Non-Essential Communication in Mobile Applications

ABSTRACT

Abstract
Abstract
Abstract
Abstract

1. INTRODUCTION

Mobile applications enjoy almost permanent connectivity and ability to exchange information with their own backends, third-party servers and other applications installed on the same device. During that information exchange, applications often release sensitive information about their users, such as location, phone number or unique device id [6, 4, 16]. With privacy being an increased concern, multiple works propose approaches for tracking the release of sensitive information [6, 4, 2, 16, 7], and, further, investigate feasibility of obfuscating or blocking it completely [9].

Yet, users often deliberately trade their privacy for receiving a desired service from an application, e.g., finding friends within a certain radius from their current location. In such cases, preventing the release of the user's location would render an application privacy-preserving but effectively useless. Existing approaches provide no means for differentiating between essential and non-essential releases of sensitive information: they flag each identified case as a potential privacy breach, regardless of its designated use. The non-trivial task of classifying the *purpose* of information release is then left to the user. Permission systems of contemporary mobile platforms also do not contribute in making such distinction as they only require applications to declare the *type* of information they want to access, and have no means to capture the "semantics" of its use. Thus, once an application gains access to the user's sensitive information, it can, in addition to utilizing the information for the purpose assumed by the user, also release it to unauthorized third parties.

This paper aims at distinguishing between these two types of information releases – those that are expected and are essential for the desired application functionality and those that are not. Specifically, we focus on identifying communication that can be eliminated without any noticeable impact on the application's functionality. Our in-depth investigation of ten top-popular Android applications from the Google Play stores shows that such communication is surprisingly common. In fact, we noticed that some applications spawn services that communicate with third-party servers even when the application itself is not active and longer and not visible in the list of active applications.

We thus propose a static application analysis technique for identifying statements that can be deemed *non-essential*: their failure are silently ignored by the applications without interfering the applications' behavior.

Communication in Android Applications. We conduct an empirical study whose goal is to explore and quantify the amount of non-essential connections performed by Android applications. We focus on the three most common connection types: HTTP, socket and RPC. The first two are used to communicate with various backend servers – the application's own and third parties'; the last one is used to communicate with other applications and services running on the same device.

Our study is dynamic in nature and is performed in three phases. In the first phase, we establish a baseline application behavior. Similarly to the approach in [9], we record a script triggering the application functionality via a series of interactions with the application's user interface. After each interaction, we capture a screenshot of the device to record the application state.

In the second phase, we instrument the application to log information about triggered connection statements. The instrumented apk is then installed and executed on a mobile device using the recorded script.

In the third phase, we investigate the impact of each triggered connection on the overall behavior of the application. Specifically, we iterate over all triggered connection, producing a new version of the application with each corresponding connection being disabled. Connection disabling is achieved by replacing the connection statement with the statement that indicates connection failure, e.g., that throws an exception that occurs when the connection fails due to the device being put in airplane mode. We then install the modified application and run it using the previously recorded script. The screenshots documenting the execution of the modified application are compared to those of the original one. We consider executions as equivalent if they result in screenshots that differ only in the content of advertisement information, messages in social network applications such as twitter, and the devices status bar. We also separately note connections that contribute to presenting the advertisement content, if the analyzed application has any.

Our analysis reveals that around 65% of the connection statements exercised by the applications do not lead to any noticeable effect on the observable application functionality. Slightly more than 25% of these correspond to HTTP and socket communication. The

rest correspond to RPC calls to internal services installed on the device: notably, but not exclusively, Google advertising and analytics, which further communicate with external services. Moreover, in applications that present advertisement material, about 60% of the connections that do affect the observable application behavior are used for the advertising purposes only.

A manual byte code inspection of the identified "hidden" connections shows that failures are often either silently ignored by the application, e.g., with an empty exception *catch* block, or written to a log file without being propagated to the user. This behavior is indicative for the connections being *non-essential* for the application behavior and, in some cases, even harmful. In fact, the only application out of ten we analyzed that leaked the unique device id to the internet did that via such a hidden connection; identifying and blocking the connection eliminated the information leakage without affecting the application behavior and other information exchange operations that the application performed.

