Covert Communication in Mobile Applications

Julia Rubin*, Michael I. Gordon*, Nguyen Nguyen[§] and Martin Rinard*
*Massachusetts Institute of Technology, USA, [§]Global InfoTek, Inc, USA
mjulia@mit.edu, mgordon@mit.edu, nguyen@uwinsoftware.com, rinard@mit.edu

***Abstract*—This paper studies communication patterns in mobile applications. Our analysis shows that 63% of the external communication made by top-popular free Android applications from Google Play has no effect on the user-observable application functionality. To detect such covert communication in an efficient manner, we propose a highly precise and scalable static analysis technique: it achieves 93% precision and 61% recall compared to the empirically determined "ground truth", and runs in a matter of a few minutes. Furthermore, according to human evaluators, in 42 out of 47 cases, disabling connections deemed covert by our analysis leaves the delivered application experience either completely intact or with only insignificant interference. We conclude that our technique is effective for identifying and disabling covert communication. We then use it to investigate communication patterns in the 500 top-popular applications from Google Play.**

I. Introduction

Mobile applications enjoy almost permanent connectivity and the ability to exchange information with their own backend and other third-party servers. This paper shows that much of this communication does not deliver any tangible value to the application's user: disabling it leaves the delivered application experience completely intact. Yet, this covert communication comes with costs such as potential privacy breaches, bandwidth charges, power consumption on the device, and the unsuspected presence of continued communication between the device and remote organizations. In fact, we observed that several popular applications, e.g., Walmart and twitter, spawn services that covertly communicate with remote servers even when the application itself is inactive and the user is unaware that the spawned services are running in the background.

This paper focuses on distinguishing between *overt communication* that contributes to the application functionality anticipated by the user, and *covert communication* that is hidden and unexpected from the user's point of view. We start by analyzing communication patterns of popular applications in the Google Play app store. Motivated by the significant amount of covert communication we found in these applications, we develop a highly precise and scalable static analysis that can identify such communication automatically. We use our analysis to further investigate this unfortunate phenomenon and report on our findings. The following research questions drive this investigation:

RQ1: How frequent is covert communication? We conduct an empirical study that focuses on investigating the nature of communication in thirteen of the top twenty most-popular applications in Google Play (twitter, Walmart, Spotify, Pandora, etc.). For each connection statement triggered by these applications, we compare the dynamic execution of the original application to that of the application with the corresponding statement being disabled.

We consider all visual content delivered by the application, including advertisement, as essential and necessary, avoiding any subjective judgment on the relevance of that information to the user. That is, we consider application executions as equivalent if they differ only in contextual information, such as *content* of advertisement information and/or messages in social network applications, e.g., twitter.

Our study reveals that 63% of the *exercised* connection statements are covert – disabling them has no noticeable effect on the observable application functionality. Interestingly, less than half of such covert connections correspond to communication initiated from known advertisement and analytics (A&A) packages included in the application. Thus, looking at the package information only is not sufficient for distinguishing between overt and covert communication.

RQ2: Can covert communication be detected statically? Detailed investigation of multiple detected cases of covert connections inspired us to develop a novel static application analysis that can detect such connections automatically. The core idea behind our analysis is to look for cases when both connection success and failure are "silently" ignored by the application, i.e., when no information is presented to the user neither on success nor on failure of the connection.

For a connection statement, we analyze the portion of the program's control flow graph that corresponds to the *direct processing* of the connection. This includes *forward stack processing* and *failure handling* executions. The former is the non-exceptional execution reachable after the connection statement but limited to processing of the Android runtime events that trigger the connection statement. The latter traverses methods up all possible call stacks from the connection statement but stops at points at which the exception raised by the connection statement and all related rethrown exceptions are cleared.

The direct processing of the connection call is searched for method call invocations that could target a predefined set of API calls that affect the user interface. If such a call is found, the connection call statement is deemed overt. Furthermore, if the exception could be propagated back to the Android runtime (causing the application to exit), the connection call is deemed overt. Otherwise, it is deemed covert.

There are two reasons for the analysis to misclassify a connection as covert. First, it does not take into account user interface (UI) updates that occur outside of the connection's direct processing but are caused by the connection altering a local or a remote state. Second, the semantics of the direct processing itself is defined solely by the application's byte code, i.e., it does not consider native code and inter-application executions. Our experiments show that these cases are rare, which, together with the high scalability of the technique, justifies our design choices.

RQ3: How well does static detection perform? To assess the precision and recall of the static analysis technique, we evaluate it on the "truth set" established during the in-depth dynamic study described above. The results show that our technique features a high precision: 93% of the covert connection calls identified by the static analysis are indeed classified as covert during the dynamic study. Moreover, even though the technique is designed to be conservative and favor high precision over high recall, it is still able to identify 61% of all covert connections in the studied applications.

There are only two connections that were misclassified as covert, both due to the reasons mentioned above. The first one occurs because a UI update – in this case, presenting icons of additional apps that can be downloaded from Google Play – happens outside of the connection's direct processing. The second case, which is responsible for presenting advertisement material via the Google Ads component installed on the device, occurs because the direct processing of the connection does not extend to asynchronous RPC communication with Google services installed on the device.

To gain further insights on the quality of the technique, we apply it to additional 47 top-popular applications from Google Play. For these applications, we disable all connection statements deemed covert by the analysis. We then employ humans to perform a *usability assessment*: we provide them with two identical devices, one running the original and the other – the modified version of the application. We ask them to track and report on any observable differences between the runs of an application on two devices.

The results of this assessment are encouraging: there are no observable differences in 63.8% of the applications (30 cases). In 25.6% of the applications (12 cases), differences are related to absence of functionality considered minor by the users, e.g., ads or decorating images. Only 10.6% of the applications (5 cases) miss functional features considered essential by the users. This result implies that our static analysis produces actionable output that could already be applied to eliminate many cases of covert communication.

RQ4: How often does covert communication occur in real-life applications and what are its most common sources? We apply the analysis on the top 500 popular applications from Google Play. This experiment reveals that 46% of connection statements encoded in these applications are deemed covert. Most common sources of covert communication are Google services and various A&A services. Yet, these are not the exclusive source of covert communication, and not all communication made from these packages is covert.

Significance of the work. In [1], [2], [3], the authors state that "the system should continuously inform the user about what it is doing". Inspired by this principle, the goal of our work is to identify application functionality hidden from the user. This paper studies the extent of that problem in benign applications downloaded from popular application stores and installed by millions of users. We believe that our findings help focus the effort of improving transparency and trust in the mobile application domain.

