11-2017

Uncovering user-triggered privacy leaks in mobile applications and their utility in privacy protection

Joo Keng Joseph CHAN
Singapore Management University, joseph.chan.2012@msis.smu.edu.sg

Citation

CHAN, Joo Keng Joseph. Uncovering user-triggered privacy leaks in mobile applications and their utility in privacy protection. (2017). Dissertations and Theses Collection.
Available at: https://ink.library.smu.edu.sg/etd_coll_all/39

MSc Dissertation

by

Joseph Chan Joo Keng

Submitted to School of Information Systems in partial fulfillment of the requirements for the Degree of Master of Science in Information Systems

Dissertation Committee:

Lingxiao JIANG (Chair)
Assistant Professor of Information Systems
Singapore Management University

Archan MISRA (Committee Member)
Associate Professor of Information Systems
Singapore Management University

Baihua ZHENG (Committee Member)
Associate Professor of Information Systems
Singapore Management University

Singapore Management University
2016

Uncovering User-Triggered Privacy Leaks in Mobile Applications and their Utility in Privacy Protection

Joseph Chan Joo Keng

Abstract

Mobile applications are increasingly popular, and help mobile users in many aspects of their lifestyle. Applications have access to a wealth of information about the user through powerful developer APIs. It is known that most applications, even popular and highly regarded ones, utilize and leak privacy data to the network. It is also common for applications to over-access privacy data that does not fit the functionality profile of the application. Although there are available privacy detection tools, they might not provide sufficient context to help users better understand the privacy behaviours of their applications.

In this dissertation, I present the design, implementation and evaluation of an Automated Privacy Testing System called MAMBA for uncovering the causes of user-triggered leaks in Android applications ('leak causes') as well as their paths taken to reach the leaks. Privacy 'leak-causes' refer to privacy usage and leak characteristics of applications as well as user-actions and activities with privacy implications. Paths refers to page transition paths as well as the sequence of user actions required to cause these transitions. This solution is based on hybrid application of dynamic/static analysis of Android applications, and it involves improving automated testing of applications for run-time verification of the 'leak causes'. The automated testing is based on directed testing, and automatically traverses applications from initial to resulting activities with potential leak behaviours, based on paths obtained from static analysis of the Android callback control flows.

I demonstrate the advantages of my automated testing system through standalone evaluations as well as comparisons with another automated testing system - Automated Model Checker (AMC) [39]. The results show that MAMBA has large

improvements in terms of less testing time required, with only a small reduction in coverage. MAMBA also has good privacy data access accuracy (Precision=79.84% , Recall=68.90%), and moderate privacy data leak accuracy (Precision=35.66% , Recall=56.10%) - (Recall values were measured relative to AMC). Privacy detectors of ProtectMyPrivacy (PMP) and TaintDroid were utilized for runtime measurements.

I also show how the resulting outputs of privacy 'leak causes' can be utilized together with a privacy message overlay mechanism for warning users of privacy triggers interactively during application uses. I have conducted field and lab user studies to show that the privacy messages can aid users to utilize applications in a way that uses less of their privacy data, if they do not agree with the privacy usages. I also found that there are other factors which influence message effectiveness.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Mobile applications (apps) are ubiquitous in the modern age of smart devices, and have become an indispensable part of life. They help to manage many aspects of urban lifestyle, e.g. from looking for locations of nearby restaurants and retails venues, to helping users' keep track of their expenditure and health. A robust ecosystem exists in the mobile application stores, such as in Google Playstore and Apple iTunes Store, where millions of apps are downloaded everyday.

It is important for good methods to be available for notifying and educating the user on apps that may inappropriately infringe on their privacy. There is an increasingly urgent need to be able to detect and profile privacy usage characteristics of mobile apps. While a great asset to users, mobile apps have access to a wealth of information about the user through powerful Application Programming Interfaces (APIs) provided by the operation system platforms. Many apps might not follow good practices in privacy design [45]. It is well known that many apps, even popular and highly regarded ones, access and leak privacy data to the network.

To aid in resolving these problems, various privacy detection tools have been created by the research community [24, 61, 3] as well as platform developers [6, 56] for detection and control of applications' uses of privacy data. There has also been various work in designing usable interfaces that make it easier for users to configure privacy decisions [5, 12].

Privacy leak detection tools [24, 6] are geared towards the monitoring of privacy infringements rather than on effective notifications to the user. The tools inform users whenever privacy infringements take place during run-time, but do not provide sufficient context or summarized insights on these infringements or app behaviours. Various works have sought to address this by improving mechanism for privacy notification [12], using and improving visual framing (e.g. icons next to text, ratings etc.) [2], as well as by the improvement of textual outlook [16], linguistic properties [32] and timing [60] of privacy messages. An aspect that is missing from current work is on providing users with additional context or insights into the privacy characteristic of apps. Doing this can potentially improve the value of privacy notifications and aid users in improving their privacy. It also might allow them to set better privacy policies in privacy data managers [6] available in current mobile OSes.

Hence, privacy detection tools might not provide sufficient context information on app behaviours (e.g. they do not show which app views actually utilize data, or whether privacy leaks are caused by user actions or system events etc.), and they also present notices in a manner that might not be easily digestible by users. Providing these additional privacy contexts in the form of tailored privacy notices could aid users to better understand their apps privacy usage and improve their privacy.

In addition, a characteristic of these tools is that the detection of privacy leaks/accesses are performed in real-time during uses by the mobile user [24, 61, 14, 3, 61, 33]. Test inputs have to be generated for traversal of app states before the privacy detectors can effectively detect privacy behaviours. This relies on the mobile user, who usually has to run through applications before notifications (e.g. Scroll-downs, Pop-ups or from within a Permission Management List etc.) appear from the privacy detector running in the background, warning the user of privacy leak or access caused by the application.

Besides real-time systems, static leak detection systems such as FlowDroid [9] and PiOS [23] are able to analyze privacy infringements in mobile apps as well.

However, these systems also do not provide sufficient context information on app behaviours, and are also not geared towards the presentation of notifications to users in an easily digestible and understandable manner.

The objectives of my work are thus to be able to accurately, efficiently and scalably output the causes of user-triggered privacy leaks and privacy characteristics of mobile apps, after which, to provide effective notifications to users by superimposition of privacy notices on top of culprit app buttons/views to improve users privacy behaviours and understanding of their apps privacy characteristics, and to evaluate the notification mechanism alongside existing mechanisms of other privacy notification systems.

In this dissertation, I present a novel hybrid static and dynamic analysis solution and system for automatically uncovering and testing the causes and paths of user-triggered privacy leaks ('leak causes') in Android applications (MAMBA System). Paths refers to page transition paths as well as the sequence of user actions required to cause these transitions. These privacy 'leak causes' serve to function as an additional context to help users better understand their apps' privacy behaviours.

My approach first utilizes static analysis to obtain causes of user-triggered privacy data accesses in applications as well as the view transitional paths that users would take towards these activities. A dynamic testing stage is then performed for runtime verification of the privacy accesses. The dynamic testing stage helps to improve on the outputs of static analysis, by verifying that the privacy accesses and leaks occur during app runtime.

App binaries are first analyzed statically to identify the Android activities that are linked to privacy sensitive APIs in the control flow. The paths towards these activities, from the initial activities, are then obtained by the analysis of control flow between call-back functions of the app, so that an activity-transition graph can be obtained. This activity-transition graph is then used to direct the transition of the app towards the privacy-related activities at run time. The testing process produces test logs which can be analyzed to uncover specific user-actionable app components on activities that cause privacy leaks. This information can be stored in a 'leak-cause' database for customizable notifications to users by a lightweight mobile app. The mobile app overlays privacy messages on top of culprit buttons and views during app usage.

I evaluate my MAMBA system against a prominent automated app testing solution - Automated Model Checking (AMC) [39] for over 500 apps. My evaluation criteria is based on the average testing time required (latency), coverage of activities that are required to be reached as well as the precision of uncovering privacy-related behaviours in these activities.

I also investigate the effects of providing the analysis outputs to real users in lab and field studies involving users on their personal devices over a period of weeks. I demonstrate the effectiveness of the outputs in changing user behaviors by showing reduced frequency of app accesses and duration of app uses for users who disagree with the app privacy behaviours. In the next sections, I detail my thesis statement, approach and contributions.

1.1 Thesis Statement

My thesis statement is as follows:

Providing the additional context on the causes and characteristics of user-triggered privacy leaks in mobile apps can help users to change behaviours that impact their privacy. These causes and characteristics can be efficiently and accurately uncovered by a hybrid application of static analysis of callback functions and runtime traversal of the apps towards targeted pages with privacy implications. This method also has advantages over the other automated app testing approaches in terms of significantly less testing time.

As stated above, the main objectives in my dissertation are to output causes and characteristics of user-triggered app privacy leaks accurately and efficiently from hybrid static and dynamic analysis of apps. The processing of apps should be done in a scalable and accurate manner, and with high coverage so that specific areas of the apps can be reached in practical amount of testing time. I evaluate and compared my automated testing system and approach against another prominent automated testing solution, Automated Model Checking (AMC). The outputs from the system can then be utilized to notify users about privacy issues.

I also aim to present these outputs in a manner that will help mobile users to improve their privacy. The improvement in their privacy will be measured in terms of decrease in their app-usage metrics (usage frequency, duration and no. of buttons clicked) for apps with privacy behaviours that they are in disagreement with.

1.2 Approach

My approach is based on the following areas, (i) A Novel Hybrid Static/Dynamic Analysis System, (ii) Evaluation of the System and (iii) User Studies to Investigate Utility of the System Outputs:

(i) Novel Hybrid Static/Dynamic Analysis System: Static analysis is used to capture activity/page transition paths of Android apps, as well as identify target activities that leak privacy data. Obtaining such transition paths require the construction of the control flows among Android call-back functions. Android call-backs are event-driven function calls within the Android framework that may be enacted by the user (via event listeners) or centrally called by the system during the Android life-cycle or occurrence of system-related events.

Due to the event-driven nature of the Android framework, the control flow among call-back functions are not available from a standard control-flow graph. To overcome this problem, I have formulated and implemented algorithms that build the control flows among call-back methods of Android activities and the unique GUI components linked to these call-back methods, from Android bytecode representations. The sequence of user actions from the control flows can then be used to build test cases for an automated tester to direct the executions of the apps towards privacy related activities, so that causes of privacy leaks can be tested at run-time. This is different from prior work, in that user-triggered activity transition paths leading towards privacy related app activities and GUI components are produced as additional context, in addition to privacy leak points.

The automated app testing part of my solution aims to test and verify privacy leaks. While a number of automated testing solutions exist [18, 43, 42, 39, 51], automated app testing remains a time and resource intensive process. My automated directed app traversal technique aims to reduce the testing time required for each app, while maintaining coverage and accuracy in uncovering user-triggered leaks.

(ii) Evaluation of the System: I conduct evaluations to compare my system with another automated testing system, Automated Model Checker (AMC) on a sizeable set of over 500 apps crawled from popular categories from the Google PlayStore. The metrics for evaluation are average app testing times required (latency), coverage of target activities and accuracy of uncovering privacy accesses and leaks. I investigate average performance as well as look at the performance distributions across the apps. I determine what advantages, trade-offs and other performance concerns my proposed system have against the other app testing system.

(iii) User Studies to Investigate Utility of the System Outputs: I aim to provide deeper context and insights into the characteristic of apps in privacy notices by displaying the causes and characteristics of user-triggered leaks in privacy notices for apps. These notices are overlaid on top of privacy invasive app buttons and views, on reaching certain in-app activities, as well as on starting up the app. To the best of my knowledge, the display of causes of user-triggered privacy leaks to users has not been studied before.

I investigate how real users in the field react in response to these notices, as well as how their mobile behaviours can be changed from knowledge obtained from the notices. In addition, I investigate and demonstrate improvements in users' recollection and understanding of apps' uses of their privacy when they are presented with notices from my mechanism. I perform such measurements by studying users' privacy-related inclinations after using applications with and without the notices.

1.3 Research Contributions

My research contributions are as follows:

- Design and implement a novel hybrid static/dynamic tool for outputting causes and paths of user-triggered privacy leaks in Android apps

- Formulate algorithms for analyzing the control flows among call-back functions that are involved in activity and page transitions in Android apps, to create an activity-transition graph

- Utilize the activity-transition graph to direct traversal of applications towards privacy-related activities for automated testing and verification at run time

- Evaluate the automated testing solution on a sizeable number of over 500 Android apps to demonstrate its scalability and performance; Also perform a comparison of the automated testing solution and algorithm against another testing system to demonstrate its advantages/trade-offs in terms of improved testing time, and relatively acceptable coverage and accuracy in privacy detection.

- Investigate the effectiveness of providing additional context of causes of privacy leaks to mobile users under real-world conditions.

- Demonstrate that users privacy behaviours are improved when notified on the causes of privacy leaks as well as app privacy characteristics, and investigate users inclinations when presented with the privacy notices, as well as the factors affecting users' privacy decisions (e.g. Prior app usage history, Agreement with privacy of app, Type of app etc.).

1.4 Summary of Thesis Results

A summary of my key thesis results is as follows:

Empirical Study on Taxonomy of User-Triggered Privacy Leaks

Based on an empirical study of 226 Androids apps from popular categories of the Google PlayStore:

- 46.5% (105) of apps were found to leak privacy data.
- Apps were found to possess a taxonomy of privacy leak behaviours of (i) User-Triggered Leaks, (ii) Start-up Leaks and (iii) Periodic Leaks.
- More than half of the apps with privacy leaks (64 apps) was found to leak due to user triggers on app widgets/buttons.
- Using Association Rules Mining, an overall knowledge database on the app views/buttons that cause user-trigger leaks can be built. Information from this knowledge database can be utilized by an Interactive Privacy Leak Reporter, for the display of privacy notices to users during app usage.

Evaluation of MAMBA Privacy System

Based on a comparison of MAMBA Privacy System with the Automated Model Checker (AMC) on a set of 500 apps:

- MAMBA has an advantage over AMC in terms of less testing time required overall (13.18 mins/app for MAMBA compared to 51.86 mins/app for AMC), with only a small reduction in activity coverage (63.96% for MAMBA compared to 70.33% for AMC).

- MAMBA has good privacy data access accuracy (Precision=79.84%, Recall=68.90%) and moderate privacy data leak accuracy (Precision=35.66%, Recall=56.10%) - (Recall values were measured relative to AMC)
- Overall, MAMBA was almost *4 times faster* than AMC (13.18 mins/app compared to 51.86 mins/app) in detecting and verifying privacy-related activities. For apps with privacy behaviours, MAMBA was *2.7 times faster* than AMC (21.52 mins/app compared to 57.11 mins/app). And for apps with no privacy behaviours, MAMBA had the largest advantage in being almost *8 times faster* than AMC (6.08 mins/app comapred to 47.4 mins/app).

User Studies on Utility of System Outputs

The following results were found from lab and field studies, using outputs of user-triggered causes of privacy leaks:

- For the lab study, it was found that there was a 0% to 80% drop in users avoiding privacy leak causing widgets/buttons for the 4 test apps. Usability was maintained for apps with and without privacy messages, based on Likert Scores.
- For the field study conducted with 42 users over a 2 week period, users who received privacy messages, disapproved of the app's data use as well as had (i) User education stimulus, (ii) High levels of privacy consciousness and (iii) High levels of surprise at message contents, were shown to have reduced duration of app usage, reduced frequency of app start ups as well as a reduction in the number of leak-causing buttons clicked upon.
- Users' Prior App Usage Time (Familiarity) had the highest statistical strength in influencing app usage behaviours. This implied that users who have been

utilizing the apps for a longer period of time were less likely to decrease utilization of apps, in spite of receiving privacy messages.

Resulting Publications

The following publications resulted from this work:

1. J. C. J. Keng, T. K. Wee, L. Jiang, and R. K. Balan. **”The case for mobile forensics of private data leaks: towards large-scale user-oriented privacy protection.”** In Proceedings of the 4th Asia-Pacific Workshop on Systems (APSys 2013), page 6. ACM, 2013.

2. J. C. J. Keng, L. Jiang, T. K. Wee, and R. K. Balan. **”Graph-aided directed testing of android applications for checking runtime privacy behaviours.”** In 11th IEEE/ACM International Workshop on Automation of Software Test (AST 2016), (Co-located with ICSE 2016), 2016.

3. J. C. J. Keng, L. Jiang, T. K. Wee, and R. K. Balan. **”Leveraging Automated Privacy Checking For Design of Mobile Privacy Protection Mechanisms.”** In 1st Workshop on Bridging the Gap Between Privacy By Design and Privacy in Practice, (Co-located with CHI 2016), 2016.

1.5 Existing Work (Literature Review)

In this section, I highlight some existing work on the analysis of mobile app privacy and make some comparisons in relation to my work. I focus on 3 aspects of mobile app privacy related work: (i) Mobile Privacy Detection & Protection Systems, (ii) Automated Testing Systems and (iii) Privacy Notification Mechanisms. The comparisons serve to bucket the different dimensions such as techniques utilized, characteristics, capabilities, usable outputs and performance measures of the various systems.

1.5.1 Mobile Privacy Detection & Protection Systems

(Please refer to Appendix A (Table A.1): Comparison Table of Mobile Privacy & Detection Systems) A detailed comparison list of some privacy systems developed in recent years by the research community is displayed [65, 50, 26, 41, 25, 9, 7, 40, 59, 23, 52]. The dimensions for comparisons are: ’Privacy-Related Behaviour Analyzed’, ’Usable Outputs’, ’Techniques’, ’Latency’, ’Coverage’ and ’Accuracy’.

From the table, the Privacy-Related Behaviours analyzed by these systems include malicious behaviours of apps, privacy leaks, app network and data transmission characteristics, privacy-risk levels, sensitive user inputs and personally identifiable information.

Usable Outputs provided by these systems range from flagging of apps which display malicious behaviours, providing privacy analysis reports/risk assessment, identification of program source/sinks of privacy leaks, additional contextual information for enabling better privacy understanding, as well as leak detection of sensitive data fields.

Techniques utilized by these systems are mainly static/dynamic analysis, as well as machine-learning and network flow detection.

Latency refers to the average length of time to process each app, and ranged from just a few minutes per app to more than 1 day per app. Coverage refers to the extent by which the apps are covered. While Accuracy is the level of correctness of the systems’ outputs.

Comparison with my Approach & System: The goals of my system are to uncover the causes and paths leading to user-triggered privacy leaks in mobiles apps, to act as additional form of contextual information to aid users in protecting their privacy. It does so by analyzing the control flows among callback functions that are involved in activity and page transitions in Android apps, to create an activity transition graph. This representation can then be utilized to improve app coverage and testing time during automated testing. It does so by directing executions of the app from root (main) to targeted activities within the app for testing and verification of privacy leaks. The knowledge obtained from the process can then be utilized as a form of additional contextual information to improve user privacy, as well as to allow them to set better privacy policies for controlling their privacy data.

For the dimension of 'Privacy-Related Behaviour Analyzed' in the comparisons, my system falls under detection of privacy leaks. The 'Usable Outputs' are providing additional contextual informational for enabling better privacy understanding. While the techniques utilized are static/dynmaic analysis.

1.5.2 Automated Testing Systems

(Please refer to Appendix A (Table A.2): Comparison Table of Automated Testing Systems) An integral part of my privacy analysis solution consists of automated app testing to achieve good runtime coverage of mobile apps, for verification of the outputs of static analysis for improved accuracy. In this section, I provide a comparison of currently available automated testing solutions and systems.

From Appendix B, the dimensions for comparison are: 'Exploration Strategy', 'Exploration Target', 'Exploration Technique', 'Inputs Simulated' and 'Purpose'. There are 4 main types of Exploration Strategies, (i) Random, (ii) Systematic Event Selection, (iii) Model-Based (Built On-the-Fly at Runtime) and (iv) Model-Based (from Static Analysis).

(i) Random exploration is based on the generation and simulation of random user inputs in order to enact app state and activity/page changes. For example, the simplest and most frequently used Monkey [18] generates random UI events for black-box testing. (ii) Systematic Event Selection is still based on random exploration, but have more systhematic selection methods of simulated user inputs. For example, Dynodroid [43] has a series of exploration strategies, *Frequency*, *UniformRandom* and *BiasedRandom* that helps to improve coverage. (iii) Model-based

(On-the-Fly at Runtime) exploration strategy builds a model of the runtime app states and user events encountered during exploration, and utilizes the model to traverse the app and keep track of states already explored. AMC [39] and DECAF [42] extracts the DOM trees from the apps at runtime to build UI models, and performs its exploration of the models.

The automated testing strategy employed in my hybrid static/analysis privacy analysis system falls under the category of (iv) Model-Based (from Static Analysis), where a model for exploration is built prior to runtime. A^{3}E [10] has a targeted exploration strategy based on following a Static Activity Transition Graph. While this approach is similar to my system, it is based on a simplified intent analysis of sources and sinks in activity startups, and does not analyze event handlers of GUI objects or their identifiers. A closer inspection at its targeted exploration algorithm reveals that it still requires extraction and iteration over the runtime GUI elements obtained from the Android Hierarchyviewer. ORBIT [64] utilizes static analysis for detection of event handlers of the apps, but does not analyze activity transition paths. The exploration model is still built from dynamic analysis. Further more, the source code of the app is required for exploration. Brahmastra [15] utilizes a similar concept to my system, in that it uses static analysis to map transition paths of Android activities for exploration. However, these transition paths do not model GUI widgets, app layout views or listener objects. Instead, Brahmastra relies on re-writing of apps to force activity transitions ('self-execution'). This has disadvantages as the paths might not be entirely representative of those triggered by a real user. The path transitions employed in my solution involves transition paths that are reachable by real users, with no artificial forcing of app activity and state transitions.

'Exploration Target' refers to the prioritization by the tester to cover specific parts of the app. For example, whether the app will try to cover all pages within the app indiscriminately, or only cover some distinct pages. 'Inputs Simulated' refers to the types of inputs that the tester uses (UI Events and/or System Inputs). While 'Purpose' refers to the reason that each system was built for. For example, for Stress

Testing or General Testing etc.

For ’Exploration Target’, my MAMBA system prioritizes app pages with privacy implications. The ’Inputs Simulated’ are UI Events and the ’Purpose’ is for privacy analysis.

1.5.3 Privacy Notification Mechanisms

(Please refer to Appendix A (Table A.3): Comparisons Table of Privacy Notification Systems) One of my main thesis components involve the usage of the outputs of my privacy analysis system to improve user privacy. As detailed in Section 2.3, my privacy notifications mechanism involves highlighting the causes/paths of privacy leaks as well as privacy leakage characteristics of mobile apps. It does so by privacy message overlays on top of leak-cause buttons/widgets, as well as the display of messages whenever the user navigates to associated app views and activities.

In this section, I perform a comparison of the various privacy notification systems and their notification mechanisms employed in these systems [61, 44, 65, 6, 48, 29, 38, 14, 5, 55, 52, 24]. As displayed in the table of Appendix C, these system are compared in the dimensions of ’Notification Content’, ’Notification Interface’, ’Frequency’, ’Action’, ’Timing’ as well as ’Granularity’.

From Appendix C, the Notification Contents range from real-time privacy data accesses, additional install time permissions control & options, summarized privacy access-control lists of all apps installed, statistics of data usage (Frequency of accesses, Categories of apps etc.) as well as additional context of app privacy usages (e.g. Network IP, Screen-capture sequences). The ’Notification Interfaces’ utilized are Dialog Boxes, Install-time lists, background Android messages, standard Android Scroll-down Notifications, as well as summarized Privacy Manager Lists.

In terms of ’Notification Contents’, my notification mechanism can best be categorized under providing users with Additional Context of app Privacy Usages, while

’Notification Interfaces’ employed are Android Toast Messages.

’Frequency’ refers to how often the privacy notifications appear, and can be at every privacy instance or configurable etc. ’Action’ refers to the notifications being either ’Blocking’ or ’Non-Blocking’. ’Blocking’ notifications are accompanied by a screen freeze, with choices required to be made by users before app usage can continue. While ’Non-Blocking’ notifications do not have any screen freezes. ’Timing’ refers to the time period where notifications appear (Before or after privacy event). And ’Granularity’ refers to the detail level of information contained by the notifications.

For ’Frequency’, my MAMBA system notifies the user at every privacy access/leak instance. For ’Action’, my privacy messages are of the ’Non-Blocking’ type. The ’Timing’ of notifications are before each privacy event. And the ’Granularity’ of MAMBA is at the Data level.

Chapter 2

Uncovering Causes and Paths of User-Triggered Privacy Leaks

In this chapter, I present my framework for privacy analysis of mobile apps at the back-end and utilization of the results to build a knowledge base to help improve user privacy. I detail some preliminary findings on the feasibility of uncovering user-triggered privacy leaks, as well as present an empirical study on the taxonomies of privacy behaviours of Playstore apps.

I describe my hybrid static/dynamic analysis solution and system (MAMBA System) for analysis and uncovering of the causes of user-triggered privacy leaks in Android apps. The system identifies Android activities that have privacy related API calls (Target Activities) and outputs their activity transition paths by static analysis. These paths are then provided to a directed automated runtime tester to transit the app towards these Target Activities, where they are verified by privacy access/leak detectors.

