NChecker: Saving Mobile App Developers from Network Disruptions

Xinxin Jin Peng Huang Tianyin Xu Yuanyuan Zhou

University of California, San Diego

{xinxin, ryanhuang, tixu, yyzhou}@cs.ucsd.edu

Abstract

Most of today's mobile apps rely on the underlying networks to deliver key functions such as web browsing, file synchronization, and social networking. Compared to desktop-based networks, mobile networks are much more dynamic with frequent connectivity disruptions, network type switches, and quality changes, posing unique programming challenges for mobile app developers.

As revealed in this paper, many mobile app developers fail to handle these intermittent network conditions in the mobile network programming. Consequently, *network programming defects (NPDs)* are pervasive in mobile apps, causing bad user experiences such as crashes, data loss, etc. Despite the development of network libraries in the hope of lifting the developers' burden, we observe that many app developers fail to use these libraries properly and still introduce NPDs.

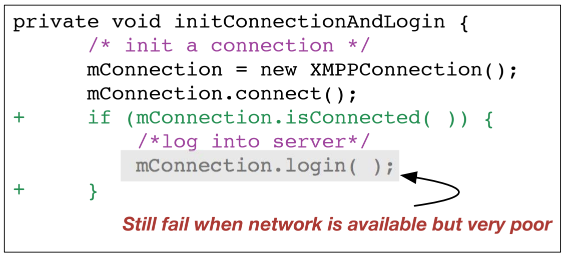
In this paper, we study the characteristics of the real-world NPDs in Android apps towards a deep understanding of their impacts, root causes, and code patterns. Driven by the study, we build NChecker, a practical tool to detect NPDs by statically analyzing Android app binaries. NChecker has been applied to hundreds of real Android apps and detected 4180 NPDs from 285 randomly-selected apps with a 94+% accuracy. Our further analysis of these defects reveals the common mistakes of app developers in working with the existing network libraries' abstractions, which provide insights for improving the usability of mobile network libraries.

1. Introduction

1.1 Motivation

User experience (UX) is the deciding factor to the success of any mobile app in the competitive app market [30, 33]. One of the major contributors to mobile UX is the app's ability to

EuroSys '16, April 18-21, 2016, London, United Kingdom
Copyright © 2016 ACM 978-1-4503-4240-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2901318.2901353



```
private void initConnectionAndLogin {
      /* init a connection */
      mConnection = new XMPPConnection();
      mConnection.connect();
+     if (mConnection.isConnected( )) {
          /*log into server*/
          mConnection.login( );
+     }
```

Still fail when network is available but very poor

Figure 1: A problematic patch of Android app ChatSecure. The patch fails to handle login() failure under intermittent network.

interact with underlying mobile networks. It is reported that 84% of the Android apps in Google App Store require the network connection to deliver key functions [66]; apps that fail to handle network problems well are likely to receive low star-rating reviews [38, 39, 53].

Mobile networks are fundamentally different from traditional desktop-/server-based networks for the dynamics incurred by mobility. In a typical mobile network, the network disruptions could frequently happen due to fluctuation of wireless signals and switches between network domains or even different network types (e.g., from cellular to WiFi networks) [43, 64]. The dynamics, together with frequent disruption events, of mobile networks pose unique programming challenges. Therefore, special attention should be paid to anticipate and tolerate network problems (e.g., disruptions) seamlessly without user awareness.

Unfortunately, as revealed in this paper, many of today's mobile app developers fail to handle these network disruptions in mobile network programming. Consequently, network-related errors are pervasive in today's mobile apps, causing bad mobile UX such as crashes, freezing, data loss, etc. In this work, we call such software defects *network programming defects (NPDs)*, which are caused by mishandling network issues in an app, resulting in negative user experiences under disruptive network environments. In our evaluation, we detect NPDs in 98+% of the evaluated mobile apps (c.f., §5).

