Non-Essential Communication in Mobile Applications

Julia Rubin*, Michael I. Gordon*, Nguyen Nguyen§ and Martin Rinard*
*Massachusetts Institute of Technology, USA
§Global InfoTek, Inc, USA

***Abstract*—This paper studies communication patterns in mobile applications. Our analysis shows that almost 70% of the external communication made by top-popular Android applications from Google Play has no effect on the user-observable application functionality. To detect such communication in an efficient manner, we propose a highly precise and scalable static-analysis technique: it achieves 93% precision and 60% 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 non-essential 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 non-essential communication and use it to investigate communication patters 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 back-end and other third-party servers. This paper shows that much of this communication delivers no value to the user of the application: disabling such communication leaves the delivered application experience completely intact. Yet, this communication comes with costs such as bandwidth charges, power consumption on the device, potential privacy breaches and analytic data release, and the unsuspected presence of continued communication between the device and remote organizations. In fact, we observed that several applications silently spawn services that communicate with third-party servers even when the application itself is no longer active, with the user completely unaware that the spawned services are still running in the background.

This paper takes the first steps towards automatically identifying and disabling these kinds of non-essential communications. We start by analyzing communication patterns of popular applications in the Google Play App Store (twitter, WalMart, Spotify, Pandora, etc.). Motivated by the significant amount of non-essential communication we found in these applications, we develop a highly precise and scalable static analysis that can identify such non-essential 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 frequently does non-essential communication occur in widely used mobile applications? To estimate the significance of the problem, we conduct an empirical study that focuses on identifying and investigating the nature of non-essential communication in thirteen of the top twenty most-popular applications in Google Play.

Our study has three major steps. First, we establish baseline application behavior. Towards this end, we record a script triggering the application functionality via a series of interactions with the application's user interface. After each interaction, the script records the application state by capturing a screenshot of the device. We then instrument the application to log information about all triggered connection statement and execute the instrumented version using the same script. Finally, we iterate over all identified triggered connections, disabling each at a time. We execute the recorded script again for the version of the application with that connection disabled and compare the obtained screenshots to those of the original run.

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 device's status bar. Our study reveals that around 70% of the *exercised* connection statements are not essential — disabling them has no noticeable effect on the observable application functionality. Interestingly, only less than half of such non-essential connections correspond to communication initiated from known advertisement and analytics (A&A) packages included in the application; other statements are triggered by the core application itself. Thus, looking at the package information only is not sufficient for distinguishing between essential and non-essential communication.

RQ2: Can non-essential communication be detected statically? Detailed investigation of multiple detected cases of non-essential 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 essential. Furthermore, if the exception could be propagated back to the Android

runtime (causing the application to exit), the connection call is deemed essential. Otherwise, it is deemed non-essential.

Our static analysis is designed to scale to large Android applications and to conservatively approximate the behavior of dynamic constructs such as reflection and missing semantics such as native methods. The analysis also reasons about application code reachable through Android API calls and callbacks by analyzing each application in the context of a rich model of the Android API [1].

We perform intra-application analysis only, i.e., ignoring effects of RPC communication between multiple applications on the same device. Moreover, our technique is not designed to handle *stateful* communication patterns with external services, i.e., those when a connection attempt leaves the target in a state different from the one it had before, and further communication is influenced by the server's state. Our experiments show that both such cases are rare.

RQ3: How well does static detection perform? To assess the precision and recall of our technique, we evaluate it on the "truth set" established during the in-depth dynamic study describe above. The results show that our static analysis features a high precision – 93% of the non-essential connection calls identified by the static analysis are indeed classified as non-essential during the manual analysis. Even though it is designed to be conservative, it is still able to identify 60% of all non-essential connection in the studied application. Only two connections in total are misclassified as non-essential: one is responsible for presenting advertisement material via the Google Ads component installed on the device, and another – for presenting icons of additional apps that can be downloaded from Google Play.

To gain further insights on the quality of our technique, we apply it on additional 47 top-popular applications from Google Play. For these applications, we disable all connection statements deemed non-essential by our 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 two versions of the application.