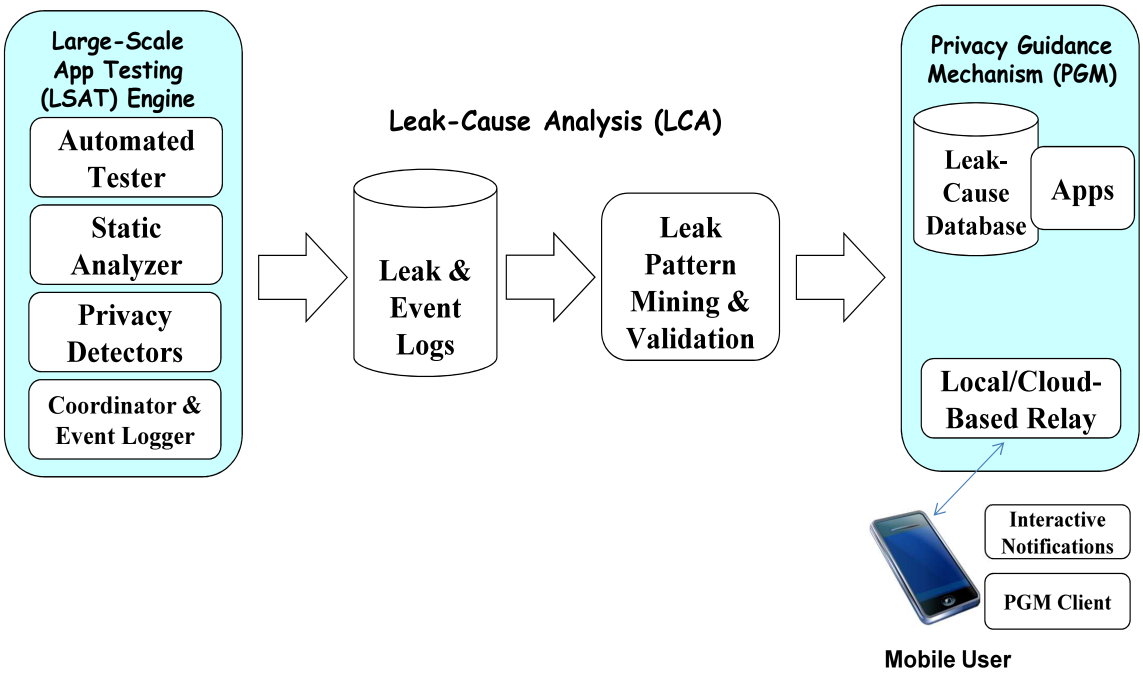

Large-Scale App Testing (LSAT) Engine
Automated Tester
Static Analyzer
Privacy Detectors
Coordinator & Event Logger
Leak-Cause Analysis (LCA)
Leak & Event Logs
Leak Pattern Mining & Validation
Privacy Guidance Mechanism (PGM)
Leak-Cause Database
Apps
Local/Cloud-Based Relay
Interactive Notifications
PGM Client
Mobile User

Figure 2.1: Diagram of Privacy Analysis Solution Framework Consisting of Subsystems: (i)Large-Scale App Testing (LSAT) Engine, (ii) Leak-Cause Analysis (LCA) and (iii)Privacy Guidance Mechanism (PGM).

2.1 Privacy Analysis Solution Framework

2.1.1 Components of Framework

The framework of my privacy analysis solution is as shown in Figure 2.1. It is based on automated analysis of mobile app binaries on back-end machines, prior to installation and release by the app store. Useful privacy-related behaviour information can then be mined and passed on to mobile users for them to make more informed and educated choices, and improve their privacy protection. The framework consists of 3 main sub-systems: the (i) Large-Scale App Testing (LSAT) Engine, (ii) Leak-Cause Analysis (LCA) and (iii) Privacy Guidance Mechanism (PGM).

(i) Large-Scale App Testing (LSAT) Engine: This sub-system is for mass dynamic/static analysis for the purpose of uncovering runtime privacy behaviours for a large number of apps. It consists of the Automated Tester, Static Analyzer, Privacy Detectors and the Coordinator & Event Logger components. The Automated Tester performs automated exploration of the app, with the aim of covering all possible UI windows that users would reach. The Static Analyzer performs static analysis on decompiled mobile app binaries, for the purposes of utilizing knowledge obtained from static analysis to aid in automated testing. This is so that testing times required can be reduced, by pruning the search space required for testing. In addition, static analysis allows the uncovering of a subset of activities that are suspected to leak or access user privacy data. Such information might be useful to users for privacy protection.

The Privacy Detectors component of LSAT detects if any privacy data on the test device is being leaked to the network, and outputs timestamped leak reports to test logs for offline analysis. I utilized the TaintDroid [24] and ProtectMyPrivacy (PMP) [48] privacy detectors to generate leak reports for privacy leaks and accesses respectively in this work.

The last component, the Coordinator & Event Logger, performs 2 roles in the system. The 1st is to coordinate testing activities on parallelized testers, and the 2nd is to capture and log all useful usage traces from automated testing for subsequent analysis. The usage traces include information on UI activities reached, window views, widgets clicked upon and app screen-captures. Such usage traces were obtained by instrumenting and extending the Android Framework and customizing the operating system (OS) image [20]. These traces allow the deduction and reconstruction of app states that occurred during automated testing. These traces are analyzed in the Leak-Cause Analysis described next.

(ii) Leak-Cause Analysis (LCA) Subsystem: The Leak-cause analysis subsystem's purpose is to mine test logs for useful privacy characteristics of mobile apps tested by the Large-Scale App Testing (LSAT) Engine, in particular privacy leak-causes of apps. It does so by the process of Frequent Item-set mining, to output leakage patterns of tested apps with sufficiently high support values. These leakage patterns include the sequences of user-events that cause privacy leaks, as well as the privacy leakage characteristics (whether privacy leaks are user-triggered, and/or occur at start-up or in a periodic fashion).

In addition to leak patterns, the LCA also performs superimposition of app screen captures with button/widget overlay boxes. This functions as a visual aid in user education. The validation process of LCA is fully automated, and records leak patterns and behaviours into a *'Leak-Cause Database'* for a knowledge-base reference to users.

(iii) Privacy Guidance Mechanism (PGM) Subsystem: The Privacy Guidance Mechanism (PGM) Subsystem serves to utilize the 'Leak-Cause Database' generated by the LCA to provide interactive notifications to the user. The notifications can be utilized in a variety of ways with good flexibility: For example in a privacy manager containing a summarized list for users to refer to privacy behaviours of apps. Or in the form of privacy messages which appear to the user during app operation.

Information from the 'Leak-cause Database' can be transmitted locally from the user device itself or by HTTP from a cloud-based reporter. Privacy notifications is performed by a notifications PGM client, which is a customized light-weight Android app that runs in the background as an Android service and detects the app usage state of the user, performing notifications at the required moments.

2.2 Hybrid Static/Dynamic Analysis Solution (MAMBA System)

The overall system diagram for my proposed Hybrid Static/Dynamic Analysis system is displayed in Figure 2.2 - MAMBA System [35]. This can be considered the design and solution for the 'Large-Scale App Testing (LSAT) Engine' of my Privacy Analysis Framework of Section 2.1.1.

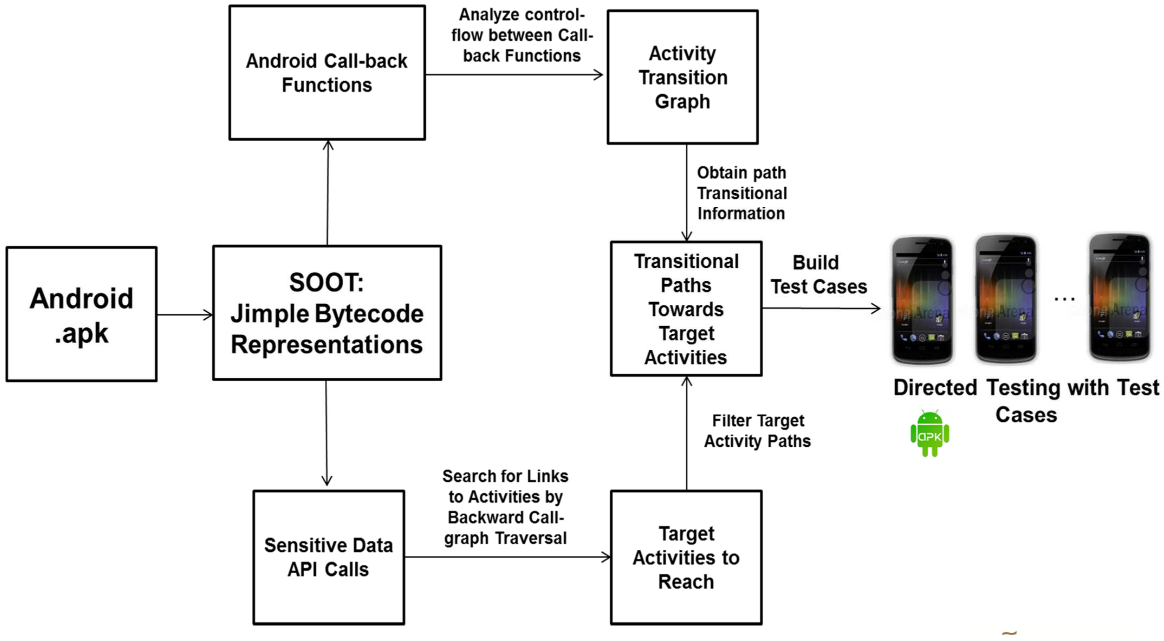

Android Call-back Functions
Analyze control-flow between Call-back Functions
Activity Transition Graph
Obtain path Transitional Information
Android .apk
SOOT: Jimple Bytecode Representations
Transitional Paths Towards Target Activities
Build Test Cases
…
Directed Testing with Test Cases
APK
Filter Target Activity Paths
Search for Links to Activities by Backward Call-graph Traversal
Sensitive Data API Calls
Target Activities to Reach

Figure 2.2: System Diagram of Proposed Hybrid Static/Dynamic Analysis System for Uncovering User-Triggered Causes and Paths of Privacy Leaks in Android Apps (MAMBA System)

From Figure 2.2, an Android .apk is decompiled into its the Jimple [58] bytecode representation. An analysis is first performed to uncover the targeted Android activities that should be reached. This is done by backwards callgraph traversal from the invocation points of Sensitive Data API calls in the bytecode.

A data/control-flow analysis is then performed on Callback functions to output an Activity Transition Graph (ATG). The ATG contains path transition information that allows the activity transitions from starting to any Android activity within the app. Information on the user events required (clickable buttons/widgets) to enable these transitions are also contained in the ATG. Knowledge on target activities is then used to filter the ATG paths, so that test cases can be built for a runtime tester to transit the app from root to any of the target activities, where the privacy access/leak behaviours are verified at runtime.

I describe the components of my solution in the following sub-sections:

2.2.1 Association of App Activities/Views with Privacy Sensitive APIs

The solution starts with identifying Android App Activities/Views that are association with privacy-related API calls. (The algorithm for association of app activities with privacy sensitive APIs is in Algorithm 1 of Appendix A.) From Figure 2.2, the binary .apk file of the Android app is first decompiled using SOOT's Android Dalvik bytecode conversion to the Jimple Representation [13]. All privacy sensitive data API invocation points which obtain some form of user data (e.g. GPS location, Contact List, Phone No., Identifiers etc.) are located. These APIs were sourced from the Android API documentation [46]. (Please refer to Appendix A for the list of APIs) My system performs a back-ward callgraph traversal from these invocation points, and identifies the app activities which have methods along the callgraph. Such activities are then designated as Target Activities that my Automated Runtime App Testing Solution should reach.

From Algorithm 1, 2 types of target activities are outputted: 'Target Activities to Reach' $\tau 1$, as well as 'Target Activities to Test' $\tau 2$. For $\tau 1$ activities, it is sufficient to perform runtime testing only to the point of reaching the activity, before the privacy accesses/leaks occur. This is because the privacy sensitive APIs exist under methods which are part of the Android creation lifecycle. For $\tau 2$ activities, privacy sensitive APIs exist under 'User-Input' handling methods (e.g. onClick). It is thus required for runtime testing to reach as well as perform testing (by enacting clickable buttons/widgets) on the activities.

Android activities that have privacy sensitive API invocation points that are neither under Android creation lifecycle or 'User-Input' handling methods are designated under the $\tau 2$ activity group.

2.2.2 Analysis of Control-Flow Between Android Callbacks

The next step in my solution involves analysis of the control flow between Callback Functions of the Android Framework to build an Activity Transition Graph (ATG) representation for guiding automated runtime testing. Android Callback Functions are event-driven. They can be driven by the user (e.g. by the user performing an action), or coordinated and driven by the system (e.g. on Activity creation/destruction, on closing of windows etc.). Because of this event-driven nature, the control-flow between Callback functions will not be available from a normal control-flow graph analysis.

To solve the problem, Yang et al. [63] performed modelling of Android callback functions, and made their work available as a tool- GATOR tool [53]. Initial versions of my system were built on top of the GATOR tool and its modelling of Android Callback functions and GUI objects. However, it was discovered that the GATOR tool suffers from generalizability issues whereby the number of apps which could be successfully analyzed was low. On performing a larger-scale evaluation, only about 80 of 600 (13.3%) Android apps could be successfully analyzed. I hence implemented my own analysis of callback functions.

A possible reason for GATOR's generalizability problems is that it tries to do very detailed modelling of control flows (e.g. The exact sequences of control flows, as well as the types and characteristics of the views being inflated). Hence it might not have been able to handle the complexities with different app implementations.

My analysis outputs Fully-Directed as well as Semi-Directed Activity Transitional Graphs (ATG) and test cases for automated testing. These are described as follows:

Fully-Directed Activity Transitional Graph (ATG)

In the Fully-Directed ATG, exact widget identifier information is contained in the activity transitional edges. The automated runtime testing thus only requires stimulating a single user-input to cause an activity transitions in each ATG edge. I formulated and implemented an algorithm for callback function analysis over SOOT to generate these ATG edges. The algorithm is based on the analysis of Android User-Input Listener Invocation Points and the dataflow tracking of listener objects across Android callback functions to identify the corresponding GUI element involved, as well as source and target activities. **(Please refer to Algorithm 2 in Appendix A)**

Figure 2.3 illustrates examples of resulting Fully Directed test cases for example activities of 2 apps: 2 different paths were found for '.PremiumActivity', the 1st shorter path being a single user-action on the widget with identifier 'action_premium', and the 2nd longer path involving transition with 2 user-actions ('action_info' and 'premiumUpgradeButton'). The other example for the activity '.koisettings' found 4 different possible paths from root to the activity, the 1st path requiring 2 user-actions, and 2nd-4th paths requiring 3 user-actions.

Target Activity: com.bitsmedia.android.quitpro.activities.PremiumActivity

(App: com.bitsmedia.android.quitpro.apk)

Test Case #1 : [action_premium]

Test Case #2 : [action_info, premiumUpgradeButton]

Target Activity: com.interestingcoolerfreegoimbh.app.koisettings

(App: com.interestingcoolerfreegoimbh.apk)

Test Case #1 : [settings, button1unlock]

Test Case #2 : [settings, RelativeLayout04, gridview]

Test Case #3 : [settings, RelativeLayout05, gridview2]

Test Case #4 : [settings, RelativeLayout03, gridviewback]

Figure 2.3: Example: Format of Fully-Directed Test Cases Built from Callback Function Analysis For Automated Testing

To increase accessibility and improve testing time, multiple paths for each activity were ordered according to length of shortest to longest, and executed shortest paths first before trying out longer paths if the shorter paths were found to be infeasible. The automated tester executes shorter paths before moving to try out the longer paths if it is unable to reach the resulting activity.

While a Fully-Directed ATG is highly efficient in guiding automated runtime testing, the generalizability of the analysis is still imperfect in uncovering paths for all apps. For apps in which fully-directed ATG cannot be obtained, the Semi-Directed ATG is utilized to generate test cases. This is described in the following sub-section.

Semi-Directed Activity Transitional Graph (ATG)

In the Semi-Directed ATG, the Android callback function analysis does not directly output exact button/widget identifier information. Semi-Directed test cases have a much higher successful generation rate ($> 70\%$) than fully directed test cases due to the simpler analysis involved. The analysis is performed over Android intents to identify activity transitions using a source-sink concept [10]. A simple listener analysis is performed to identify actionable GUI elements within each activity. The lists of actionable widgets are provided in the ATG information for use by the automated runtime tester. **(Please refer to Algorithm 3 in Appendix A)**

Automated runtime testing proceeds by trying each potential widget identifier associated with the current activity, and determining if the resulting activity reached corresponds to the target node of the current ATG edge. if the target node of the current edge is not reached, a 'back track' command is issued to bring the app back to the source activity. The next potential widget is then tried until the next activity in the ATG edge is reached.

Upon reaching the target node in the ATG, the remaining actionable buttons/widgets identifiers in the source node of the edge is discarded and replaced with that of the target node. In this way, activity transitions proceed until the target activity containing privacy sensitive APIs is reached.

Figure 2.4 displays the format of the semi-directed test cases for 2 target activities of apps. The transiting activities are in bold, with the widget identifiers contained within the brackets.

Target Activity: com.interestingcoolerfreegoimbh.app.koisettings

(App: com.interestingcoolerfreegoimbh.apk)

Test Case:

[**com.interestingcoolerfreegoimbh.app.menuicon**{settings,imageView1,morefreapps ,setwallpaper}; **com.interestingcoolerfreegoimbh.app.GalleryWallpaperSettings**{image View1,butt2a,butt1,button1unlock,tvSeekBar11};**com.interestingcoolerfreegoimbh.app .koisettings**{unlockbuttommain,backone,textapp}]

Target Activity: com.lyrebirdstudio.promodialog.PromoActivity

(App: com.lyrebirdstudio.montagenscolagem.apk)

Test Case:

[**com.lyrebirdstudio.montagenscolagem.SelectPhotoActivity**{my_awesome_toolbar, montagens_show_case, montagens_main_ad_id};**com.lyrebirdstudio.promodialog .PromoActivity**{}]

Figure 2.4: Example: Format of Semi-Directed Test Cases Built from Callback Function Analysis For Automated Testing

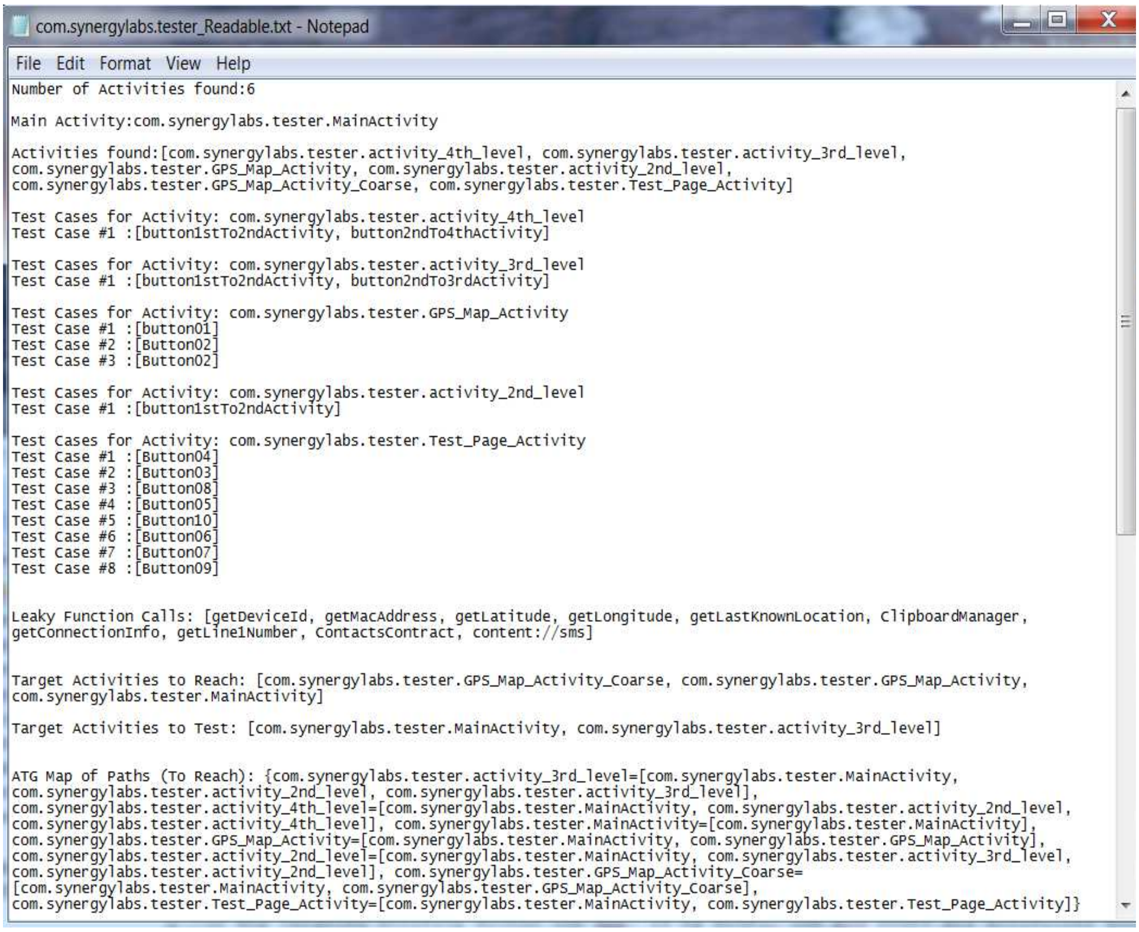
In the example of Figure 2.4, the test case directs the app towards target activities called 'com.interestingcoolerfreegoimbh.app.koisettings' and 'com.lyrebirdstudio.promodialog.PromoActivity'. In order for 'com.interestingcoolerfreegoimbh.app.koisettings' to be reached, it has to transit through 2 other activities, ’com.interestingcoolerfreegoimbh.app.menuicon’ and ’com.interestingcoolerfreegoimbh.app.GalleryWallpaperSettings’. It does so by filtering and only stimulating user action on the widget identifiers within the brackets (e.g.{settings, imageView1, morefreapps, setwallpaper}), and discarding the rest which are not involved in enabling transitions.

2.2.3 Reports on Activity Transition Paths and Results of Static Analysis

To facilitate reporting of static analysis results, human-readable reports in text format were outputted in addition to the test cases. Figure 2.5 illustrates the report outputs of an example app. As can be seen from the report, the information provided includes:

- Total number/names of activities
- Main (Root) activity of app
- Widget identifiers in directed test cases required to reach each of the activities
- Privacy infringing APIs found
- Target activities that have callgraph links to privacy APIs
- ATG map containing the required activity transitions of each of the activities in the app
- Widget identifiers corresponding to the ATG map

It should be highlighted that the MAMBA system outputs the results of static analysis including the ATG and paths of all app activities in the report, but filters results to generate test cases for automated runtime testing to reach only the target activities.



com.synergylabs.tester_Readable.txt - Notepad
File Edit Format View Help

```
Number of Activities found:6

Main Activity:com.synergylabs.tester.MainActivity

Activities found:[com.synergylabs.tester.activity_4th_level, com.synergylabs.tester.activity_3rd_level,
com.synergylabs.tester.GPS_Map_Activity, com.synergylabs.tester.activity_2nd_level,
com.synergylabs.tester.GPS_Map_Activity_Coarse, com.synergylabs.tester.Test_Page_Activity]

Test Cases for Activity: com.synergylabs.tester.activity_4th_level
Test Case #1 :[button1stTo2ndActivity, button2ndTo4thActivity]

Test Cases for Activity: com.synergylabs.tester.activity_3rd_level
Test Case #1 :[button1stTo2ndActivity, button2ndTo3rdActivity]

Test Cases for Activity: com.synergylabs.tester.GPS_Map_Activity
Test Case #1 :[button01]
Test Case #2 :[Button02]
Test Case #3 :[Button02]

Test Cases for Activity: com.synergylabs.tester.activity_2nd_level
Test Case #1 :[button1stTo2ndActivity]

Test Cases for Activity: com.synergylabs.tester.Test_Page_Activity
Test Case #1 :[Button04]
Test Case #2 :[Button03]
Test Case #3 :[Button08]
Test Case #4 :[Button05]
Test Case #5 :[Button10]
Test Case #6 :[Button06]
Test Case #7 :[Button07]
Test Case #8 :[Button09]


Leaky Function Calls: [getDeviceId, getMacAddress, getLatitude, getLongitude, getLastKnownLocation, ClipboardManager,
getConnectionInfo, getLine1Number, ContactsContract, content://sms]


Target Activities to Reach: [com.synergylabs.tester.GPS_Map_Activity_Coarse, com.synergylabs.tester.GPS_Map_Activity,
com.synergylabs.tester.MainActivity]

Target Activities to Test: [com.synergylabs.tester.MainActivity, com.synergylabs.tester.activity_3rd_level]


ATG Map of Paths (To Reach): {com.synergylabs.tester.activity_3rd_level=[com.synergylabs.tester.MainActivity,
com.synergylabs.tester.activity_2nd_level, com.synergylabs.tester.activity_3rd_level],
com.synergylabs.tester.activity_4th_level=[com.synergylabs.tester.MainActivity, com.synergylabs.tester.activity_2nd_level,
com.synergylabs.tester.activity_4th_level], com.synergylabs.tester.MainActivity=[com.synergylabs.tester.MainActivity],
com.synergylabs.tester.GPS_Map_Activity=[com.synergylabs.tester.MainActivity, com.synergylabs.tester.GPS_Map_Activity],
com.synergylabs.tester.activity_2nd_level=[com.synergylabs.tester.MainActivity, com.synergylabs.tester.activity_3rd_level,
com.synergylabs.tester.activity_2nd_level], com.synergylabs.tester.GPS_Map_Activity_Coarse=
[com.synergylabs.tester.MainActivity, com.synergylabs.tester.GPS_Map_Activity_Coarse],
com.synergylabs.tester.Test_Page_Activity=[com.synergylabs.tester.MainActivity, com.synergylabs.tester.Test_Page_Activity]}
```

Figure 2.5: Snippet of Human-Readable Report in Text Format on Results of Static Analysis (MAMBA System)

2.2.4 Automated Runtime App Testing

As mentioned, the Activity Transition Graph (ATG) generated from the analysis in the previous Subsection 2.2.2 is used to guide traversal of the app from root (main) activity to any targeted activity within the app. In my model, the ATG nodes are activities and graph edges are the directed edges between each activity. Directed edges contain callback function information of activities, which include widget identifiers of the GUI components that will cause transitions between any 2 activities defined in the app. Widget identifiers are utilized for edges because Android's Hierarchy Viewer [47] provides visibility on clickable Android UI components from these identifiers. The ATG representation is shown in Figure 2.6. GATOR's CCFG class was utilized as a stub to generate the ATG abstraction.

From Figure 2.6, the callback function contained by the ATG node as well as the activity name is highlighted in bold. In the example, the callback function of the node is 'onCreate', which represents the starting up of an Android activity. While the activity containing this callback is of the name 'com.bitsmedia.android.quitpro.activities.PremiumActivity'. The edge contains a source as well as a target node, and also illustrates the callback functions as well as activity names in bold. A descriptor is present at the start of the node, indicating the type of node. In this particular example of the ATG edge, the start node is of an Android view inflation type, represented by 'Inflate_Node' while the target node is of an activity start type.