Unlike desktop and server applications that are typically developed and maintained by experienced developer teams, mobile apps are often written by novice developers or small teams (attributable to the mature SDK) [8, 9, 32, 37]. Many of these novice app developers lack professional programming training, and thus have trouble reasoning about potential network-related issues such as disruptions and reconnections, not to mention to handle these issues well. Some mobile developers come from the background of building desktop or server software and have not been used to the practices of mobile network programming.

Moreover, individuals and small developer teams are usually limited in development and testing resources. With tight release cycles, many of them tend to focus more on features and UIs to attract first-time users. Some big companies (e.g., Facebook) chose to build different versions of apps to accommodate different network conditions [10], and develop large-scale testing frameworks [2, 7, 11] to test the app under different environments. However, this is clearly not affordable for most app developers.

Even if app developers find NPDs in their code, fixing these issues properly may pose another level of challenges. For example, we often notice that developers' patches for NPDs to be incomprehensive to address all the potential network disruptions.

Figure 1 shows a real-world NPD from a popular Android app, ChatSecure, in which the app tries to initialize an XMPP connection and log into the server. However, due to the network disruption, the `login()` statement can fail and cause the app to crash. To fix the bug, a patch is added to only enable login when network is available. Unfortunately, this patch did not completely rule out the problem, as the developer made a wrong assumption that the login request will succeed as long as there is a connection. It is a reasonable assumption in a desktop environment with a stable network, but is not true in mobile environments —`login()` can still fail with poor mobile network signals.

Figure 2 gives another example of NPD from a popular Android app Telegram. When the connection is unavailable, `connect()` would fail and enter an exception handler, where the reconnect timer is set to 500ms and executes `connect()` repeatedly until it succeeds. This leads to 100% CPU usage and excessive battery drain. The developers observed the symptom and added a patch to check if the network is available before connecting to the server. However, when the network is unstable, `connect()` could still fail and the symptom remains. The real problem here is the aggressive re-connection interval. A desirable solution is to back off retries if `connect()` fails repeatedly.

1.2 Are Existing Solutions Sufficient?

From the developers' perspective, a desirable solution is the one that saves them from the complicated and error-prone network programming. Therefore, a network library, encapsulating the boilerplate low-level network API calls, is an attractive solution to this end. Indeed, in the hope of lifting app developer's burden, a number of mobile network libraries have been built for a variety of network protocols,

```
public void connect() {
+   if(!isNetworkOnline()) {
+       Log.d("Connection not available");
+       return;
+   }
+
    selector.scheduleTask(new Runnable() {
        public void run(){
            try {
                client = selector.connect();
            } catch (Exception e) {
             /*Re-schedule selector.connect()
               every 500ms */
                …
            }
        }}
    }
}
```

Still execute the catch block when network is available but poor

Figure 2: A buggy patch of Telegram. The patch fails to handle the frequent re-connections under poor network conditions.

such as OkHttp [25] and Volley [34] for HTTP, aSmack [6] for XMPP, PJSIP for SIP [26], etc. These libraries provide app developers with high-level APIs. Some of them also enable certain fault tolerant features, such as catching runtime exceptions and retrying request several times if necessary.

While these libraries hide many low-level programming details and provide handy abstractions and primitives, they cannot fundamentally prevent inexperienced app developers from making mistakes for the following reasons.

First, there is no "one-size-fits-all" library. Thus, mobile developers still have to take the responsibility to set the library APIs for their own applications. The following is a conversation between an app developer and a library designer [35]. The app developer complained about the default timeout value being too short that caused frequent timeout exceptions. But in the library developer's opinion, setting a timeout value that fits all apps is out of the library's scope.