The results of this assessment are encouraging: there were no observable differences in 63.8% of the application (30 cases). In 25.6% of the applications (12 cases) differences where related to absence of functionality considered minor by the users, e.g., ads or decorating images. Only 10.6% of the applications (5 cases) missed functional features considered essential by the users. These results imply that our technique produces actionable results that could already be applied to eliminate many cases of non-essential communication.

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

Significance of the Work. Our work focuses on benign applications that can be downloaded from popular application stores and that are installed by millions of users. By identifying and highlighting application functionality hidden from the user, the goal is to encourage application developers to produce more transparent and trustworthy applications. The identification of potential privacy violations in previous versions of popular Android applications [2], [3], [4] followed by the elimination of these violations in current Android applications provides encouraging evidence that such an improvement is feasible.

Contributions. The paper makes the following contributions:

1) It sets *a new problem* of distinguishing between essential and non-essential communication in an automated manner. The goal is to improve the transparency and trustworthiness of mobile applications.
2) It proposes *a semi-automated dynamic approach* for detecting non-essential communication in Android applications that 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 affect 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 69% 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 non-essential connections – those that do not directly propagate any effects back to the application's user. 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% work with only insignificant interference when disabling all connections identified by the technique.
5) It provides *quantitative evidence* for the prevalence of non-essential connections in the 500 top-popular free applications on Google Play and identifies common patters of such communication.

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

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

TABLE II. ANALYZED APPLICATIONS.

Applications	jar size (MB)	Total # of connection statements	# of triggered connection statements	# of non-essentials (% of trig.)	# of non-essentials in known A&A (% of total non-essential)
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%)	8 (57.1%)
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%)	4 (44.4%)
com.spotify.music	5.4	20	7	3 (42.9%)	1 (33.3%)
com.twitter.android	5.9	21	4	3 (75.0%)	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	52.0	330	75	52 (69.3%)	25 (48.1%)

`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.

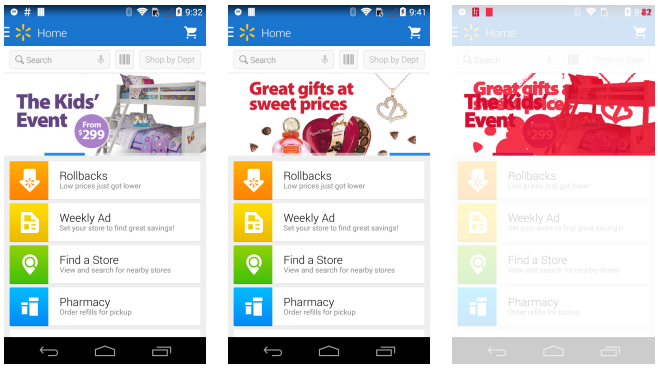
2) Application Instrumentation: As input to our study, we assume an Android application given as an apk file. We use the dex2jar tool suite [5] to extract the jar file from the apk. We then use the asm framework [6] to implement two types of transformations:

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

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

As a side effect of resigning applications, their authentication when communicating with services such as Google Plus APIs might be broken [7], preventing the users from signing in with their Google Plus account or performing in-app purchases from the Google Play store. Due to this known limitation, we refrain from executing such scenarios in our analysis, as discussed below.

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. As the first step, we use the Android getevent tool [8] 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



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

Fig. 1. Visual differences.

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 [9] tool. While this tool is able to generate 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 [10], 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 and twitter applications. We used the ImageMagick compare tool [11] to 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 the screenshots in Figs 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 Figs 1(d) and (e). These two figures are thus also deemed similar.

In one out 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 10 minutes, exploring all its functionality visible to us. We refrained from executing functionality related to signing-in into the user's Google Plus account or performing in-app purchases though, due to the instrumentation-related limitations mentioned above. During that execution, we recorded the execution script that captured all triggered actions.

We then re-installed the application and re-ran the script to collect schreenshots 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 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 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.

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.

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 become 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 sizes are given in the second column of the table. We did not extend our dynamic analysis beyond these thirteen applications because the inspection of our findings indicated that we reached saturation: while it is clearly infeasible to explore all possible scenarios, we observed similar trends in all analyzed applications.

B. Results

The quantitative results of the study are presented in Table II. Column 3 and 4 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, A&A libraries. In fact, we identified 14 different A&A libraries used by the 13 applications that we analyzed, and many times a single applications uses multiple such libraries.