Detecting Non-Essential Connections. Inspired by the findings, we devise a static application analysis technique for detecting cases when connection failures are "silently" ignored by the application, i.e., when information about a connection failure is not propagated back to the end user. Our technique searches for cases when exceptions resulting from connection attempts are caught by an application without giving any visual message to the user. **TDB: designed to be conservative etc.**

We evaluate the technique on the "truth set" produced manually during the above empirical analysis and show that it is able to identify xx% of the "silent" information releases (xxx cases for all analyzed applications), introducing only yy false-positive and zz false-negative results.

Applying the analysis on additional zz top-popular applications from Google Play reveals that ww% of connection sites established by these applications can be deemed unnecessary. **TBD**

Significance of the Work. Our work focuses on benign mobile applications that can be downloaded from the popular application stores and that are installed by thousands, if not millions, of users. By identifying and highlighting application functionality hidden from the user, it aims at improving transparency and, ultimately, quality of these applications.

We are largely inspired by the positive trend of improvement that we recently observed in the application development community. One of its manifestations is the increased awareness of privacy considerations. It is likely that such awareness was affected by multiple studies that highlighted potential privacy violation in applications from popular application stores [6, 4, 16]. Indeed, we observed that newer versions of the analyzed applications in many cases no longer exhibit the previously identified privacy breaches. This improvement trend reassures and further emphasizes the significance of this work as an additional step towards achieving better application quality and increasing trust between users and application developers.

Add research questions, gradual functionality disabling, waste of bandwidth argument, walmart example. Targets are not indicative, AppFence case.

We show that most malware writers basically copy/paste code from fellow developer code and from public tutorials/samples from the Web [1]

RQ1: What % of connections is non essential? RQ2: How many can be found? RQ3: What types of non-essential communication exists?

Contributions. The paper makes the following contributions:

Table 1: Considered Connection Statements.

	Class or Interface	Method	Indication of Failure
1.	java.net.URL	openConnection	java.io.IOException
2.	java.net.URLConnection	connect	java.io.IOException
3.	org.apache.http.client.HttpClient	execute	java.io.IOException
4.	java.net.Socket	getOutputStream	java.io.IOException
5.	android.os.IBinder	transact	android.os.RemoteException

1. It sets a new problem of distinguishing between essential and non-essential release of information by mobile applications in an automated manner. This problem is orthogonal and complementary to that of identifying sensitive information flow, which was the focus of numerous earlier works. Highlighting non-essential information releases in existing mobile applications is expected to improve transparency and contribute to the overall quality of the field.

2. It proposes a dynamic approach for detecting non-essential releases of information in Android applications which does not require access to the application source code. The approach relies on interactive injection of connection failures and identification of cases in which the injected failures do not effect the observable application functionality.

3. It provides empirical evidence for the prevalence of such non-essential connections in real-life applications. Specifically, it shows that 90% of the connections attempted by ten top-popular free applications on Google Play fall into that category.

4. It proposes a static technique that operates on application binaries and identifies non-essential connections – those where failures are not propagated back to the application's user. The precision and recall of the technique is xx and yy, respectively, when evaluated against the empirically established truth set. When applied on the 50 top-popular free applications on Google Play, the technique is able to identify xx% of connections made by these applications as non-essential.

The remainder of the paper is structured as follows. Section 2 describes the empirical study we conducted for gaining insights into the nature of information releases in mobile applications. Section 3 presents the static analysis technique designed for identifying non-essential information releases. Section 4 discusses results of its evaluation on real-live examples. Section 5 discusses the limitation of our work. Section 6 presents the related work, while Section 7 concludes the paper and discusses future work.

2. COMMUNICATION IN ANDROID APPLICATIONS

In this section, we first describe the design of the study that we conducted to gain more insights into the nature of communication performed by Android applications. We then discuss the study results.

2.1 Design of the Study

Connection Statements. The list of the connection statements that we considered in our study is given in Table 1. The first three are responsible for establishing HTTP connections with backend servers, the forth one provides socket-based communication and the last one allows RPC communication with other applications and services installed on the same mobile device.