Node (Activity with Callback Functions):

[START] Node ⟨Activity[com.bitsmedia.android.quitpro.activities.PremiumActivity],⟨com.bitsmedia.android.quitpro.activities.PremiumActivity: void **onCreate**(android.os.Bundle) ⟩ ,implicit_lifecycle_event,Activity[**com.bitsmedia.android.quitpro.activities.PremiumActivity**] ⟩

Edge (Source and Target Activities together with Their Callback Function):

Node⟨Inflate_Node[android.widget.Button,WID[**premiumUpgradeButton**]],⟨com.bitsmedia.android.quitpro.activities.InfoActivity$1: void **onClick** (android.view.View)⟩, click, ACT[**com.bitsmedia.android.quitpro.activities.InfoActivity**]⟩ -> [START] Node⟨Activity[com.bitsmedia.android.quitpro.activities.PremiumActivity],⟨com.bitsmedia.android.quitpro.activities.PremiumActivity: void **onCreate** (android.os.Bundle)⟩,implicit_lifecycle_event,Activity[**com.bitsmedia.android.quitpro.activities.PremiumActivity**]⟨ [normal, start_activity, [window_must_start]]

Figure 2.6: Node & Edge Representations in ATG, containing Callback Information (Represented with GATOR tool's CCFG class, which was utilized as a stub)

I built an automated walking tool on top of AMC, which utilizes test cases generated from the ATG as inputs to guide its testing. Each test case is made up of the user actions that are required to be sent to the UI to cause activity transitions, which are executed in sequence one after another. The testing thus proceeds by

’walking’ the application from the initial root activity to a target activity, before terminating and restarting from the root activity again for the next test case. Testing now proceeds in a more deterministic and guided fashion, which improves testing efficiency.

Multiple Paths & Path Feasibility: The analysis encountered cases whereby there were multiple paths from an activity back to the root activity. While my solution logs all possible paths, it does not identify which paths are actually feasible or can be reached during run-time. Infeasible paths might occur if some condition is required before the next resulting activity can be started. For example, an activity might only be accessible if the user is on WiFi. Another example could be in a shopping app, in which a checkout activity might only be accessible if the user had checked on certain items in a list.

2.2.5 Analysis of Causes of User-Triggered Leaks & Characteristics

As mentioned, MAMBA’s automated runtime testing produces log files that contain execution traces of the automated tester, information on Android activities and buttons/widgets that are reached as well as privacy access and leak reports from ProtectMyPrivacy (PMP) [48] and TaintDroid [24]. MAMBA processes the resulting leak and event log files to obtain useful outputs on the privacy behaviours of apps.

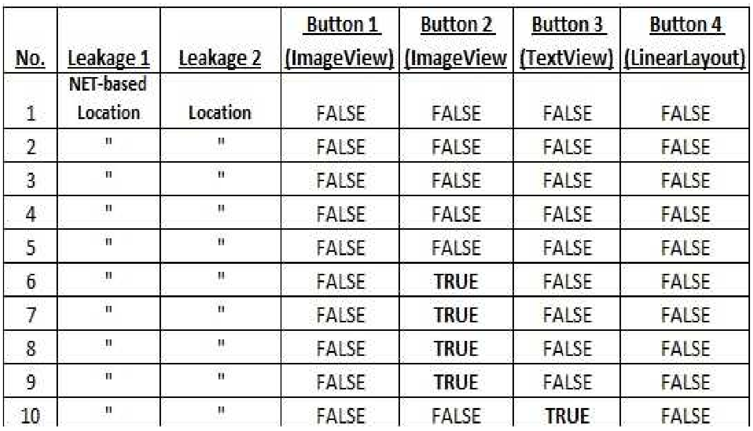
Analysis on App Usage Logs without Static Analysis Outputs

An early version of the tool [37] was based on manual/automated testing without knowledge from static analysis, and relied on uncovering user-triggered causes of privacy leaks (leak-cause analysis [36]) using Association Rule Mining [4]. In this leak-cause analysis, I established linkages between user actions to privacy leaks using WEKA [30]. The tool converts raw log files to .csv format for input and processing, and outputs the results into a ’Leak-cause Database’ for storage.

Association Rule Mining [4] identified *correlation* based on how frequently two concerned events (a particular user-action and a particular leak report) occured together within a chosen time window (5 seconds). The assumption and intuition for applying this technique is as follows: the leaking behaviour of an application stays constant with time, similar behaviour is expected to occur repeatedly if sufficient logs are collected. Thus patterns can be extracted to identify causes of various leaking behaviour. For example, if a news app is reported to leak IMEI as many times as the ”Load More News” button was touched by the user, it can be inferred that the user-action of touching this button is highly correlated with the leak of the IMEI in this app.

The outputs of the analysis are expressed as *association rules* whose left sides indicate the causes of the leaks on the right sides. For example: Click of ”Load More News”$\rightarrow$ Leak of IMEI. I utilized the Apriori algorithm available in WEKA to obtain such association rules with absolute support and confidence thresholds of 2 and 0.5 respectively. The thresholds were tuned based on samples of 20 apps, and fitted reasonably well with the test logs. For utilization with WEKA, my tool transforms raw logs into a tabular format as displayed in Figure 2.7. In this format, each clickable component in the app has a column representing co-occurrence with leak within the 5 second time window. If yes, it is indicated by a ’True’, otherwise a ’FALSE’ appears in the tab.

Non user-triggered privacy behaviours (Start-up/Periodic leaks) are also uncovered by the mining process. For Start-up leak behaviours, instances of app launches were co-located with leak events within the 5 second window. While an app was deemed to have periodic privacy leak behaviours when there were consistent occurrences of leak events at every minute time interval.



No.	Leakage 1	Leakage 2	Button 1 (ImageView)	Button 2 (ImageView	Button 3 (TextView)	Button 4 (LinearLayout)
1	NET-based Location	Location	FALSE	FALSE	FALSE	FALSE
2	"	"	FALSE	FALSE	FALSE	FALSE
3	"	"	FALSE	FALSE	FALSE	FALSE
4	"	"	FALSE	FALSE	FALSE	FALSE
5	"	"	FALSE	FALSE	FALSE	FALSE
6	"	"	FALSE	TRUE	FALSE	FALSE
7	"	"	FALSE	TRUE	FALSE	FALSE
8	"	"	FALSE	TRUE	FALSE	FALSE
9	"	"	FALSE	TRUE	FALSE	FALSE
10	"	"	FALSE	FALSE	TRUE	FALSE

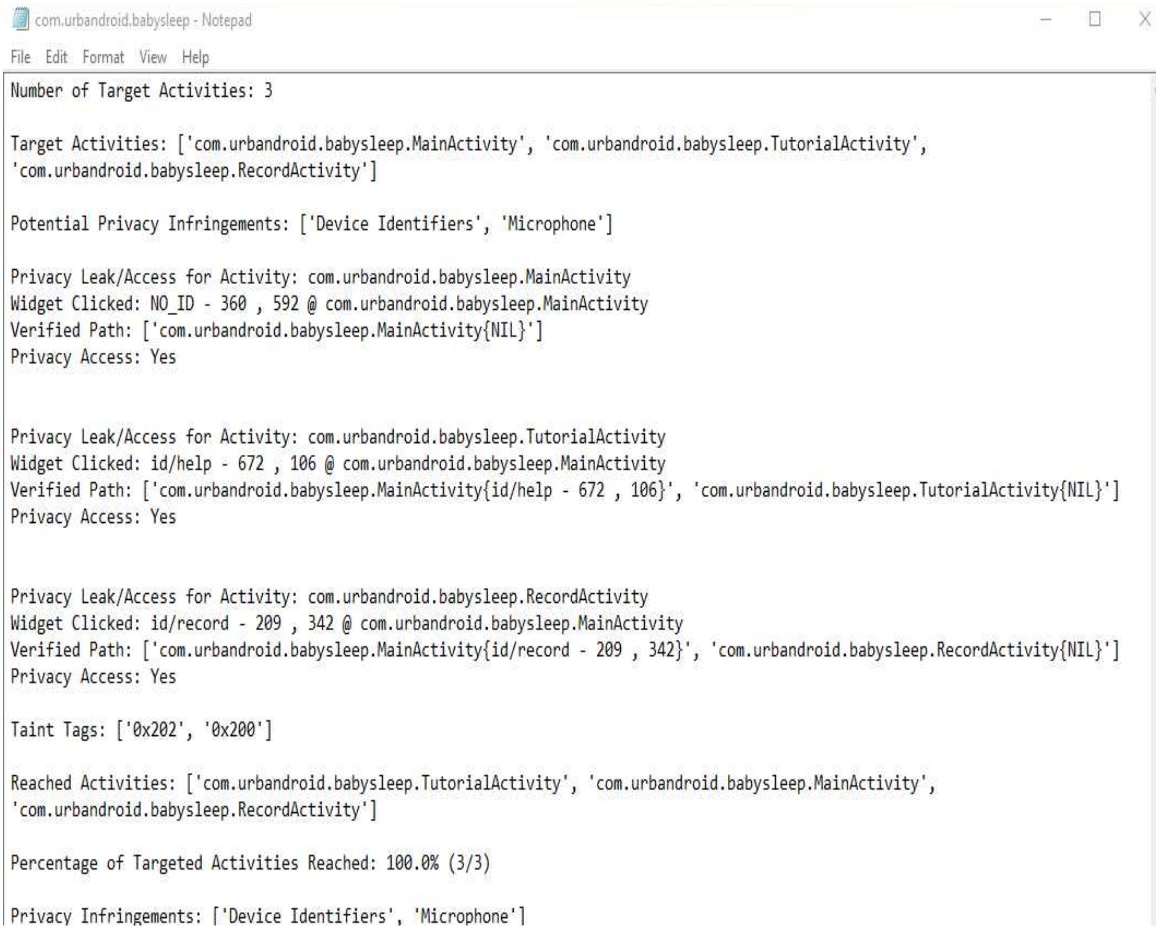
Figure 2.7: Example of Transformed WEKA Table from Raw Logs

MAMBA's Output of Runtime Verified Privacy Causes

MAMBA outputs a text format human-readable report of runtime verified privacy causes from the static analysis (Please refer to Figure 2.8). The report lists the number and names of target activities (activities with privacy-related APIs), target activities that automated testing managed to reach and coverage percentages, verified runtime paths towards target activities, the button/widget identifiers acted upon at runtime to reach these target activities as well as their screen coordinates, privacy access/leak reports associated with activities, as well as their privacy data types.

Figure 2.8 displays an example of a final report of runtime verification by MAMBA for an app (com.urbandroid.babysleep.apk). The report lists that all of the 3 target activities ('MainActivity', 'TutorialActivity' and 'RecordActivity') were reached by automated testing, and had privacy accesses of device identifiers as well as microphone sensor. The button/widget identifiers and their corresponding on-screen coordinates which were verified to reach each of these activities are displayed under the tag 'Widget Clicked'. In this example, the buttons with the 'id/help' and 'id/record' identifiers were clicked upon to reach activities 'TutorialActivity' and 'RecordActivity' respectively. Each of these identifiers had on-screen display coordinates of '672 , 106' and '209 , 342'.

The verified runtime paths towards these target activities are also displayed in the example Figure 2.8 under the tag 'Verified Path'. The example shows that the


com.urbandroid.babysleep - Notepad
File Edit Format View Help

```
Number of Target Activities: 3

Target Activities: ['com.urbandroid.babysleep.MainActivity', 'com.urbandroid.babysleep.TutorialActivity',
'com.urbandroid.babysleep.RecordActivity']

Potential Privacy Infringements: ['Device Identifiers', 'Microphone']

Privacy Leak/Access for Activity: com.urbandroid.babysleep.MainActivity
Widget Clicked: NO_ID - 360 , 592 @ com.urbandroid.babysleep.MainActivity
Verified Path: ['com.urbandroid.babysleep.MainActivity{NIL}']
Privacy Access: Yes


Privacy Leak/Access for Activity: com.urbandroid.babysleep.TutorialActivity
Widget Clicked: id/help - 672 , 106 @ com.urbandroid.babysleep.MainActivity
Verified Path: ['com.urbandroid.babysleep.MainActivity{id/help - 672 , 106}', 'com.urbandroid.babysleep.TutorialActivity{NIL}']
Privacy Access: Yes


Privacy Leak/Access for Activity: com.urbandroid.babysleep.RecordActivity
Widget Clicked: id/record - 209 , 342 @ com.urbandroid.babysleep.MainActivity
Verified Path: ['com.urbandroid.babysleep.MainActivity{id/record - 209 , 342}', 'com.urbandroid.babysleep.RecordActivity{NIL}']
Privacy Access: Yes

Taint Tags: ['0x202', '0x200']

Reached Activities: ['com.urbandroid.babysleep.TutorialActivity', 'com.urbandroid.babysleep.MainActivity',
'com.urbandroid.babysleep.RecordActivity']

Percentage of Targeted Activities Reached: 100.0% (3/3)

Privacy Infringements: ['Device Identifiers', 'Microphone']
```

Figure 2.8: Human-Readable Final Report in Text Format of MAMBA's Runtime Verification

target activities had user-triggered activity transition paths from the main activity 'MainActivity'.

The identifer and coordinate information can be utilized by the customized Interactive Privacy Notifications App described in Section 2.3. Whenever the Notifications App detects that the button/widget/view with identifier and coordinates is being displayed on the user device, it fires off a privacy message which overlays itself on top of the button. The privacy message displays the data type as well as privacy implication (data access or leak) of user action. In this manner, users can be provided with better and more in-depth context of the privacy implications of using their apps.

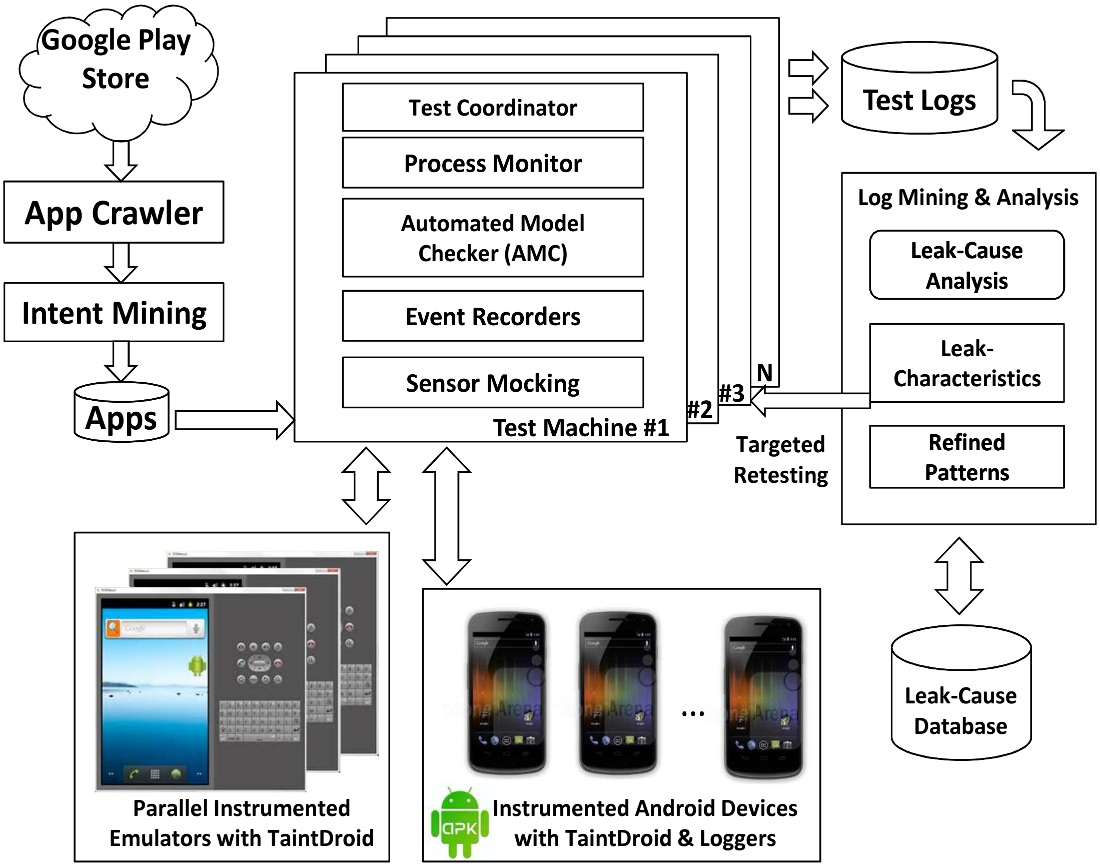


Figure 2.9: Architecture of Parallelized Automated Application Testing System

2.2.6 Scaling up App Testing By Parallelized Test Instances

In order to scale up application testing, I leveraged on parallelization over multiple Android emulator test instances on commercial clouds (Amazon EC-2) and physical test phones. Figure 2.9 displays the system architecture for the automated app testing system.

A *Test Machine* is a single CPU running either on the Amazon EC-2 cloud or a physical CPU tethered to physical test phones. Each Test Machine runs a test coordinator program for control and coordination of multiple *Testing Instances*. Each Testing Instance is a customized Android OS image, that runs the Android Framework and leak detection instrumentations. These instrumentations allow output of information about various app states (e.g. app starts, widget draw events, listeners etc.) and actions of the user (click, key events). From Figure 2.9, each Test Machine also runs Process Monitors, Automated Model Checker (AMC), Event Recorders as well as Sensor Mockers to enable Mobile Forensics testing on the emulator. The Parallelized Automated Testing System runs on each Test Machine, and has a role of ensuring smooth and continual testing of mobile apps in a sequential fashion. It contains fault handling routines for the Android emulator, which is known to suffer from stability issues [31].

The resulting event logs obtained from the Test Machines are processed by the Log Mining & Analysis Sub-system. Leakage patterns associates the data leaks to the actions captured in the logs. Patterns are obtained and stored in a Leak-Cause Database.

The following component modules of the Parallelized Automated App Testing System, displayed in Figure 2.9, are described as follows:

1) Application Crawler & Intent Miner: I utilized an existing Android Market API [17] to create a crawler that discovers and downloads a large number of Android apps (.APK) from the Google Play Store. With the term search API, a 32,000 word dictionary [22] was used to discover a list of about 200,000 unique apps from the store (This is not exhaustive, and more can actually be discovered). Using the list, the crawler then downloads apps at a rate of 500 apps/hour.

The Intent Miner module extracts apps' start-up intents and permissions from their manifest files. The start-up intent provides the means to start the app from the Test Coordinator, while permissions identify the platform (device or emulator) the app is suitable to be tested on. The reason is that Android emulators lack microphone emulation, hence apps that require the microphone permission is directed to physical test phones instead of the emulator.

2) Test Coordinator: The Test Coordinator ensures that the testing of the applications on each Testing Instance proceeds in a smooth and sequential fashion. The inputs to the Test Coordinator are application apk files placed in a specified folder, and the outputs are the corresponding test logs saved by individual application package names. The Coordinator installs the applications on each available Testing Instance, starts it up, saves the logs upon completion of test and starts up the testing of the next application automatically. The logs saved include the leak-reports from TaintDroid, widget and page transition data, all in-application widget and button clicks/interactions made by the AMC's MonkeyRunner as well as screen captures of all unique pages traversed.

3) Process Monitor: To ensure smooth testing and deal with instabilities with the Android Emulator and the Android Device Bridge(ADB), I developed a Process Monitor to manage the test processes such as the Android Debug Bridge (ADB), Model-Checking as well as logger processes. Problems in application testing on the Android Emulator have been highlighted previously [31], with some works specifically avoiding to run their test processes on the Emulator due to their slow speed and instabilities. Issues with emulators are aggravated especially for continuous and prolonged running with multiple instances. It was also observed that zombie ADB processes tend to accumulate at the end of each application test run, which could cause additional unwanted memory loads on the test machines causing slowdowns. As such, the Process Monitor was specifically designed for unhindered testing on the Android Emulator.

The Process Monitor tracks and monitors the Android Emulator, ADB and other running binary processes of the front-end testing system. Various fault handling routines were implemented, including the shutting and restarting of the Emulator, ADB and Android Linux system processes before and during each test run, if various stalling conditions occurred. Heuristics were used to determine stalling or hanging conditions in the test instances. The heuristics were based on whether the size of test logs and number of images being captured increased continuously during the testing runs. If the logs captured did not increase by a specified time (3 minutes), the Process Monitor shuts down and restarts the individual emulator and Android SDK processes automatically and continues with the testing instance until completion. On emulator crashes, appending of logs from the previous testing process was performed. These implementations worked well to allow proper and uninterrupted sequential testing of applications.

4) Automated Model Checker (AMC): The default app testing tool utilized is the Automated Model Checker (AMC) [39] to automatically traverse and explore the application GUIs. This module is readily switched with my MAMBA automated tester that I created (Described in Section 2.2). Switching of tester is done with a simple executable binary file swap.

5) Event Recorder: Event Recorders were implemented for each testing instance while running the Automated Model Checkers on the test instances. The Event Recorder automatically logged and saved the traces of the user-interaction with the layouts and widgets of the application. Data logged included page-transition notices, widgets and buttons tapped or clicked by the model checker, coordinates of buttons and layouts accessed by the model checker as well as taint reports provided by TaintDroid. This includes a screen-capture component to take and save images of unique pages and layouts in the applications accessed by the model checker. As the model checker could possibly access the same application page multiple times, heuristics involving simple matching of some layouts in the pages accessed were utilized to prevent multiple captures of the same application page to save storage space.

6) Sensor Mocking: I implemented a simple Sensor Mocking program to send fake running GPS coordinates to the emulator test devices through the telnet ports. The purpose of which to elicit similar behaviours and data-leaks to physical phone devices. The mocking of other sensors besides GPS can be accomplished by the open-source project Sensor Simulator [28].

2.3 Notifying Users with Privacy Outputs

With the outputs of user-triggered causes and privacy characteristics of apps uncovered in Section 2.2, the next part of my solution involves effective utilization of its outputs to improve user privacy. Improvement in user privacy is sought, which is observed in the form of reduction in user behaviours that cause privacy accesses and/or leaks. In addition, I seek to investigate if the outputs improve privacy data recollection and knowledge of the user.

Creation of Visual Leaky Layouts

Creation of Visual Leaky Layouts: For easier notification and illustration of leak causes and potential users of the rules for end users, a tool was implemented [37] to utilize the outputs of leak-cause analysis of Section 2.2.5. The tool creates visual presentations by automatically overlaying outlined coloured boxes on the top of leaky widgets in the app screen-captures. This enables the leak-cause information to be much more user friendly and easier for users to digest. An example of the generated visual leaky layout is displayed in Figure 2.10. Such visual layouts can be utilized by an analyst to make quick determinations as to whether apps are appropriately utilizing privacy data. They can be also shown to app users for more informed access control decisions of their apps.

Interactive Privacy Leak Notification App

To utilize the outputs of the hybrid static/dynamic app privacy system, an Interactive Privacy Leak Notifications app was also developed. This Notifications app works on un-rooted phones, and pulls data from the knowledge-base stored in the 'leak-cause' database. This app runs an Android service on the user phone, which continuously monitors the app state that the user is on. On detection of that the user is on a desired app page, the Notification app fires of privacy messages using Android Toast Messaging [21]. The toast messages appear on: (i) App Start-Ups, (ii) On the

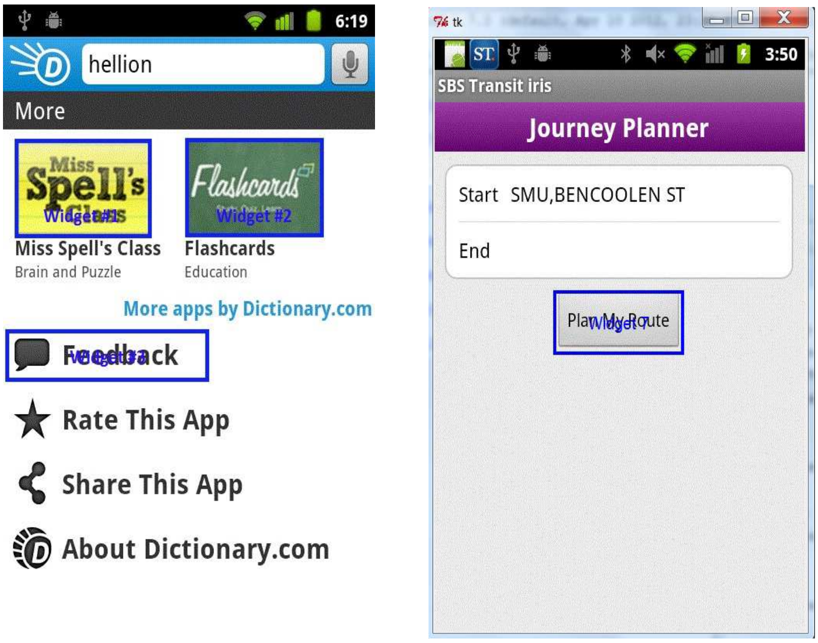


Figure 2.10: Visual Overlay of Coloured Boxes on top of Leaky Widgets in App Screen-Captures by Leak-Cause Analysis Tool

appearance of certain in-app layouts and (iii) Beside culprit buttons/widgets within the app. The Notifications app works by reading event messages provided by the Android Accessibility Service [19].

The notification app parses accessibility event messages to infer page transitions, the exact application that users are in as well as the page-layout that users are currently accessing to determine the exact moments to display notifications to users. An example of the formatted accessibility event service sequence 4-tuple is:

{1: com.facebook.katana

2: [”$ButtonText$”]

3: android.widget.EditText

4: Layout Element Count:6}

The first sequence of the 4-tuple above indicates the application package name, the second sequence refers to the text on the app widget clicked on by the user, the third sequence indicates the widget type that was clicked upon and the last sequence provides the element count in the particular layout. I found that for the majority of the page transitions within the applications, it was possible to sufficiently identify the current application page and state that a user is presently on for the field study in Section 4.2.

To enable proper display of notification message overlays regardless of user-phone screen size, scaling of the coordinates from the leak-cause database was performed. The coordinates to display message overlays were scaled over the test device (Galaxy Nexus) to user device screen sizes. i.e... $x - Coordinate = \frac{Display\ x-Coodinate}{Test\ Device\ Length} \times User\ Device\ Length$, $y - Coordinate = \frac{Display\ y-Coodinate}{Test\ Device\ Height} \times User\ Device\ Height.$

2.4 Initial Feasibility Study on Uncovering User-Triggered Privacy Leaks and Characteristics of Apps

In this section, I present the results of an initial feasibility study for uncovering user-triggered privacy leaks and privacy characteristics of Android apps (Privacy ’Leak-Causes’). I also investigate the taxonomy of app privacy leak behaviours. This study was performed using manual testing of apps.

Empirical Study: Taxonomy of Privacy ’Leak-Causes’

An empirical study was performed on popular apps from the Google Play store [1] to observe the proportions of apps with user-triggers, as well as to study the taxonomy of app privacy leak behaviours.

Applications: I sampled the top 10 apps from the (previously) 22 categories from the Google Play store during Nov 2012. (Total 226 apps) The 6 additional apps were randomly added to make up for cases where some apps crashed.

