Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2015-015 May 4, 2015

Non-Essential Communication in Mobile Applications

Julia Rubin, Michael I. Gordon, Nguyen Nguyen,
and Martin Rinard

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

CSAIL

Non-Essential Communication in Mobile Applications

Julia Rubin[1], Michael I. Gordon[1], Nguyen Nguyen[2], and Martin Rinard[1]
[1]Massachusetts Institute of Technology
[2]Global InfoTek, Inc

ABSTRACT

This paper studies communication patterns in mobile applications. Our analysis shows that 65% of the HTTP, socket, and RPC communication in top-popular Android applications from Google Play have no effect on the user-observable application functionality. We present a static analysis that is able to detect non-essential communication with 84% -90% precision and 63%-64% recall, depending on whether advertisement content is interpreted as essential or not. We use our technique to analyze the 500 top-popular Android applications from Google Play and determine that more than 80% of the connection statements in these applications are non-essential.

1. INTRODUCTION

Mobile applications enjoy almost permanent connectivity and the ability to exchange information with their own back-end, third-party servers and other applications installed on the same device. 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. But this communication comes with costs such as bandwidth charges, power consumption on the device, potential privacy and analytic data release, and the unsuspected presence of continued communication between the device and remote organizations. We have even observed applications that silently spawn services that communicate with third-party servers *even when the application itself is no longer active*, with the user completely unaware that the spawned services are still running in the background.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 identifying and disabling non-essential communication in widely used mobile applications such as ten of the top fifteen most popular applications in the Google Play App Store (twitter, Wal-Mart, Spotify, Pandora, CandyCrush, etc.). Motivated by the significant amount of non-essential communication we found in these applications, we next developed a static analysis that can automatically identify non-essential communication and used this analysis in our further investigation of this unfortunate phenomenon. 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 ten top-popular applications in Google Play. We focus on the three most common connection types: HTTP, socket and RPC. The former two are used to communicate with various back-end servers – the application's own and third parties'; the latter one is used to communicate with other applications and services running on the same device.

Baseline Behavior: We first 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, we capture a screenshot of the device to record the application state.

Instrumentation: We next instrument the application to log information about triggered connection statements. The instrumented version of the application is then installed and executed on a mobile device using the recorded script.

Disable Connections: We disable each triggered connection in turn by replacing the statement that establishes the connections with a statement that throws an exception that indicates that the connection failed because the device was in a disconnected mode.

Run Modified Application: We install the modified application and run it using the previously recorded script. Similarly to the approach in [18], the screenshots documenting the execution of the modified application are compared to those of the original one. We consider executions as equivalent if they result in screenshots that differ only in the content of advertisement information, messages in social network applications such as twitter, and the device's status bar. We also separately note connections that contribute to presenting advertisement content, if the analyzed application has any.

Result Summary: Our study reveals that around 65% of the exercised connection statements are not essential — disabling them has no noticeable effect on the observable application functionality. Slightly more than 25% of these correspond to HTTP and socket communication. The rest correspond to RPC calls to internal services installed on the device: notably, but not exclusively, Google advertising and analytics, which further communicate with external services. Moreover, in applications that present advertisement material, about 60% of the connections that do affect the observable application behavior are used for advertising purposes only.

RQ2: Can non-essential communication be detected statically? Inspired by our findings, we develop a novel static application analysis that can detect connection failures that are "silently" ignored by the application, i.e., when information about a connection failure is not propagated back to the end user. The static analysis classifies each connection call by inspecting the execution of the application during *failure handling* of the connection call. Failure handling begins when the exception is propagated to the connection call and ends when the execution exits the exception handler of the exception or the handlers of all rethrown exceptions that are raised during handling. If a failure handling exception could affect the user interface through a call to a predefined set of API calls, we classify the connection call to be *essential*. We classify it as *essential* also if there is a failure handing path that could exit the program, because a thrown exception propagates back into the Android runtime.

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 [16].