Column 4 of the table lists exceptions indicating connection failures that occur when the desired server is unavailable, or when a device is put in the disconnected or airplane mode. When investigating the significance of a connection on the overall behavior of an analyzed application, we inject connection failures by replacing connection statements with statements that throw exceptions of the appropriate type. This approach was chosen as it leverages the applications' native mechanism for dealing with failures, thus reducing side-effects introduced by our instrumentation to a minimum.

Application Instrumentation. As input to our study, we assume an Android application given as an apk file. We use the dex2jar tool suite[1] to extract the jar file from the apk. We then use the asm framework[2] to implement two types of transformations:

1. *A monitoring transformation* which produces a version of the original application that logs all executions of the connection statements in Table 1.

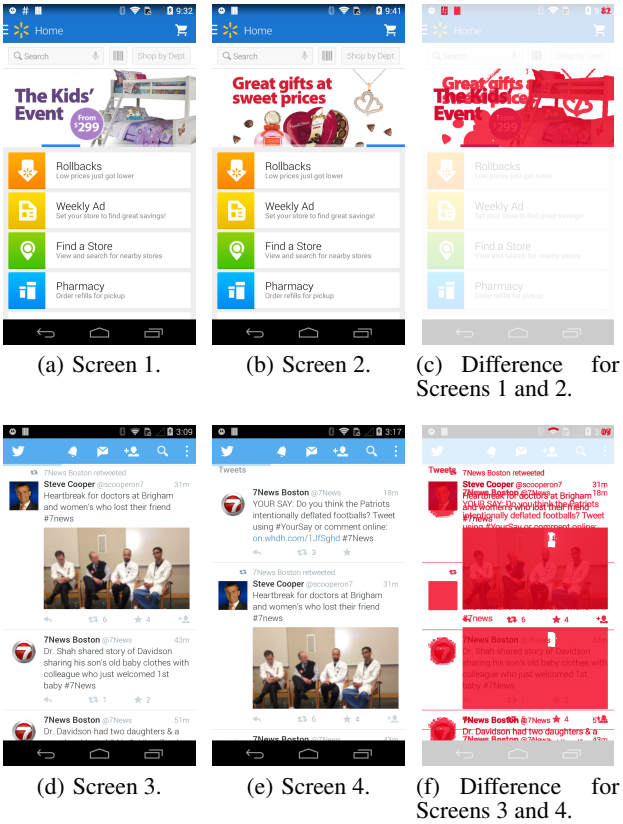
2. *A blocking transformation* which obtains as additional input a configuration file that specifies the list of connection statements to disable. It then produces a version of the original application in which the specified connection statements are replaced by statements that throw exceptions of the corresponding type, as specified in Table 1.

The jar file of the transformed application is then converted back to apk using the dex2jar tool suite and signed with the jarsigner tool distributed with the standard Java JDK.

Automated Application Execution and Comparison. Comparison of user-observable behavior requires dynamic execution of the analyzed applications. The main obstacle in performing such comparison is the ability to reproduce program executions in a repeatable manner. To overcome this obstacle, we produce a script that automates the execution of each application. As the first step, we use the Android getevent tool[3] that runs on the device and captures all user and kernel input events to capture a sequence of events that exercise an application behavior. We make sure to pause between user gestures that assume application response. We then enhance the script produced by getevent to insert a screen capturing command after each pause and also between events of any prolonged sequences. We upload the produced script onto the device and run it for each version of the application.

We deliberately opt not to use Android's UI/Application Exerciser Monkey[4] tool. While this tool is able to generates a repeatable sequence of pseudo-random streams of events such as clicks, touches and gestures, in our case, it was unable to provide a reasonably exhaustive coverage of application functionality. Even for applications that do not require entering any login credentials, it quickly locked itself out of the application by generating gestures that the analyzed application cannot handle. We thus have chosen to manually record the desired application execution scenario, which also included any "semantic" user input required by the application, e.g., username and password.

For the comparison of application executions, we started by following the approach in [9], where screenshots from two different runs are placed side-by-side, along with a visual diff of each two corresponding images, as shown in Figure 1, for the walmart and twitter applications. We used the ImageMagick compare tool[5] to



(a) Screen 1. (b) Screen 2. (c) Difference for Screens 1 and 2.

