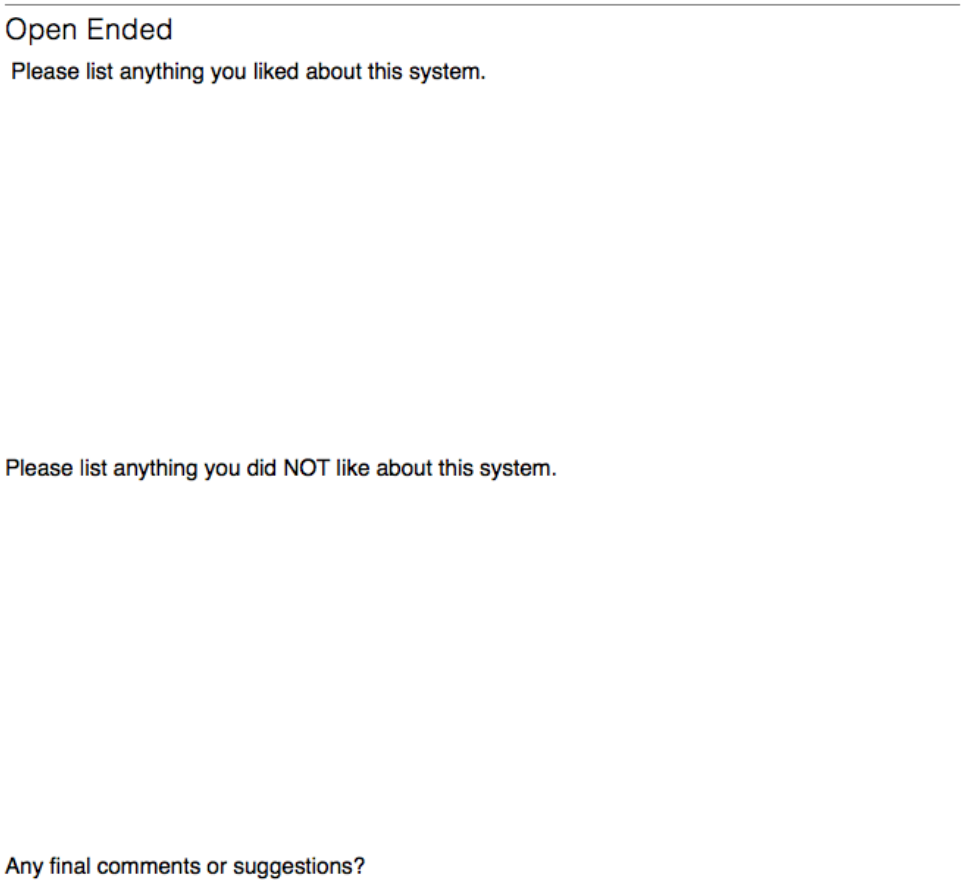

Figure 6.4: Example of the overall prototype questionnaire. Page 3.

Table 6.3: Aggregate metric results: edge identification in the user study.

Task Number	Set Size or View	Average Time (in seconds)	Average Accuracy (as a percentage)	Total
1	Small Set	2.1	1.00	10
2	Small Set	3	1.00	10
3	Small Set	1.7	1.00	10
4	Small Set	1.9	1.00	20
5	Small Set	1.3	1.00	10
6	Small Set	1.5	1.00	30
1	Large Set	1.5	1.00	10
2	Large Set	2.4	1.00	30
3	Large Set	1.4	1.00	10
4	Large Set	2.8	1.00	40
5	Large Set	1.3	1.00	10
6	Large Set	1.3	1.00	100
8	Small Set	1.1	1.00	10
9	Small Set	1.0	1.00	10
10	Small Set	1.1	1.00	10
11	Small Set	1.4	1.00	30
12	Small Set	1	1.00	10
13	Small Set	1.1	1.00	120
8	Large Set	1.1	1.00	10
9	Large Set	1.3	1.00	30
10	Large Set	1.0	1.00	10
11	Large Set	2.6	1.00	60
12	Large Set	1.1	1.00	10
13	Large Set	1.1	1.00	140
16	Component Flow View	49.8	0.975	40

Table 6.4: Aggregate metric results: ICC classification in the user study.