There are two special cases that our technique is not designed to handle: (1) *optional* behaviors, for which failing connections are silently ignored, but successful connections result in presenting additional information to the user; advertisement content usually falls into that category. (2) *stateful* communication, for which failures leave the connection target in a state different from the one it has after a successful communication, and further communication is influenced by the server's state. Our experiments show that such cases are rare.

RQ3: How well does static detection perform? To assess the quality of our technique, we evaluate it on the "truth set" established during our empirical analysis of applications from Google Play. The results show that it features a high precision – 83% of the identified connection (64 out of 82) are indeed classified as non-essential during the manual analysis. Even though it is designed to be conservative, it is still able to identify 64% of all non-essential connection (64 out of 106). There are 18 connections in total that are miss-classified as non-essential. Out of these, 16 correspond to optional application behaviors and the remaining 2 – to stateful communication. Counting advertisement content as non-essential gives the overall precision and recall of 90% and 65%, respectively.

RQ4: How often does non-essential communication occur in real-life applications and what are its most common destinations? Applying the analysis on the top 500 popular applications from Google Play reveals that 84% of connection sites encoded in these applications can be deemed non-essential. Most common target of non-essential communication are various Google services for mobile developers. We conjecture that applications commonly register for various such services without eventually using them. Additional common targets are advertisement, analytics and gaming services.

Significance of the Work. Our work focuses on benign mobile 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 [13, 11, 28] 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 release of information by mobile applications 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 releases of information in Android applications which does not require access to the application source code. The approach relies on interactive injection of connection failures and identification of cases in which the injected failures do not 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 65% of the connections attempted by ten top-popular free applications on Google Play fall into that category.

4. It proposes *a static technique* that operates on application binaries and identifies non-essential connections – those where failures are not propagated back to the application's user. The precision and recall of the technique is 83% and 63%, respectively, when evaluated against the empirically established truth set. The precision and recall increases to 90% and 64%, receptively, when considering the advertisement content as non-essential.

5. It provides *quantitative evidence* for the prevalence of non-essential connections in the 500 top-popular free applications on Google Play, showing that 84% of connections encoded in these applications can be deemed as non-essential.

2. 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.

2.1 Design of the Study

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

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

Application Instrumentation. As input to our study, we assume an Android application given as an apk file. We use the dex2jar tool suite [10] to extract the jar file from the apk. We then use the asm framework [5] to implement two types of transformations:

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

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

Table 1: Considered Connection Statements.

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

Table 2: Analyzed Applications.

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

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 [2] that runs on the device and captures all user and kernel input events to capture a sequence of events that exercise an application behavior. We make sure to pause between user gestures that assume application response. We then enhance the script produced by getevent to insert a screen capturing command after each pause and also between events of any prolonged sequences. We upload the produced script onto the device and run it for each version of the application.

We deliberately opt not to use Android's UI/Application Exerciser Monkey [3] tool. While this tool is able to generates a repeatable sequence of pseudo-random streams of events such as clicks, touches and gestures, in our case, it was unable to provide a reasonably exhaustive coverage of application functionality. Even for applications that do not require entering any login credentials, it quickly locked itself out of the application by generating gestures that the analyzed application cannot handle. We thus have chosen to manually record the desired application execution scenario, which also included any "semantic" user input required by the application, e.g., username and password.

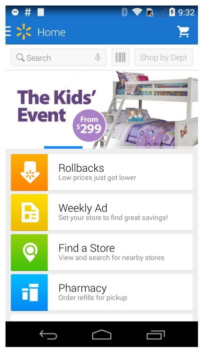
For the comparison of application executions, we started by following the approach in [18], where screenshots from two different runs are placed side-by-side, along with a visual diff of each two corresponding images, as shown in Figure 1, for the Walmart and twitter applications. We used the ImageMagick compare tool [20] 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 screenshots in Figures 1(a) and (b) similar. We also ignored the exact content of widgets that are populated by applications in a dynamic manner and are designed to provide continuously updated information that is expected to differ between applications runs, such as tweets in Figures 1(d) and (e). These two figures are thus also deemed similar.