Contributions. The paper makes the following contributions:

1) It sets *a new problem* of distinguishing between overt and covert communication in an automated manner.
2) It proposes *a semi-automated dynamic approach* for detecting covert communication in Android applications. The approach relies on interactive injection of connection failures in application binaries and identification of cases in which such injections do not affect the observable application functionality.
3) It provides *empirical evidence* for the prevalence of covert connections in real-life applications. Specifically, it shows that 63% of the connections attempted by thirteen top-popular free applications on Google Play fall into that category.
4) It proposes *a static technique* that operates on application binaries and identifies covert connections. The technique is highly scalable and precise: out of 47 highly popular applications on Google Play, 63.8% worked without any interference and further 25.6% worked with only insignificant interference when disabling all covert connections identified by the technique. The technique also features 93% precision and 61% recall when evaluated on the "truth set" established dynamically.
5) It provides *quantitative evidence* for the prevalence of covert connections in the 500 top-popular free applications on Google Play and identifies their common patterns.

II. Communication in Android

In this section, we 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.

A. Design of the Study

1) Connection Statements: Table I lists the base classes and their corresponding methods that we consider in our study. We also include all sub-classes of those listed in the table.

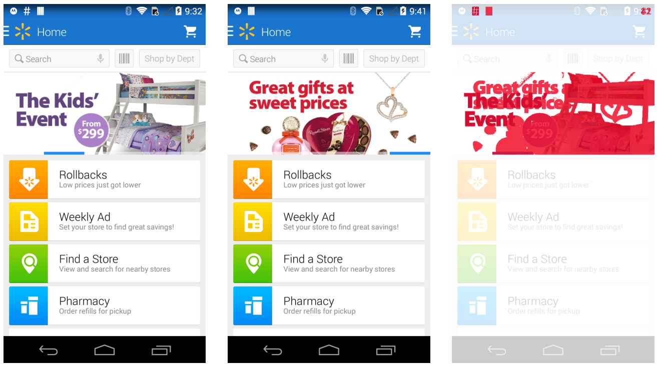
When a connection failure occurs, e.g., when the desired server is unavailable, or when a device is put in disconnected or airplane mode, each of these methods throws a `java.io.IOException` exception. Thus, for investigating the significance of a connection for the overall behavior of an analyzed application, we inject a connection failure by replacing the connection statement with a statement that throws such an exception. 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.

2) Application Instrumentation: As input to our study, we assume an Android application given as an apk file. We use the dex2jar tool suite [4] to extract the jar file from the apk. We then use the ASM framework [5] to implement two types of transformations:

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

TABLE I. Considered Connection Statements.

	Class or Interface	**Method**
1.	java.net.URL	openConnection
2.	java.net.URLConnection	connect
3.	org.apache.http.client.HttpClient	execute
4.	java.net.Socket	getOutputStream


Home
Search
Shop by Dept
The Kids' Event
From $299
Great gifts at sweet prices
Rollbacks
Low prices just got lower
Weekly Ad
Set your store to find great savings!
Find a Store
View and search for nearby stores
Pharmacy
Order refills for pickup

(a) Screen 1. (b) Screen 2. (c) Diff. for (a) and (b).

Fig. 1. Visual differences.

- *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.

The jar file of the transformed application is then converted back to an apk using the dex2jar tool suite and signed with our own signature, using the jarsigner tool distributed with the standard Java JDK. As a known side effect of resigning applications, their authentication for services such as Google Plus APIs might be broken [6]. As a result, users might be unable to sign in with their Google Plus account or perform in-app purchases from the Google Play store. For that reason, we refrain from executing such scenarios in our analysis.

3) 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.

We experimented with Android's Monkey tool [7], but it was unable to provide a reasonably exhaustive coverage as it quickly locked itself out of the application by generating gestures that the analyzed application cannot handle. We thus recorded the desired application execution scenario manually, including in the recording any "semantic" user input required by the application, e.g., username and password. We used the Android getevent tool [8] that captures all user and kernel input events. We made sure to pause between user gestures that assume application response. We then enhanced the script produced by getevent to insert a screen capturing command after each pause and between events of any prolonged sequences.

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 Fig. 1, for the Walmart application. We used the ImageMagick compare tool [10] to produce the visual diff images automatically. We then manually scanned the produced output while ignoring differences in *content* of widgets that are populated by applications in a dynamic manner because such widgets are *expected* to differ between applications runs. That is, we ignored differences in the exact *content* of the advertisement messages (but not their overall presence/absence and position), *content* of social network messages, e.g., tweets, and the status of the device, e.g., the exact time. As such, the screenshots in Figs 1(a) and (b) were deemed similar: they differ only in the content of the advertisement information and the information in the status bar.

In one of the 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.

4) 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 for around 10 minutes, exploring all its functionality visible to us. Our goal was to (a) achieve sufficient coverage and (b) keep the application active long enough to allow any background data fetch processes to manifest themselves in the application's UI. Yet, we refrained from executing functionality related to signing-in into the user's Google Plus account or performing in-app purchases, due to the resign-related limitations mentioned above. We recorded the execution script that captured all triggered actions. We then re-installed the application and re-ran the script to collect screenshots that were used as the baseline for further comparisons.

In the second phase, we used the monitoring transformation to produce a version of the application that logs information about all existing and triggered connection statements. We ran this version using the recorded execution script and collected the statistics about its communication statements.

In the third phase, we iterated over all *triggered* connection statements, disabling them one by one, in order to assess the necessity 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 *covert*, kept it disabled for the subsequent iterations and proceed to the next connection in the list. Otherwise, we marked the exercised connection as *overt* and kept it enabled in the subsequent iterations. We continued with this process until all connections in the list were explored.

As the final quality measure, we manually inspected the execution of the version in which all covert connections were blocked, to detect any possible issues missed by the automated analysis.

5) Subjects: As the subjects of our study, we downloaded the 20 top-popular applications available on the Google Play store in November 2014. We excluded from this list 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 applications whose ASM-based instrumentation failed, most probably because

TABLE II. ANALYZED APPLICATIONS.