Task	ICC View Set Size	Average Time (in seconds)	Average Accuracy (as a percentage)	Total
Classify a false-positive ICC	Small	22.1	1.00	30
Classify a true-positive ICC	Small	19.0	0.8	10
Classify a false-positive ICC	Large	3.24	1.00	50
Classify a true-positive ICC	Large	4.63	0.95	40

Table 6.5: Aggregate metric results: individual view survey.

View	Q 1	Q 2	Q 3	Q 4	Q 5	Q 6	Q 7	Q 8	Q 9
App Communication	3.6	3.6	3.6	3.6	3.7	3.6	3.7	3.8	3.7
ICC	3.5	3.8	3.6	2.9	3.5	3.7	3.7	3.4	3.8
App Component	3.8	3.8	3.8	3.7	3.8	3.7	3.8	3.6	3.8
Component Flow	3.4	3.5	3.6	2.8	3.6	3.7	3.6	3.4	3.7

Table 6.6: App sets evaluated by user study participants.

Question Number	Question Set	Average Answer
1	System Usability Scale	3.6
2	System Usability Scale	3.7
3	System Usability Scale	3.6
4	System Usability Scale	3.7
5	System Usability Scale	1.4
6	System Usability Scale	3.6
7	System Usability Scale	3.4
8	System Usability Scale	0.9
1	Usefulness	3.6
2	Usefulness	3.1
1	Ease of Use	3.6
2	Ease of Use	3.6
3	Ease of Use	3.7
1	Ease of Learning	3.6
2	Ease of Learning	3.5
3	Ease of Learning	3.7
1	Satisfaction	3.6
2	Satisfaction	3.4

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we presented a web-based visualization approach to post-classification collusion visualization which simplifies the complex network of ICC communications and data-flows between a set of potentially collusive apps. Our minimalistic design creates an interface which is orderly, easy to use, and will not overwhelm the human analyst with excessive information. The tool effectively provides a new countermeasure to the growing threat of colluding Android malware by providing a human analyst with all the information he or she needs to identify colluding applications in a reliable and efficient manner. Furthermore, the tool's interface enables the analyst to identify false-positive and false-negative classifications of ICCs. Our evaluation of our approach via a user study revealed that our approach is usable and allows analysts to accurately and efficiently identify malicious and benign data-flows and collusive ICCs. In our evaluation, users were able to identify false-positive classifications and differentiate between data flows with 100 precent accuracy in an average time of less than ten seconds. Although the accuracy for identifying true-positive ICC classifications was only 92 precent, we believe that our target users will achieve a higher accuracy due to their expertise over our user study participants.

However, our approach is not a complete solution to the problem of Android security. For example, presently, our approach does not consider opportunistic collusion or other communication channels other than explicit ICCs. Furthermore, meta-data about a set of applications under inspection (such as authors, the most frequent way the app is installed on users' Android devices, etc.) can be useful when identifying app collusion. Presently, our approach does not incorporate any of this meta-data into the layout. Thus our work can be viewed as a first step in usable visualization tools which combat Android collusion.

7.2 Future Work

Future work for our approach could include:

Detection for Implicit Intents and Other Communication Channels Presently, as stated above, our tool does not consider the threat of opportunistic Android collusion via implicit ICCs or collusion though other communication challenges. Thus our approach may miss several collusive attacks. Future would could be to add a toggle system at the top of each view to show explicit ICCs, implicit ICCs, other types of communication, or all of the above. This will allow the user to use our approach to visualize all collusive inter- and intra-communications between a set of apps.

Biset ICC View Our ICC view currently visualizes ICCs between only two sets of components belonging to two distinct apps. However, our ICC view could be expanded to take the form of a biset visualization as detailed by Sun et al. [42]. It is possible to visualize ICCs between n, where n is the number of apps in the app set, sets of app components using this format in a usable format. However, we would suggest allowing the user to gradually increase the number of components sets present in the ICCs view so the user does not become overwhelmed.