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

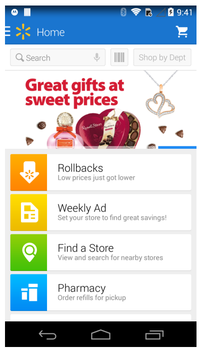
Execution Methodology. We performed our study in three phases. In the first phase, we installed the original version of each analyzed application on a Nexus 4 mobile device running Android version 4.4.4. We manually exercised the application, exploring all its functionality visible to us, and recorded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

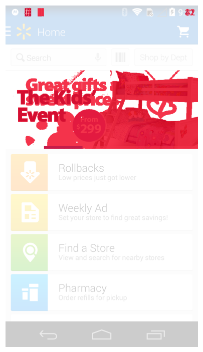
In the second phase, we used the Monitoring Transformation to produce a version of the original application that logs information about all existing and triggered connection statements. We ran the produced version using the execution script and collected the statistics about its communication patterns.

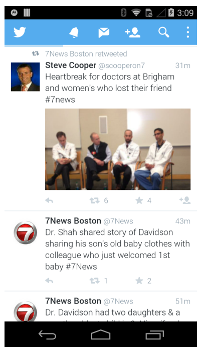
In the third phase, we iterated over all *triggered* connection statements, disabling them one by one, in order to assess the necessity of each connection for preserving the user-observable behavior of



(a) Screen 1.


Home
Search
Shop by Dept
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
(b) Screen 2.

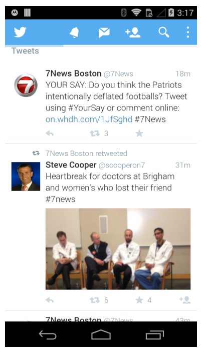

(c) Difference for Screens 1 and 2.


7News Boston retweeted
Steve Cooper @scooperon7 31m
Heartbreak for doctors at Brigham and women's who lost their friend #7news

7News Boston @7News 43m
Dr. Shah shared story of Davidson sharing his son's old baby clothes with colleague who just welcomed 1st baby #7News

7News Boston @7News 51m
Dr. Davidson had two daughters & a

(d) Screen 3.


Tweets

7News Boston @7News 18m
YOUR SAY: Do you think the Patriots intentionally deflated footballs? Tweet using #YourSay or comment online: on.whdh.com/1JfSghd #7News

7News Boston retweeted
Steve Cooper @scooperon7 31m
Heartbreak for doctors at Brigham and women's who lost their friend #7news

(e) Screen 4.

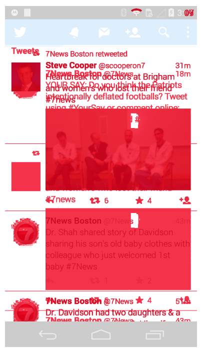

(f) Difference for Screens 3 and 4.

Figure 1: Visual differences.

Table 3: Communication Types.

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

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.

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

The remaining ten applications are listed in the first column of Table 2; their corresponding sizes are given in the second column of the table. We did not extend our dynamic analysis beyond these ten applications because the inspection of our findings indicated that we reached saturation: while it is clearly infeasible to explore all possible scenarios, we observed similar trends in all analyzed applications. As such, inclusion of additional ones was not expected to provide substantially new insights.

2.2 Results

The quantitative results of the study are presented in columns 3–7 of Table 2. 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, advertising and analytics libraries and more. In fact, we identified nine different advertising and analytics libraries used by the ten applications that we analyzed, and many times a single applications uses multiple such libraries.

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

Classification of the Triggered Statements. Column 5 of Table 2 shows the number of connection statements that we determined as non-essential during our study. Averaged for all applications, 65% of the connections fall in that category. This means that only 35% of the connection statements triggered by an application affect its observable behavior, when executed for the exact same scenario with the connection being either enabled or disabled (see column 6 of Table 2).

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

To answer RQ1, we conclude that non-essential communication often occur in real-world applications: 65% of the triggered connection statements can be deemed non-essential.