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 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. We also excluded from the further analysis the candycrushsaga application (row 8 in Table II), as it did not exhibit any non-essential connections.

1) Classification of the Triggered Statements: Column 5 of Table II shows the number of connection statements that we determined as non-essential during our study. Averaged for all applications, 69% of the connections fall in that category. This means that only 31% 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.

> *To answer RQ1, we conclude that non-essential communication often occur in real-world applications: 69% of the triggered connection statements can be deemed non-essential.*

The last column of Table II presents statistic considering the communication with known A&A services. It turns out that only 48% of all non-essential connections target such libraries. That is, more than half of non-essential connections originate from application-specific libraries and their purpose is unknown to us.

2) 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 [10], where the authors show that blocking all messages to A&A services made more than 60% of the applications either less functional or completely dysfunctional. Moreover, some malicious applications deliberately hide their payloads by using package names which look legitimate and benign [12]. We conclude that a more sophisticated technique 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. We noticed that, in a large number of cases, connection successes do not trigger any notifications in the application's user interface and, at the same time, connection failures are also silently ignored by the applications without producing any visual indication to the user. For the failures, the exception triggered by 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 warning 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.

To answer RQ2, we conjecture that non-essential connections can be detected by inspecting updates to GUI elements on both the success and the failure path of a connection statement. Lack of such GUI updates is indicative for a connection being non-essential 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 *essential* or *non-essential*. We begin with machinery that allows us to define essential and non-essential connection statements with precision.

Definition (Rethrown Exception). A rethrown exception occurs when a `catch` block catches an exception, but before the block is exited, a statement reachable from the block explicitly throws the same exception object, or throws a new exception object. The process of searching the stack for a handler begins anew.

Definition (Failure Handling). The failure handling of a connection call s for exception type e is defined as the execution path that starts when an exception of type e propagates to connection call s and ends when the *last* `catch` block is exited that handles e or a rethrown exceptions of e.

Intuitively, the failure handling of s on e is the computation that handles e and any failure 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.

Definition (Essential and Non-essential Connections). For connection statement s, s is classified as an *essential connection* if meets at least one of the following criteria:

1) **User-Interface Cue on Failure**: When s triggers an exception, the user may be notified of the failure via a user interface cue during failure handling.
2) **Program Exit on Failure**: When s triggers an exception, the program may stop executing due to a failure handling path not terminating at an application event handler method.
3) **User-Interface 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 non-essential connection call does not meet any of the criteria.

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 [1]. Static analysis must either model the Android application execution environment, or account for possible dynamic program behaviors with conservative analysis choices; otherwise some runtime behaviors could be unconsidered. Our static analysis over-approximates the runtime behaviors of the applications, and under-approximates the connection calls that could be non-essential.

Our analysis is implemented in the Soot Java Analysis Framework [13] and utilizes libraries and the Android API model provided by DroidSafe [1]. The presentation of the analysis below assumes the application is represented in the Jimple intermediate language [13].

The design of our analysis prioritizes high precision over high recall because we do not want to block connection calls that perform essential communication. Our analysis employs a class hierarchy analysis (CHA) [14] to build a call graph with refinement achieved by intra-procedural data-flow analysis. After much experimentation with higher precision, though brittle, points-to analysis techniques, this analysis combination gave us the best performance for the classification task.

To compute a call graph, we augment the application code with the DroidSafe Android Device Implementation (ADI) [1]. The ADI is a Java-based model of the Android runtime and API that attempts to present full runtime semantics for commonly-used classes of the runtime and API. Our call graph construction does not traverse into Android API methods. However, we found it necessary to account for API calls that immediately jump back into the application. For example, if an application method, m calls `Thread.start()` on a receiver that is an application class, t, we found it necessary to add the edge to the call graph (m, t.`run()`). This includes the started thread t in failure handler if m is encountered.

To achieve this in general, we add to the call graph edges of the type (m, n) where there is an edge (m, `api-method`), the call of `api-method` is passed a value that is a reference to an application class, and `api-method` calls method n on the passed application class value. This strategy adds to our callgraph the edges for the `Thread.start()` to the `Thread.run()` discussed above.