Applications	jar size (MB)	Total # of connection statements	# of triggered connection statements	# of covert (% of trig.)	# of covert in known A&A (% of total covert)
air.com.sgn.cookiejam.gp	2.7	17	3	2 (66.7%)	1 (50.0%)
com.crimsonpine.stayinline	3.2	15	2	2 (100.0%)	2 (100.0%)
com.devuni.flashlight	1.4	16	3	1 (33.3%)	1 (100.0%)
com.emoji.Smart.Keyboard	0.8	3	3	2 (66.7%)	0 (0.0%)
com.facebook.katana	0.6	3	0	-	-
com.grillgames.guitarrockhero	6.2	51	14	14 (100.0%)	6 (42.8%)
com.jb.emoji.gokeyboard	5.2	42	10	7 (70.0%)	0 (0.0%)
com.king.candycrushsaga	2.6	15	1	0 (0.0%)	-
com.pandora.android	5.7	57	12	9 (75.0%)	3 (33.3%)
com.spotify.music	5.4	20	7	2 (26.6%)	1 (50.0%)
com.twitter.android	5.9	21	10	3 (30.0%)	1 (33.3%)
com.walmart.android	5.8	33	8	5 (62.5%)	3 (60.0%)
net.zedge.android	6.5	37	8	4 (50.0%)	4 (100.0%)
Total		330	81	51 (62.9%)	22 (43.1%)

they use language constructs that are not supported by that framework. The remaining thirteen applications are listed in the first column of Table II; their corresponding archived byte code sizes are given in the second column of the table.

B. Results

The quantitative results of the study are presented in Table II. Columns 3 and 4 of the table show that only a relatively small number of connection statements encoded in the applications are, in fact, triggered dynamically. Some of the non-triggered statements correspond to execution paths that were not explored during our dynamic application traversal. Yet, the vast majority of the statements originate in third-party libraries included in the application, e.g., Google and Facebook services for mobile developers and various A&A libraries. Such libraries are often only partially used by an application, which could explain the incomplete coverage.

An interesting case is the Facebook application (row 5 in Table II), where most of the application code is dynamically loaded at runtime from resources shipped within the apk file. Our analysis was unable to traverse these custom-packed resources, and we thus excluded the application from the further analysis, noting that the only three connection statements in the application jar file are never triggered. We also excluded from the further analysis the Candy Crush Saga application (row 8 in Table II), as it did not exhibit any covert connections.

1) Classification of the Triggered Statements: Column 5 of Table II shows the number of connection statements that we determined as covert during our study. Averaged for all applications, 62.9% of the connections fall in that category. This means that only 37% 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.

Determining the original purpose of these covert connections is a non-trivial task: as implied by their nature, they do not exhibit any observable behavior. Due to obfuscation, virtually no "semantic" insights about the purpose of these connections can be drawn from the manual analysis of the application binaries either. In an attempt to shed some light on the essence of these connections, we inspected package and class names, as well as execution stack traces, of the methods containing the covert connection statements. We also inspected the data traffic that corresponds to these connections by routing the communication through a proxy that is able to sniff the data and decrypt the SSL encoding, if needed [11].

We discovered that only 22 out of 51 connections (43%) originate from the known A&A libraries, as shown in the last column of Table II. Another 11 connections (21%) appear to be responsible for the A&A content as well. However, they come either from application-specific packages or from third-party libraries that cannot be immediately linked to A&A. For example, the Walmart application triggers a covert connection from the *com/walmartlabs/analytics/* package, which appears to be a proprietary analytics service.

Analytics services collect information about application performance, crash and usage data, as well as the exact actions the user performs within the app. While this information has a clear value to the developer, no apparent description specifying the nature and frequency of the data collection is presented to the user. In fact, some applications start collecting analytics information even before they get activated. For example, twitter, Walmart and Pandora start their data collection as soon as the phone is booted and continue, periodically, during the phone's entire up time, even if the applications themselves were never used. In most cases, the user cannot opt-out from such data sharing without uninstalling the application.

Beyond A&A, several applications release information to their own and/or third-party services, without causing any effect on the applications' observable behavior. For example, twitter uses covert connections to collect information about videos and other rich media attachments followed by the users in tweets. The GO Keyboard application sends, via a covert connection, a set of ids to the *launchermsg.3g.cn* server; it also sends some encrypted data, which we could not decode, to *nextbrowser.goforandroid.com*. Both Pandora and Spotify music players use Facebook's social graph services [12], sending out information about the application usage. As another example, the Walmart application incorporates the barcode scanner library provided by Red Laser [13] – an eBay company that specializes in comparing prices. This library causes the application to send out information about the scanned barcode to the *data.redlaser.com* server. Yet, blocking that release of information does not harm the scanning capabilities.

To answer RQ1, we conclude that covert communication often occurs in real-world applications: 62.9% of the triggered connection statements can be deemed covert.

2) Lessons Learned: As shown in the last column of Table II, only 43% of the covert connections originate in the known A&A libraries. As such, distinguishing between overt and covert connections only by considering their package name schema is ineffective. Moreover, some malicious applications deliberately hide their payloads by using package names which look legitimate and benign [14]. We conclude that a more sophisticated technique for identifying the covert communication performed by the applications is required.

We manually investigated binaries of the analyzed applications, to gain more insights into the implementation patterns of covert connections. We noticed that, in a large number of cases, neither successes nor failures of such connections trigger visual notifications to the user. For the failures, the triggered exception is often either caught and silently ignored 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 warning message is written to the Android log file. However, this file is mostly used by developers rather than end-users.

To answer RQ2, we conjecture that covert connections can be detected by inspecting updates to UI elements on both the success and the failure path of a connection statement. Lack of such UI updates is indicative for a connection being covert for the application execution as the user is unaware of both the success and the failure of the communication.

III. Static Analysis for Classifying Connections

In this section we describe the static analysis algorithm we employ to automatically classify connections. Android applications are developed in Java as a series of external event handler routines, e.g., button click and application exit. Given an Android application, the static analysis classifies each statement that may invoke a connection call as either *overt* or *covert*. We begin by giving a precise definition of overt and covert connections assumed by the analysis.

Definition (Overt and Covert Connections). For connection statement s, s is classified as *overt* if it meets at least one of the following criteria:

1) **UI Cue on Failure**: When s triggers an exception e, the user may be notified of the failure via a user interface cue during *failure handling* of s on e.
2) **Program Exit on Failure**: When s triggers an exception e, the program may stop executing due to an execution path that propagates e back up to the Android runtime.
3) **UI Cue on Success**: When s succeeds, there exists a possible modification to the user interface *before* either (a) the next external event is handled by the application or (b) processing continues beyond the *failure handling* of s.