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

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

Lessons Learned. The collected statistics show that no principle distinction between essential and non-essential connections can be made just by considering connection types and their destinations. That observation is consistent with findings in [18], where authors show that blocking all messages to advertising and analytics services made more than 60% of the applications either less functional or completely dysfunctional. We conclude that a more sophisticated 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 of each of the identified type and communication target. We noticed that, in a large number of cases, connection failures are silently ignored by the applications without producing any visual indication to the user. That is, the exception triggered by the connection failure of a non-essential connection is either caught and ignored locally in the method that issues the connection or, more commonly, propagated upwards in the call stack and then ignored by one of the calling methods.

In several cases, an error or 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 connection failure paths. The lack of updates to GUI elements on the failure path is indicative for a connection being silently ignored by the application, thus being non-essential for the application execution.

```
1  public class ApplicationClass {
2      void f() {
3          try {
4              g();
5              stmtA;
6          } catch (AdvertisingException e) {
7              stmtB;
8          }
9          stmtC;
10     }
11 }
12 public class AdvertisingAPIClass {
13     void g() throws AdvertisingException {
14         try {
15             stmtD;
16             connect();  //throws RemoteException
17             stmtE;
18         } catch (RemoteException e) {
19             stmtF;
20             throw new AdvertisingException();
21         }
22         stmtG;
23     }
24 }
```

Figure 2: An example of failure handling.

3. FAILURE-HANDLING ANALYSIS

In this section we describe the static analysis algorithm we employ to automatically classify connections. Given an Android application, the static analysis classifies each statement that may invoke a connection call as either *essential* or *non-essential*. Intuitively, we define an essential connection statement, s, as meeting either of the following criteria:

1. **User-Interface Cue**: When s triggers an error, the user may be notified of the error via a user interface cue during error handling.
2. **Program Exit**: When s triggers an error, the program may stop executing.

Conversely, a non-essential connection call does not meet either of the two criteria.

Android applications are developed in Java, and program execution follows the semantics of Java. In an Android application, each connection call s may generate an exception (of dynamic type e) that reaches a subset of the program's `catch` blocks. At runtime, when e is triggered by s, the executing method's trap table is consulted, and if no `catch` blocks are defined to handle e at s, then e is passed back up the stack to the calling method at the calling statement, and the process repeats. If the Android runtime is returned to during the stack unwinding, the application is typically exited with an error.

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.

Figure 2 give a simplified representation of failure handling pattern that we observed in the twitter application. Method `f` invokes method `g`. In `g`, a connection call is encountered on line 16; assume during execution this connection call throws a `RemoteException`. The failure handling for this connection call and exception is the set of statements: *stmtF*, `throw new AdvertisingException()`, and *stmtB*. These statements are executed from the start of the handling of thrown `RemoteException` to the end of the handling of the rethrown `AdvertisingException`.

PROBLEM: Analyze each connection call, s, in an Android application to determine whether the application could possibly exit on a failure at s or could modify the user interface during a failure handling path of s.

To solve this problem statically, our failure-handling analysis conservatively calculates all possible failure handling for exceptions that denotes connection failure for each connection call in the application. If there exists a failure-handling path of s on e that may include a call to a method that notifies the user of the failure, then s is considered essential. If it is possible for e, when triggered at s, to propagate back up the stack to the Android runtime, s is considered essential.

3.1 Static Analysis of Android Applications

Static analysis of Android applications is notoriously difficult because of issues including [16]:

- Android applications execute in the context of the Android API and runtime. The application thus represents an incomplete program.
- The Android API and runtime comprises multiple millions of lines of code implemented in multiple programming languages. Furthermore, much of the implementation is left for device manufactures to implement, and is thus proprietary and closed-source.
- Applications are event-driven and dynamic by nature. Applications define event handlers for possible runtime events that are triggered in the Android runtime, and passed to the application for handling.
- Applications interact heavily with the Android API. The Android API includes most of the Java standard library, plus additional utility and resource access classes.
- Android application packages typically ship with third-party libraries for performing operations such as advertising, analytics, and interaction with remote services. These libraries are commonly large, obfuscated, and include heavy use of reflection.