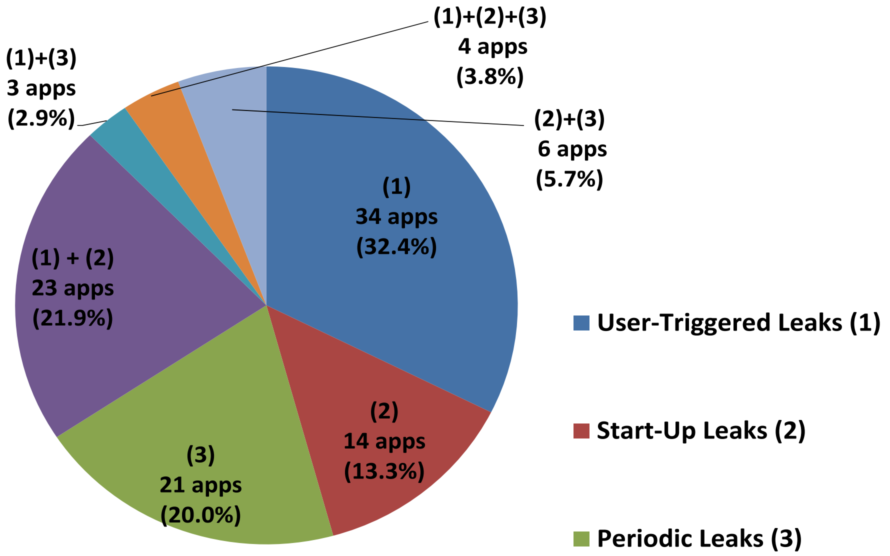


Figure 2.11: Distribution of Various Leak Causes

Testing Environment: The test was performed on a Google Nexus one phone running Android v2.3.4 with a custom ROM that includes TaintDroid leak detector [24] and the relevant OS instrumentations to capture UI events and user-actions.

Taxonomy of Leaky Applications: Out of the 226 apps studied, 121 apps were found not to be leaking private data (non-leaking apps). Among the 105 apps (46.5%) leaking privacy data, 64 apps (28.3%) were found to leak private data due to user actions on application widgets. With leak-cause analysis, I defined a taxonomy of privacy leak behaviours for apps, based on the observation of how apps can leak data: (1) User-Triggered Leaks (identified by association rules), (2) Start-Up or One-Time Leaks, and (3) Periodic Leaks. Defining such a taxonomy helps in properly classifying and describing the privacy leak behaviours of apps.

The distribution of the 105 apps with various leak causes is shown in Figure 2.11. 32.4% (34) of the leaky apps leak private data solely due to (1) User-Triggered actions on widgets. 13.3% (14) leak solely on (2) Start-Up. 20% (21) leak data solely in a (3) Periodic fashion. 21.9% (23) can leak data by either (1) User-Triggers or (2) Start-Up. 2.9% (3) leak data either by (1) User-Triggers or (3)

Periodically. 3.8% (4) applications leak data by any of the three means. 5.7% (6) have no User-Triggered leaks, but leak data both on (2) Start-up and (3) Periodically.

Distribution of Leak Data Types: Figure 2.12 shows the distribution of various types of private data that was leaked by the test apps. 9 different leaked data types have been found: {Phone Contacts, GPS Location, ICCID (SIM card identifier), IMEI, Last-Known Location, Location (Non-GPS), Microphone Input, Net-based Location, Phone Number}. It is noticed that a majority of leaks are of 3 types: IMEI, Location (Non-GPS), and Net-Based location. As most of the processes in the applications required an active internet connection, it is unsurprising perhaps that much of the private data leaked is of their Net-Based location, which is the coarse-grained geographical location obtained from mobile users's IP addresses. However, it is surprising that many applications are leaking IMEIs and non-GPS location, which is the user location obtained from cellular towers, to external servers. From our observation, IMEIs are often used (but possibly unnecessary) as a unique identifier to link mobile users to activities such as user feedback, comments, and record-keeping for access when requesting server-based information by the applications.

2.4.1 Accuracy of Association Rules Mining

User-triggered leaks and privacy characteristics were uncovered by the mining of user-triggered causes and characteristics of privacy leaks from the outputs and logs of mobile app testing described in Section 2.2.5. As mentioned, the processing of logs is performed using *Association Rules Mining* with the WEKA machine learning tool. Whereby app user-triggers and pages with privacy leaks that are correlated with privacy reports (i.e. have high support and confidence values) are outputted for building of a knowledge base in the form of a 'leak-cause' database for advisory to users.

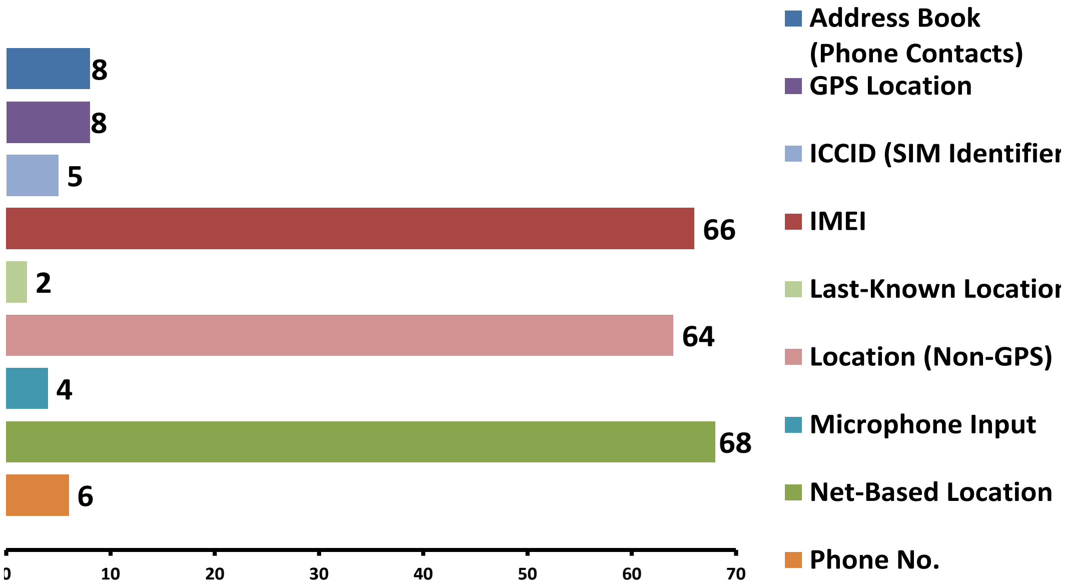


Figure 2.12: Distribution of Types of Leaked Data

Figure 2.13 displays the accuracy results of the association rules mining as a distribution. An overall mining accuracy of 67% was obtained. These accuracies were obtained based on a manual pattern validation across a set of 64 apps with user-triggered privacy leaks. Accuracy is defined to be the percentage of leak causes that have been defined to be correct, out of the total number of leak causes returned from the miner. It can be observed that the miner is very accurate (80-100%) for 37.5% (24) of the apps. However, there was a small percentage of apps with poor accuracies (<40) - 26.6% (9) apps. Some of the reasons for lower accuracies are highlighted next.

Causes of Mining Inaccuracies: Based on manual inspections of apps with lower accuracies, I highlight a few reasons. It was observed that there were situations where multiple non-leaky widgets were often situated in close proximity to a leaky widget on the screen layouts. This often caused false positives due to these non-leaky widgets showing up as false candidates in the Association Rules mining. Co-occurrences of start-up/periodic leaks together with user-triggered leaks can confuse the rule mining and cause false positives. Also, there are situations in

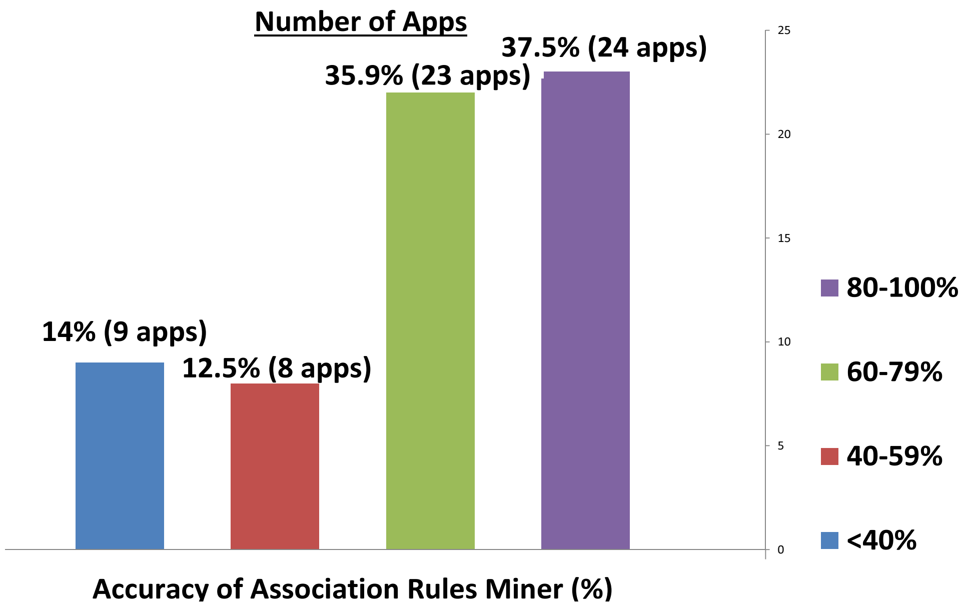


Figure 2.13: Distribution of Leak-Cause Accuracies

which widgets might leak infrequently (e.g. only once before stopping), and this contributes to false negatives. Also, it is also known that TaintDroid tool may generate false positives due to the usage of tag aggregation at various points in the system to reduce storage costs. All these reasons contribute to mining inaccuracies.

2.4.2 Feasibility in Creation of a 'Leak-Cause' Database

I built a knowledge-base of app leak causes from Mobile Forensics back-end testing framework ('Leak-Cause' Database). Building such a knowledge-base can help to enhance the usability and trustworthiness of app stores. Notifications can be performed by a customized 3rd-party app that I created, that is linked up to this 'Leak-Cause' Database. The database contains relevant information pertaining to the leak behaviours of the apps: Types of private data leaks, leak-widgets identified by the leak-cause miner, characteristics of leaks from taxonomy (User-triggered, Start-Up, Periodic), App name, screen-captures, legitimacy recommendations etc. Table 2.1 shows sample database contents for 4 apps: 'Dictionary', 'HungryGoWhere', 'GMail' and 'MessengerWithYou'.

No.	App Name	Leaks Found	Association Rules(Sample)	Leak Behaviour	Recommended?
1	Dictionary.com	1) Location 2) Net-based Location	1)(ImageView #1) → (Location)+(Net-based Location) 2)(ImageView #2) → (Location)+(Net-based Location) 3)(TextView #1) → (Net-Based)+(Location) 4) (Button #1) + (ImageView #3) → (Location)+(Net-based Location)	-Start-up -User-Triggered	No
2	HungryGoWhere	1) ICCID (SIM Card Identifier) 2) IMEI	1)(TextView #1) → (ICCID)+(IMEI) 2)(LinearLayout #1) → (ICCID)+(IMEI) 3)(LinearLayout #2) → (ICCID)+(IMEI) 4) (RelativeLayout #1) → (ICCID)+(IMEI)	-Start-up -User-Triggered	No
3	Gmail	1) Address Book (Phone Contacts) 2) Phone No. 3) GPS Location 4) Net-based Location	1)(LinearLayout #1) → (Address Book)+(Phone Book) +(GPS Location)+(Net-based Location) 2)(Button #1) → (Address Book)+(Phone Book) +(GPS Location)+(Net-based Location)	-User-Triggered	Yes
4	Messenger WithYou	1) IMEI	1)(Button #1) → (IMEI) 2)(Button #2) → (IMEI) 3)(Button #3) → (IMEI) 4)(Button #4) → (IMEI)	-User-Triggered	Yes

Table 2.1: Example of Content Fields in Leak-Cause Database

As an example from Table 2.1, the rule 1) for 'Dictionary.com' indicates that touching 'ImageView #1' would leak users' locations. The leaky widgets can be uniquely identified by 8-digit java has code identifiers and (x,y) layout coordinates, with their corresponding leak type associations. A total of 647 leak causes were mined from the 64 apps with user-triggered leaks. I thus demonstrated that a leak-cause database can be created for privacy notifications to the mobile user. Information from this database can be utilized by an Interactive Privacy Leak Reporter, which I describe in Section 2.3.

2.5 Scaled-Up Testing Experiment with Parallelized App Testing System

I ran the parallelized Automated Testing System described in Section 2.2.6 on a set of 1,700 Android apps downloaded from the Application Crawler. The apps were selected to provide an equal representation among popular and unpopular applications, and consisted of 840 applications from the 26 most popular categories on Google Play as well as a set of 890 randomly selected applications which are not in any of the popular categories.

I utilized 10x Amazon EC-2 cloud machines to run Android Emulator Testing

Total # of Apps	1,700
# of Apps with Privacy Leaks	462
Average Running Time per Application	17.9 minutes
Average # of Unique Pages Transited per Application	6.3

Table 2.2: Run-Time Statistics in Automated Testing Experiment

instances, as well as 3 actual physical Android phones (2 x Galaxy Nexus; 1 x Nexus 4). While it was found that up to 3 emulator test instances could be run on a quad-core CPU machine with 4GB RAM, it would be more cost effective to run a single emulator test instance on a dual-core CPU machine with 2GB RAM in the EC-2 cloud. I set a maximum memory allocation of 768 MB for each emulator, and an upper-bound running time of 60 minutes for each application.

Some statistics of the experiment is as shown in Table 2.2. Of the 1,700 applications tested, we found that 462 applications (27.2%) leak user-data to the external network. The average running time of each application was 17.9 minutes (upper bounded by 60 mins), and it was found that the model checker traversed through an average of 6.3 unique pages per application. In this case, a unique page is defined to be a page that has a significantly different layout (greater than 3 layout widgets/views) than the rest of the pages in the application. In total, 507 machine hours were utilized for the testing of all applications.

In the run-time testing experiment performed in [39], the exploration times required for the apps had a large variation from 10 minutes to over 10 hours. As the aim is to simply to illustrate an automated approach for user-centric leaks, the design choice was to upper bound the run-time to 60 mins/app. However, it is possible for all testing to proceed to full coverage if a larger number of machines are utilized.

Coverage Measurements of App Testing: I performed activity coverage measurements for a number of 187 randomly selected apps. After decompiling, the AndroidManifest and XML layout resources of these decompiled apps were analyzed to obtain a measure of the number of unique layouts and pages required for model checking. This was then compared to the number of actual unique pages accessed by the Automated Testing System. The results are displayed in Figure 2.14. 83 of the 187 applications (44.4%) had a coverage of more than 80%, 39 applications (20.9%) had a coverage of between 40% to 80%, while 65 applications (34.8%) had a coverage of less than 39%.

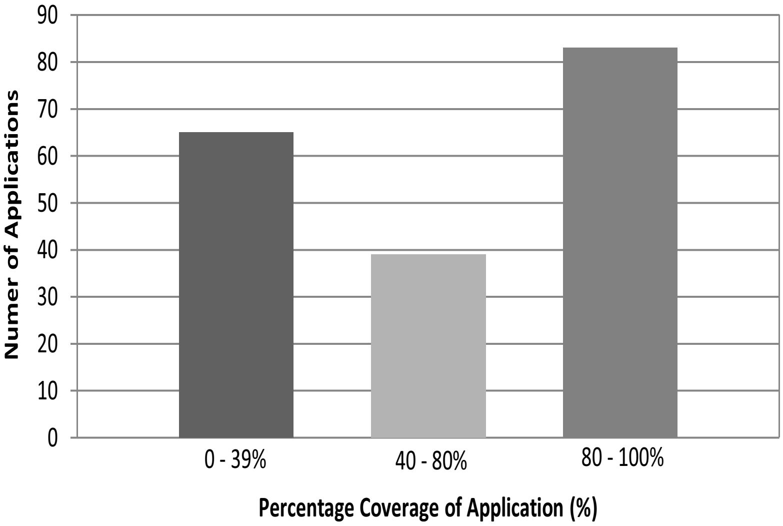


Figure 2.14: Distributions in Application Activity Coverage (Percentage)

Processing of Leak-Causes: I performed processing and discovery of leak-causes and characteristics for the apps. The apps were classified into 3 forms of leak characteristics: Applications that leak a users' privacy data on (i) Start-up, (ii) In a Periodic fashion and (iii) On User-Triggers. As before, frequent item-set mining was utilized. For every occurrence of a leak event reported by TaintDroid in the log files of each application, a look-back window of 5 seconds to co-locate user-activities with these leak events. Similarly for Start-up leak behaviours, instances of application launches were co-located to leak events within a 5 second window as well. For Periodic leakage characteristics, consistent occurrences of at least 1 leak event

# of User-Triggered Leak Rules Found	**# of Applications**
1-3	11
4-6	72
>**6**	5

Table 2.3: Distribution of Numbers of User-Triggered Leak Rules Found

at every minute time intervals were identified. The data leak was deemed to be periodic if at least 75% of the minute time intervals had a leak report. WEKA [30] was utilized with absolute support values of value 2. Targeted re-testing was performed on leaking apps to validate the leak characteristics.

Table 2.3 presents a distribution of the number of user-triggered leak rules for 88 applications found to leak user-data on pressing in-app widgets. Of these applications, 11 were found to have 1 to 3 triggers, a majority of 72 applications had 4 to 6 triggers and 5 applications had more than 6 user triggers that cause privacy leaks. These results illustrate that most user-triggered leaky applications typically have between 4 to 6 widgets that would cause a privacy leak when clicked.

Distribution & Characteristics of Leaks Found: Figures 2.15 displays the proportions of applications displaying each of the 3 different leak characteristics and the types of user leaks found respectively in the 462 applications. 148 applications (32.0%) were found to have start-up leaks, 378 applications (81.8%) were found to have periodic leaks, and 88 applications (19.0%) of the applications had leaks due to user triggers.

Figure 2.16 highlights the types and numbers of leaks found in the applications. The leaks were found to disclose users' physical locations, address books, protected storage (SMSes, Photo & Media Files etc.), phone identity (IMEI) as well as the SIM card identifier (ICCID). A large number of 337 applications were found

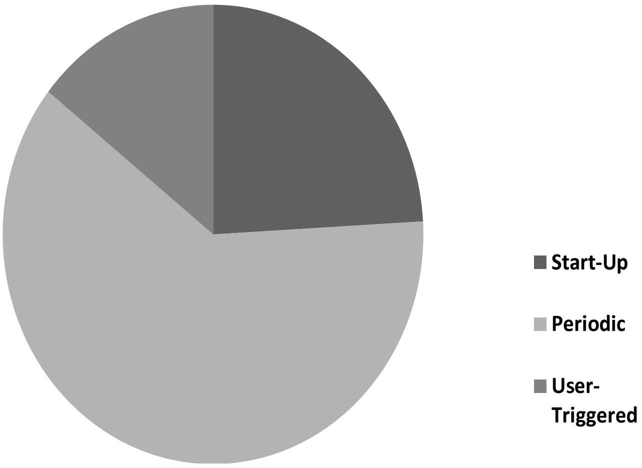


Figure 2.15: Proportions of Leak-Characteristics

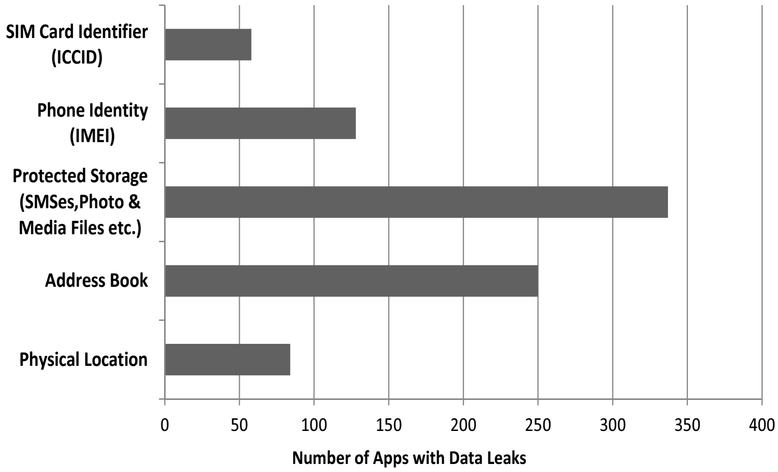


Figure 2.16: Types of Privacy Leaks Found

to disclose user protected storage, while a surprisingly large number of 250 applications disclosed the address book to an external server.

Chapter 3

Evaluation Results of MAMBA System

In this chapter, I compare my automated testing solution, the MAMBA system, to another testing solution, the Automated Model Checker (AMC) [39]. The comparison is performed over 500 Android apps, with the comparisons metrics being the (i) Testing Times Required (Latency), (ii) Activity Coverage, as well as the (iii) Precision of my MAMBA testing in uncovering runtime privacy accesses/leaks.

3.1 The Automated Model Checker (AMC)

I provide some preliminaries on the Automated Model-Checker (AMC), which is an automated app testing solution for Android apps from MobiSys 2013 [39]. In AMC, no target activities were specified, but available user-actions for each clickable widget/button on each activity encountered were ordered sequentially in a top-to-bottom fashion based on the layout and each tried in-turn. On reaching a next activity state, the app was backtracked to the previous activity before selection of another clickable widget/button for tapping. In this fashion, AMC tries to reach all activities in the app. AMC is based on state equivalence heuristics, and the comparisons of the app states that automated testing has reached after performing user actions on app widgets.

After the inputs of user actions, activity transitions might not necessary occur. And many user actions might not necessarily cause transitions but merely changes in the activity's state (For example clicking a view causes some additional information to be displayed).

While it isn't difficult for a human to deduce whether an activity transition has occurred instead of merely view changes, the problem is not trivial for a machine. AMC solves this problem by calculating structural hashes of the current activity and comparing it to that of a previous state. An activity transition is deemed to have occurred if the structural hashes calculated are different. AMC also holds in memory a collection of states that testing has already reached, and builds an transitional representation of all states visited at runtime. In addition, it uses equivalence heuristics to determine if multiple transitions from a particular activity closely resemble each other. If it does, it deduces that it is only sufficient to test the first few buttons in the page. (For example, it is sufficient to test only the first few news items in an news list of an activity)

AMC does not hold any prior knowledge of activity transitions of apps, but builds up this transitional knowledge during runtime testing. This is in contrast to MAMBA, which obtains activity transitional information of apps from static analysis prior to runtime testing. While this speeds up runtime testing, a downside of MAMBA is that additional processing time is required for static analysis. However, as the program sizes of mobile platforms are designed to be compact, the times required for static analysis will usually not be exceedingly large.

3.2 Small-Scale Evaluation of MAMBA using GATOR's Callback Analysis

I present the results of a small-scale evaluation of 24 apps using MAMBA with test cases generated by GATOR's [53] Callback Analysis (MAMBA-GATOR). Testing time required as well as activity coverage of MAMBA was evaluated against the Automated Model Checker (AMC) [39]. The test cases generated were entirely fully-directed with no semi-directed ones. As mentioned in previous sections, GATOR's Callback Analysis was found to have a low success rate. The test apps were thus selected based on apps which could be successfully processed.

3.2.1 Results of AMC Comparison with MAMBA-GATOR

The total time required for each of the 24 apps for both AMC and MAMBA-GATOR are displayed in Table 3.1. AMC results are displayed in column 'AMC (Testing Time)', and consists of time for runtime testing in '[hh:mm:ss]' format. The total testing time required for MAMBA-GATOR is a combination of the time required for static test case generation as well as the automated running of the test cases on the automated tester.

The total testing times required for AMC Model Checking ranged from over 2 minutes (0:02:32) to more than 3 hours (3:23:10). This is comparison to total testing time required for MAMBA-GATOR's directed testing ('MAMBA-GATOR - Total [hh:mm:ss]'), which ranged from 2 minutes (0:02:04) to just over 15 minutes (0:15:40). These testing total times required for MAMBA-GATOR were therefore significantly shorter than that required for AMC. MAMBA-GATOR was 6.1 times faster across all the 24 apps (Average 0:31:15 compared to 0:05:03), and was up to 58 times faster (No. 21 app: 'Speak Mandarin Free').

Sensitive APIs & Target Activities of Interest: From Table 3.1, the Privacy Sensi-

						AMC (Testing Time)	MAMBA - GATOR (Testing Time)		
No.	App Name	Package	Privacy Data	Total No. of Activities	No. of Target Activities	Total [hh:mm:ss]	Test Case Generation [s]	Test Case Running [s]	Total [hh:mm:ss]
1	Album Cover Finder	com.ftpcafe.coverart.trial	GPS	8	6	1:18:25	59	337	0:06:36
2	A+ Certification Lite	com.mhazzm.APlusCertification702Lite	GPS	4	1	0:03:10	45	101	0:02:26
3	Advanced Task Killer	com.rechild.advancedtaskkiller	GPS	6	1	1:04:11	47	140	0:03:07
4	African American Quotes	com.hmobile.africanamericanquote	GPS	11	11	1:18:18	37	610	0:10:47
5	Android Book App	com.appmk.book.main	GPS	5	3	0:19:41	56	146	0:03:22
6	Arsenal Fan Club	com.arsenalfanclub	Nil	9	0	0:46:55	57	N.A.	Nil
7	Blood Alcohol Tracker	com.promille	GPS	7	7	0:27:00	32	411	0:07:23
8	Call and SMS Easy Blocker	com.ekaisar.android.eb	GPS, Contact-List ,SMSes	9	2	0:17:17	94	168	0:04:22
9	Car Performance Free	com.unnull.apps.carperformancefree	GPS	7	2	0:12:51	26	224	0:04:10
10	CLT vs. PJ	com.neocode.cltxpj	GPS	3	3	0:03:35	30	185	0:03:35
11	Driving Skill Monitor	com.drismo	GPS	14	3	0:34:09	50	216	0:04:26
12	Fat Burning Foods	com.v1_4.fatburningfoods.com	GPS, IMEI	15	15	0:11:27	42	535	0:09:37
13	Font for Galaxy	com.hongik.fontomizerSP	GPS	2	2	0:03:09	38	114	0:02:32
14	Interesting Cooler	com.interestingcoolerfreegoimbh.app	GPS	10	1	0:17:56	55	69	0:02:04
15	Interpret Your Dream	com.dreamforth.iyd	GPS	6	6	0:06:01	34	296	0:05:30
16	JaquecApp	com.terranology.jaquecapp	GPS	18	1	0:37:35	49	109	0:02:38
17	My Display Check	com.jensu.screenchecker	GPS	7	7	0:15:40	30	357	0:06:27
18	Shikoku Railway	com.appspot.noritsubushi.shikoku	GPS	6	6	0:20:49	38	422	0:07:40
19	Solar Battery Charger	com.solar.charger.battery	GPS	3	2	0:07:12	39	123	0:04:42
20	Space War APK	com.space_war_free_10	GPS, IMEI Phone No.	13	10	0:19:26	40	900	0:15:40
21	Speak Mandarin Free	com.chineseskill.lan_tool.sc	Microphone, Device IDs	3	2	3:23:10	61	148	0:03:29
22	Super Runner Boy	com.pack.Super-RunnerBoyTrial	GPS	4	1	0:02:32	68	80	0:02:28
23	TooLate Lateness	afw.allforweb.toolate	Contacts	6	1	0:08:34	42	95	0:02:17
24	Trios Labs Reader	com.triosLabs.hadithreader	GPS	8	8	0:26:47	47	313	0:06:00

Table 3.1: Results of Automated Testing - (Model-Checking) AMC vs. Graph-Directed Tester (MAMBA)

tive API association of Section 2.2.1 uncovered that 21 apps accessed users' GPS data, 2 of the apps accessed the Contact List, 3 apps Device Identifiers, and 1 app each accessed users' Microphone and SMSes. 1 app (Arsenal Fan Club) did not access any privacy data. The total no. of activities each app contained as well as the no. of activities found to be accessing user privacy data are illustrated in the table.