Conversely, a covert connection call does not meet any of the criteria.

Failure handling is defined precisely in Section III-B. Intuitively, the failure handling of connection statement s on exception type e are the computation paths that handle e and any failures triggered by the handling of e (through rethrown exceptions). Failure handling is finished when all exceptions triggered by e are handled and flow returns to normal execution.

In what follows, we discuss how we compute a static call graph that represents possible method targets of each call expression in the Android application. We then discuss our analysis on top of this graph. For each connection statement, the analysis consists of three steps: failure handler analysis, success forward analysis, and success backward analysis.

Table I lists the target methods that are considered as connection calls. The set of target methods that are considered as affecting the user interface are listed in Table III (also included are all overriding methods).

A. Constructing an Accurate Call Graph for Android Apps

Static analysis of Android applications is notoriously difficult because of the complexity and dynamic nature of the Android runtime and API; precise, whole-program analysis runs the high-risk of missing dynamic program behavior and not scaling to real-world Android applications. Static analysis must either model the Android execution environment or account for possible dynamic program behaviors with conservative analysis choices; otherwise some runtime behaviors could be unconsidered.

Our analysis over-approximates the runtime behaviors of the applications, and under-approximates the connection calls that could be covert. The analysis prioritizes high precision over high recall because we do not want to block connection calls that perform overt communication. Our analysis employs a class hierarchy analysis (CHA) [15] to build a call graph with refinement achieved by intra-procedural data-flow analysis, as discussed later. After much experimentation with higher precision, though brittle, points-to analysis techniques, this analysis combination gave us the best performance for the classification task. Our analysis is implemented in the Soot Analysis Framework [16] and utilizes the Android API model provided by DroidSafe [17]. The presentation of the analysis below assumes the application is represented in the Jimple intermediate language [16].

To compute a call graph, we augment the application code with the DroidSafe Android Device Implementation (ADI) [17]. The ADI is a Java-based model of the Android runtime and API. It attempts to present full runtime semantics for commonly-used classes of the runtime and API. Our call graph construction begins at each application method that could be a target of the Android runtime, i.e., the event handler methods of the application. Call graph construction does not traverse into Android API methods. As such, the call graph is actually a forest of graphs, each rooted at an event handler of the application, and each graph includes only application-defined methods. However, we found it necessary to account for API calls that directly call back into the application, e.g., `Thread.start()`. We achieve that by replacing the API call with a direct call to the application method(s) that the API call could invoke, e.g., `Thread.run()`.

The call graph is augmented to account for reflected method calls using the following policy. When a reflected call is found, we add edges to the graph that target all methods of the same package domain as the caller (e.g., `com.google`, `com .facebook`). The edges are pruned by the following strategy: if the number of arguments and argument types to the call can be determined using a def-use analysis [18], then we limit the edges to only targets that have the same number and types of arguments. This strategy works well for us in practice and aggressively accounts for reflection semantics.

B. Failure Handler Analysis

Failure analysis determines if a connection call modifies the UI or could exit the application on exception. We organize the static failure-handling analysis as a recursive traversal on the call graph. An iterator over all application statements calls the analysis separately for the combination of each statement in the application that could target a connection call and an exception that indicates communication failure.

Conceptually, the failure handling of s on e is defined as a slice of instructions. In the slice are all statements that are

TABLE III. CONSIDERED UI ELEMENTS.

	Class or Interface	Methods
1.	android.app.Dialog	setContentView
2.	android.support.v7.app. ActionBarActivityDelegate	setContentView
3.	android.view.View	onLayout, layout, onDraw, onAttachedToWindow
4.	android.view.ViewGroup	addView, addFocusables, addTouchables, addChildrenForAccessibility
5.	android.view.ViewManager	addView, updateViewLayout
6.	android.view.WindowManagerImpl. CompatModeWrapper	addView
7.	android.webkit.WebView	loadData, loadDataWithBaseURL, loadUrl
8.	android.widget.TextView	append, setText
9.	android.widget.Toast	makeText

```
 1: procedure FINDCATCHES(meth, stmt, ex, visiting, stack, cg)
 2:     if (stmt, ex) $\in$ visiting or stmt $\in$ overt then return end if
 3:     visiting $\leftarrow$ visiting $\cup$ (stmt, ex)
 4:     catchBlockStart $\leftarrow$ FINDCOMPATCATCH(meth, stmt, ex)
 5:     if catchBlockStart $=$ null then
 6:         if ISEVENTHANDLER(meth) then
 7:             overt $\leftarrow$ overt $\cup$ stmt
 8:             return
 9:         end if
10:         for (predStmt, predMeth) $\in$ GETPREDS(cg, meth) do
11:             if stack $\neq \emptyset$ and (predStmt, predMeth) $\neq$ PEEK(stack) then
12:                 continue
13:             end if
14:             newStack $\leftarrow$ stack
15:             POP(newStack)
16:             FINDCATCHES(predMeth, predStmt, ex,
17:                         visiting, newStack, cg)
18:             if predStmt $\in$ overt then
19:                 overt $\leftarrow$ overt $\cup$ stmt
20:                 return
21:             end if
22:         end for
23:     else
24:         catchStmts $\leftarrow$ GETCATCHSTMTS(catchBlockStart, meth)
25:         ANALYZEHANDLER(meth, stmt, catchStmts,
26:                        visiting, $\emptyset$, stack, cg)
27:     end if
28: end procedure
```

Fig. 2. Find **catch** blocks for exception thrown at statement.

reachable from all **catch** blocks that can handle e. Then, for each reachable statement in the slice that could throw an exception of type f, we recursively add to the slice all statements that are reachable from all **catch** blocks that could dynamically handle f. Next, we recursively process those newly added statements that could throw an exception. In the processing, Android API methods are not considered, and searching for a handler does not propagate into the Android runtime environment.

The analysis starts with the FINDCATCHES procedure listed in Fig. 2. It employs a set of helper procedures; the non-obvious ones are listed in Fig. 4, and the full list is available in [19]. FINDCATCHES is called separately for each connection statement and exception pair, stmt and ex, respectively. The arguments to the procedure also include stmt's enclosing method (meth); the set of currently visiting statements and exception pairs (visiting), initially empty; the call stack built when searching forward on a path of the call graph (stack), initially empty; and the pre-calculated call graph (cg).