It is not feasible for a static analysis to include analysis of the source code of the Android runtime and API because of the size and multi-language nature of this code base. Thus 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. Precise, whole-program analysis runs the high-risk of missing dynamic program behavior and not scaling to real-world Android application [16].

Our analysis employs a class hierarchy analysis (CHA) [9] 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. We augment the call graph to account for reflected method calls, and conservatively account for exceptions that can be thrown by native code. Our failure-handling analysis over-approximates the runtime behaviors of the applications, and under-approximates the connection calls that could be non-essential.

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

Our analysis is implemented in the Soot Java Analysis Framework [29] and utilizes libraries and the Android API model provided by DroidSafe [16]. The presentation of the analysis below assumes the application is represented in the Jimple intermediate language [29].

3.2 Call Graph Construction

Our algorithm first computes a static call graph based on CHA analysis. To compute a call graph, we augment the application code with the DroidSafe Android Device Implementation (ADI) [16]. 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 [1], 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.

3.3 Failure Handler Analysis

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. Table 1 lists the target methods that we consider connection calls and each method's associated failure exception.

The analysis starts with the FINDCATCHES procedure listed in Figure 3. For each start of the analysis on a 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, handler analysis of the reachable statements of the **`catch`** block is performed (FINDCATCH, line 26). Figure 4 gives the listing of the

```
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
```

Figure 3: Find `catch` blocks for exception thrown at statement.

handler analysis procedure ANALZYEHANDLER. 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 recursive 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 error handling affects the user interface (line 9).

FINDCATCHES and ANALYZEHANDLER maintain a set of statement and exception pairs, essential, that records pairs that are calculated as essential. After all connection call statement and exception pairs are analyzed, pairs not in the essential set are considered non-essential.

The set of target methods that are considered as affecting the user interface are listed in Table 4. We also define all overriding methods of the methods listed in the table as UI methods.

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. The stack is pushed on line 18 of ANALYZEHANDLER for each method call of a reachable handler code. During the handler search of the execution stack in

```
 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
```

Figure 4: Analyze reachable statements during handling for UI interaction or rethrown exceptions.

FindCatches, the stack is consulted to guide the search on line 13, only visiting the edge is at the head of the stack. The stack is popped when visiting a caller method of the current method in FindCatches line 17.

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. The classification of the thrown statement is propagated to the current excepting statement in line of AnalyzeHandler.

Figure 5 gives the algorithm for GetPossibleThrownTypes. First, the method calculates the `try` block that associates with the `catch` block that encloses *stmt*. Next, 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

[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.

```
 1: procedure GetPossibleThrownTypes(meth,stmt)
 2:     thrownTypes = []
 3:     tryStmts = GetTryBlock(meth,stmt)
 4:     for tryStmt ∈ tryStmts do
 5:         if IsThrowStmt(tryStmt) then
 6:             for defStmt ∈ GetLocalDefs(GetOp(stmt)) do
 7:                 if IsAlloc(defStmt) then
 8:                     rethrownTypes ⟵ rethrownTypes ∪
 9:                                 GetAllocType(defStmt)
10:                 else if IsCaughtExceptionStmt(defStmt) then
11:                     reThrownTypes ⟵ rethrownTypes ∪
12:                                 GetPossibleThrownTypes(meth, defStmt)
13:                 else
14:                     return ∅
15:                 end if
16:             end for
17:         else if HasInvoke(tryStmt) then
18:             for (succStmt, succMeth) ∈ GetSuccs(cg, tryStmt) do
19:                 thrownTypes ⟵ thrownTypes ∪
20:                             GetThrowsExceptions(succMeth)
21:             end for
22:         end if
23:     end for
24:     return thrownTypes
25: end procedure
```

Figure 5: Calculate exception types caught at statement.

Table 4: 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`

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 the procedure returns null to denote that it cannot calculate the thrown type (line 14). This situation causes the examined connection call or rethrown statement in ANALYZEHANDLER to be labeled essential (we have not included the code in the algorithms to propagate this situation to the essential set, though it is in our implementation).