> ***App developer:*** *"My requests to call this API often timeout when the default timeout is 2500ms. Should the API be redesigned?"*
>
> ***Library designer:*** *"Generally speaking, I'd hope a server would be able to perform most read operations in well under a second. I don't think there's a one size fits all answer."*

To demonstrate the impact of the default API parameters under adverse mobile networks, we download files of different sizes from a HTTP server under different packet loss rates using Google's Volley library with the default API parameter settings (the default timeout is 2500ms). If the request fails, the library will automatically retry once. We use Network Link Conditioner [19] to control and throttle the network traffic. Figure 3 shows the success rate of downloading different sizes of files under different networks. In general, the downloading failure rate increases with the file sizes and packet loss increasing. The results indicate that app developers should tune the API parameters carefully for

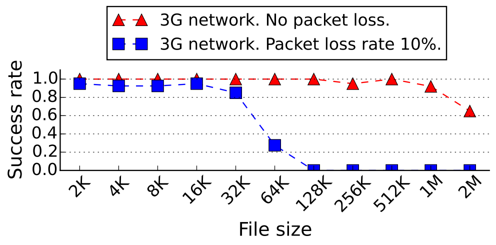


Figure 3: The sensitivity of default API parameters to different network conditions

different mobile networks and application requirements, instead of blindly trusting the default values.

Second, novice developers may lack the knowledge or expertise to use the mechanisms provided by the library APIs, especially the fault tolerance related APIs. For example, a developer uses Android's Asynchronous HTTP library to upload a large file for his app. But he saw in logs a `SocketTimeoutException` raised by the library and had no clue what happened or how to fix it [1]. Although this library provides the `setTimeout()` API for developers to set their own timeout values flexibly, if the app developer lacks the understanding of the low-level socket mechanisms, he or she can hardly reason about the correct API usage and therefore the problems still remain.

1.3 Our Contribution

The goals of this paper are two-fold: (1) helping non-expert mobile app developers detect network programming defects in the mobile apps that use existing mobile network libraries; (2) improving the design of existing mobile network libraries to prevent future NPDs in the first place. By accomplishing the above goals, we make the following contributions:

(1) We conduct the first empirical study of NPDs in mobile apps based on 90 real-world NPDs collected from 21 Android apps. The study provides a deep understanding of NPD characteristics, including their impact on user experience, root causes, and code patterns. These understandings can directly benefit NPD detection and prevention practices. Additionally, our study reveals that many mobile developers lack the experience in handling common disruption events in mobile networks.

(2) We develop NChecker, a practical tool to help mobile app developers detect NPDs. NChecker statically analyzes the binaries of Android apps and identifies NPDs when developers misuse network library APIs. NChecker has been applied to 285 real Android apps and found 4180 NPDs with 94% accuracy. We demonstrate the practicality of NChecker through a user study that shows even inexperienced developers can fix the NPDs within 2 minutes on average with NChecker's reports.

(3) We further analyze the common mistakes manifested in the 4180 detected defects. Based on the analysis, we provide the insights for further improving the usability of mobile network libraries, especially for non-expert mobile developers.

App/Sys	Category	#Installs
Chrome	Communication	>500M
Barcode scanner	Tools	>100M
Firefox	Communication	>50M
Telegram	Communication	>10M
K9	Communication	>5M
XBMC	Media & Video	>1M
Wordpress	Social	>1M
Sipdroid	Communication	>1M
ConnectBot	Communication	>1M
NPR news	News & Magazines	>1M
Csipsimple	Communication	>1M
Signal private messenger	Communication	>1M
ChatSecure	Communication	>100K
Owncloud	Productivity	>100K
GTalkSMS	Tools	>50K
Yaxim	Communication	>50K
Jamendo Player	Music & Audio	>10K
Hacker News	News & Magazines	>10K
BombusMod	Social	>10K
Kontalk	Communication	>10K
Android Framework	System	built-in

Table 1: 21 Android apps used in the study

2. An Empirical Study

To understand the characteristics of NPDs, we conduct an empirical study of 90 real-world NPDs in popular Android apps. We attempt to answer the following research questions from the study:

- How do NPDs impact user experience?
- What are the root causes of NPDs?

We also summarize the different code patterns of the studied NPDs, which will be presented in §4 as part of NPD detection methodology.