Expand upon the App Component View Our App Component view currently a gateway to the various Component Flow views for each app component. However, we could add several features, such as summary tables for each component as well as the ability to include and filter components from other apps how how they communicate with current components, into the view. This would allow our users to access and explore collusive data in ways not currently available in the prototype. For example, if an analyst wanted to see which other components communicate with a component currently rendered in the App Communication view, the App Component view could be expanded to allow for this inspection.

Interchangeable Static Analysis Modules The static analysis our approach uses is context- and flow-sensitive. However, there are many other program analysis sensitivities, such as parameter sensitivity, that would increase the accuracy of our data flow information and collusion threat level. We could provide the option to increase or decrease the number of sensitivities our analysis provides by incorporating several static program analysis modules and allowing the user to select which one they want to use. This effectively gives users the ability to adjust the accuracy and performance of the analysis.

Bibliography

[1] Apktool: A tool for reverse engineering Android apk files. `https://code.google.com/p/android-apktool/`.

[2] Spring Framework. `http://projects.spring.io/spring-framework/`.

[3] B. Amos, H. Turner, and J. White. Applying machine learning classifiers to dynamic android malware detection at scale. In *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 1666–1671, July 2013.

[4] L. O. Andersen. Program analysis and specialization for the c programming language. Technical report, 1994.

[5] M. Angelini and G. Santucci. Modeling incremental visualizations. In *Proc. of the EuroVis Workshop on Visual Analytics (EuroVA '13)*, pages 13–17, 2013.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[7] M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-Driven Documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2011.

[8] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on Android. In *Proc. of the 19th Annual Network and Distributed System Security Symposium*. The Internet Society, 2012.

[9] S. Cabello and B. Mohar. Adding one edge to planar graphs makes crossing number and 1-planarity hard. *CoRR*, 2012.

[10] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Proc. of the 9th Int'l Conference on Mobile Systems, Applications, and Services*, pages 239–252. ACM, 2011.

[11] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[12] W. Dietl and P. Mller. Universes: Lightweight ownership for jml. *JOURNAL OF OBJECT TECHNOLOGY*, 4(8):5–32, 2005.

[13] K. Elish, D. Barton, D. Yao, and B. Ryder. Inter-app data-flow analysis for app collusion and vulnerability detection in Android. In *Technical Report, Computer Science, Virginia Tech*, 2015.

[14] K. Elish, X. Shu, D. Yao, B. Ryder, and X. Jiang. Profiling user-trigger dependence for Android malware detection. *Computers & Security*, 49:255–273, 2015.

[15] K. Elish, D. Yao, and B. Ryder. On the need of precise inter-app ICC classification for detecting Android malware collusions. In *Proc. of the IEEE Mobile Security Technologies (MoST) workshop, in conjunction with the IEEE Symposium on Security and Privacy*, 2015.

[16] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 393–407, Berkeley, CA, USA, 2010. USENIX Association.

[17] W. Fang, B. P. Miller, and J. A. Kupsch. Automated tracing and visualization of software security structure and properties. In *Proceedings of the Ninth International Symposium on Visualization for Cyber Security*, VizSec '12, pages 9–16, New York, NY, USA, 2012. ACM.

[18] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[19] P. Fiaux, M. Sun, L. Bradel, C. North, N. Ramakrishnan, and A. Endert. Bixplorer: Visual analytics with biclusters. *Computer*, (8):90–94, 2013.

[20] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012.

[21] R. Hartson and P. Pyla. *The UX Book: Process and Guidelines for Ensuring a Quality User Experience*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[22] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Not.*, 23(7):35–46, June 1988.

[23] W. Huang, Y. Dong, A. Milanova, and J. Dolby. Scalable and precise taint analysis for android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 106–117, New York, NY, USA, 2015. ACM.

[24] E. Karapistoli, P. Sarigiannidis, and A. A. Economides. SRNET: A real-time, cross-based anomaly detection and visualization system for wireless sensor networks. In *Proceedings of the Tenth Workshop on Visualization for Cyber Security*, VizSec '13, pages 49–56, New York, NY, USA, 2013. ACM.

[25] J. Kim, Y. Yoon, K. Yi, and J. Shin. ScanDal: Static analyzer for detecting privacy leaks in android applications. In H. Chen, L. Koved, and D. S. Wallach, editors, *MoST 2012: Mobile Security Technologies 2012*, Los Alamitos, CA, USA, May 2012. IEEE.