For stmt and ex, the procedure first consults stmt's containing method to find an appropriate **catch** (line 4). If ex is not caught locally, and meth is an event handler (called from the Android runtime), then we conservatively calculate that stmt could cause application exit, and it is added to the overt set (lines 6-9). Otherwise, the analysis recursively visits all direct caller methods to find **catch** blocks that trap the call graph edge (lines 10-21), as discussed later.

```
 1: procedure ANALYZEHANDLER(meth, exceptStmt, stmts, visiting, handledStmts,
    stack, cg)
 2:     if stmts $\in$ handledStmts then return end if
 3:     handledStmts $\leftarrow$ handledStmts $\cup$ stmts
 4:     for each stmt $\in$ stmts do
 5:         if HASINVOKE(stmt) then
 6:             for (succStmt, succMeth) $\in$ GETSUCCS(cg, stmt) do
 7:                 if ISUIMETHOD(succMeth) then
 8:                     overt $\leftarrow$ overt $\cup$ exceptStmt
 9:                     return
10:                 else if ISNATIVEMETHOD(succMeth) then
11:                     for nativeEx $\in$ GETTHROWSEXCEPTIONS(succMethd) do
12:                         FINDCATCHES(meth, stmt, nativeEx, visiting, stack, cg)
13:                     end for
14:                 else
15:                     newStack $\leftarrow$ stack
16:                     PUSH(newStack, (succStmt, succMeth))
17:                     succStmts $\leftarrow$ GETBODYSTMTS(succMeth)
18:                     ANALYZEHANDLER(succMeth, exceptStmt, succStmts,
19:                                    visiting, handledStmts, newStack, cg)
20:                 end if
21:             end for
22:         else if ISTHROWSTMT(stmt) then
23:             rethrownTypes = $\emptyset$
24:             for defStmt $\in$ GETLOCALDEFS(GETOP(stmt)) do
25:                 if ISALLOC(defStmt) then
26:                     rethrownTypes $\leftarrow$ rethrownTypes $\cup$ GETTYPE(defStmt)
27:                 else if ISCAUGHTEXCEPTIONSTMT(defStmt) then
28:                     rethrownTypes $\leftarrow$ rethrownTypes $\cup$
29:                         GETPOSSIBLETHROWNTYPES(meth, defStmt)
30:                 else
31:                     overt $\leftarrow$ overt $\cup$ exceptStmt
32:                     return
33:                 end if
34:             end for
35:             for rethrownType $\in$ rethrownTypes do
36:                 FINDCATCHES(meth, stmt, rethrownType, visiting, stack, cg)
37:                 if stmt $\in$ overt then
38:                     overt $\leftarrow$ overt $\cup$ exceptStmt
39:                     return
40:                 end if
41:             end for
42:         end if
43:     end for
44: end procedure
```

Fig. 3. Analyze reachable statements during failure handling.

If a compatible handler is found locally (lines 24-26), the analysis calls the procedure ANALYZEHANDLER (Fig. 3) on the statements of the compatible block. This procedure analyzes the reachable statements of the handler. If a call that could target a UI method is encountered, then the statement that began the handler analysis is considered overt since the failure handling affects the user interface (lines 7-9 in Fig. 3). If the analysis finds a call to a native method, we assume that the method will throw all exceptions it is defined to throw, and the handler analysis spawns a FINDCATCHES instance for each exception declared throws (lines 10-13). When the analysis finds a call to an application method, it pushes the current statement and method onto the stack and recursively calls itself for the new method to analyze the new method's statements (lines 14-20).

If the analysis finds a **throw** statement, it spawns a new FINDCATCHES analysis to find all the possible handlers of each rethrown exception (lines 22-42). Towards this end, it first calculates local def-use chains to obtain the types of the exception. In lines 23-34, the analysis considers all local reaching defs of the thrown value. If an allocation statement

FindCompatCatch($meth$,$stmt$,ex): Return the first statement of the `catch` block that will handle an exception of type ex thrown at statement stmt in method meth.

GetCatchStmts($stmt$,$meth$): Given the start of a `catch` block defined in the trap table of method $meth$, return all statements that were defined in the source code for the `catch` block of $stmt$. This method calculates an over-estimation of `catch` block extents, e.g., it includes `finally` blocks.

GetPossibleThrownTypes($meth$,$stmt$): Calculate the possible exception types caught at the catch block that begins with $stmt$ of $meth$. The statements of the `try` block that associates with the `catch` block that encloses $stmt$: (1) For a call statement, the procedure adds to the return list all exception types declared throws by all methods that the call can target, (2) For a `throw` statement, the reaching definitions of the thrown value are calculated. If the reaching definition is an allocation, then add to the return list the type of the allocation. If the reaching definition is a caught exception statement, then GetPossibleThrownExceptions recursively calls itself to find the nesting try block statements and continue the calculation. If a definition of any other statement type can reach the thrown value, then return null to denote that it cannot calculate the thrown.

Fig. 4. Failure handling analysis helper functions.

reaches the **`throw`**, then the allocated type is added to the set of possible types of the rethrown exception. If a caught exception reference, c, reaches the **`throw`** statement, then the **`try`** block associated with **`catch`** block of c is analyzed for all checked exceptions that could be thrown. This is performed in the GetPossibleThrownTypes call in line 29 of AnalyzeHandler. If only allocations and caught exception statements reach the thrown value, then the handler analysis spawns a new FindCatches instance to analyze the failure handling.

A stack of pairs of method call statement and target method is maintained. The analysis uses the stack to focus the handler search in FindCatches after a method call has been performed by a handler further up the stack (lines 11-15 in Fig. 2). When we initiate the analysis for a connection call, the stack is empty and the analysis in FindCatches has to search all possible stacks (predecessor of the containing method) for handlers of the connection statement's exception. However, once a handler is found, and the handler calls a sequence of methods that ends in a possible rethrown exception, the sequence of methods defines the only stack that should be searched for a handler of the rethrown exception (line 11).

The stack is pushed in line 16 of AnalyzeHandler for each method call of a reachable handler code. During the handler search of the execution stack in FindCatches, the stack is consulted to guide the search in line 11, only visiting the edge at the head of the stack. The stack is popped when visiting a caller method of the current method in FindCatches line 15.

C. Success Analysis

For connection statement stmt, if the failure handling analysis concludes that no UI call could be invoked during failure handling and all stacks handle stmt's exception, then the analysis continues with the success paths of stmt. The success analysis determines if there is any UI modification after the connection succeeds but before control returns to the Android runtime environment and before control merges back to the failure handling paths.