The apps were found to each contain between 2 to 18 activities in total. 10 of the 24 apps had 100% of their activities accessing privacy data, with varying numbers (0-100%) for the rest. These privacy accessing activities were flagged out to be reached by the directed tester.

Generating Test Cases for Directed Testing: The processing times required for generating test cases for target activities of each app are tabulated in Table 3.1 (seconds). Test Case Generation time includes bytecode and CCFG generation, CCFG traversal and search for paths as well, Privacy Sensitive API association as the generation of user-action sequences for the automated tester. Test cases were generated on an AMD Opteron 4386, 3.1GHz CPU and running 64-bit Debian Linux.

Test Case Generation took between 26 to 94 seconds for each app (Under column 'MAMBA-GATOR : Test Case Generation (s)'). The results indicated that generation time had no clear relation with the total number of activities in the app or the number of target activities, although it may be dependent on app size and complexity. An example was the No. 12 app: 'Fat Burning Foods', which had 15 activities in total and 15 target activities, but required 42 seconds for processing. This is in comparison to No. 2 app: 'A+ Certification Lite' that required a longer generation time of 45 seconds but had smaller number of only 4 activities and 1 target activity.

Running Test Cases: The testing times required for running the resulting test cases on an automated tester are tabulated under the column 'MAMBA-GATOR: Test Case Running (s)'. Automated testing performed on 3 hardware Galaxy Nexus devices running Android OS 4.1.1 which were connected to a testing program running on a Windows 7 machine. Testing can be carried out on emulators as well, but we only utilized hardware devices for consistency in this evaluation.

Running a single test case took about over a minute on average. Automated app testing took between 69 seconds (1 minute 9 seconds) to 900 seconds (15 minutes) for each app, and were dependent on the number of target activities as well as the number of test cases.

Discussion on Findings

Why is my Solution Faster? MAMBA-GATOR's Graph-Aided Directed Testing was faster due to the immediate and expedited transitions to target activities via app buttons that were already known beforehand from test cases built. This was in contrast to AMC, which had to spend time exploring all clickable UI components in each app activity, many of which might cause state changes but not activity transitions. Examples of such occurrences are Apps #1, #4 and #21, which required over 1-3+ hours respectively.

The branching structure of activity transitions also played a part to increase test times in AMC. From inspection, apps with more highly cascaded activity transitions required more time in AMC Testing, due to the time required for back-tracking to previous activities, which graph-directed testing did not require.

It can be observed that there were situations where MAMBA-GATOR testing was only slightly faster than AMC. (For example for App No. 15: 'Interpret Your Dream' and App No. 22: 'Super Runner Boy'.) From manual inspection of the apps, these arose in circumstances where activity transitions were only branched around the root (main) activity, such that the automated tester only needed to explore all buttons on the root activity page to transit to other activities within the app.

It is highlighted that the GATOR's CCFG analysis generated only contains explicit activity starting ('startActivity') intents and do not include implicit intents. Explicit intents consists of activity type classes that have been defined within developer code of the app, while implicit intents consists of transitions to external window states (e.g. Transit from app to a web browser or email client).

Coverage of Target Activities

Figure 3.1 displays the Coverage of Target Activities (%) over the Total Testing Time (seconds) for 3 example apps (App #1, #4 and #20). Although AMC was also able to cover target activities, it required significantly higher testing times compared to MAMBA-GATOR. It can be seen that the Coverage of Target Activities in graph-directed testing converges faster compared to AMC. Coverage of activities in directed testing proceeded at a steady pace until completion, compared to the prolonged times required and stagnation at certain activity points for AMC.

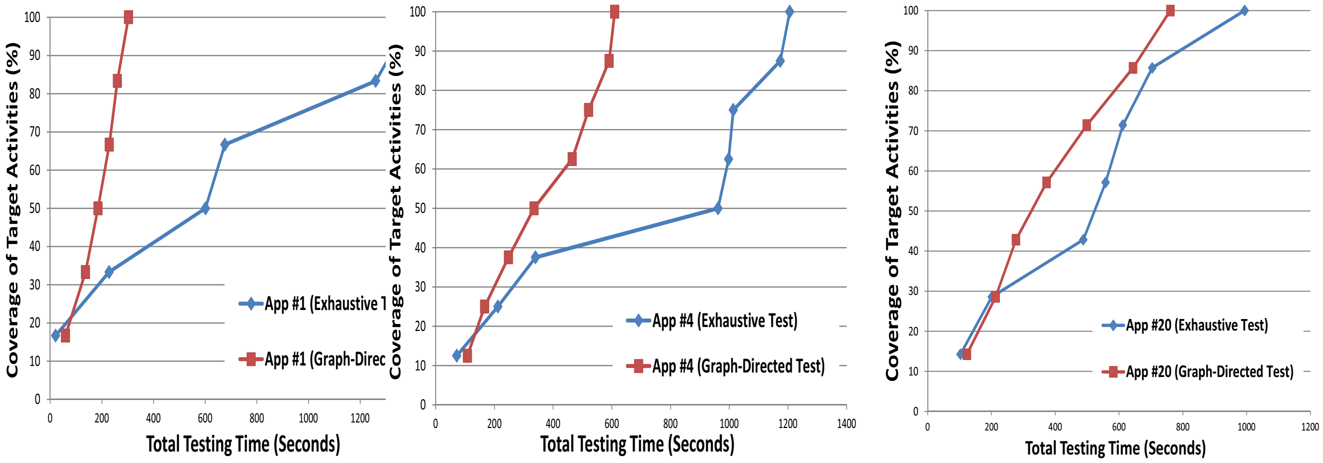


Figure 3.1: Coverage of Target Activities (%) over Total Testing Time (Seconds) for 3 Example Apps - (Left- App #1: ”Album Cover Finder”; Middle- App #4: ”African American Quotes” ; Right- App #20: ”Space War APK”)

Both my graph-directed testing solution and AMC were each able to cover 86 of the 101 target activities in Table 3.1 (85.1% coverage). From inspection, the remaining activities were inaccessible by our testers due to a few reasons. The first was that there were some defunct activities which were defined by developers and in app code, but did not contain start intents. There were also a few which required some hard to fulfil conditions that normal user events could not meet. For example, App #11: Driving Skill Monitor had an activity that required input from GPS sensors for a certain distance before it could be accessed. App #20: Space War APK had 3 activities that required completion of game levels before it could be accessed. Allowing coverage for such activities falls outside the scope of my work.

Verification of Privacy Usage Behaviour in Apps

Leak reports from 2 privacy detectors, PMP and TaintDroid were logged during the automated test runs. PMP detected run-time privacy data accesses, while TaintDroid was able to detect privacy leaks over the network. Based on the logs captured from graph-directed testing, I verified that 45 of the 101 target activities (44.6%, over 13 apps) had privacy data accesses, and 27 of these activities (26.7%, over 8 apps) leaked private data to the network. Such privacy behaviours can be recorded in offline databases for future notification to users. The logs from AMC also indicated the same findings, which showed that my Directed-Testing solution was able to perform verification of privacy usage behaviour in apps just as well as exhaustively testing the app, but in a significantly reduced time.

3.3 Large-Scale Evaluation & Comparison of MAMBA with Automated Model Checker (AMC)

In this section, I present the results of a larger scale evaluation & comparison of my MAMBA system with the Automated Model-Checker (AMC). This was conducted on a set of 500 apps crawled from the Google Playstore. In contrast to the smaller-scale evaluation described in previous Section 3.2, which had a low successful analysis rate, the test cases were generated entirely by MAMBA's Callback Analysis without using GATOR. My evaluation metrics were testing times required for test completion (latency), coverage/recall of target activities that had privacy infringing APIs, as well as precision in uncovering activities with privacy accesses/leaks.

3.3.1 Conduct of the Experiment

In this section, I describe the test conditions under which the experiments were carried out. 500 test apps were analyzed each by MAMBA as well as by AMC. The parallelized app testing system described in Section 2.2.6 was utilized to scale up testing of both testers, with phone images running both PMP and TaintDroid to detect data accesses and leaks. APK binaries were installed on test devices and tested sequentially, with test logs being saved and processed at the end of testing.

For the MAMBA system, testing consists of static analysis of apks to uncover paths and target activities, generation of test cases followed by apk installation and automated testing on physical devices. For AMC testing, testing consists of only apk installation and automated testing steps. A maximum time limit of 3 hours/app was set on AMC. The results of testing of MAMBA and AMC are then compiled and compared side-by-side.

Test Environment and Selection of Test Apps

Test cases were generated on an AMD Opteron 4386, 3.1GHz CPU running 64-bit Debian Linux. The test cases were then transferred to parallelized app testing instances running on a Quad-core i7, 2.2GHz laptop CPU running Windows 7, with multiple Android test devices tethered with a multi-port USB hub. The Android devices utilized were Galaxy Nexus phones running Android 4.1.2.

The static analysis, consisting of privacy API association to identify target activities as well as uncovering testing paths towards these activities, was performed on a set of 1,000 apps. These apps were crawled from the Google Playstore, with an even distribution of apps from the 26 app categories. From the 1,000 apps, it was found that just over 400 apps (40%) invoked at least one privacy sensitive API. 500 test apps were then randomly selected from this set of 1,000 app to provide a roughly equal distribution of apps with privacy APIs (230 apps from 400 privacy-related apps; 270 from remaining 600 apps).

The MAMBA, AMC and parallelized test coordination and logging programs were packaged and ran as Python and Java binaries (.jar) on separate threads on the i7 laptop CPU. The logs were saved and processed into designated folders under the name of individual apk files.

46 out of the 230 apps with privacy-related APIs required user login. Accounts were created for these apps manually, and the automated testing was performed on these apps in their logged-in state.

3.3.2 Results of Comparison of MAMBA with AMC

After running MAMBA and AMC on the 500 test apps, the testing times required and coverage of target activities are then compared. The results demonstrated that MAMBA has advantages in significantly reduced overall testing times of apps for reaching the specified targeted activities, and verifying runtime privacy accesses/leaks (13.18 minutes/app compared to 51.86 minutes/app). Only a small trade-off in a slightly reduced coverage of target activities (70.33% by AMC to 63.96% by MAMBA) was required.

The precision of MAMBA's static analysis in uncovering runtime app privacy behaviours was 79.84% for data accesses and 35.66% for data leaks. MAMBA also had recall values of 68.90% for data accesses and 56.10% for data leaks, as measured relative to privacy behaviours uncovered by AMC. These results are explained in the next sections.

Testing Time Required for App Testing (Latency)

We compare the testing times required for both MAMBA and AMC for the 500 apps. MAMBA's required testing time was 13.18 minutes/app compared to 51.86 minutes/app for AMC across 500 apps. We take a closer look and differentiate the testing times required for 2 distinct groups of apps: (i) Apps with at least one privacy-related API (#1-#230), and (ii) Apps with No privacy-related APIs (#231-#500). Figures 3.2 and 3.3 displays the distribution of testing times required for (i) and (ii) respectively. The testing times for AMC are in blue, while MAMBA's plot is in red.

(i) Apps with at least one Privacy-Related API: Figure 3.2 displays the distribution of total app testing times required for each of the apps with at least one privacy-related API (Apps #1-#230). AMC has a mean testing time of 57.11 minutes/app (Std. Deviation = 58.93) - (Min.: 0.67 minutes; Max.: 208.3 minutes) compared

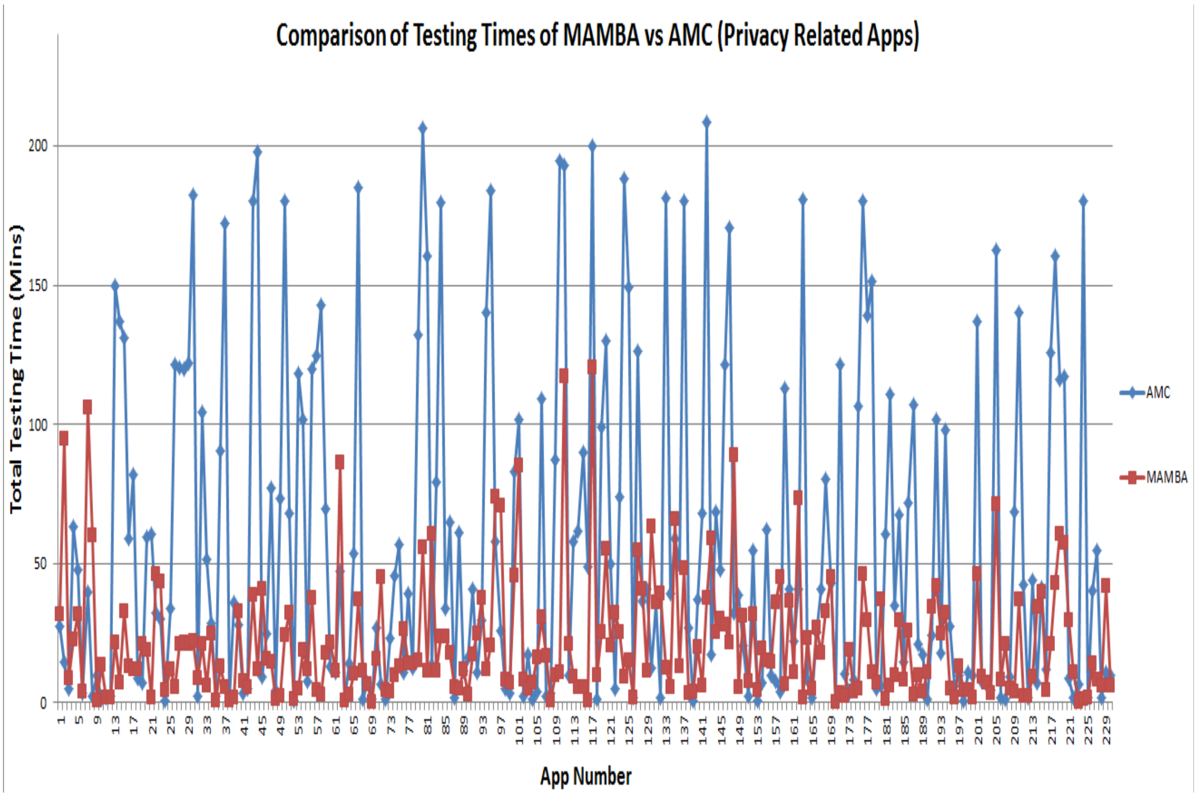


Figure 3.2: Distribution Plot of Total Testing Time (Mins) against App Number (#1-#230) for (i) Apps with at least one Privacy-Related API: MAMBA (Red) - (Mean=21.52 mins/app, Std. Dev.=21.91) ; AMC (Blue) - (Mean=57.11 mins/app, Std. Dev.=58.93)

to MAMBA's mean testing time of 21.52 minutes/app (Std. Deviation = 21.91) - (Min.: 0.25 minutes; Max.: 120.68 minutes). From the graph's distribution, it can be observed that the testing times required for MAMBA is dominantly lower for the vast majority of apps compared to AMC.

It can be observed from Figure 3.2 that there were a few apps for AMC with total testing time larger than the 3 hour time limit set. This was due to the actual runtime being longer than the programmed time limit, due to the presence of other processes (e.g. writing of test logs, installation of app etc.) that were also involved in the testing.

(ii) Apps with No Privacy-Related API: Figure 3.3 displays the distribution of total app testing times required for each of the apps with no privacy-related APIs (Apps

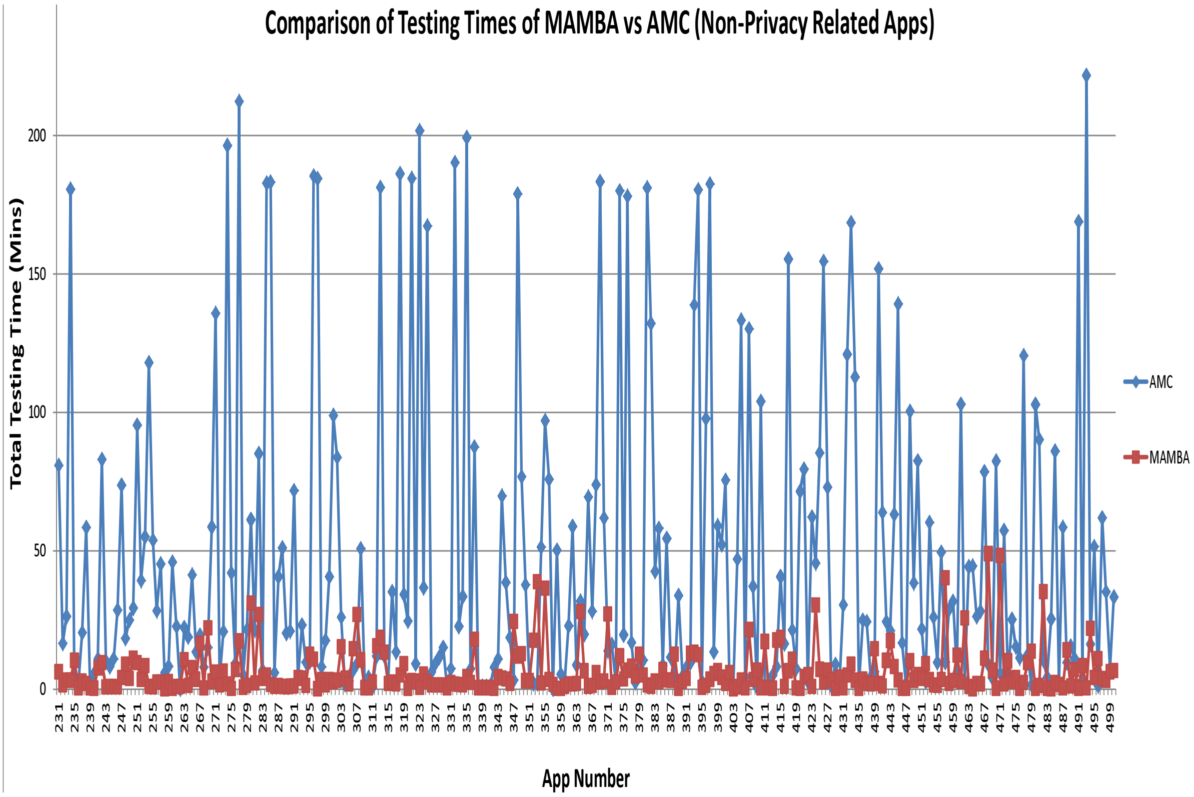


Figure 3.3: Distribution Plot of Total Testing Time (Mins) against App Number (#231-#500) for (ii) Apps with No Privacy-Related API: MAMBA (Red) - (Mean=6.08 mins/app, Std. Dev.=8.16) ; AMC (Blue) - (Mean=47.40 mins/app, Std. Dev.=55.35)

#231-#500). AMC has a mean testing time of 47.40 minutes/app (Std. Deviation = 55.35) - (Min.: 0.02 minutes; Max.: 221.73 minutes) compared to MAMBA's mean testing time of 6.08 minutes/app (Std. Deviation = 8.16) - (Min.: 0.08 minutes; Max. 48.95 minutes). From the graph's distribution, the testing time required for MAMBA is clearly much lower across all apps. This is because for MAMBA, apps with no privacy-related APIs do not require runtime testing and time is only required to be spent on static analysis.

Coverage of Target Activities

We measure the extent by which MAMBA and AMC is able to reach the specified activities that have been uncovered by MAMBA's privacy API association analysis of Section 2.2.1. Coverage is measured based on the ability of both testers to reach

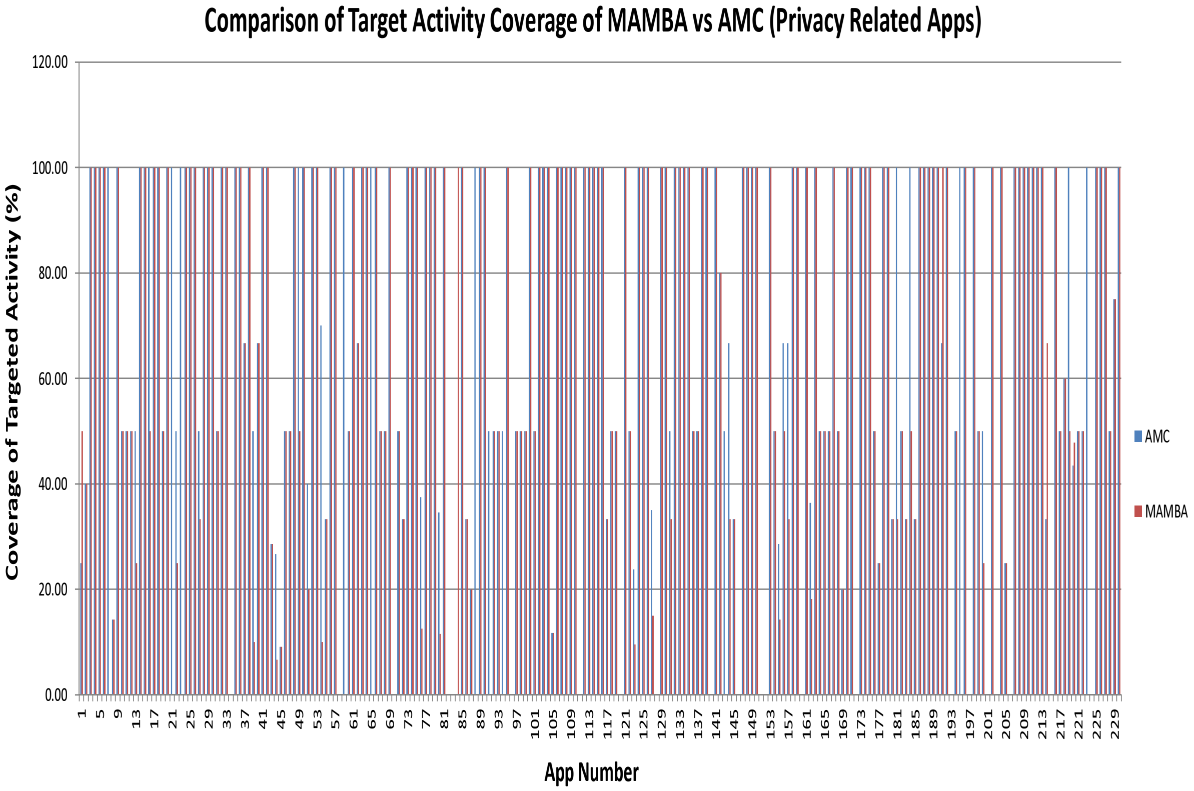


Figure 3.4: Distribution Plot of Coverage of Target Activities against App Number Coverage for AMC (Blue) - (Mean=70.33%) and MAMBA (Red) - (Mean=63.95%)

the activity at runtime. AMC was able to cover on average, 70.33% of target activities compared to 63.95% of MAMBA. This means that MAMBA only had a slight decrease in the coverage as compared to AMC. Figure 3.4 illustrates the coverage distribution of AMC (Blue) and MAMBA (Red). It can be observed that there is no large discernable difference across the apps.

For the target activities that could be reached by AMC but not MAMBA, this was due to imperfections in some of the test cases generated. There were some transitional or widget identifier information that MAMBA's static analysis was unable to output. Also, an observation is that certain apps had target activities that could not be reached by both AMC and MAMBA. A reason is that there were some defunct or dead activity-code that were uncovered during the static analysis, but were actually unreachable at runtime. There were also some activities that required some complex sequences of user inputs or situations. (For example, some activities required user-location to move a certain distance to be displayed, or the input of special keystrokes) Reaching such activities fall outside the scope or capabilities of MAMBA or AMC.

There were also a small number of target activities that MAMBA could reach but not AMC. This illustrates that in some cases, MAMBA's concept of pre-computing paths prior to runtime testing can have advantages in better reachability.

Overall Coverage: AMC had an overall activity coverage of 36.99% compared to 21.97% for MAMBA. For AMC, this could be attributed to the time bounding of AMC to 3 hours and the presence of dead activity code. Whereas for MAMBA, the low overall coverage was due to the condition that only test cases towards target activities were ran. However, these values also illustrate that MAMBA only needs to cover a smaller proportion of overall activities within the apps compared to AMC to be able to reach and test the required targeted activities.

Accuracy in Detection (Precision & Recall) of Privacy-Infringing Android Activities

We measure MAMBA's accuracy in uncovering privacy accessing and privacy leaking Android activities. As mentioned, the PMP and TaintDroid runtime detectors were instrumented for detection. By definition, privacy behaviours detected at runtime are assumed to be the ground truth. Within the test apps, MAMBA was able to output 206 activities that had privacy data accesses, and 92 activities with data privacy leaks with complete accuracy.

We next measure the accuracy (precision/recall) of MAMBA's static analysis. We define the precision to be the number of correctly detected privacy accessing/leaking activities over the number of accessing/leaking activities that MAMBA outputted. Recall for MAMBA is defined to be the number of correctly detected accessing/leaking activities by MAMBA over all accessing/leaking activities found by AMC. The total number of privacy accessing/leaking activities detected during AMC's runtime testing was used as an estimate for all true privacy-related activities.

This estimate is reasonable because AMC does not have a set target for exploration, but attempted to reach all activities within the app.

MAMBA's precision in outputting privacy accessing activities was 79.84% (206 activities/258 activities), and 35.66% (92 activities/258 activities) for privacy leaking activities. MAMBA's recall in outputting privacy accessing activities was 68.9% (206 activities/299 activities), and 56.10% (92 activities/164 activities) for privacy leaking activities.

The 270 apps with no privacy-related behaviours were tested by AMC to detect any privacy behaviours. It was found that MAMBA was very accurate in detecting apps that had no privacy-related behaviours. Across the 270 apps that MAMBA flagged to be non privacy invasive, there were only 1 app with privacy access reports and 2 apps with privacy leak reports.