2.1 Methodology

As we need to understand the detailed root cause of each issue at the source-code level, we study mobile app projects with open issue tracking and version control systems, including the app projects hosted on GitHub [14], Google Code [15], Mozilla's Bugzilla [23], and Android Framework [4]. Among all these app projects, we first search network-related keywords (e.g., "network", "WiFi", "3G", and "connect") to collect potential issues from the issue tracking system and commit logs. Then, we review all the potential issues and decide NPD related issues based on the triggering conditions (a typical NPD is triggered by sudden disconnection, weak signals, network switches, or of-

ID	Category	App	NPD description	Developer's resolution
(i)	Dysfunction	Firefox	The download fails due to transient network errors	Add retry on connection failures
(ii)	Dysfunction	Yaxim	The sent message is lost on network failure	Queue the message for re-sending
(iii)	Unfriendly UI	Hacker News	No indication if the feeds loading fails	Add error message
(iv)	Crash	ChatSecure	Do not handle no connection exception on login	Add catch blocks
(v)	Freeze	Chrome	Failed XMLHttpRequest on webpage freezes the WebView	Cancel the request on failure
(vi)	Battery drain	Kontalk	Frequent synchronizations in offline mode	Disable synchronization in offline

Table 2: Representative NPDs found in real world mobile apps

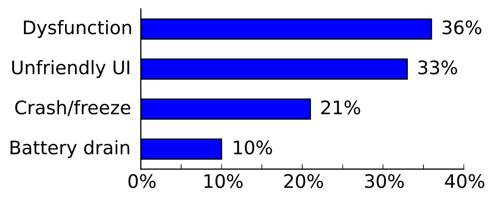


Figure 4: Distribution of NPD impact on user experience

fline conditions). In total, we collected 90 NPDs from 21 open source Android app projects (Table 1).

Threats to Validity. Similar to the previous works, real world characteristics studies are all subject to a validity problem. The potential threats to the validity of our study are the representativeness of the apps, and our collection method.

For the app representativeness, the studied apps cover various types of mobile apps (including messaging, news, web browsing, etc.) and use various network protocols such as HTTP, XMPP, IMAP, etc. As for our collection method, we carefully select generic network-related keywords, and manually examine each issue using multiple information sources (e.g., patches, discussion threads, and comments). The collection itself takes two man-month.

In summary, while we cannot draw any general conclusions that our study are applicable to all mobile apps, we believe that our study does capture the common characteristics of real-world NPDs. We do not emphasize any quantitative results. The purpose of this study is to shed light on what is error-prone in mobile network programming and how to help developers avoid such problems.

2.2 Impact on User Experience

We first categorize the studied NPDs according to their impacts on user experiences (UX) as the symptoms of the NPDs. Figure 4 shows the four categories of common UX impacts and their distributions.

Dysfunction (36%): defects that impair app network operations and break the expected functionality. Dysfunction could be very severe. For example, the defect in Table 2(ii) causes users' data losses. Other dysfunction NPDs bother users to redo certain operations, such as re-sending the message in Table 2(i)'s example.

Unfriendly UI (33%): defects that do not deliver friendly or seamless UIs for network failure notification. Mobile apps should react gracefully to network failures, even to permanent failures. Because apps are user interactive, when a user request encounters network failures, it is better to let the user aware what is wrong rather than the silent failure [19]. We observe that many patches have been added to display error messages upon network failures, such as the ones in Table 2(iii). Without error messages, users cannot reason about the cause of app inactions. In Table 2(iii)'s example, it is hard to differentiate the lack of new feeds from network problems.

Crash/Freeze (21%): defects that terminate the running app abnormally. Even though Java compilers perform certain compile-time exception checks that help reduce crashes, many NPDs still cause exceptions that escape compilers, such as the issue shown in Table 2(iv). In addition, as mobile apps commonly updates UI components after sending requests or receiving responses, NPDs could cause the UI to freeze, like the example in Table 2(v).

Battery drain (10%): excessive network requests can cause heavy energy usage and drain the battery fast. The example in Figure 2 belongs to this category. Table 2(vi) shows a similar example.