Furthermore, the call graph is augmented to account for reflected method calls in the application 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 [15], 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.

The static analysis is performed in three steps for each connection statement: (1) failure handler analysis, (2) success forward analysis, and (3) success backward analysis. The analysis will terminate if a step determines that the connection is essential. Otherwise, a connection is considered non-essential.

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) ∈ visiting —— (stmt, ex) ∈ essential then
3:         return
4:     end if
5:     visiting ← visiting ∪ (stmt, ex)
6:     catchBlockStart ← FINDCOMPATCATCH(meth, stmt, ex)
7:     if catchBlockStart = null then
8:         if ISEVENTHANDLER(meth) then
9:             essential ← essential ∪ (stmt, ex)
10:            return
11:        end if
12:        for (predStmt, predMeth) ∈ GETPREDS(cg, meth) do
13:            if stack ≠ ∅ and (predStmt, predMeth) ≠ PEEK(stack) then
14:                continue
15:            end if
16:            newStack ← stack
17:            POP(newStack)
18:            FINDCATCHES(predMeth, predStmt, ex,
19:                                visiting, newStack, cg)
20:            if (predStmt, ex) ∈ essential then
21:                essential ← essential ∪ (stmt, ex)
22:                return
23:            end if
24:        end for
25:    else
26:        catchStmts ← GETCATCHSTMTS(catchBlockStart, meth)
27:        ANALYZEHANDLING(meth, stmt, catchStmts,
28:                            visiting, ∅, stack, cg)
29:    end if
30: end procedure
```

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

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

Fig. 3. Analyze reachable statements during handling for UI interaction or rethrown exceptions.

Table I lists the target methods that we consider connection calls and each method's associated failure exception. The set of target methods that are considered as affecting the user interface are listed in Table III. We also define all overriding methods of the methods listed in the table as UI methods.

B. Failure Handler Analysis

Failure analysis determines if a connection call modifies the user interface or could exit the application on exception. We organize the static failure-handling analysis as a recursive traversal on the call graph for ease of understanding. 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.

The analysis starts with the FINDCATCHES procedure listed in Fig. 2. For each statement and exception pair, s and e, respectively, the procedure first consults s's containing method to find an appropriate `catch`; if e is not caught locally, the analysis recursively visits all direct predecessors of the method to find `catch` blocks that trap the call statement edge (lines 7-23). For each predecessor, p, if a catch is not found that wraps the call edge, then p's direct predecessors are visited, and so on.

For each `catch` that is found during the FINDCATCH, ANALZYEHANDLER of Fig. 3 analyzes the reachable statements of the handler. The analysis considers the reachable statements inter-procedurally and flow-insensitively. Handler analysis searches for: (1) calls to application methods, (2) `throw` statements (3) calls to native methods, and (4) possible calls to UI methods. 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 16-21). If analysis finds a `throw` statement, the handler analysis spawns a new FINDCATCHES analysis to find all the possible handlers of each rethrown exception (lines 25-44). If analysis finds a call to a native method, we assume that it will throw all exceptions it is defined to throw, handler analysis spawns a FINDCATCHES instance for each exception declared throws (line 12). If a call is encountered that could target a UI method, then the statement that began the handler analysis is considered essential since the failure handling affects the user interface (line 9).

A stack of pairs of method call statement and method is maintained during the analysis. 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. 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.

During handler search in FINDCATCHES, if no handler is found locally, and the method is a possible entry point called from the Android runtime, then we conservatively calculate that the exception and excepting statement could cause application exit, so the pair is added to the essential set (line 8 of FINDCATCHES).

During handler analysis, if a `throw` statement is encountered in reachable code of a handler, the analysis needs to determine the possible type of the thrown value, and then start a new search for the handler. The ANALYZEHANDLER procedure calculates local def-use chains for the method it is analyzing. It uses the local def-use information to calculate the types of the exception. In lines 25 through 37 the analysis considers all local reaching definitions of the thrown value. If an allocation statement reaches, then add the allocated types to the possible types of rethrown exceptions. If a caught exception statement[1], 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 GETPOSSIBLETHROWNTYPES call on line 32 of ANALYZEHANDLER.

If any other type of statement is a definition that reaches the thrown value, then the analysis cannot determine the exception type and the connection call (or rethrown exception statement) is considered essential (line 33). 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.

Fig. 4 lists the helper procedures employed in the failure handling analysis.

C. Success Analysis

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

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

[1]A caught exception statement is a statement that defines that start of a `catch` block and assigns a local variable to the exception object caught by the block.