(d) Screen 3. (e) Screen 4. (f) Difference for Screens 3 and 4.

Figure 1: Visual differences.

produce the visual diff images automatically. We then manually scanned the produced output while ignoring differences in content of advertisement messages and the status of the device, deeming screenshots in Figures 1(a) and (b) similar. We also ignored the exact content of widgets that are populated by applications in a dynamic manner and are designed to provide continuously updated information that is expected to differ between applications runs, such as tweets in Figures 1(d) and (e). These two figures are thus also deemed similar.

In one out of ten analyzed cases, we had to revert to manual execution and comparison of the application runs. That case involved interactions with a visual game that required rapid response time, thus the automated application execution was unable to provide reliable results.

Execution Methodology. We performed our study in three phases. In the first phase, we installed the original version of each analyzed application on a Nexus 4 mobile device running Android version 4.4.4. We manually exercised the application, recording the execution script that captured all triggered actions, as described above. We then re-installed the application to recreate a "clean" initial state and ran the produced execution script. We used screenshots collected during this run as the baseline for further comparisons.

In the second phase, we used the Monitoring Transformation to produce a version of the original application that logs information about all existing and triggered connection statements. We ran the produced version using the execution script and collected the statistics about its communication patterns.

In the third phase, we iterated over all *triggered* connection statements, disabling them one by one, in order to assess the necessity

[1]https://code.google.com/p/dex2jar/
[2]http://asm.ow2.org/
[3]https://source.android.com/devices/input/getevent.html
[4]http://developer.android.com/tools/help/monkey.html
[5]http://www.imagemagick.org/script/compare.php

of each connection for preserving the user-observable behavior of the application. That is, we arranged all triggered connection statements in a list, in a lexical order and then applied the Blocking Transformation to disable the first connection statement in the list. We ran the produced version of the application using the recorded execution script and compared the obtained screenshots to the baseline application execution. If disabling the connection statement did not affect the behavior of the application, we marked it as *non-essential*, kept it disabled for the subsequent iterations and proceed to the next connection in the list. Otherwise, we marked the exercised connection as *essential* and kept it enabled in the subsequent iterations. We continued with this process until all connections in the list were explored.

In all iterations, we also disabled all connections that were classified as non-triggered during the second phase. That was done to improve the accuracy of our analysis by proactively preventing applications from taking a new, previously unexplored, path when they detect connection failures. As the final quality measure, we manually introspected the execution of the version in which all non-essential connections were blocked, to detect any possible issues missed by the automated analysis.

Subjects. As the subjects of our study, we downloaded 15 top-popular applications available on the Google Play store on November 2014. We excluded from this list three chat applications, as our evaluation methodology does not allow assessing the usability of a chat application without a predictably available chat partner. We also excluded two applications whose asm-based instrumentation failed, most probably become they use language constructs that are not supported by that framework.

The remaining ten applications are listed in the first column of Table 2. The second and the third columns list the size of each application and its corresponding the instrumentation time, averaged for multiple performed runs. These measures were taken on a computer running Mac OS X Version 10.9.5, with 16GB of memory and a 3 GHz Intel Core i7 processor.

We did not extend our dynamic analysis beyond these ten applications because the inspection of our findings indicated that we reached saturation: while it is clearly unfeasible to explore all possible scenarios, we observed similar trends in all analyzed applications. As such, inclusion of additional ones was not expected to provide substantially new insights.

2.2 Results

The quantitative results of the study are presented in columns 4–8 of Table 2. The forth and the fifth columns of the table show that only a small number of connection statements encoded in the applications are, in fact, triggered dynamically. While some of the non-triggered statements can correspond to execution paths that were not explored during our dynamic application traversal, the vast majority of the statements originate in third-party libraries included in the application but only partially used, e.g., various Google services for mobile developers, advertising and analytics libraries and more. In fact, we identified nine different advertising and analytics libraries used by the ten applications that we analyzed, and many times a single applications uses multiple such libraries.