2.3 Root Cause Analysis

Table 3 categorizes the root causes of the studied NPDs and their distributions.

Cause 1: No connectivity checks (30%). In a dynamic mobile network, connections can drop frequently. While a proper error handling logic is necessary to tolerate all kinds of applications' network failures, checking connectivity before a network request is critical for mobile apps considering the mobile UX and limited resources. Both Android and iOS developer guides [19, 24] recommend the following practices for disconnected conditions: if the request is initiated by users, the app should display the network status information to users; if the request is initiated in background, the

Root cause	# Cases (%)
No connectivity checks	27 (30%)
Mishandling transient error	12 (13%)
Mishandling permanent error	24 (27%)
Mishandling network switch	27 (30%)

Table 3: Root causes of studied NPDs

app should cache and stop the operation to avoid wasting energy and mobile data every time the app initiates a connection, irrespective of the size of the associated data transfer. Therefore, it is desirable to check the network connectivity before *every* network request and gracefully react to connection changes.

Missing connectivity check is the most prevalent cause in the studied NPDs, perhaps because developers often incorrectly assume a stable network. Another reason is checking connectivity for every network operation could be tedious and error-prone, so developers can easily forget to do it.

Cause 2: Mishandling transient errors (13%). Retrying is the primary practice of handling transient network errors. However, defining retry policies is non-trivial for mobile apps and should follow the following principles [19]: (1) retry policies should keep users' waiting time in mind and be able to give users feedback timely; (2) retry policies should avoid unnecessary retries as each retry attempt consumes energy.

We observe the following two types of buggy retry behaviors that violate the above principles:

- *Cause 2.1: No retry for time-sensitive requests (55%).* We define a network request to be *time-sensitive* if it is initiated by an app user. It is important to retry the network operations to bypass transient errors and deliver the response to the user timely.
- *Cause 2.2: Over-retry (45%).* Over retry may cause unnecessary energy consumption and other side effects. The problematic retry decisions include: *(a) retry non-time-sensitive requests:* When the network request is initialized by background services rather than end users, it is usually *not* time-sensitive, and the retry should be disabled to save energy and mobile data; *(b) retry on non-idempotent requests:* repeatedly sending HTTP POST requests could be bugs [28]. As defined in HTTP/1.1: "A user agent MUST NOT automatically retry a request with a non-idempotent method."

Cause 3: Mishandling permanent error (27%). We observe three types of NPDs in handling permanent network errors (errors that are not recoverable).

- *Cause 3.1: No timeout setting (33%).* Timeout is a common mechanism to report failures in transmission. If the request fails to finish in a predefined period, the app would receive a timeout exception. However, the default Android network API performs a blocking connect, which probably takes several minutes to get a TCP timeout if the connection fails [29]. Such a long waiting time is obviously over the normal user expectation. Therefore, if the developer does not explicitly use the timeout API, users may never know whether the request succeeds even under permanent network errors.
- *Cause 3.2: Absent/Misleading failure notification (44%).* Not notifying users upon network failures is not acceptable, as exemplified by Table 2(iii). What is worse is the misleading error message. In one bug in iOS Wordpress [36], when the user lost network connectivity upon sending a post, he got the error message *"Couldn't Sync - A server with the specified host name could not be found."* This message only makes users more confused.
- *Cause 3.3: No validity check on network response (23%).* Developers sometimes assume the API's return value is always valid. However, it can be `null` under network disruptions and cause a crash if not checked.

Cause 4: Mishandling network switches (30%). Mobile apps face frequent network switching events, e.g., switching from cellular to WiFi to tethering hotspots. Each network switch means disconnecting from the old network and establishing a new network connection. However, many developers fail to handle the network switch correctly.