3.4 Helper Procedures

The analysis employs the following helper procedures:

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 $stmt$. Since the dex bytecode does not provide the ending statement of traps, we need to calculate the extent of the catch block. GETCATCHSTMTS takes advantage of the property that Java compilers do not produce code that jumps from outside of the catch block into the middle of a catch block. So to calculate the `catch` block's extent, GETCATCHSTMTS (1) produces a control flow graph (CFG) for $meth$, (2) colors all statements reachable from $stmt$ in the CFG, (3) for each statement, c of $meth$, if all predecessors of c in the CFG are colored then c is included in the set of statements that are returned ($stmt$ is also included in the return set). 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 $stmt$.

4. EXPERIMENTS

To establish the quality of our static analysis algorithm, we evaluate it on the "truth set" established during our in-depth case study (see Section 2). We then use it to gather information and report on non-essential communication in the 500 top-popular Android applications from Google Play.

4.1 Evaluation of the Static Analysis

For our evaluation, we limit the set of results reported by the static analysis to those that were, in fact, triggered during our dynamic study (see Table 2 in Section 2): these are the connection statements for which we have reliable information to compare against. We assess the results, for each application individually and averaged for all applications, using the metrics below. The results are summarized in Table 5.

1. *Expected*: the size of the predetermined expected result, i.e., the number of connections listed as non-essential in Table 2.
2. *Reported*: the number of connections deemed as non-essential by the static analysis.
3. *Correct*: the number of non-essential connections correctly identified by the static analysis, i.e., those that were deemed as non-essential in the dynamic study as well.
4. *Precision*: the fraction of relevant results among those reported, calculated as $\frac{Correct}{Reported}$.
5. *Recall*: the fraction of relevant results among those expected, calculated as $\frac{Correct}{Expected}$.
6. *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.

As can be seen in the second and the third columns of Table 5, the overall averaged precision of our analysis is 83.3%. The analysis correctly identified 64 non-essential connections out of the total 82 reported. 18 connections were mis-classified, out of which 16 correspond to *optional* application behaviors, i.e., when connection failures are indeed ignored and the applications proceed without the missing information. The remaining two cases correspond to a *stateful* communication within the application. The details of each of these cases are given below:

- 13 connections are used for presenting *optional* advertisement content: 3 in the *net.zedge.android* application, and 5 in *com.crimsonpine.stayinline* and *com.grillgames.guitarrockhero* each.
- 3 connections correspond to *optional* application behaviors: 2 in the *com.walmart.android* application, responsible for providing location-aware search, and 1 in the *com.spotify.music* application, responsible for enhancing the presented album with images.
- 2 connections, both in *com.spotify.music*, correspond to *stateful* communication within the application. Blocking each of these connections, individually, harms the application's search capabilities.

Considering the advertisement information as non-essential gives an overall average precision of 90.8%, as shown in column 4 of Table 5. That is, depending on the user's perspective, between 83% and 90% of the cases identified as non-essential by the static analysis indeed do not affect the behavior of the applications.

While designed to be conservative, our analysis is able to correctly identify 64 out of 106 connection statements deemed non-essential in the empirical study, resulting in the overall recall of 63.8% (column 3 in Table 5). Considering advertisement non-essential results in a slight increase in recall, to 64.6% (column 5 in Table 5). The technique correctly identifies the majority of non-essential connections.

Finally, the last column of table Table 5 shows that our analysis is highly efficient – it runs in a matter of minutes even on large applications.

To answer RQ3, we conclude that a static analysis can be applied for an accurate detection of non-essential connections. Our highly scalable technique features up to 90% precision and 64% recall, respectively.

4.2 Non-Essential Communication In the Wild

We apply our technique on the 500 most popular Android application downloaded from the Google Play store in January 2015. Our goal is to investigate how often non-essential communication occurs in real-life applications and what are its most common destinations.

Table 5: Comparison with the Manually Established Results.