3.3.3 Discussion

Advantages and Trade-Offs of MAMBA

The testing time and coverage results demonstrate that MAMBA has a significant advantage in a lower testing time requirement per app (latency) for testing of activities. Across all 500 apps, MAMBA was almost *4 times faster* (13.18 mins/app compared to 51.86 mins/app) in detecting and verifying privacy-related activities. For the 230 apps with privacy behaviours, MAMBA was *2.7 times faster* (57.11 mins/app compared to 21.52 mins/app). And for the 270 apps with no privacy behaviours, MAMBA had the largest advantage in being almost *8 times faster* (47.4 mins/app compared to 6.08 mins/app). These testing time advantages are weighted against a marginal 6.4% decrease in coverage by MAMBA compared to AMC (63.95% to 70.33%). MAMBA is thus a very time efficient system for detection of activities that have privacy implications compared to the AMC automated testing method.

While the test cases were generated from a server, the automated testing times could be achieved by a laptop CPU. This demonstrates that MAMBA has good scalability.

MAMBA's static analysis component had a high precision of 79.84% and recall of 68.9% in outputting correct privacy accesses, but had only a moderate precision of 35.66% and recall of 56.10% in outputting privacy leaks. The lower accuracy of privacy leak detection is to be expected however, as MAMBA does not analyze any source-sink networked API calls. These accuracy results indicate that MAMBA's static analysis outputs (especially privacy accesses) might be directly applied to the the Interactive Privacy Notifications App of Section 2.3, to save on automated app testing time. There is, of course, a trade-off by a roughly 20% decrease in accuracy.

Chapter 4

Understanding Utility of User-Triggered Privacy Leak Messages

In this section, I investigate and validate the utility of presenting information on user-triggered privacy leaks uncovered in Chapter 2 to users. This is performed using lab and field studies, and utilizes the Privacy Notification Mechanism detailed in Section 2.3.

I chose to utilize a *'Non-Blocking'* mechanism of privacy notifications, in which messages appear to users in the the background during various points of app usage [54]. Such messages indicate to users the privacy cost or implications of app usage or on performing certain in-app actions, but without providing users direct control over allowing or blocking the data usage. This is in contrast to *'Blocking'* mechanisms, in which messages are accompanied with a GUI 'freeze', where the user is required to make a selection on the data access before app functionality can be restored.

Why did I choose to validate using a 'Non-Enforcing' mechanism?: While Enforcing mechanisms have been utilized and studied in various privacy systems and user-studies contexts [57, 14, 3, 5, 27] and have been shown to be effective, the Non-Enforcing mechanism has been less studied for mobile app privacy. The Non-Enforcing mechanism also fits my proposed framework better, due to the characteristic of more fine-grained notifications having to be passed to the user during app usage (e.g. Widget overlays and appearance of notifications on certain app layouts etc.).

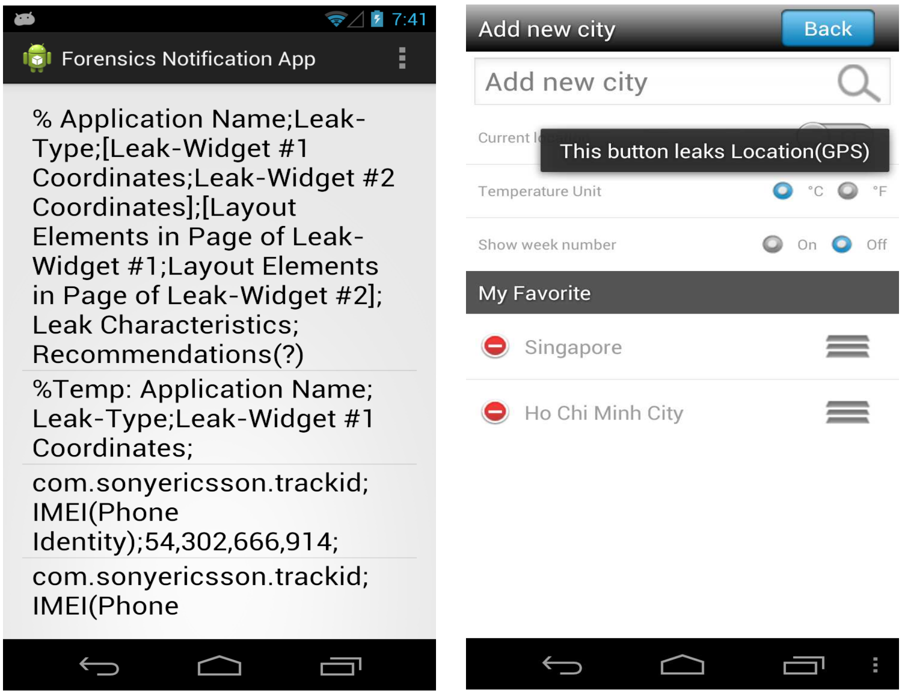
4.1 Small-Scale Lab-Study

4.1.1 Conduct of Lab-Study

I conducted a pre-deployment study of the privacy notifications app in the lab with a small set of 12 users. The goal of the lab study was to observe if privacy messages were effective for users in leak avoidance. The users consisted of students and staff (8 males, 4 females) from the School of Information Systems in Singapore Management University. The users had an average age of 29 with 66% of participants being expert Smartphone users.

The users were divided into 2 separate groups of 6 users each: a (i) Test Group (with notifications) and (ii) Control Group (without notifications). Each user was not informed which group they belonged to. Instructions were provided to users that they might be observing display messages during application usage, and that they were to utilize applications in as natural a manner as possible as if in a personal capacity. No further context information was provided to them. The notification app logged all widget click and page transition data of the users, and performed a comparison between the two group. At the end of the study, participants were asked to rate their ease of application usage on a Likert Scale, to measure differences in usability between the 2 groups.

A screen capture of the Notification App as well as an example of a privacy



% Application Name;Leak-Type;[Leak-Widget #1 Coordinates;Leak-Widget #2 Coordinates];[Layout Elements in Page of Leak-Widget #1;Layout Elements in Page of Leak-Widget #2]; Leak Characteristics; Recommendations(?)

%Temp: Application Name; Leak-Type;Leak-Widget #1 Coordinates;

com.sonyericsson.trackid; IMEI(Phone Identity);54,302,666,914;

com.sonyericsson.trackid; IMEI(Phone

This button leaks Location(GPS)

(a) Notification App

(b) Toast Messages

Figure 4.1: (a) Notification App displaying leaky widgets and characteristics obtained from the leak-cause database. (b) Privacy Message displaying the leak of GPS Location on a clickable view (Flashes and disappears after a 3 second period)

Application	Description	Leak Type	Cause
Sony TrackID	Music Recognition App	IMEI	Button on start page
Telemaque Horoscope	Lifestyle Horoscope App	GPS & IMEI	4 widgets at different places
Linpus Weather	Weather App	GPS	Button on main page
Super Backup: SMS & Contacts	Tools Backup App	Address Book & IMEI	4 widgets at different places

Table 4.1: List of applications used in lab-study

message is in Figure 4.1. I selected 4 apps for the study. The app names as well as their leak-types and characteristics and causes are illustrated in Table 4.1 These apps were chosen on the basis that they were relatively obscure apps in which users would likely be unfamiliar with their layouts.

The user study was divide into two experiments, each involving 2 applications:

1. Experiment 1 (No instructions provided): We informed the participants that they would be taking part in a privacy study, but did not tell them about the existence of messages or provided further instructions. The participants were then asked to use the applications for as long as they required, until they are

satisfied that they have utilized or accessed every part of the application.

2. Experiment 2 (More detailed instructions): We informed the participants about the existence of possible notifications, and they should try to make their own decisions as to how they should utilize the application. The users were also then asked to use the applications for as long as they required.

I utilized the Sony TrackID and Linpus Weather apps for Experiment 1, and Telemaque Horoscope and Super Backup SMS apps for Experiment 2.

4.1.2 Results of Small-Scale Lab Study

Results of Experiment 1:

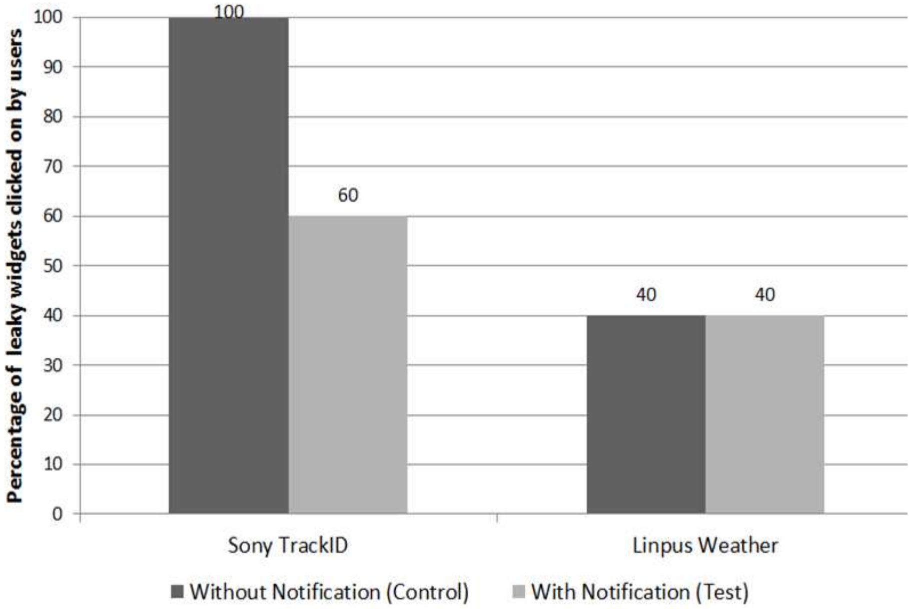


Figure 4.2: Percentage of Leaky Widgets Clicked on by Users in Experiment 1 (No Instructions Provided) for 'Sony TrackID' and 'Linpus Weather' Apps

(Pls refer to Figure 4.2) For the 'Sony TrackID' app, 100% of leaky widgets were clicked on by users in the control group, compared to 60% in the test group. This represented a 40% drop in the number of leaky widgets clicked on by participants when presented with the notification. This was an encouraging result as it shows that the behaviour of the user changes when presented with the notification mechanism.

For the 'Linpus Weather' application, there were no changes observed in behaviours among both test and control groups as participants from both groups clicked on 40% of the leaky widgets. From our observation of the kind of leak that the application was causing (GPS Location), it might be because participants did not mind sharing their GPS location for this particular app. However, other reasons such as users' unfamiliarity and lack of knowledge that there would be notifications might have played a part as well.

Results of Experiment 2:

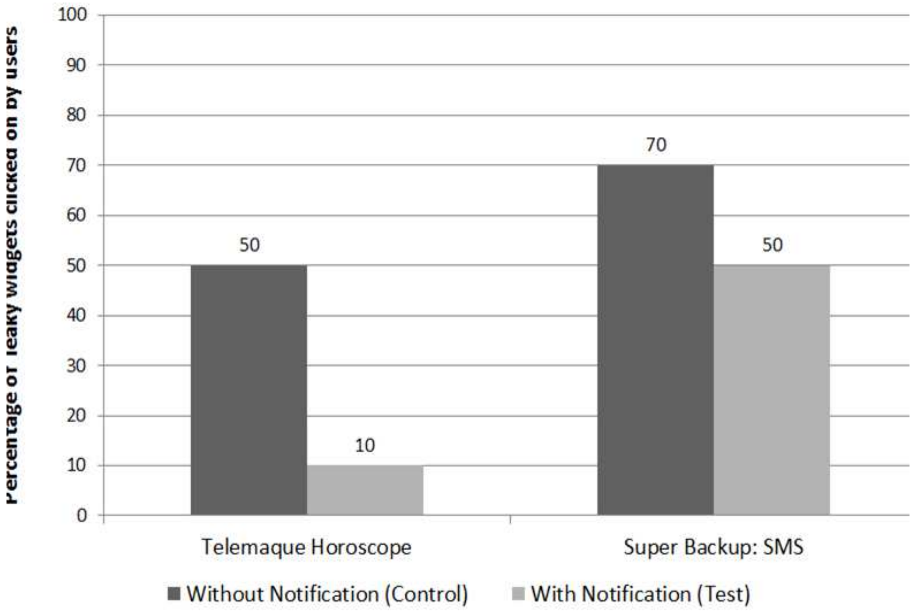


Figure 4.3: Percentage of Leaky Widgets Clicked on by Users in Experiment 2 (More Detailed Instructions) for 'Telemaque Horoscope' and 'Super Backup: SMS' Apps

(Pls refer to Figure 4.3) For the 'Telemaque Horoscope' app, 10% of leaky widgets were clicked on by users in test group, compared to 50% in the control group. Whereas for the 'Super Backup: SMS' app, 50% of leaky widgets were clicked on by users in the test group, compared to 70% in the control group. There were therefore decreases of 80% and 28.6% of number of leaky widgets clicked upon for the

Telemaque Horoscope and Super Backup SMS apps respectively. These results are promising as it shows that users' privacy behaviours are positively affected, by them clicking on fewer leaky widgets, when presented with the leak information and the basic knowledge of what leak notifications are.

Based on user comments collected, 3 of the participants using the Telemarque Horoscope application expressed discomfort with the data that the application was utilizing and sending over the network (IMEI & GPS Location). 2 of the participants using the Super Backup SMS application expressed some discomfort with the application send Address Book and IMEI data. But another 2 of the participants did also mentioned that they could understand why the Super Backup SMS application required the use of their Address Book, and felt that this was ok. Overall, the results made sense as the general drop in the number of leaky widgets clicked by the users in Experiment 2 was corresponding to the increased concerns that they had with regards to the types of data the applications were sending. Furthermore, their comments also indicated that there was good awareness of how the applications were using and sending data, based on the appearance of the temporary messages besides leaking widgets.

Usability Results:

Both test and user groups were asked to rate the usability of the applications on a 7 point Likert (7-Highly Usable to 1-Not Usable at All). The differences in the Likert Scores between both Test and Control groups were not statistically significant, and it was observed that 70% (with notification) and 80% (without notification) of users gave a rating of 5 and above (10% decrease overall).

The results show that our mechanism does not greatly affect usability while still notifying users of privacy leaks.

Discussion

The user study demonstrates that under lab settings, the privacy messages are successful in decreasing user leaks cause by clicking on leaky-causing widgets/buttons during app usage. There were promising results with a 0% to 80% drop in users avoiding the leaky widgets, while still maintaining usability with statistically insignificant difference in usability ratings between the test and control groups. Furthermore, feedback obtained from the users demonstrated increased user perception and understanding of application data usage.

4.2 Large-Scale Field-Study

Next, I validated my notification mechanism on a larger number of users over a longer time period of 2 weeks. Users on non-rooted devices were targeted, and the study was designed to take place on their own personal mobile phones. The Field Study was performed using a 'between-participants' design, whereby 2 different groups of users were randomly selected to be subjected to Test (With Notifications) and Control (Without Notifications) conditions.

4.2.1 Conduct of Field-Study

Approach of the study:

The study was conducted from Carnegie Mellon University (CMU) and Singapore Management University (SMU) campuses, after clearing the Institutional Review Board (IRB) requirements of each institute. The study consisted of 3 phases. In the 1st phase, we conducted pre-study surveys and briefings, as well as installed our customized Notifications Application on the personal Android smartphones of each user. In the 2nd phase, the users utilized their smartphones and applications in their own capacity for a period of 2 weeks. The 3rd phase was conducted after the 2-week study, whereby the users were requested to complete a post-study survey.

Application	App Type	Privacy Leaks Displayed	Message Locations Within App
Facebook	Social-Media	(Address-Book, Camera, IMEI, Location)	(On Start-up; Beside 3 widgets)
Instagram	Social-Media	GPS	On Start-up
ChannelNewsAsia	News	Microphone	On advert tab
The Straits Times	News	IMEI	(On Start-up; Beside news items)
GMAIL	Utility	(Address-Book, Calendar , IMEI, Media Files)	(On Start-up; Beside 2 widgets)
UC Web-Browser	Utility	GPS & IMEI	(On Start-up; Beside 1 widget)
YouDao Dictionary	Utility	(Camera, IMEI, Microphone)	(On Start-up; Beside 2 widgets)
Temple Run 2	Game	GPS	On Start-up
Subway Surfers	Game	IMEI	On Start-up
Trivial Crack	Game	Address-Book	(On Start-up; Beside 4 widgets)

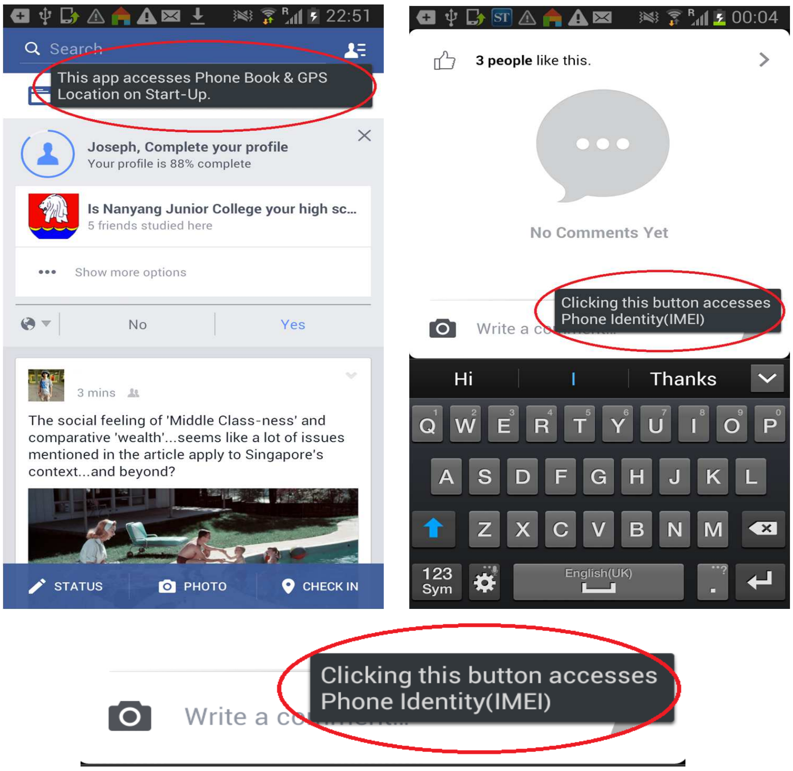
Table 4.2: List of Applications used in Field-Study

User characteristics and perceptions were measured from the pre and post-study surveys.

Characteristics of Notifications and Test Apps:

The Interactive Notifications App (described in Section 2.3) was utilized in the field study to display privacy messages to users interactively during application operation on their personal phones, using Android toast messages. (Refer to Table. 4.2 for the apps and placement of notifications) As the work does not study or compare the impact of phrasing or words used in the privacy messages, the messages were designed to be non-alarming and concise. The messages contained sentences such as *'This app accesses'* or *'Clicking this button accesses'*, followed by the data type. To ensure that users did not miss reading messages, they appeared for a 5-second period. Example of data types indicated included GPS Lication, Phone Identifier(IMEI), Phone Book and Media Files etc. (Figure. 4.4 shows the screencaptures)

10 test applications were utilized in the study (Table. 4.2), and were selected on the basis that they were already in regular usage by most of the test subjects. The Notifications Applications also sends back usage statistics to a data collection



This app accesses Phone Book & GPS Location on Start-Up.

Joseph, Complete your profile
Your profile is 88% complete

Is Nanyang Junior College your high sc...
5 friends studied here

The social feeling of 'Middle Class-ness' and comparative 'wealth'...seems like a lot of issues mentioned in the article apply to Singapore's context...and beyond?

3 people like this.

No Comments Yet

Clicking this button accesses Phone Identity(IMEI)

Clicking this button accesses Phone Identity(IMEI)

Figure 4.4: (Messages Circled) Left: Privacy Message warning of phone book and GPS Location Access ; Right: Privacy message appearing over a button warning of access of phone identifiers on clicking the button. Bottom: A Blow-Up of a Privacy Message

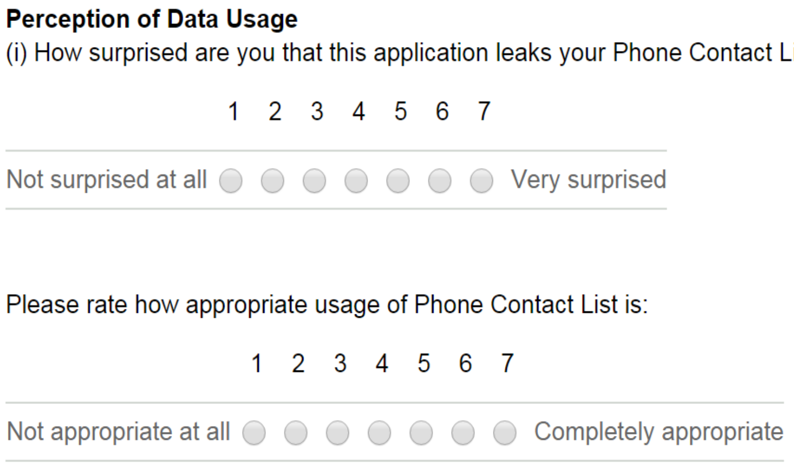
server for storage and processing. As shown in Table 4.2, the apps were chosen to be representative of important app categories: Social-Media, News, Utility and Game Apps. The privacy messages appeared on app start-ups as well as on-top of various widgets and views when users arrived at them during usage.

Participants

The field study had 47 users (37 from CMU and 10 from SMU). They were randomly divided into separate groups of Test (With Messages), and Control (Without Messages). (25 Users in Test, 22 Users in Control) They consisted of staff/students as well as external working adults. 37 of the participants were from CMU, while 10 of the participants were from SMU. Test subjects were recruited via posters displayed on campus grounds, as well as from online adverts from Craigslist Pittsburgh. The subjects were made up of staff/students, as well as external working adults from the ages of 19 to 42 years of age (33 males and 14 females). Each subject from CMU was compensated USD30.00, while each subject from SMU was compensated SGD30.00 for their participation in the 2-week study. Participants were required to be at least 18 years of age and own an Android smartphone with an OS of Android v4.1 and above.

Capture of User-Characteristics as Independent Variables

Characteristics of users were captured from pre and post-study surveys on 7-point Likert-Scale as well as Yes/No questions. (**Please refer to Appendix A, Figures A.1 & A.2 for the Survey Forms**) These characteristics include users' *Prior app Usage-time (Familiarity)*, *Surprise level*, *Usability*, *Privacy Consciousness level* and *Disapproval of app data-usage*. In addition to privacy messages, a subset of 17 users (10 males, 7 females) were randomly selected and provided with a stimulus of additional User Education on the meaning of messages (e.g. IMEI were identifiers that could allow tracking etc.) as well as implications of allowing accesses (private data could be sent over the network and stored by developers). This allows the inclusion of the effects of users being provided additional User Education in a regression model.



Perception of Data Usage
(i) How surprised are you that this application leaks your Phone Contact List?

1 2 3 4 5 6 7

Not surprised at all ○ ○ ○ ○ ○ ○ ○ Very surprised

Please rate how appropriate usage of Phone Contact List is:

1 2 3 4 5 6 7

Not appropriate at all ○ ○ ○ ○ ○ ○ ○ Completely appropriate

Figure 4.5: Questions in Post-Study Survey Form posed to each user asking them how surprised they were with the privacy leakages of each application, as well as to rate the appropriateness of the data access on a 7-Point Likert Scale

As can be seen from the form in Figure. 4.5, the appropriateness rating of each data type per application was phrased as the question: *Please rate how appropriate usage of "Data Type" is:* The rating was based on a 7-point Likert, with the maximum score of 7("Completely Appropriate") and a minimum score of 1("Not Appropriate at All"). We also gauged the surprise levels to the data leaks by the question: *How surprised are you that this application leaks "Data Type?"* This was again on a Likert scale, with 7("Very surprised") and 1("Not surprised at all."). They were also asked the question: *"Knowing the data leak characteristics of the app, will you still continue to use it?"* (Yes/No/Unsure)

Using the post-study survey form, we were able to measure users' level of approval/disapproval for each data-type leak for all of the test applications. A user was deemed to be 'OK' with the application's data leak behaviour if he/she answered 'Yes' to continuing to use the application on knowing its leak characteristics. Otherwise, the user was deemed to be 'Not-OK' with the application's leak behaviour. If the user replied 'Unsure', he/she would be deemed 'OK' if an average rating score of 3 and above for the Data-type appropriateness was provided. (Note: The survey question was directed towards the apps' usage privacy data and not towards individual privacy messages) We posted the same form for the users in both the Treatment and Control groups at the end of 2-weeks. We use these survey findings of user variability to analyze the experimental study results in the next section.

Multiple Linear Regression

I conducted a multivariate regression with characteristics as Independent variables, and 3 variables of (i) Frequency of App Start-Ups (Log), (ii) Duration of App Usage (Log) and (iii) No. of Leak Buttons Clicked (Log) as Dependent variables. We investigated these Dependent variables as they were proportional to the amount of privacy data being leaked by the application. The Independent Variables, their descriptions as well as the questions posed in the surveys to capture them are displayed in Table 4.3.