GETPOSSIBLETHROWNTYPES($meth$,$stmt$): Calculate the possible exception types caught at the catch block that begins with $stmt$ of $meth$. The method calculates the `try` block that associates with the `catch` block that encloses $stmt$. The procedure examines `throw` and call statements of the `try`. For a call statement, the procedure adds to the return list all exception types declared throws by all methods that the call can target. 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.
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.
ISEVENTHANDLER($meth$): Return true if method $meth$ overrides a method defined in the Android API. This method over-approximates the methods that can be called by the Android runtime to handle events.
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.
GETTRYBLOCK($meth$,$stmt$): Given a statement $stmt$ that begins a `catch` block in method $meth$, return the list of statement of try block associated with the enclosing `catch` block of $stmt$.

Fig. 4. Failure Analysis Helper Functions.

1) Success Forward Analysis: Code reachable from the statement immediately after s is searched for a connection call. This is accomplished by traversing all paths in the interprocedural control flow graph (CFG) starting at s and following method invoke expressions. We follow both normal and exceptional control paths when analyzing the CFG. The analysis does not follow call graph edges into the Android API.

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

D. Design Discussion and Limitations

The intuition for our static analysis is that essential connection calls *always* affect the user interface either on success or failure. Furthermore, the effect is *direct* on success. As a marker for the notion of a *direct effect*, an execution trace that handles a single runtime event naturally denotes a unified block of processing, so our analysis reasons about event handlers to bound the success search to represent the direct processing of a connection call.

Another insight gleaned from examination of the applications in the study in Section II is that direct processing associated with a successful invocation of connection call s in method m ends when the successful control flow merges back with the exceptional control flow. This is the motivation for why the success backward analysis only examines methods that were analyzed by the failure analysis. We assume that for all stacks of m, methods beyond the failure analysis do not contribute to processing of s.

TABLE IV. COMPARISON WITH THE MANUALLY ESTABLISHED RESULTS.

Applications	Correctly detected non-essential		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%)	1/3 (33.3%)	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	23/25 (93.2%)	23/52 (60.0%)	2min 48s

The presented analysis has the following limitations. Dynamically loaded code is not considered. The analysis considers only checked exceptions. We use a best-effort, though aggressive, policy to account for reflection semantics; this policy could miss possible runtime semantics.

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 non-essential 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 non-essential are disabled.

1) Accuracy: For the accuracy evaluation, we look again at the applications listed in Table II, excluding facebook and candycrush because these two applications did not exhibit any non-essential 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 non-essential among those reported by the technique.
- *Recall*: the fraction of connection statements correctly identified as non-essential among those expected, i.e., marked as non-essential 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 non-essential 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 GUI updates for these icons happen after the success and failure paths unify, and are thus 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 60% of statements deemed non-essential 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 *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 Table IV.

2) Usability Assessment: To check whether our technique is able to provide actionable results, we further select 100 applications that persisted in the list of the 500 most popular free applications on Google Play in November 2014 and May 2015 samples. We used two Nexus devices running Android v4.4.4, installing the original version of an applications on one. On the other, we installed a modified version that was produced by employing the *blocking transformation* (see Section II) to disable all calls identified as non-essential 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 10 minutes, and to record all differences observed during the execution. 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.

To analyze the results of that experiment in a reliable manner, we exclude 14 applications that were non-operational (either did not run, in both versions, 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 Play applications that we could not re-install on a device; 5 chat applications; 4 applications that either contained no connection statements or had no non-essential connection statements detected; and 11 applications for which no non-essential 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.

TABLE V. TOP 10 NON-ESSENTIAL COMMUNICATION CALLERS.

	Package	Description	Used in # (%) of Apps	Non-essential 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%)

Missing advertisement: 9 (19.2%). Similarly to the zedge example described above, advertisement information was missing in 9 cases.
Missing minor functionality: 3 (6.4%). The participants observed absence of features that they perceived as minor: 2 cases of missing icons, in the flashlight and the talkingben applications, and 1 case were they could not 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 might me related to resigning issues, but we could not verify that.