An interesting case is the facebook application (row 3 in Table 2), where most of the application code is dynamically loaded at runtime from resources shipped within the apk file. Our analysis was unable to traverse this dynamically loaded code, and we thus excluded the application from the further analysis, noting that the only three connection statements that existed in the application jar file are never triggered.

Table 3: Communication Types.

	HTTP and Socket	RPC
Triggered	35 (30.7%)	79 (69.3%)
Non-essential (total)	18 (25.5%)	53 (74.6%)
Non-essential (Google and Known A&A Services)	8 (17.7%)	37 (82.2%)

Classification of the Triggered Statements. Column 6 of Table 2 shows the number of connection statements that we determined as non-essential during our study. Averaged for all applications, around 65% of the connections fall in that category. This means that only 35% of the connection statements triggered by an application affect its observable behavior, when executed for the exact same scenario with the connection being either enabled or disabled.

Four of the analyzed applications contained advertisement material. For these applications, 71% of the connections deemed essential were used for advertising purposes, as shown in the last column of Table 2.

Table 3 shows the distribution of the triggered connection statements into external communication performed via HTTP and sockets, and internal RPC communication. Overall, 30% of all triggered connection statements correspond to external communication while 70% – to internal one, as shown in the second row of the table. The breakdown is similar for the connection statements that we deemed non-essential: slightly more than 25% correspond to external communication and the remainder – to the internal communication with services installed on the same device, as shown in the third row of Table 3.

The last row of the table present statistic considering the communication with known advertisement and analytic services. The table shows that almost 18% of the non-essential connections used for these purpose flow to the external services and 82% – to internal ones, which further communicate with external services to deliver the required content. Google services are commonly, but not exclusively, used by numerous applications.

Lessons Learned. The collected statistics show that no principle distinction between essential and non-essential connections can be made just by considering connection types and their destinations. That observation is consistent with findings in [9], where authors show that blocking all messages to advertising and analytics services made more than 60% of the applications either less functional or completely dysfunctional. We conclude that a more sophisticated techniques for identifying the non-essential communication performed by the applications is required.

We manually investigated binaries of several analyzed applications, to gain more insights into the way applications treat non-necessary connections of each of the identified type and communication target. We noticed that, in a large number of cases, connection failures are silently ignored by the applications without producing any visual indication to the user. That is, the exception triggered by the connection failure of a non-essential connection is either caught and ignored locally in the method that issues the connection or, more commonly, propagated upwards in the call stack and then ignored by one of the calling methods.

In several cases, an error or warring message is written to the Android log file. However, this file is mostly used by application developers and is rarely accessed by the end-user.

As the result of the study, we conjecture that non-essential connections can be detected by inspecting connection failure paths. The lack of updates to GUI elements on the failure path is indicative for a connection being silently ignored by the application, thus

Table 2: Analyzed Applications.

Applications	jar size (MB)	Instrumentation time (ms)	Total # of connection statements	# of triggered statements	# of non-essentials (% of trig.)	# of essentials (% of trig.)	# of ads (% of essentials)
air.com.sgn.cookiejam.gp	2.7	4,321	639	7	3 (42.9%)	4 (57.1%)	-
com.crimsonpine.stayinline	3.2	24,184	842	18	13 (72.2%)	5 (27.8%)	5 (100.0%)
com.facebook.katana	0.6	1,192	3	0	-	-	-
com.grillgames.guitarrockhero	6.2	44,381	882	35	30 (85.7%)	5 (14.3%)	5 (100.0%)
com.king.candycrushsaga	2.6	19,573	638	4	3 (75.0%)	1 (25.0%)	-
com.pandora.android	5.7	38,479	532	13	9 (69.2%)	4 (30.8%)	1 (25.0%)
com.spotify.music	5.4	39,067	181	28	21 (75.0%)	7 (25.0%)	-
com.twitter.android	5.9	8,043	314	13	6 (46.2%)	7 (53.8%)	-
com.walmart.android	5.8	72,351	446	10	5 (50.0%)	5 (50.0%)	-
net.zedge.android	6.5	42,889	871	21	16 (76.2%)	5 (23.8%)	3 (60.0%)
Totals (average)	4.5	29,448	535	14.9	11.8 (65.8%)	4.8 (34.2%)	3.5 (71.3%)

Table 6: Insert Caption here.