No.	Independent Variable	Description	Range
1	Privacy Messages	Did user receive privacy messages?	(Yes=1; No=0)
2	Disapproval of Data Usage	Does user disapprove of the data accessed for a particular app? -(From post-study survey)	(Yes=1; No=0)
3	User Education	Did user receive additional briefing session on data-type meanings and privacy implications?	(Yes=1; No=0)
4	App Type	Category of particular app	Categorical (1.Social Media, 2.News, 3.Utility, 4.Games)
5	Privacy Consciousness	Question to user: ”How privacy-conscious would you describe yourself to be when using mobile apps?” -(From pre-study survey)	Likert Scale (7:”Extremely Consciousness” - 1:”Not Consciousness at all”)
6	Usability of App	Question to user: ”Please rate how usable (convenient to use) the application is.” -(From post-study survey)	Likert Scale (7:”Very Usable” - 1:”Very Unusable”)
7	Surprise at Message Content	Question to user: ”How surprised are you that this application leaks your *”data-type”*?” -(From post-study survey)	Likert Scale (7:”Very Surprised” - 1:”Not Surprised at all”)
8	Prior App Usage Time (Familiarity)	Question to user: ”How long have you been using this app prior to this user-study?”” -(From post-study survey)	Values (3:Years, 2:Months, 1:Weeks, 0:Not used prior to study)

Table 4.3: Description of Independent Variables in Multiple Regression

Each of the 3 Dependent variables (Freq. of Start-Ups, Duration of App Usage (s) and No. of Leak Buttons Clicked) were analyzed separately in a log-linear regression model. The model consists of the logarithmic form of each of these 3 metrics as dependent variables, with the independent variables of Table. 4.3. We also included 2 and 3-factor interactions of Presence of Privacy Messages and Disapproval of Data Usage with the other independent variables. The regression model utilized is as follows:

Regression Model:

log(Frequency of Start Ups/Duration of App Usage/No. of Leak Buttons Clicked) $=$

$$Is\_Notification\_Present + App\_OK\_Not\_OK + Is\_Context\_Present + factor(AppType) + Privacy\_Consciousness + Usability + Familiarity + Surprise\_Level + Usability : Privacy\_Consciousness + Is\_Notification\_Present : App\_OK\_Not\_OK + Is\_Notification\_Present : Is\_Context_Present + Is\_Notification\_Present : factor(AppType) + Is\_Notification\_Present : Privacy\_Consciousness + Is\_Notification\_Present : Usability + Is\_Notification\_Present : Familiarity + Is\_Notification\_Present : Surprise\_Level + App\_OK\_Not\_OK : Is\_Notification\_Present : Is\_Context\_Present + App\_OK\_Not\_OK : Is\_Notification\_Present : factor(AppType) + App\_OK\_Not\_OK : Is\_Notification\_Present : Privacy\_Consciousness +$$

$App\_OK\_Not\_OK : Is\_Notification\_Present : Usability + App\_OK\_Not\_OK : Is\_Notification\_Present : Familiarity + App\_OK\_Not\_OK : Is\_Notification\_Present : Surprise\_Level$

4.2.2 Results of Field-Study

The results of the multivariate regressions are in Table. 4.4. The p-values of the individual independent variables are displayed in the last column. We look out for variables at 99%, 95% and 90% intervals (p-value$\leq$0.001, 0.05 & 0.1), and negative estimates that indicate the effectiveness of privacy messages by usage decreases. The explanatory values of R^2 were moderate from 0.294 to 0.442.

From Table. 4.4, of the individual main effects (List items 1-8), and for the mobile usage metrics of (i) Frequency and (ii) Duration of App Usage, 2 factors of 'Application Type' as well as 'Prior App Usage Time (Familiarity)' were found to be significant. For example in the 1st row of Table. 4.4, line 8, users who had used the application previously for a longer period of time (Familiarity) had a significant higher frequency of app start-ups (Estimate=0.15, p-value=1.83×10^{-7}). For the metric of of (iii) No. of Leak Widgets Clicked, 'App Type' and 'User Education' had significant effects.

I found significant 3-factor interaction effects of 'Privacy Messages' and the 'Disapproval of Data Usage' to: 'User Education', 'Privacy Consciousness', 'App Type (Games)' and 'Surprise at Message Content'. For example in the 2nd row of Table. 4.4, line 9, users who received privacy messages and disapproved of the data usages had a decrease in the Duration of App Usage, if he/she had the additional stimulus of user education (Estimate=-1.23, p-value=1.38×10^{-4}).

Effectiveness of Privacy Messages Mechanism

While the results indicate that the effects of Privacy Messages did not significantly alter user behaviour across all users, the study highlights that a 'Non-Blocking'

Depend. Variable	Independent Variable	Estimate	Std. Error	*p*-value
(i) Freq. of App Start-Ups (Log)	1. Privacy Messages	-0.20	0.43	0.63
	2. Disapproval of Data Use	-0.08	0.09	0.37
	3. User Education	0.07	0.09	0.45
	4a. (Social Media)	Baseline	Baseline	Baseline
	4b. (News)**	**-0.39**	**0.18**	**0.03**
	4c. (Utility)***	**-0.30**	**0.11**	**0.005**
	4d. (Games)***	**-0.52**	**0.11**	**-9.87×10^{-6}**
	5. Privacy Consciousness	1.32×10^{-3}	0.06	0.98
	6. Usability of App	-0.02	0.05	0.74
	7. Surprise at Message Cont.	0.01	0.02	0.62
	8. Prior App Usage Time (Familiarity)***	**0.15**	**0.03**	**1.83×10^{-7}**
	9. Interactions (1 : 2 : 3)***	**-0.46**	**0.16**	**0.005**
	10. Interactions (1 : 2 : 5)*	**-0.12**	**0.06**	**0.06**
	(Adjusted R^2 = 0.442)			
(ii) Dur. (sec) of App Usage (Log)	1. Privacy Messages	-0.13	0.83	0.88
	2. Disapproval of Data Use	-0.01	0.18	0.95
	3. User Education	0.02	0.17	0.90
	4a. (Social Media)	Baseline	Baseline	Baseline
	4b. (News)	0.11	0.35	0.76
	4c. (Utility)**	**-0.50**	**0.20**	**0.016**
	4d. (Games)**	**-0.45**	**0.22**	**0.04****
	5. Privacy Consciousness	0.12	0.12	0.33
	6. Usability of App	0.06	0.10	0.54
	7. Surprise at Message Cont.	0.04	0.04	0.31
	8. Prior App Usage Time (Familiarity)***	**0.24**	**0.05**	**1.23×10^{-5}**
	9. Interactions (1 : 2 : 3)***	**-1.23**	**0.32**	**1.38×10^{-4}**
	10. Interactions (1 : 2 : 5)**	**-0.28**	**0.12**	**0.021**
	11. Interactions (1 : 2 : 7)*	**-0.15**	**0.08**	**0.07**
	(Adjusted R^2 = 0.316)			
(iii) No. of Leak Buttons Clicked (Log)	1. Privacy Messages	0.04	1.73	0.98
	2. Disapproval of Data Use	-2.52	1.61	0.12
	3. User Education*	**-0.83**	**0.47**	**0.08**
	4a. (Social Media)	Baseline	Baseline	Baseline
	4b. (News)	-0.43	0.67	0.52
	4c. (Utility)***	**-2.38**	**0.46**	**5.34×10^{-7}**
	4d. (Games)***	**-2.51**	**0.88**	**4.66×10^{-3}**
	5. Privacy Consciousness	-0.09	0.31	0.76
	6. Usability of App	-0.13	0.23	0.58
	7. Surprise at Message Cont.	-0.03	0.12	0.78
	8. Prior App Usage Time (Familiarity)	0.10	0.12	0.40
	9. Interactions (1 : 2 : 3)***	**-3.83**	**0.95**	**7.86×10^{-5}**
	10. Interactions (1 : 2 : 4d)**	**-3.89**	**1.58**	**0.01**
	11. Interactions (1 : 2 : 7)*	**-0.44**	**0.25**	**0.08**
	(Adjusted R^2 = 0.294)			

Table 4.4: Results: Multiple Linear Regression of Field Study (***99%, **95%, *90% Confidence Intervals)

mechanism can be still be effective under certain conditions. The significant 3-factor interaction effects uncovered that user-disapproval with accesses, user-education, inherent levels of privacy consciousness as well as high surprise-levels are required for the effectiveness of a 'Non-Blocking' mechanism. Although a comparison study on non-blocking and blocking mechanisms is beyond the scope of this thesis, I see that, for stronger overall effects, especially for users who have a longer history of usage of the particular app as well as for warning of more serious issues such as malware, 'Blocking' mechanisms would be advocated.

4.2.3 Discussion & Factors Influencing Usage Behaviours

Users utilized Social Media apps significantly more than the other app categories. The Prior App Usage Time (Familiarity) had the highest statistical strength in influencing app usage behaviours of App Start-up Frequency and App Usage Durations across all users. This implies that users who have been utilizing the applications for a longer period of time were less likely to decrease utilization of the applications, in spite of receiving the Privacy Messages.

This has both positive and negative implications. A positive implication is that implementing privacy messages on applications is unlikely to cause its overall rate and duration of usages to drop. And this supports the notion that mobile platforms or developers can readily deploy privacy messages to users without badly decreasing overall usage rates. A negative implication is that privacy messages might not function well in scenarios where strong overall effects on users are desirable (e.g. Warning of security issues).

As mentioned earlier, all significant interactions found included both Privacy Messages together with the Disapproval of Data Use. The user's disapproval of data use by an app was thus critical in determining the effectiveness of privacy messages. We did not find any significant 3-factor interactions involving Prior App Usage Time (Familiarity),(i.e.Interactions(1:2:8)). This indicates that users who received privacy messages and disapproved were unaffected by App Familiarity.

From in Table. 4.4, the strongest interaction for all 3 usage metrics was between (Privacy Messages: Disapproval of Data Usage: User Education). Users who received privacy messages, disapproved of data usage indicated and had additional user education had decreased app usage metrics. User Education was thus a strong factor allowing for the effectiveness of the privacy messages. The implications are that it is important for app stores and developers to ensure that users have a proper understanding of the meanings as well as consequences of privacy related messages to optimize its effectiveness. It would be greatly beneficial for platforms to include educational aspects in the privacy messaging framework. A possible idea might be the implementation of voice messages or sliding screens detailing privacy implications and explanations of data meanings, which might improve the effectiveness of privacy messages. There exists opportunities for future exploration in this area.

Another significant interaction found was that between (Privacy Messages: Disapproval of Data Usage: Privacy Consciousness) for Frequency of Start-ups, as well as for Duration of App Usage. Users who received privacy messages and disapproved of the data usages had decreased app usage metrics if they had a higher level of Privacy Consciousness. This was an indication that users who disapproved of the privacy message contents were reacting accordingly to their privacy comfort levels.

A third significant interaction was between (Privacy Messages: Disapproval of Data Usage: Surprise at Message Content) for Duration of App Usages and Number of Widgets clicked. Users who were more surprised at the messages, and disapproved of the data usages, were thus more likely to have decreased app usage metrics. Users were more likely to react properly to messages in which there were strong contrasts between legitimate and questionable usage, which increased surprise. This suggests that it might be beneficial for platforms to utilize message designs that increase this contrasts, so that users have a higher level of surprise towards the privacy accesses indicated.

Chapter 5

Conclusion

In this dissertation, my main motivation is the investigation into the novel and unique context of causes of user-triggered privacy app behaviours as well as the activity transition paths towards these behaviours. I presented the design, implementation and evaluation of my solution (MAMBA) for achieving this goal. The solution involves a hybrid application of static analysis and automated runtime testing of app binaries. And the techniques utilized are control-flow analysis as well as dataflow analysis for callback functions.

My MAMBA solution achieved a much higher success rate (Over 70% compared to 13%) in the analysis of callback functions than previous recent work (GATOR [63]), for the purposes of generating an activity transition graph (ATG) that guides automated runtime testing of Android apps. The solution also demonstrated large time savings compared to another automated testing system, the Automated Model Checker (AMC), with only a small trade-off in activity coverage. MAMBA's static analysis also had high accuracy in outputting activities with privacy access behaviours (79.84%), and had a moderate accuracy in outputting activities with privacy leak behaviours (35.66%).

User studies demonstrated that presentation of the additional context can help users to improve their app usage behaviours. Improvements in app usage behaviours were measured by the decrease in app start up frequencies, usage durations as well as clicking on fewer privacy culprit buttons/widgets, for the apps with privacy usages which users disapproved of. It was also found that for the 'Non-Blocking' delivery mechanism, the users' prior app usage history was the strongest determining factor influencing privacy behaviours. And there were certain conditions which enhanced the effectiveness of the 'Non-Blocking' mechanism - Namely users' disapproval of privacy messages, sufficient understanding on the meaning and implication of notices, being inherently privacy conscious as well as surprise towards the app's privacy behaviours in the notices.

I describe some possible extensions for future work in the next section.

5.1 Future Work

5.1.1 Extensions to MAMBA System

Static Analysis of App Privacy Leaks

An extension to MAMBA's static analysis to output target activities with privacy leak behaviours would be very useful. At the moment, MAMBA does not analyze the control flow of network function calls (e.g. HTTP and POST), and is thus unable to differentiate between privacy access and privacy leak behaviours.

This extension could be performed by utilizing FlowDroid's [9] static taint tracking to analyze network calls as 'sink' points, to observe if sensitive data variables are linked to network function calls. This would likely increase the precision of MAMBA's privacy leak detection from its current 35.7%, and allow the direct use of static analysis outputs for user notifications without the need for runtime verification with TaintDroid.

A performance concern however, would be an increase in the time required for completion of static analysis. While FlowDroid's taint tracking is known to have reasonable overheads [9], an evaluation would be required for accurate determination.

Improved Analysis of Android Callback Functions

At the moment, while MAMBA has a high success rate for analysis of Android Callback Functions, it suffers from generalizability issues. This reduces the capability to output fully directed test cases, which it overcomes by producing semi directed test cases. These issues stem from MAMBA missing out on possible variants in the implementation of listener invocations within the Android framework, and the problem is compounded by differences in Android API versions as well as the utilization of custom view libraries by developers. In-depth modelling of listener invocations could be thus performed to enhance MAMBA's capability of analyzing callback functions.

Improved Runtime Privacy Detection

MAMBA current instruments 2 privacy detectors, PMP and TaintDroid. While these privacy detectors are well regarded and extensively utilized by the research community, they have reported vulnerabilities that malicious apps can potentially exploit and bypass [11]. Future work could be aimed towards patching up some of these vulnerabilities of privacy detectors for improved privacy detection. The solution would require the proposal of defences against some of the reported attacks in existing literature (e.g. 'System-File Attack' or 'File Length Attack' etc.).

Another possibility to improve runtime privacy detection is the tracking of kernel related system calls which are linked to users' data privacy. A survey would be required to document which system calls are related to the privacy data as well as network and other calls that could indicate potential privacy leaks on Android. Such a survey might be done manually or by using automated tools (e.g. SUSI [49]). While the tracking of kernel system calls would certainly result in an improvement in the accuracy of runtime privacy detection, a concern is that large overheads on the phone would be incurred. However, as MAMBA is based on back end analysis, these overheads would not be relevant as they would be incurred on test devices instead of on user devices.

Extended Comparisons of Automated Testers

From Appendix A, I detailed 4 types of automated testing solutions, based on a survey of existing research. These are: (i) Random, (ii) Systematic Event Selection, (iii) Model-Based (Built at runtime) and (iv) Model-Based (Built from static analysis). A comparison could be made by the re-implementation of representative systems from each of these groups (e.g. Monkey [18], VanarSena [51], PUMA [31] and A^3E [10]), and evaluating them with the MAMBA automated tester on a scaled up number of a few thousand apps. A significant contribution can be claimed if MAMBA can demonstrate improvements in terms of lower testing time, higher coverage as well as the reachability of certain runtime app states that other automated testers might be unable to reach.

5.1.2 Privacy User Studies

Improvement & Extension of Field Study

The field study conducted in Chapter 4.2 can be improved by changing the study from a 'Between-participants' to a 'Within-participants' design, extending the number of test apps, controlling users' prior knowledge on the test apps as well as by improving the user survey mechanism.

The current field study was conducted with a 'Between-participants' design, in which the test stimulus was conducted over 2 different groups of users. While this was acceptable, it requires a large number of users to be statistically confident. While 47 users cannot be considered a small number for a HCI study, a larger number of users would increase the validity of the findings. Instead of conducting the study over more users, it would however be advantageous to change the study design to a 'Within-participants' design, whereby the comparisons would be made based on test and control stimulus presented to the same user.

While the current study's 10 test apps were solicited from a pre-study survey, the current app list might be too small to optimally represent the apps that all users were already regularly using. There were users who were regularly using only a small number of apps from the list prior to the study. An improvement can be made by observation of users' app usage histories (this can be done during the control phase of a 'Within-participants' design), and extending the list of test apps based on these user histories.

Another improvement can be made by controlling users' prior knowledge on the test apps in which privacy messages would appear on. In the current field study, users were aware of the test apps that were being utilized in the study, as they were required to install any apps that they did not already have from the list of 10 test apps. This could have introduced biases in the study, as users could be utilizing the test apps more with the mistaken idea that this would help the study. Users should thus be withheld any information with regard to the names of the test apps in future studies.

Finally, the survey mechanism can be improved in future studies. In the current field study, users were surveyed only twice - Pre study and Post study. The users can be surveyed more frequently at regular intervals during the duration of the study (e.g. Survey at every 3-day interval). A customized survey mechanism would have to be developed that can send prompts at regular intervals to users, as well as generate survey forms for users based on their app usage during in between the survey time intervals. More fine grained surveys will mean that more fine grained user characteristics can be obtained (e.g. User approval at the data level for each app, or at the message level granularity).

In addition, usage of a customized survey mechanism can allow extension of the field study to investigate if the privacy notices can allow users to set better privacy policies in data access control of their apps. These might be posed in the form of survey questions, such as whether they would prefer to allow or deny access of the data utilization behaviour of their test apps. This can be used to demonstrate that providing the additional context of causes of user-triggered privacy leaks can allow users to set better privacy policies, so that apps' privacy behaviours follow closer to their provided functionalities.

Study on Privacy Notification Mechanisms

In Appendix A, a comparisons table for Privacy Notification Systems is provided. The table details the notification mechanisms utilized for the privacy systems surveyed. Future work might be performed to compare between the dimensions of 'Notification Content' and 'Notification Interface' of these other privacy systems with that of MAMBA's privacy notification mechanism. In addition, the different factors of 'Frequency', 'Timing' as well as 'Granularity' might be investigated for their efficacy in the effectiveness of the notification mechanisms.

The 'Notification Content' of my MAMBA system can be compared together with 2 existing types: (i) Notices on real time privacy data accesses and (ii) Summarized privacy access control lists can be evaluated together. Ideally, the evaluation should be performed in field studies. But for ease of conduct can also be performed on mechanical turk platforms, where users could be shown mock ups of these contents or screen captures. Users would be surveyed on the privacy access policies that they would set (Allow/Deny). The evaluation metrics could be - Functionality profile of the app (App functionality requires/does not require privacy data type) as well as the Correctness of privacy profile set by user (% of privacy data access control decisions that fits app's functionality profile).

The 'Notification Interface' of my MAMBA system can also be compared with a few types of: (i) Install Time List, (ii) Android Notifications (Standard scroll down), (iii) Dialog Box and (iv) Summary List. This might be performed under a lab setting, where users would be asked to complete tasks on the apps, and observing the appearance of different types of notification interfaces. The evaluation metrics could be - User feedback on usability and preference, Recall of users, Task completion time and Effectiveness of privacy leak avoidance.

Bibliography

[1] Google play. `https://play.google.com`.

[2] I. Adjerid, A. Acquisti, L. Brandimarte, and G. Loewenstein. Sleights of privacy: Framing, disclosures, and the limits of transparency. In *Proceedings of the Ninth Symposium on Usable Privacy and Security*, page 9. ACM, 2013.

[3] Y. Agarwal and M. Hall. Protectmyprivacy: detecting and mitigating privacy leaks on ios devices using crowdsourcing. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 97–110. ACM, 2013.

[4] R. Agrawal, R. Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994.

[5] H. Almuhimedi, F. Schaub, N. Sadeh, I. Adjerid, A. Acquisti, J. Gluck, L. F. Cranor, and Y. Agarwal. Your location has been shared 5,398 times!: A field study on mobile app privacy nudging. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 787–796. ACM, 2015.

[6] R. Amadeo. App ops: Android 4.3's hidden app permission manager, control permissions for individual apps! *http://www.androidpolice.com/2013/07/25/app-ops-android-4-3s-hidden-apppermission-manager-control-permissionsfor-individual-apps*, 2013.

[7] S. Amini, J. Lin, J. I. Hong, J. Lindqvist, and J. Zhang. Mobile application evaluation using automation and crowdsourcing. 2013.

[8] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 59. ACM, 2012.

[9] 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. *ACM SIGPLAN Notices*, 49(6):259–269, 2014.

[10] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. *ACM SIGPLAN Notices*, 48(10):641–660, 2013.

[11] G. S. Babil, O. Mehani, R. Boreli, and M.-A. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *Security and Cryptography (SECRYPT), 2013 International Conference on*, pages 1–8. IEEE, 2013.

[12] R. Balebako, J. Jung, W. Lu, L. F. Cranor, and C. Nguyen. Little brothers watching you: Raising awareness of data leaks on smartphones. In *Proceedings of the Ninth Symposium on Usable Privacy and Security*, page 12. ACM, 2013.

[13] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 27–38. ACM, 2012.

[14] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th workshop on mobile computing systems and applications*, pages 49–54. ACM, 2011.

[15] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall. Brahmastra: Driving apps to test the security of third-party components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1021–1036, 2014.

[16] E. K. Choe, J. Jung, B. Lee, and K. Fisher. Nudging people away from privacy-invasive mobile apps through visual framing. In *IFIP Conference on Human-Computer Interaction*, pages 74–91. Springer, 2013.

[17] A. M. Developer. Android Market API, Oct 2013. `https://code.google.com/archive/p/android-market-api/`.

[18] A. Developers. The developers guide. ui/application exerciser monkey, 2012.

[19] A. Developers. Accessibility Service: Developer guide, Oct 2013. `http://developer.android.com/reference/android/accessibilityservice/AccessibilityService.html`.

[20] A. Developers. Android Build Source: Developer guide, Mar 2016. `https://source.android.com/source/building.html`.

[21] A. Developers. Android Toast: Developer guide, Mar 2016. `http://developer.android.com/guide/topics/ui/notifiers/toasts.html`.

[22] Dictionary. Dictionary of english words, Oct 2013. `http://wordlist.sourceforge.net/12dicts-readme.html`.

[23] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *NDSS*, pages 177–183, 2011.

[24] W. Enck, P. Gilbert, S. Han, V. Tendulkar, 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. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[25] P. Faruki, S. Bhandari, V. Laxmi, M. Gaur, and M. Conti. Droidanalyst: Synergic app framework for static and dynamic app analysis. In *Recent Advances in Computational Intelligence in Defense and Security*, pages 519–552. Springer, 2016.

[26] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 576–587. ACM, 2014.

[27] H. Fu, Y. Yang, N. Shingte, J. Lindqvist, and M. Gruteser. A field study of run-time location access disclosures on android smartphones. *Proc. USEC*, 14, 2014.

[28] GitHub. Sensor simulator, April 2016. `https://github.com/openintents/sensorsimulator`.

[29] L. P. Guard. Lbe privacy guard, Mar 2016. `https://www.amazon.com/Team-LBE-Privacy-Guard/dp/B004UR5KSG`.

[30] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[31] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 204–217. ACM, 2014.

[32] M. Harbach, S. Fahl, P. Yakovleva, and M. Smith. Sorry, i dont get it: An analysis of warning message texts. In *International Conference on Financial Cryptography and Data Security*, pages 94–111. Springer, 2013.

[33] R. Holly. Using app permissions in android m, June 2015.

[34] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang. Supor: Precise and scalable sensitive user input detection for android apps. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 977–992, 2015.

[35] J. C. J. Keng, L. Jiang, T. K. Wee, and R. K. Balan. Graph-aided directed testing of android applications for checking runtime privacy behaviours. In *11th IEEE/ACM International Workshop on Automation of Software Test (AST 2016)*, 2016.

[36] J. C. J. Keng, T. K. Wee, L. Jiang, and R. K. Balan. The case for mobile forensics of private data leaks: towards large-scale user-oriented privacy protection. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, page 6. ACM, 2013.

[37] J. C. J. Keng, T. K. Wee, L. Jiang, and R. K. Balan. Demo of mobile forensics: Identification of leak causes in mobile applications. In *4th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys 2013)*, 2013.

[38] M. Kern and J. Sametinger. Permission tracking in android. In *The Sixth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies UBICOMM*, pages 148–155, 2012.

[39] K. Lee, J. Flinn, T. J. Giuli, B. Noble, and C. Peplin. AMC: Verifying user interface properties for vehicular applications. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 1–12. ACM, 2013.

[40] M. Lindorfer, M. Neugschwandtner, and C. Platzer. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, volume 2, pages 422–433. IEEE, 2015.

[41] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. Van Der Veen, and C. Platzer. Andrubis–1,000,000 apps later: A view on current android malware behaviors. In *2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, pages 3–17. IEEE, 2014.

[42] B. Liu, S. Nath, R. Govindan, and J. Liu. Decaf: detecting and characterizing ad fraud in mobile apps. In *Proc. of NSDI*, 2014.

[43] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[44] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 328–332. ACM, 2010.

[45] [Online]. You think facebook privacy is bad? take a look at smartphone games, Nov. 2014.

[46] [Online]. Android developers api class reference, http://developer.android.com/reference/classes.html, Oct. 2015.

[47] [Online]. Hierarchy viewer android, http://developer.android.com/tools/help/hierarchy-viewer.html, Jan. 2016.

[48] [Online]. Protectmyprivacy (pmp) for android, http://www.android.protectmyprivacy.org/, Jan. 2016.

[49] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, 2014.

[50] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 209–220. ACM, 2013.

[51] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and scalable fault detection for mobile applications. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 190–203. ACM, 2014.

[52] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes. Recon: Revealing and controlling pii leaks in mobile network traffic. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (New York, NY, USA, 2016), MobiSys*, volume 16, 2016.

[53] A. e. a. Rountev. Gator: Program analysis toolkit for android, 2016. `http://web.cse.ohio-state.edu/presto/software/gator/downloads/`.

[54] F. Schaub, R. Balebako, A. L. Durity, and L. F. Cranor. A design space for effective privacy notices. In *Eleventh Symposium On Usable Privacy and Security (SOUPS 2015)*, pages 1–17, 2015.

[55] P. Silva, V. J. Amorim, F. N. Ribeiro, and I. Muzetti. Privacymod: Controlling and monitoring abuse of privacy-related data by android applications. In *2015 Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 42–47. IEEE, 2015.

[56] D. Stites and K. Skinner. User privacy on ios and os x, 2014.

[57] J. Tan, K. Nguyen, M. Theodorides, H. Negrón-Arroyo, C. Thompson, S. Egelman, and D. Wagner. The effect of developer-specified explanations for permission requests on smartphone user behavior. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pages 91–100. ACM, 2014.

[58] R. Vallee-Rai and L. J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations. 1998.

[59] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Profiledroid: multi-layer profiling of android applications. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, pages 137–148. ACM, 2012.

[60] M. S. Wogalter, V. C. Conzola, and T. L. Smith-Jackson. Research-based guidelines for warning design and evaluation. *Applied ergonomics*, 33(3):219–230, 2002.

[61] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 539–552, 2012.

[62] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan. Intentfuzzer: detecting capability leaks of android applications. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 531–536. ACM, 2014.

[63] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *International Conference on Software Engineering (ICSE)*, 2015.

[64] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated gui-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering*, pages 250–265. Springer, 2013.

[65] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054. ACM, 2013.