On average, 2.6 non-essential call statement per an application were triggered at runtime (min: 1, max: 9, mdn: 2). Counting all dynamic call instances of these statements gives us the average of 299 non-essential call instances per an application (min: 1, max: 4011, mdn: 11). High average numbers are due to applications that, once installed, are constantly executed in the background, and, as it turns out, attempt to communicate with the net. 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 non-essential connections. The technique is precise, highly scalable and provides actionable output that can be directly used for disabling non-essential communication in a vast majority of cases.

B. Non-Essential Communication in the Wild

We next apply our technique on the 500 most popular Android application downloaded from the Google Play store. By considering such a large data set, our goal is to investigate how often non-essential 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 non-essential (8,539 connection out of 18,480 in total). These results are consistent with the observation of our empirical study described in Section II.

Table V presents the top 10 packages in which non-essential connections occur. As the numbers are aggregated for 500 applications, it is no surprise that Google services, as well as gaming, advertisement and analytics services, are on the top of the list – numerous applications use these services, as shown in the third column of Table V. More surprising is the *com.gameloft* package (row 2 of Table V) that is part of only 17 different mobile applications by the same company. Yet, the number of non-essential connection statements these game applications contain is notable.

The last column of Table V shows the percentage of non-essential 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 use to determine its impact on the application behavior.

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

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. We attempted to mitigate this problem by performing 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.

During our analysis, we disabled connections one by one, iterating over their list arranged in a lexicographic (i.e., semantically random) order. As such, we could miss cases when several connections can be excluded altogether, but not individually. Since exploring all connection state combinations is exponential, we opted for this linear approach that still guarantees correct, albeit possibly over-approximate results.

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 essential if it is essential for at least one of such behaviors.

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 Non-Essential Connections: Our technique deems as non-essential *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 the future work.

Some of the non-essential 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 in Mobile Applications: Huang et al. [16] 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 [17] 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. [18] 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 [2], 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 [19], [1], [20], [21]. 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 [10] 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 [22] and AppContext [23] build up on the FlowDroid static information flow analysis system [19] 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. [24] 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.

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 [25], [26], [27], [28], [29], [30], [31]. 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 [27], [28], [30]), 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

Non-essential 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 non-essential 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 non-essential communication and promote the development of more transparent and trustworthy mobile applications.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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)*, 2014, pp. 175–190.

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

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

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

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

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

[10] 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.

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

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

[13] R. Vallée-Rai, E. Gagnon, and L. Hendren, "Optimizing Java bytecode using the Soot framework: Is it feasible?" *CC*, 2000.

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

[15] A. Aho, M. Lam, R. Sethi, and J. Ullman, *No TitleCompilers: Principles, Techniques, and Tools*, 2nd ed. Addison Wesley, 2006.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] Byeong-Mo Chang, Jang-Wu Jo, and Soon Hee Her, "Visualization of exception propagation for Java using static analysis," in *Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE Comput. Soc, 2002, pp. 173–182. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1134117

[26] B.-M. Chang, J.-W. Jo, K. Yi, and K.-M. Choe, "Interprocedural exception analysis for Java," in *Proceedings of the 2001 ACM symposium on Applied computing - SAC '01*. New York, New York, USA: ACM Press, Mar. 2001, pp. 620–625. [Online]. Available: http://dl.acm.org/citation.cfm?id=372202.372786

[27] 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.

[28] C. Fu and B. G. Ryder, "Exception-chain analysis: Revealing exception handling architecture in Java server applications," in *Proceedings - International Conference on Software Engineering*, 2007, pp. 230–239.

[29] 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.

[30] X. Qiu, L. Zhang, and X. Lian, "Static analysis for java exception propagation structure," in *Proceedings of the 2010 IEEE International Conference on Progress in Informatics and Computing, PIC 2010*, vol. 2, 2010, pp. 1040–1046.

[31] G. Kastrinis and Y. Smaragdakis, "Efficient and effective handling of exceptions in java points-to analysis," in *Compiler Construction*, 2013. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-37051-9_3