Applications	Correctly detected non-essential		Correctly detected non-essential (counting ads as non-essential)		Execution time
	Precision	Recall	Precision	Recall	
air.com.sgn.cookiejam.gp	1/1 (100.0%)	1/3 (33.3%)	1/1 (100.0%)	1/3 (33.3%)	1min 50s
com.crimsonpine.stayinline	13/18 (72.2%)	13/13 (100.0%)	18/18 (100.0%)	18/18 (100.0%)	1min 52s
com.grillgames.guitarrockhero	17/22 (77.3%)	17/30 (56.7%)	22/22 (100.0%)	22/35 (62.9%)	2min 54s
com.king.candycrushsaga	3/3 (100.0%)	3/3 (100.0%)	3/3 (100.0%)	3/3 (100.0%)	1min 53s
com.pandora.android	4/4 (100.0%)	4/9 (44.4%)	4/4 (100.0%)	4/9 (44.4%)	2min 13s
com.spotify.music	4/7 (57.1%)	4/21 (19.0%)	4/7 (57.1%)	4/21 (19.0%)	2min 18s
com.twitter.android	4/4 (100.0%)	4/6 (66.7%)	4/4 (100.0%)	4/6 (66.7%)	2min 33s
com.walmart.android	3/5 (60.0%)	3/5 (60.0%)	3/5 (60.0%)	3/5 (60.0%)	2min 17s
net.zedge.android	15/18 (83.3%)	15/16 (93.8%)	18/18 (100.0%)	18/19 (94.7%)	2min 31s
Totals (average)	64/82 (83.3%)	64/106 (63.8%)	77/82 (90.8%)	77/119 (64.6%)	2min 15.7s

Table 6: Top 20 Non-Essential Communication Callers.

		Description	Used in # (%) of Apps
1.	com.google.android.gms	Google mobile services	403 (80.6%)
2.	com.facebook	Facebook services	190 (38.0%)
3.	com.android. vending.billing	Google in-app billing	139 (27.8%)
4.	com.chartboost.sdk	Gaming services	116 (23.2%)
5.	com.flurry.sdk	Advertising, monetization and analytics services	79 (15.8%)
6.	com.millennialmedia. android	Advertising, monetization and analytics services	76 (15.2%)
7.	com.mopub.mobileads	Advertising, monetization and analytics services	70 (14.0%)
8.	com.tapjoy	Advertising, monetization and analytics services	47 (9.4%)
9.	com.bda.controller	PhoneGap game controller	23 (4.6%)
10.	com.unity3d. plugin.downloader	Gaming Services	21 (4.2%)

Our analysis reveals that 84.2% of all connections encoded in these application can be considered non-essential (283,159 connection out of 336,203 in total). These results are consistent with the observation of our empirical study described in Section 2. Table 6 presents the top 10 packages in which non-essential connections occur. As the numbers are aggregated for 500 applications, it is not a 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 last column of Table 6. By manually investigating some of the most-popular connections in reverse-engineered versions of the applications, we observed that those connections are designed to be "best-effort" only. For example, an application might attempt to obtain user-specific advertisement information, but continues with generic advertisement if that attempt fails. The prevalence of mobile services and their "best-effort" behavior make us believe that it would be beneficial if these services were designed to allow users to select the level of support they wish to obtain, instead of relying merely on connectivity for that purpose.

To answer RQ4, we conclude that non-essential communication is very common in real-life applications. Most such communication is performed with various mobile services that are designed to be "best-effort only", i.e., communication failures do not prevent successful application execution. Designing mobile services that allow the user to select the preferred level of support instead of relying merely on connectivity would be beneficial.

5. LIMITATIONS AND THREATS TO VALIDITY

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, thus the findings apply only to the execution paths explored during the analysis. Even though we made an effort to cover all application functionality visible to us, we probably 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 application versions 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.

A Static Technique For Detecting Non-Essential Connections. Our technique deems *optional* behaviors – those when failing connections are ignored by the application, but successful connections result in presenting additional information to the user – as non essential. As an application can proceed when optional behaviors are excluded, it is debatable whether they are really essential for the application functionality or not. In fact, we believe that it is up to the users to decide whether an optional behavior is indeed essential for their needs.