	Non-Essential	Essential	Unknown	Exec. time
Total (xx apps)	x	x	x	x
Avg. per app	x	x	x	x

being non-essential for the application execution.

3. DETECTING NON-ESSENTIAL CONNECTIONS

4. EXPERIMENTS

1) Comparison with the manually established results

As shown in Table 5, there are only two mis-classified non-essentials: in spotify and walmart. A manual inspection shows that connection failures are indeed ignored and the application proceed without the missing information: album images in the first case and location-based store selection in the second.

2) Run on 100 apps and report the number of findings, execution time, most common non-essential APIs(?)

1) Total number of all non-essential connections in all apps 2) Avg. number of non-essential connections per app 3) Total number of all essential connections in all apps 4) Avg. number of essential connections per app 5) Total number of all unknown connections in all apps 6) Avg. number of unknown connections per app 7) Top 20 callers in non-essential connections, calculated as follows: a number of occurrences of each non-essential connection in all apps is calculated, and for the top 20 most frequent results, the callees are reported (the "from" part of the reported line)

5. LIMITATIONS AND THREATS TO VALIDITY

Limitation of dynamic analysis – it only considers the executed passes. This is how we compared. Several operations in the same connection statement. Cannot handle cases when two blocked points depend on each other.

Limited number of subjects.

6. RELATED WORK

Most close to our are works that employ user-centric analysis to identify spurious application behaviors and that investigate information propagation in mobile applications. From a technical point of view, our work relies on static application analysis for Android and on exception analysis for Java. We thus discuss the related work along these four dimensions.

User-Centric Analysis for Identifying Spurious Behaviors in Mobile Applications. Huang et al. [10] propose a technique, AsDroid, for identifying contradictions between a user interaction function and the behavior that it performs. This technique associates intents with certain sensitive APIs, such as HTTP access or SMS send operations, and tracks the propagation of these intents through the application call graph, thus establishing correspondence between APIs and the UI elements they affect. It then uses the established correspondence to compare intents with the text related to the UI elements. Mismatches are treated as potentially stealthy behaviors. In our work, we do not assume that all operations are triggered by the UI and do not rely on textual descriptions of UI elements.

CHABADA [8] compares natural language descriptions of applications, clusters them by description topics, and then identifies outliers by observing API usage within each cluster. Essentially, this system identifies applications whose behavior would be unexpected given their description. Instead, our approach focuses on identifying unexpected behaviors given the actual user experience, not just the description of the application.

Elish et al. [5] propose an approach for identifying malware by tracking dependencies between the definition and the use of user-generated data. They deem sensitive function calls that are not triggered by a user gesture as malicious. However, in our experience, the absence of a data dependency between a user gesture and a sensitive call is not always indicative for suspicious behavior: applications such as twitter and walmart can initiate HTTP calls to show the most up-to-date information to their user, without any explicit user request. Moreover, malicious behaviors can be performed as a side-effect of any user-triggered operation. We thus take an inverse approach, focusing on identifying operations that do not affect the user experience.

Information Propagation in Mobile Applications. The most prominent technique for dynamic information propagation tracking in Android is TaintDroid [6], which detects flows of information from a selected set of sensitive sources to a set of sensitive sinks. TaintDroid is used, extended and customized by several follow-up research projects. For example, the Kynoid system [14] extends it with user-defined security policies, which include temporal constraints on data processing as well as restrictions on destinations to which data is released. Several static information flow analysis techniques for tracking propagation of information from sensitive sources to sinks have also been recently developed [2, 7, 11, 12]. The first two ensure accurate detection of information flows within

Table 4: Considered UI Elements.