[26] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. Android taint flow analysis for app sets. In *Proc. of the 3rd International Workshop on the State of the Art in Java Program Analysis (SOAP)*. ACM, 2014.

[27] L. Li, A. Bartel, T. Bissyande, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 280–291, May 2015.

[28] A. Long, J. Saxe, and R. Gove. Detecting malware samples with similar image sets. In *Proceedings of the Eleventh Workshop on Visualization for Cyber Security*, VizSec '14, pages 88–95, New York, NY, USA, 2014. ACM.

[29] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: statically vetting Android apps for component hijacking vulnerabilities. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 229–240. ACM, 2012.

[30] C. Marforio, A. Francillon, and S. Capkun. Application collusion attack on the permission-based security model and its implications for modern smartphone systems. In *Technical Report, Department of Computer Science, ETH Zurich*, 2010.

[31] C. Marforio, H. Ritzdorf, A. Francillo, and S. Capkun. Analysis of the communication between colluding applications on modern smartphones. In *Proc. of the 28th Annual Computer Security Applications Conference*. ACM, 2012.

[32] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[33] D. Octeau, S. Jha, and P. McDaniel. Retargeting Android applications to Java bytecode. In *Proc. of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2012.

[34] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In *Proc. of the 22nd USENIX Security Symposium*. USENIX Association, 2013.

[35] R. Pienta, A. Tamersoy, H. Tong, A. Endert, and D. H. P. Chau. Interactive querying over large network data: Scalability, visualization, and interaction design. In *Proceedings of the 20th International Conference on Intelligent User Interfaces Companion*, IUI Companion '15, pages 61–64, New York, NY, USA, 2015. ACM.

[36] T. Ravitch, E. R. Creswick, A. Tomb, A. Foltzer, T. Elliott, and L. Casburn. Multi-app security analysis with FUSE: Statically detecting Android app collusion. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, PPREW-4, pages 4:1–4:10, New York, NY, USA, 2014. ACM.

[37] T. Reps. Program analysis via graph reachability. In *Proceedings of the 1997 International Symposium on Logic Programming*, ILPS '97, pages 5–19, Cambridge, MA, USA, 1997. MIT Press.

[38] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *Selected Papers from the 6th International Joint Conference on Theory and Practice of Software Development*, TAPSOFT '95, pages 131–170, Amsterdam, The Netherlands, The Netherlands, 1996. Elsevier Science Publishers B. V.

[39] R. Schlegel, K. Zhang, X.-y. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. *NDSS*, 11:17–33, 2011.

[40] H.-J. Schulz, M. Angelini, G. Santucci, and H. Schumann. An enhanced visualization process model for incremental visualization.

[41] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. "andromaly": A behavioral malware detection framework for android devices. *J. Intell. Inf. Syst.*, 38(1):161–190, Feb. 2012.

[42] M. Sun, P. Mi, C. North, and N. Ramakrishnan. Biset: Semantic edge bundling with biclusters for sensemaking. *Visualization and Computer Graphics, IEEE Transactions on*, 22(1):310–319, 2016.

[43] S. C. Sundaramurthy, J. McHugh, X. S. Ou, S. R. Rajagopalan, and M. Wesch. An anthropological approach to studying CSIRTs. *IEEE Security & Privacy*, pages 52–60, 2014.

[44] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[45] T. Vidas, D. Votipka, and N. Christin. All your droid are belong to us: A survey of current android attacks. In *Proceedings of the 5th USENIX Conference on Offensive Technologies*, WOOT'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[46] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1329–1341, New York, NY, USA, 2014. ACM.

[47] B. Wolfe, K. Elish, and D. Yao. Comprehensive behavior profiling for proactive Android malware detection. In *Proc. of 17th International Information Security Conference (ISC)*, 2014.

[48] T. Wüchner, A. Pretschner, and M. Ochoa. DAVAST: Data-centric system level activity visualization. In *Proceedings of the Eleventh Workshop on Visualization for Cyber Security*, VizSec '14, pages 25–32, New York, NY, USA, 2014. ACM.