Our technique also deems as non-essential *stateful* communication that toggles the state of a connection target but does not perform any operations in fault-processing code. In many cases, detecting such communication statically is impossible because the code executed on the target is unknown and unavailable. Even when communication is performed via RPC within the same application, it is exceedingly costly for an analysis to determine, with precision, whether a connection (or set of connections) is stateful.

Some of the non-essential connections that we identified statically might never be triggered dynamically. In fact, our empirical study shows that only less than 5% of the connection statements in the analyzed applications were indeed triggered. Some of such connections belong to execution paths that were not explored during our dynamic application traversal. Yet, a large percentage of these connections originate in third-party libraries that are included in the application but only partially used. Analyzing them is still beneficial as this code might be used in other applications. Nevertheless, our approach could be combined with techniques for detecting dead code, to provide better results.

6. RELATED WORK

Work related to this paper falls into three categories: (1) user-centric analysis to identify spurious application behaviors (2) information propagation in mobile applications, and (3) static exception analysis for Java.

User-Centric Analysis for Identifying Spurious Behaviors in Mobile Applications. Huang et al. [19] 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. [12] propose an approach for identifying malware by tracking dependencies between the definition and the use of user-generated data. They deem sensitive function calls that are not triggered by a user gesture as malicious. However, in our experience, the absence of a data dependency between a user gesture and a sensitive call is not always indicative for suspicious behavior: applications such as twitter and Walmart can initiate HTTP calls to show the most up-to-date information to their user, without any explicit user request. Moreover, malicious behaviors can be performed as a side-effect of any user-triggered operation. We thus take an inverse approach, focusing on identifying operations that do not affect the user experience.

Information Propagation in Mobile Applications. The most prominent technique for dynamic information propagation tracking in Android is TaintDroid [13], 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 [4, 16, 23, 24]. 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 [18] 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 [6] and AppContext [30] build up on the FlowDroid static information flow analysis system [4] 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. [26] 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.

Exception Analysis for Java. A rich body of static analysis techniques has been developed to analyze and account for exceptional control and data flow [7, 8, 14, 15, 21, 25, 22]. 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 [14, 15, 25]), we designed our technique to conservatively, though aggressively, consider difficult to analyze Android application development idioms such as reflection, RPC, native methods, and missing program semantics of the Android API defined in non-Java languages. We initially experimented with high-precision abstraction techniques such a deep object-sensitive points-to analysis [27]; however, the abstraction choices either did not scale or the precision and recall of the full analysis was unacceptable.

7. 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.

8. REFERENCES

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

[2] Android getevent Tool. https://source.android.com/devices/input/getevent.html.

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

[4] 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.

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

[6] 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.

[7] 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*, pages 173–182. IEEE Comput. Soc, 2002.

[8] 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*, pages 620–625, New York, New York, USA, Mar. 2001. ACM Press.

[9] 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.

[10] dex2jar. https://code.google.com/p/dex2jar/.

[11] 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.

[12] 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.

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

[14] C. Fu, A. Milanova, B. G. Ryder, and D. G. Wonnacott. Robustness testing of Java server applications. *IEEE Transactions on Software Engineering*, 31:292–311, 2005.

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

[16] 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.

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

[19] 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.

[20] ImageMagick Compare Tool. http://www.imagemagick.org/script/compare.php.

[21] J. W. Jo, B. M. Chang, K. Yi, and K. M. Choe. An uncaught exception analysis for Java. *Journal of Systems and Software*, 72:59–69, 2004.

[22] G. Kastrinis and Y. Smaragdakis. Efficient and effective handling of exceptions in java points-to analysis. In *Compiler Construction*, 2013.

[23] 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.

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

[25] 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*, volume 2, pages 1040–1046, 2010.

[26] 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.

[27] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick Your Contexts Well: Understanding Object-Sensitivity. In *POPL*, 2011.

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

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

[30] 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.