	Class or Interface	Methods
1.	android.view.ViewGroup	addView, addFocusables, addTouchables, addChildrenForAccessibility
2.	android.widget.TextSwitcher	addView
3.	android.widget.ViewSwitcher	addView
4.	android.view.WindowManagerImpl	addView
5.	android.view.WindowManagerImpl$CompatModeWrapper	addView
6.	android.view.ViewManager	addView, updateViewLayout
7.	android.app.Dialog	setContentView
8.	android.support.v7.app.ActionBarActivityDelegate	setContentView
9.	android.support.v7.app.ActionBarActivityDelegateBase	setContentView
10.	android.support.v7.app.ActionBarActivityDelegateICS	setContentView
11.	android.webkit.WebView	loadData, loadDataWithBaseURL, loadUrl
12.	android.view.View	onLayout, layout, onDraw, onAttachedToWindow
13.	android.widget.ImageView	onLayout, layout, onDraw, onAttachedToWindow
14.	android.inputmethodservice.KeyboardView	onLayout, layout, onDraw, onAttachedToWindow
15.	android.widget.AnalogClock	onLayout, layout, onDraw, onAttachedToWindow
16.	android.widget.TextView	onLayout, layout, onDraw, onAttachedToWindow, append, setText
17.	android.widget.Toast	makeText

Table 5: Comparison with the Manually Established Results.

Applications	Correctly detected non-essential		Correctly detected non-essential (counting ads as non-essential)	
	Precision	Recall	Precision	Recall
air.com.sgn.cookiejam.gp	1/1 (100.0%)	1/3 (33.3%)	1/1 (100.0%)	1/3 (33.3%)
com.crimsonpine.stayinline	9/14 (64.3%)	9/13 (69.2%)	14/14 (100.0%)	14/18 (77.8%)
com.grillgames.guitarrockhero	15/20 (75.0%)	15/30 (50.0%)	20/20 (100.0%)	20/35 (57.1%)
com.king.candycrushsaga	1/1 (100.0%)	1/3 (33.3%)	1/1 (100.0%)	1/3 (33.3%)
com.pandora.android	0/0	0/9 (0%)	0/0	0/9 (0%)
com.spotify.music	2/3 (66.7%)	2/21 (9.5%)	2/3 (66.7%)	2/21 (9.5%)
com.twitter.android	3/3 (100.0%)	3/6 (50.0%)	3/3 (100.0%)	3/6 (50.0%)
com.walmart.android	3/4 (75.0%)	3/5 (60.0%)	3/4 (75.0%)	3/5 (60.0%)
net.zedge.android	13/16 (81.3%)	13/16 (81.3%)	16/16 (100.0%)	16/19 (84.2%)
Totals (average)	82.8%	43.0%	92.7%	45.0%

a single application, while the last two – across multiple applications.

McCamant and Ernst [13] take a quantitative approach to information flow: they cast information-flow security to a network-flow-capacity problem and describe a dynamic technique for measuring the amount of secret data that leaks to public observers. Tripp and Rubin [16] propose to extend the information flow analysis with a Bayesian notion of statistical classification, which conditions the judgment whether a release point is legitimate on the evidence arising at that point, e.g., the similarity between the data values about to be released and the data obtained via the source APIs.

Our work is orthogonal and complimentary to all the above: while they focus on providing precise information flow tracking capabilities and detecting cases when sensitive information flows outside of the application and/or mobile device, our focus is on distinguishing between essential and non-essential flows.

The authors of AppFence [9] build up on TaintDroid and explore approaches for either obfuscating or completely blocking the identified cases of sensitive information release. Their study shows that blocking all such cases renders more than 65% of the application either less functional or completely dysfunctional, blocking cases when information flows to advertisement and analytics services "hurts" 10% of the applications, and blocking the communication with the advertisement and analytics services altogether – more than 60% of the applications. Our work has a complementary nature as we rather attempt to identify cases when communication can be disabled without affecting the application functionality. Our approach for assessing the user-observable effect of that operation is similar to the one they used though.

Both MudFlow [3] and AppContext [17] build up on the FlowDroid static information flow analysis system [2] and propose approaches for detecting malicious applications by learning "normal" application behavior patterns and then identifying outliers. The first work considers flows of information between sensitive sources and sinks, while the second – contexts, i.e., the events and conditions, that cause the security-sensitive behaviors to occur. Our work has a complementary nature as we focus on identifying non-essential rather than malicious behaviors, aiming to preserve the overall user experience.