Here we summarize the success analysis of connection call stmt enclosed in method meth by presenting a high-level description of two conceptual phases:

1) Success Forward Analysis: Code reachable from the statement immediately after stmt is searched for a connection call. This is accomplished by traversing all paths in the interprocedural control flow graph (CFG) starting at stmt and following method invoke expressions via the call graph. We follow both normal and exceptional control paths when analyzing the CFG.

2) Success Backward Analysis: For all methods, caller, such that there is a path from caller to meth in the call graph and caller was searched during the failure handling analysis, examine all statements of caller for calls that affect the user interface. This has the effect of traversing the call graph backwards from meth at stmt for UI calls, stopping at event handlers (that are called by the Android API) and, on a path, stopping once the exception of stmt is handled.

If the success forward and success backward analysis do not find any calls that could affect the user interface, then the connection call is classified as covert.

D. Design Discussion

The intuition for our static analysis is that overt connection calls always affect the user interface either on success or failure. Another insight gleaned from the study in Section II is that normal processing of a connection often ends when the exceptional and non-exceptional control flow paths merge. Hence, the success backward analysis only examines methods that were analyzed by the failure analysis. Furthermore, we use the enclosing runtime event handlers to bound the connection call processing.

IV. Experiments

We start by assessing the quality of our static analysis technique. We then apply the technique to gather information about common patterns of covert communication in the 500 most popular Android applications on Google Play.

A. Quality of the Static Analysis

We first evaluate the accuracy, i.e., precision and recall, of our technique on the "truth set" established during our in-depth case study (see Section II). Then, via a usability assessment, we evaluate the user-experience when running a version of an application in which all connections deemed covert are disabled.

1) Accuracy: For the accuracy evaluation, we look again at the applications listed in Table II, excluding Facebook and Candy Crush because these applications did not exhibit any covert communication. We limit the set of results reported by the static analysis to those that were, in fact, triggered dynamically, as only for these we have the "ground truth" established. We assess the results, for each application individually and averaged for all applications, using the metrics below:

- *Precision*: the fraction of connection statements correctly identified as covert among those reported by the technique.
- *Recall*: the fraction of connection statements correctly identified as covert among those expected, i.e., marked as covert during the dynamic study.

- *Execution time*: the execution time of the analysis, measured by averaging results of three runs on an Intel® Xeon® CPU E5-2690 v2 @ 3.00GHz machine running Ubuntu 12.04.5. The machine was configured to use at most 16GB of heap and to perform no parallelization for a single application, i.e., each application uses one core only.

The results of this experiment are summarized in Table IV. The second column of the table shows that the overall averaged precision of our analysis is 93.2%. The analysis correctly identifies all but two covert connections. The first one, in *com.devuni.flashlight*, is responsible for presenting icons of application extensions that can be downloaded from Google Play. The misclassification stems from the fact that UI updates for these icons happen after the success and failure paths unify, and thus are missed by our search.

The second misclassified connection, in *net.zedge.android*, is responsible for presenting advertisement material and belongs to the *com.mopub.mobileads* A&A service library packaged with the application. That library relies on asynchronous RPC communication with Google services installed on the same device. Our static analysis is not designed to track inter-application communication between various applications and services on the device, hence the false-positive result.

Even though our analysis is designed to be conservative, it is able to correctly identify 61.5% of statements deemed covert in the empirical study (see column 3 in Table IV). The major reasons for why we do not achieve higher recall are (1) a conservative, though feasibly analyzable, definition of direct processing related to a connection call, and (2) conservative call graph construction, specifically w.r.t. reflection. Such solution is aligned with our goal of providing actionable results, which are "safe" albeit under-approximate.

Finally, the analysis is highly efficient and runs in a matter of minutes even on large applications, as shown in the last column of Table IV.

2) Usability Assessment: To check whether our technique is able to provide actionable results, we further selected 100 applications that persisted in the list of the 500 most popular free applications on Google Play in the January 2015 and May 2015 samples. We installed the original version of each application on a Nexus device running Android v4.4.4. On an identical device, we installed a modified version of each application that was produced by employing the *blocking transformation* (see Section II) to disable all calls identified as covert by the static analysis.

We recruited two human subjects, both experienced software developers, and paired each with an author of this paper. Each pair was given one device with the original and one with the modified versions of the applications. We asked them to execute the same application simultaneously on both devices for around 10 minutes, and to record all differences observed during the execution. Similar to the experiment described in Section II, our goal was to ensure sufficient coverage and manifestation of background data fetch processes in the application's UI. We asked the participants to avoid signing in with a Google Plus account or performing in-app purchases from the Google Play store, as these features are not supported in resigned applications, as discussed in Section II.

TABLE IV. COMPARISON WITH THE MANUALLY ESTABLISHED RESULTS.

Applications	**Correctly detected covert**		**Execution time**
	Precision	Recall	
air.com.sgn.cookiejam.gp	1/1 (100.0%)	1/2 (50.0%)	2min 11s
com.crimsonpine.stayinline	2/2 (100.0%)	2/2 (100.0%)	2min 24s
com.devuni.flashlight	1/2 (50.0%)	1/1 (100.0%)	1min 44s
com.emoji.Smart.Keyboard	2/2 (100.0%)	2/2 (100.0%)	1min 16s
com.grillgames.guitarrockhero	1/1 (100.0%)	1/14 (7.1%)	6min 14s
com.jb.emoji.gokeyboard	4/4 (100.0%)	4/7 (57.1%)	3min 22s
com.pandora.android	4/4 (100.0%)	4/9 (44.4%)	2min 41s
com.spotify.music	1/1 (100.0%)	1/2 (50.0%)	2min 51s
com.twitter.android	1/1 (100.0%)	1/3 (33.3%)	3min 3s
com.walmart.android	3/3 (100.0%)	3/5 (60.0%)	3min 2s
net.zedge.android	3/4 (75.0%)	3/4 (75.0%)	4min 13s
Average	93.2%	61.5%	2min 48s

To analyze the results of that experiment in a reliable manner, we exclude 14 applications that were non-operational (either did not run in the original version or required payment to continue running); 17 applications for which ASM-based instrumentation failed or the instrumented version did not run due to the issues related to the resigning process; 2 Google applications that we could not re-install on a device; 5 chat applications; 4 applications that either contained no connection statements or had no covert connection statements detected; and 11 applications for which no covert connection statement were triggered during the dynamic execution of the application.