Appendix A

Appendix

List of Privacy Sensitive Data API Invocation Calls Utilized By MAMBA

- getLatitude
- getLongitude
- getLastKnownLocation
- requestLocationUpdates
- getLastLocation
- getConnectionInfo
- getMacAddress
- getDeviceId
- getSubscriberId
- getSimSerialNumber
- ContactsContract
- pictureFile
- getCameraInfo
- FEATURE_CAMERA
- getLine1Number
- startRecording
- MediaRecorder
- mRecorder
- CallLog.Calls
- content://sms
- CalendarContract
- ClipboardManager
- Secure.ANDROID_ID

Algorithm 1 Algorithm For Sensitive API Association with App Activities

```
INPUT: Sensitive Data APIs
OUTPUT: Target App Activities Associated with APIs To Reach/Test

JimpleSet $\leftarrow$ getJimpleBytecodeRepresentations()
sensitiveAPISet $\leftarrow$ getSensitiveAPISet()
activityClassSet $\leftarrow$ getActivityClasses()
activityMethodSet $\leftarrow$ getMethods()

begin

for each Activity $\in$ activityClassSet do

    for each Method $\in$ activityMethodSet do
        Method Body $\beta \in$ getActiveBody()
        All Sensitive API Invocation Points $\alpha \leftarrow$
        getAllSensitiveInvocationPoints()

        for each Invocation Point in $\alpha$ do
            Get Methods along back-tracked CFG, backTrackMethodSet $\leftarrow$
            backTrackCFG($\alpha$)

            for each Method $in$ backTrackMethodSet do
                Activity Class containing Method $\mu \leftarrow$ getClass(Method)
                if Method is of 'onCreate' lifecycle then
                    if $\mu \in$ activityMethodSet then
                        Target Activity to Reach $\tau 1 \leftarrow$
                        targetActivityToReachSet.add($\mu$)

                else if Method is of 'User-Input' Handling Type then
                    if $\mu \in$ activityMethodSet then
                        Target Activity to Test $\tau 2 \leftarrow$
                        targetActivityToTestSet.add($\mu$)

end
```

Algorithm 2 Algorithm For Generating Fully-Directed Activity Transition Graph (ATG)

```
INPUT: Application Binary
OUTPUT: Edges of Activity Transition Graph (ATG)

JimpleSet $\leftarrow$ getJimpleBytecodeRepresentations()
CallbackSet $\leftarrow$ getCallbackClasses()
activityClassSet $\leftarrow$ getActivityClasses()

begin

for each Activity $\in$ activityClassSet do

    for each Callback Method $\in$ in Activity do

        if Method is of 'onCreate' lifecycle then
            if findStartActivityInvocation() is True then
                ATG Edge $\varepsilon$ $\leftarrow$ ATG.add(callback_of_previousActivity,
                callback_of_nextActivity)

        Listener Invocation Point $\lambda$ $\leftarrow$ getListenerInvocation()

        Get Statements Along Backwards Data Flow of Listener, $\Phi$ $\leftarrow$
        dataFlowTracking($\lambda$,Method)

        for each Statement in $\Phi$ do
            if Statement $\in$ View ID Retrieval from XML Resources then
                Get View ID, ID $\leftarrow$ getID(Statement)
                Get Widget Identifier, I $\leftarrow$ getXMLResource(ID)

            if Statement Contains Object File of Listener then

                Get Active Body of User-driven Callback Function
                of Object File, body $\leftarrow$
                getActivityBodyObjectListener(Object File)

                Get Next Activity from Bytecode body, nextActivity $\leftarrow$
                getNextActivity(body)

        Get User-driven Callback Function of Previous Activity,
        callbackPreviousActivity $\leftarrow$ getCallBack(Activity)

        Get 'onCreate' Callback Function of Next Activity,
        callbackNextActivity $\leftarrow$ getCallBack(nextActivity)

        ATG Edge $\varepsilon$ $\leftarrow$ ATG.add(callbackPreviousActivity,callbackNextActivity
        , I)

end
```

Algorithm 3 Algorithm For Generating Semi-Directed Activity Transition Graph (ATG)

```
INPUT: Application Binary
OUTPUT: Edges of Activity Transition Graph (ATG)

JimpleSet $\leftarrow$ getJimpleBytecodeRepresentations()
CallbackSet $\leftarrow$ getCallbackClasses()
activityClassSet $\leftarrow$ getActivityClasses()

begin

for each Activity $\in$ activityClassSet do

    for each Callback Method $\in$ in Activity do

        Listener Invocation Point $\lambda \leftarrow$ getListenerInvocation()

        if $\lambda$ != Null then
            Get All Statements in Method, $\mu \leftarrow$ getAllStatements(Method)

            for each Statement in $\mu$ do
                if Statement $\in$ View ID Retrieval from XML Resources then
                    Get View ID, ID $\leftarrow$ getID(Statement)
                    Get Widget Identifier, I $\leftarrow$ getXMLResource(ID)

                    Add to List of Possible Widget Identifiers,
                    possibleWidgetIDs.add(I)

    Get All Listener Object Files of Activity, $\tau$
    $\leftarrow$ getAllListenerObjects(Activity)

    for each Object File $O$ in $\tau$ do

        Get Active Body of User-Driven Callback Function of Object File, body
        $\leftarrow$ getActivityBodyObjectListener(Object File)

        if startActivity Intent in body then
            Get Next Activity from Bytecode Body, nextActivity
            $\leftarrow$ getNextActivity(body)

            Add to List of Possible Activity Transitions, possibleActivity_
            Transitions.add(nextActivity)

    for each Activity, nextActivity, in possibleActivityTransitions do
        for each Widget Identifier,I, in possibleWidgetIDs do

            ATG Edge $\varepsilon \leftarrow$ ATG.add(callbackPreviousActivity,callbackNextActivity
            , I)

end
```

Table A.1: **Comparison Table of Privacy Detection & Protection Systems**

System	**Privacy-Related Behaviour Analyzed**	**Usable Outputs**	**Technique for App Coverage**	**Latency**	**Coverage**	**Accuracy**
1. AppIntent [65]	Data transmissions and the UI states during such occurrences for an analyst to make determinations.	Sequence of app screen captures of app operation during data transmissions.	Static taint analysis/Dynamic Analysis (Symbolic Execution)	2 hours/app	High	Not reported.
2. AppsPlay-Ground [50]	Malicious behaviour in apps. (e.g. root exploits, background SMS sending etc.)	Flagging of apps as having malicious behaviour and data leaks.	Dynamic Analysis (Fuzz testing)	Medium to low	33% code coverage	Not reported.
3. Apposcopy [26]	Privacy malware.	Flagging of apps with control/data-flow signatures matching known privacy malware signatures.	Static analysis (Malicious Signature Matching)	5.8 mins/app	High	90%
4. ANDRUBIS [41]	Malicious behaviours	Analysis report on malicious app behaviour from an online submission.	Static/Dynamic Analysis (Machine-Learning Classification)	20 mins/app	Not reported.	>90%
5. DroidAnalyst [25]	Signature-based malware detection.	Analysis report on malicious behaviour.	Static/Dynamic Analysis (Machine-Learning Classification)	>20 mins/app	50-80%	85%
6. FlowDroid [9]	Taint analysis of privacy data.	Program Sources & Sinks of privacy leaks.	Static analysis	2.5 mins/app	High	93%
7. GORT [7]	Privacy Behavioural heuristics	Provides a GUI for analysts to easily digest privacy behaviours of apps.	Static/Dynamic Analysis	> 60 mins/app	49.5% activity coverage	Not reported.
8. MARVIN [40]	App privacy risk level	Provides a privacy risk assessment score.	Static/Dynamic Analysis (Machine-Learning Classification)	>20 mins/app	Not reported.	98%
9. Profile-Droid [59]	Network traffic of app	Provides origins/sources of the network traffic.	Static/Dynamic Analysis	>1 day/app (Measurements taken multiple times)	Nil	Nil
10. PiOs [23]	Taint analysis of privacy data.	Program Sources & Sinks of privacy leaks.	Static analysis	Not reported.	High	Not reported.
11. SUPOR [34]	Sensitive user inputs (e.g. passwords, credit card nos. etc.)	Flag vulnerabilities with sensitive user inputs.	Static analysis	3.7 mins/app	High	97.3%
12. ReCon [52]	Personally identifiable information (PIIs) leaked from apps.	Information on leaks of PIIs by apps.	Dynamic analysis (Machine-Learning from Network Flows)	Few mins/app	Not reported.	98.1%

Table A.2: **Comparison Table for Automated Testing Systems**

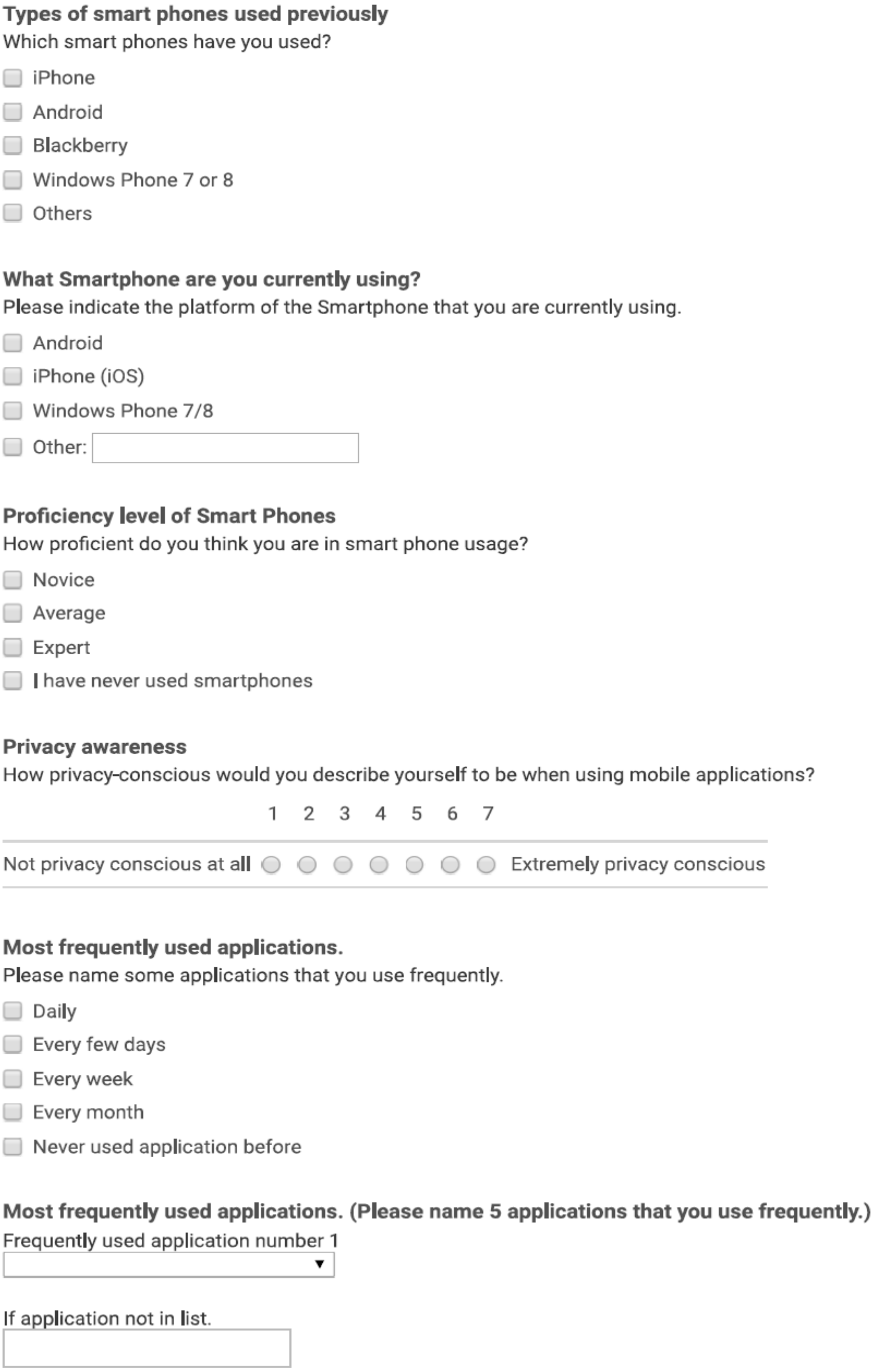
System	**Exploration Strategy**	**Exploration Target**	**Exploration Technique**	**Inputs Simulated**	**Purpose**
1. IntentFuzzer [62]	(i) Random	All Pages	Dynamic Analysis	UI Events	Stress Testing
2. Monkey [18]	(i) Random	All Pages	”	UI Events	General Testing
3. Dynodroid [43]	(ii) Systematic Event Selection	All Pages	”	UI Events/ System Inputs	Crash Testing
4. VanarSena [51]	(ii) Systematic Event Selection	All Pages	”	UI Events	Crash Testing
5. AppsPlayground [50]	(ii) Systematic Event Selection	All Pages	”	UI Events	Privacy/ Fuzz Testing
6. ACTEve [8]	(ii) Systematic Event Selection	All Pages	”	UI Events/ System Inputs	Concolic Testing
7. PUMA [31]	(iii) Model-Based (On-the-Fly at Runtime)	Flexible/ Customizable	”	UI Events	General Testing
8. AMC [39]	(iii) Model-Based (On-the-Fly at Runtime)	Distinct Pages	”	UI Events	UI Checking
9. A^{3}E(Depth-First Mode) [10]	(iii) Model-Based (On-the-Fly at Runtime)	All Pages	”	UI Events	General Testing
10. DECAF [42]	(iii) Model-Based (On-the-Fly at Runtime)	Distinct Pages	”	UI Events	Ad layouts
11. A^{3}E(Targeted Mode) [10]	(iv) Model-Based (Pre-Runtime Analysis)	Distinct Pages	Dynamic & Static Analysis	UI Events	General Testing
12. ORBIT [64]	(iv) Model-Based (Pre-Runtime Analysis)	All Pages	”	UI Events	General Testing
13. Brahmastra [15]	(iv) Model-Based (Pre-Runtime Analysis)	Distinct Pages	”	UI Events	Ad Checking

Table A.3: **Comparison Table for Privacy Notification Systems**

System	Notification Content	Notification Interface	Frequency	Action	Timing	Granularity
1. Aurasium [61]	Real-time privacy data access; Network IP details; Privilege Escalation	Dialog Box	Every access instance; Set Rule	Blocking	Before event	Data level
2. APEX [44]	Permission data control at install time.	Install-time List; Dialog Box	Every install instance; Every access instance	Blocking	After event	App/Data level
3. AppIntent [65]	Sequence of screen captures of app operation with notice appearing on privacy data sending. (Meant for analyst to check)	Android Toast Message	Every access instance	Non-Blocking	After event	Data level
4. AppOps [6]	Real-time privacy data access; Summarized access control of privacy data	Privacy Manager List; Dialog Box	First access instance; Configurable	Blocking	Before event	App/Data level
5. ProtectMyPrivacy (PMP) [48]	Real-time privacy data access; Summarized access control of privacy data	Privacy Manager List; Android Notifications	First access instance; Configurable	Blocking	Before event	App/Data level
6. LBE Privacy Guard [29]	Real-time privacy data access; Summarized access control of privacy data	Privacy Manager List; Android Notifications	First access instance; Configurable	Blocking	Before event	App/Data Level
7. Permission Tracker [38]	Provide 3 different levels of details with regards to data permission: (i) App categories, (ii) Permission categories, (iii) Frequency of permissions.	Privacy Manager List; Dialog Box	Every access instance; Configurable	Blocking	Before event	Data level

System	Notification Content	Notification Interface	Frequency	Action	Timing	Granularity
8. Mock-Droid [14]	Real-time privacy data access; Summarized access control of privacy data	Privacy Manager List; Android Notifications	Every access instance; Set Rule	Blocking	Before event	Data level
9. Privacy Nudging [5]	Informs user on frequency of privacy data accesses and number of apps that accessed the data over a time period.	Privacy Manager List; Android Notifications	Periodic (default: once daily)	Non-Blocking	After many events	App/Data Level
10. Privacy Mod [55]	Background notice informs user that app is using data. Informs user on frequency of privacy data accesses.	Privacy Manager List; Android Toast Message	Every access instance	Non-Blocking	After event	Data level
11. ReCon [52]	Informs user on frequency of personally identifiable information (PII) and number of apps that accessed the data over a time period.	Summarized UI	-	Non-Blocking	After event	Data level
12. Taint-Droid [24]	Real-time privacy data access	Android Notifications	Every leak instance	Non-Blocking	After event	Data level

Figure A.1: Pre Study Survey Form



Types of smart phones used previously
Which smart phones have you used?

□ iPhone
□ Android
□ Blackberry
□ Windows Phone 7 or 8
□ Others

What Smartphone are you currently using?
Please indicate the platform of the Smartphone that you are currently using.

□ Android
□ iPhone (iOS)
□ Windows Phone 7/8
□ Other:

Proficiency level of Smart Phones
How proficient do you think you are in smart phone usage?

□ Novice
□ Average
□ Expert
□ I have never used smartphones

Privacy awareness
How privacy-conscious would you describe yourself to be when using mobile applications?

	1	2	3	4	5	6	7	
Not privacy conscious at all	○	○	○	○	○	○	○	Extremely privacy conscious

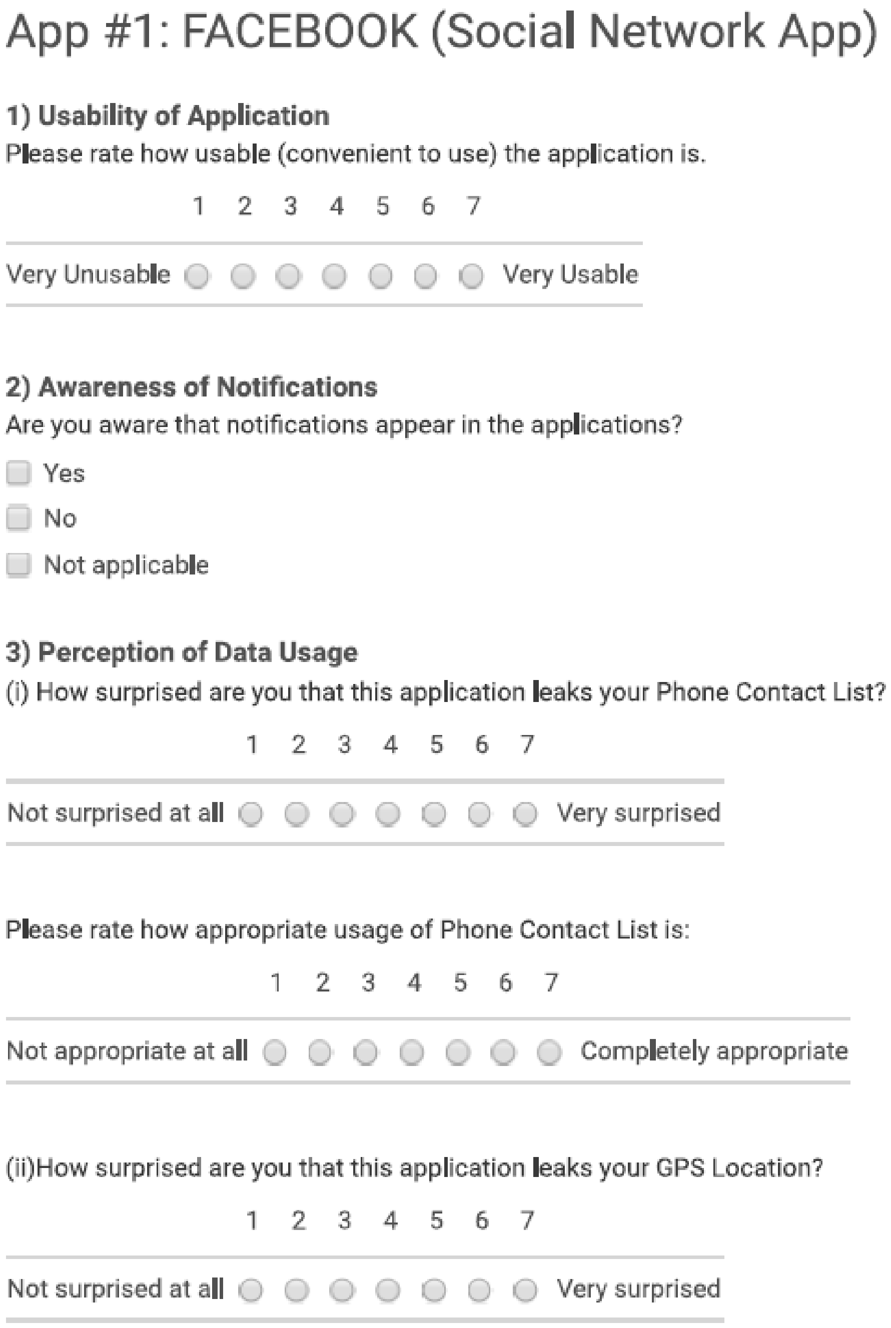
Most frequently used applications.
Please name some applications that you use frequently.

□ Daily
□ Every few days
□ Every week
□ Every month
□ Never used application before

Most frequently used applications. (Please name 5 applications that you use frequently.)
Frequently used application number 1
▼

If application not in list.

Figure A.2: Post Study Survey Form (Example: Facebook App)



App #1: FACEBOOK (Social Network App)

1) Usability of Application

Please rate how usable (convenient to use) the application is.

	1	2	3	4	5	6	7	
Very Unusable	○	○	○	○	○	○	○	Very Usable

2) Awareness of Notifications

Are you aware that notifications appear in the applications?

□ Yes

□ No

□ Not applicable

3) Perception of Data Usage

(i) How surprised are you that this application leaks your Phone Contact List?

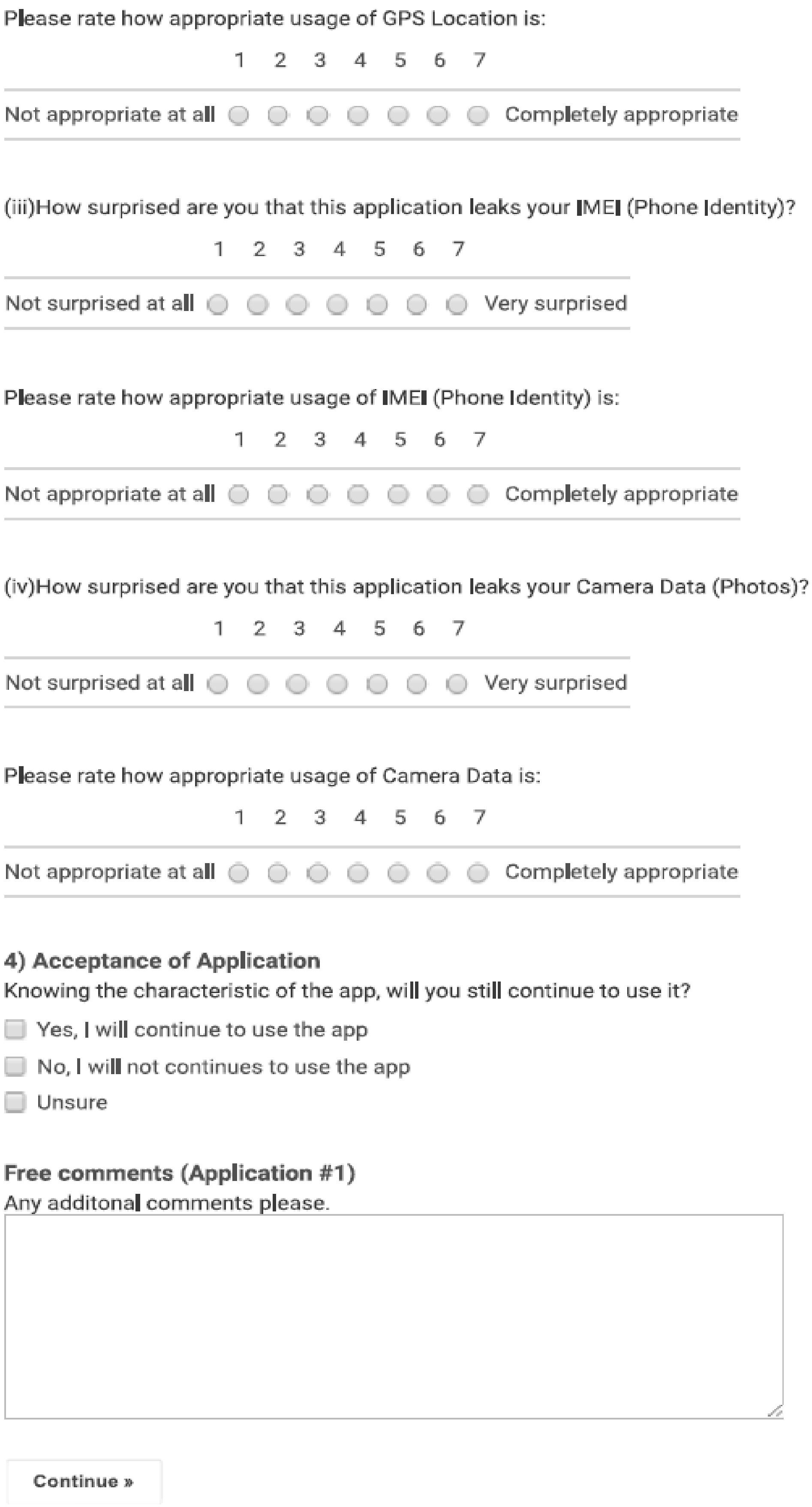
	1	2	3	4	5	6	7	
Not surprised at all	○	○	○	○	○	○	○	Very surprised

Please rate how appropriate usage of Phone Contact List is:

	1	2	3	4	5	6	7	
Not appropriate at all	○	○	○	○	○	○	○	Completely appropriate

(ii)How surprised are you that this application leaks your GPS Location?

	1	2	3	4	5	6	7	
Not surprised at all	○	○	○	○	○	○	○	Very surprised



Please rate how appropriate usage of GPS Location is:

	1	2	3	4	5	6	7	
Not appropriate at all	○	○	○	○	○	○	○	Completely appropriate

(iii)How surprised are you that this application leaks your IMEI (Phone Identity)?

	1	2	3	4	5	6	7	
Not surprised at all	○	○	○	○	○	○	○	Very surprised

Please rate how appropriate usage of IMEI (Phone Identity) is:

	1	2	3	4	5	6	7	
Not appropriate at all	○	○	○	○	○	○	○	Completely appropriate

(iv)How surprised are you that this application leaks your Camera Data (Photos)?

	1	2	3	4	5	6	7	
Not surprised at all	○	○	○	○	○	○	○	Very surprised

Please rate how appropriate usage of Camera Data is:

	1	2	3	4	5	6	7	
Not appropriate at all	○	○	○	○	○	○	○	Completely appropriate

4) Acceptance of Application

Knowing the characteristic of the app, will you still continue to use it?

□ Yes, I will continue to use the app

□ No, I will not continues to use the app

□ Unsure

Free comments (Application #1)

Any additonal comments please.