Shen et al. [15] contribute FlowPermissions – an approach that extends the Android permission model with a mechanism for allowing the users to examine and grant permissions per an information flow within an application, e.g., a permission to read the phone number and send it over the network or to another application already installed on the device. While our approaches have a similar ultimate goal – to provide visibility into the holistic behavior of the applications installed on a user's phone – our techniques are entirely orthogonal.

Static Application Analysis for Android. Zhang et al. [18] contribute a reflection-aware call graph construction algorithm for mul-

tithreaded GUI applications. They instantiate it for four popular Java GUI frameworks, including Android. **TBD.**

Exception Analysis for Java.

7. CONCLUSIONS

8. REFERENCES

[1] K. Allix, Q. Jérome, T. F. Bissyandé, J. Klein, R. State, and Y. L. Traon. A Forensic Analysis of Android Malware - How is Malware Written and How it Could Be Detected? In *Proc. of the IEEE 38th Annual Computer Software and Applications Conference (COMPSAC'14)*, 2014.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, 2014.

[3] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. Mining Apps for Abnormal Usage of Sensitive Data. In *Proc. of the 37th International Conference on Software Engineering (ICSE'15) (to appear)*, 2015.

[4] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proc. of the Network and Distributed System Security Symposium (NDSS'11)*, 2011.

[5] K. O. Elish, D. D. Yao, and B. G. Ryder. User-Centric Dependence Analysis for Identifying Malicious Mobile Apps. In *Proc. of IEEE Mobile Security Technologies Workshop (MoST'12)*, 2012.

[6] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*, pages 1–6, 2010.

[7] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information Flow Analysis of Android Applications in DroidSafe. In *Proc. of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[8] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking App Behavior against App Descriptions. In *Proc. of the 36th International Conference on Software Engineering (ICSE'14)*, 2014.

[9] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These Aren't the Droids You're Looking for: Retrofitting Android to Protect Data from Imperious Applications. In *Proc. of the 18th ACM Conference on Computer and Communications Security (CCS'11)*, pages 639–652, 2011.

[10] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang. AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction. In *Proc. of the 36th International Conference on Software Engineering (ICSE'14)*, 2014.

[11] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. Android Taint Flow Analysis for App Sets. In *Proc. of the 3rd ACM SIGPLAN International Workshop on the State Of the Art in Java Program analysis (SOAP'14)*, 2014.

[12] L. Li, A. Bartel, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. I Know What Leaked in Your Pocket: Uncovering Privacy Leaks on Android Apps with Static Taint Analysis. *arXiv Computing Research Repository (CoRR)*, abs/1404.7431, 2014.

[13] S. McCamant and M. D. Ernst. Quantitative Information Flow as Network Flow Capacity. In *Proc. of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI'08)*, 2008.

[14] D. Schreckling, J. Posegga, J. Köstler, and M. Schaff. Kynoid: Real-Time Enforcement of Fine-Grained, User-Defined, and Data-Centric Security Policies for Android. In *Proc. of the 6th IFIP WG 11.2 International Workshop on Information Security Theory and Practice (WISTP'12)*, 2012.

[15] F. Shen, N. Vishnubhotla, C. Todarka, M. Arora, B. Dhandapani, E. J. Lehner, S. Y. Ko, and L. Ziarek. Information Flows as a Permission Mechanism. In *Proc. of the ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*, 2014.

[16] O. Tripp and J. Rubin. A Bayesian Approach to Privacy Enforcement in Smartphones. In *Proc. of the 23rd USENIX Conference on Security Symposium (SEC'14)*, pages 175–190, 2014.

[17] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. AppContext: Differentiating Malicious and Benign Mobile App Behavior Under Contexts. In *Proc. of the 37th International Conference on Software Engineering (ICSE'15) (to appear)*, 2015.

[18] S. Zhang, H. Lü, and M. D. Ernst. Finding Errors in Multithreaded GUI Applications. In *Proc. of the 2012 International Symposium on Software Testing and Analysis (ISSTA'12)*, 2012.