Information about the remaining 47 applications is below.
Identical: 30 (63.8%). Our participants did not observe any noticeable differences in these 30 applications, which confirms that the connection statements deemed covert by the static analysis have indeed no effect on the user-observable application behavior.
Missing advertisement: 9 (19.2%). Advertisement information was missing in 9 cases, for the same reason as in the Zedge example described above.
Missing minor functionality: 3 (6.4%). The participants observed the absence of features that they perceived as minor: 2 cases of missing icons, in Talkingben as well as in the Flashlight application discussed above; and 1 case where they were unable to create an account for the antivirus application, but the core functionality of that application was intact.
Missing essential functionality: 5 (10.6%). Only 5 applications were missing essential functionality: Battery Saver, Spider-Man and Minion Rush games, Microsoft Office Mobile and PicsArt Photo Studio. We conjecture that the last case is related to resigning issues. Other cases stem from the limitations of our static analysis techniques: performing intra-application analysis, ignoring stateful communication and restricting the search for UI updates only to statements that occur before the success and failure paths unify. We believe that the low number of such cases, together with the high scalability of our analysis, justifies these choices.

On average, 2.6 covert call statements per application were triggered at runtime (min: 1, max: 9, mdn: 2). As each statement can be executed multiple times, we also counted all dynamic call instances of these statements, obtaining the average of 299 covert call instances per application (min: 1, max: 4011, mdn: 11). The high average numbers are due to applications that, once installed, are constantly executed in the background, and, as it turns out, attempt network communication. Examples of such applications are *com.cleanmaster.mguard* and *com.ijinshan.kbatterydoctor_en*.

> *To answer RQ3, we conclude that the static analysis proposed in this paper can be applied for an accurate detection of covert connections. The technique is precise, highly scalable and provides actionable output that can be directly used for disabling covert communication in a vast majority of cases.*

B. Covert Communication in the Wild

We next apply our technique to the 500 most popular Android applications downloaded from the Google Play store in January 2015. By considering such a large data set, our goal is to investigate how often covert communication occurs and what its most common sources are.

Our analysis reveals that 46.2% of all connection statements in these application can be considered covert (8,539 connections out of 18,480 in total). These results, adjusted by the 93% precision and 61% recall rates of the analysis, are consistent with the observation of our empirical study described in Section II.

Table V presents the top 10 packages in which covert connections occur. As we analyzed free applications that frequently use third-party packages and then aggregated the numbers for 500 such applications, it is no surprise that Google services, gaming, and A&A services are at the top of the list; the number of applications that use each of these packages is shown in the third column of Table V. More surprising is the *com.gameloft* package (row 2 of Table V); even though it is part of only 17 different mobile applications published by the same company, the number of covert connection statements these game applications contain is notable.

The last column of Table V shows the percentage of covert connections out of all connection statements in the corresponding package. This number varies between 24% and 87%, confirming, again, our initial observation in Section II that the source of a connection cannot be used to determine its impact on the application behavior.

> *To answer RQ4, we conclude that covert communication is common in real-life applications. Such communication is not exclusive to A&A packages, and not all communication stemming from these packages is covert.*

V. Limitations and Threats to Validity

1) Empirical Study: Our empirical study has a dynamic nature and thus suffers from the well-known limitations of dynamic analysis: it does not provide an exhaustive exploration of an application's behavior. Even though we made an effort to cover all application functionality visible to us, we might have missed some behaviors, e.g., those triggered under system settings different from ours. To minimize unexpected results, we performed all our dynamic experiments on the same device, at the same location and temporally close to each other. We also automated our execution scripts in order to compare behaviors of different variants under the same scenario and settings. We only report on the results comparing these similar runs.

TABLE V. Top 10 Covert Communication Callers.

	Package	**Description**	**Used in # (%) of Apps**	**Covert Calls (% of total calls)**
1.	com.google.android	Google services	382 (76.4%)	1913 (49.9%)
2.	com.gameloft	Mobile games	17 (3.4%)	784 (87.4%)
3.	com.inmobi	A&A services	61 (12.2%)	615 (67.6%)
4.	com.millennialmedia. android	A&A services	78 (15.6%)	447 (58.8%)
5.	com.mopub.mobileads	A&A services	72 (14.4%)	320 (56.9%)
6.	com.tapjoy	A&A services	49 (9.8%)	277 (43.8%)
7.	com.facebook	Facebook services	112 (22.4%)	222 (24.3%)
8.	com.unity3d	Gaming services	77 (15.4%)	203 (41.8%)
9.	(default)	Default package of an application	23 (4.6%)	178 (48%)
10.	com.flurry	A&A services	95 (19%)	175 (35.3%)

Yet, due to the limitations of dynamic analysis, we could encounter false-positive results: a connection statement that we classified as covert could have an effect on an unexplored part of an application. At the same time, we could also have false-negative results: changes in user interface could stem from the non-determinism in an application itself rather than the absence of communications. Moreover, by focusing on individual connection statements, we cannot distinguish between multiple application behaviors that communicate via the same statement in code. We thus conservatively deem a connection as overt if it is overt for at least one of such behaviors. Exploring more sophisticated techniques for identification of covert connections could be a subject of possible future work.

Finally, our study only includes a limited number of subjects, so the results might not generalize to other applications. We tried to mitigate this problem by not biasing our application selection but rather selecting top-popular applications from the Google Play store, and by ensuring that we observe similar communication patterns in all analyzed applications.

2) A Static Technique For Detecting Covert Connections: Our technique deems as covert *stateful* communication that toggles the state of a connection target but does not present any information to the user. In many cases, detecting such communication statically is impossible because the code executed on the target is unknown and unavailable. For a similar reason, in this work, we do not consider RPC communication with applications installed on the same device. We might explore that direction as part of future work.

Moreover, our analysis searches for communication that affects the application UI in a direct manner rather than transitively, through other resources. Extending the analysis to cover such cases, while maintaining its scalability and precision, is another subject for possible future work.

Some of the covert connections that we identified statically might never be triggered dynamically. A large percentage of these connections originate in third-party libraries that are included in the application but only partially used. As such, analyzing them is still beneficial as this code might be used in other applications.

VI. Related Work

Work related to this paper falls into three categories:

1) User-Centric Analysis for Identifying Spurious Behaviors: Huang et al. [20] 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.

Ko and Zhang [3] propose a system, FeedLack, for identifying usability problems in web applications. The system looks for control flow paths that originate from user input but lack UI-affecting code. Our work is similar as it relies on the same underlying principle of user feedback necessity and also searches for code affecting the UI. Yet, our goal is different: we look for any hidden behavior rather than missing feedback loops for user-triggered operations. Also, our analysis is tailored for mobile rather than web applications and, unlike FeedLack, focuses not only on success paths but takes failure paths into account as well.

CHABADA [21] 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. [22] 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.

2) Information Propagation in Mobile Applications: The most prominent technique for dynamic information propagation tracking in Android is TaintDroid [23], which detects flows of information from a selected set of sensitive sources to a set of sensitive sinks. Several static information flow analysis techniques for tracking propagation of information from sensitive sources to sinks have also been recently developed [24], [17], [25], [26]. 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 overt and covert 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 applications 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.

Both MudFlow [27] and AppContext [28] build up on the FlowDroid static information flow analysis system [24] 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 covert rather than malicious behaviors, aiming to preserve the overall user experience.

Shen et al. [29] 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 over the holistic behavior of the applications installed on a user's phone – our techniques are entirely orthogonal.

3) Exception Analysis for Java: A rich body of static analysis techniques has been developed to analyze and account for exceptional control and data flow [30], [31], [32], [33], [34], [35], [36]. Most of these techniques define a variant of a reverse data-flow analysis and use a program heap abstraction (e.g., points-to analysis or class hierarchy analysis) to resolve references to exception objects and to construct a call graph. Our technique follows a similar strategy, using class hierarchy analysis with intra-procedural analysis refinement. Though some of the prior analysis techniques will provide higher precision than our technique (namely [32], [33], [35]), we designed our technique to conservatively, though aggressively, consider difficult to analyze Android application development idioms such as reflection, native methods, and missing program semantics of the Android API defined in non-Java languages.

VII. CONCLUSIONS

Covert communication can impair the transparency of device operation, silently consume device resources, and ultimately undermine user trust in the mobile application ecosystem. Our analysis shows that covert communication is quite common in top-popular Android applications in the Google Play store. Our results show that our static analysis can effectively support the identification and removal of covert communication and promote the development of more transparent and trustworthy mobile applications.

Acknowledgments. This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0110. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

REFERENCES

[1] J. Nielsen and R. Molich, "Heuristic Evaluation of User Interfaces," in *Proc. of the International Conference on Human Factors in Computing Systems (CHI'90)*, 1990, pp. 249–256.

[2] M. H. Blackmon, P. G. Polson, M. Kitajima, and C. H. Lewis, "Cognitive Walkthrough for the Web," in *Proc. of the International Conference on Human Factors in Computing Systems (CHI'02)*, 2002, pp. 463–470.

[3] A. J. Ko and X. Zhang, "FeedLack Detects Missing Feedback in Web Applications," in *Proc. of the International Conference on Human Factors in Computing Systems (CHI'11)*, 2011, pp. 2177–2186.

[4] dex2jar, "https://code.google.com/p/dex2jar/."

[5] ASM Java Bytecode Manipulation and Analysis Framework, "http://asm.ow2.org/."

[6] Google APIs Console Help, "https://developers.google.com/console/help."

[7] Android's UI/Application Exerciser Monkey, "http://developer.android.com/tools/help/monkey.html."

[8] Android Getevent Tool, "https:// source.android.com/devices/input/getevent.html."

[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)*, 2011, pp. 639–652.

[10] ImageMagick Compare Tool, "http:// www.imagemagick.org/script/compare.php."

[11] Charles – an HTTP proxy / HTTP monitor / Reverse Proxy, "http://www.charlesproxy.com/."

[12] Facebook's Social Graph, "https://developers.facebook.com/docs/graph-api."

[13] Red Laser – an eBay company, "http://redlaser.com/."

[14] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," in *Proc. of the IEEE Symposium on Security and Privacy (SP'12)*, 2012, pp. 95–109.

[15] J. Dean, D. Grove, and C. Chambers, "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis," in *Proc. of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, 1995.

[16] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing Java Bytecode Using the Soot Framework: Is It Feasible?" in *Proc. of the 9th International Conference on Compiler Construction (CC'00)*, 2000, pp. 18–34.

[17] 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.

[18] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison Wesley, 2006.

[19] Helper Function Definitions for the Connection Analysis, "http://people.csail.mit.edu/mjulia/covert-helpers.pdf."

[20] 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.

[21] 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.

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

[23] 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)*, 2010, pp. 1–6.

[24] 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.

[25] 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.

[26] 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)*, vol. abs/1404.7431, 2014.

[27] 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)*, 2015.

[28] 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)*, 2015.

[29] 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.

[30] Byeong-Mo Chang, Jang-Wu Jo, and Soon Hee Her, "Visualization of Exception Propagation for Java Using Static Analysis," in *Proc. of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation*, 2002, pp. 173–182.

[31] B.-M. Chang, J.-W. Jo, K. Yi, and K.-M. Choe, "Interprocedural Exception Analysis for Java," in *Proc. of the 2001 ACM Symposium on Applied Computing (SAC'01)*, 2001, pp. 620–625.

[32] C. Fu, A. Milanova, B. G. Ryder, and D. G. Wonnacott, "Robustness Testing of Java Server Applications," *IEEE Transactions on Software Engineering*, vol. 31, pp. 292–311, 2005.

[33] C. Fu and B. G. Ryder, "Exception-Chain Analysis: Revealing Exception Handling Architecture in Java Server Applications," in *Proc. of the International Conference on Software Engineering (ICSE'07)*, 2007, pp. 230–239.

[34] J. W. Jo, B. M. Chang, K. Yi, and K. M. Choe, "An Uncaught Exception Analysis for Java," *Journal of Systems and Software*, vol. 72, pp. 59–69, 2004.

[35] X. Qiu, L. Zhang, and X. Lian, "Static Analysis for Java Exception Propagation Structure," in *Proc. of the 2010 IEEE International Conference on Progress in Informatics and Computing (PIC'10)*, vol. 2, 2010, pp. 1040–1046.

[36] G. Kastrinis and Y. Smaragdakis, "Efficient and Effective Handling of Exceptions in Java Points-to Analysis," in *Proc. of the 22nd International Conference on Compiler Construction (CC'13)*, 2013, pp. 41–60.