- *Cause 4.1: No reconnection on network switch (67%).* For XMPP and VoIP apps aiming at providing continuous a connection and smooth UX, the failure of catching network switches and re-establishing the connection is problematic. In a bug report of the chat app GtalkSMS [17], the app fails to respond to chat requests under intermittent networks—when the network status changes, the app still tries to receive data from the stale connections.
- *Cause 4.2: No automatic failure recovery (34%)* Mobile app may frequently experience disconnection. Failing to automatically recover the failed request also frustrate users [21]. Some patches cache users' requests and monitor changes of the connectivity status. When the network transits from a disconnected state to a connected state, the app automatically resends cached requests without user interventions.

3. Why Do Network Libraries Fail to Prevent NPDs?

There have been a number of mobile network libraries developed recently with the ambition of helping developer program mobile network correctly and easily. Many of them do provide certain features for handling network disruptions. This section attempts to understand these features and answer the fundamental question, "*why do these network libraries fail to prevent NPDs?*"

NPD Causes	**HttpURL Connection**	**Apache HttpClient**	**Volley Library**	**OkHttp Library**	**Android Async**	**Basic HTTP**
No connectivity check	◯	◯	◯	◯	◯	◯
No retry on transient error	★	◯	★	★	◯	★
Over retry	◯	◯	◯	◯	◯	◯
No timeout	◯	◯	★	◯	★	★
No/Misleading Failure notification	◯	◯	◯	◯	◯	◯
No invalid response check	◯	◯	★	◯	◯	◯
No reconnetion on net switch	◯	◯	◯	◯	◯	◯
No auto failure recovery	◯	◯	◯	◯	◯	◯

Table 4: Top libraries and their abilities in tolerating NPDs. ★ indicates tolerating the NPD automatically while ◯ indicates providing APIs but requiring developers to set explicitly.

We study the six most widely used mobile network libraries [31], including HttpURLConnection client, Apache HttpClient, Google Volley, OkHttp, Android Asynchronous HTTP client, and Basic HTTP client. The first two are Android native libraries and the others are third-party libraries. Table 4 summarizes how they handle NPDs. Column 1 lists the NPD causes obtained from § 2.3. Column 2–7 list the studied libraries. ★ means this library tolerates this type of NPD automatically, while ◯ means it may offer the error handling APIs, but developers have to set APIs explicitly on their own. For example, OkHttp does not set request timeouts by default, but it provides `setTimeout()` to allow developers to set the value on demand.

Our key observation is that *most mobile network libraries value too much on the flexibility of control and provide more flexibility than what the developers can handle, in comparison to the ease-of-use and robustness*. One reason is that many mobile network libraries are evolved from the ones designed for desktop apps with the assumption of experienced developers. As inexperienced app developers may not have the domain knowledge to use these library APIs correctly against network disruptions, the over-designed flexibility causes NPDs. As observed in our study (c.f., § 2), many developers are even not aware of the APIs for timeout settings and do not set it at all. Therefore, although some existing network libraries provide fault tolerant features, they cannot prevent NPDs.

One fundamental way to prevent NPDs is to rethink the design of the mobile network library, making them more user-friendly and robust to network errors. On the other hand, since the vast majority of existing apps are using existing libraries, it is hard to force all the apps to switch to new libraries. To help existing apps get rid of NPDs, we decide to build a tool to automatically detect NPDs from the app's binary code.

4. NChecker: NPD Detection

This section discusses the design and implementation of our NPD detection tool: NChecker.

4.1 Overview

NChecker detects NPDs in apps (as exemplified in Table 4). As apps primarily use libraries (either native or third-party) for network operations, NChecker identifies NPDs caused by developers misusing network library APIs in particular. We implement NChecker's analysis algorithms using the Java analysis framework Soot [67]. Before doing the analysis, we transform the APK into an intermediate representation of a Java program Jimple [45] and build the program call graph by extending FlowDroid [42].

Since NChecker detects NPDs in apps using network libraries, it requires the annotations of library APIs. Then, based on the annotations and the app's call graph, NChecker performs both control-flow and data-flow analysis to detect NPDs in the app and generate warning reports.

NChecker addresses three major challenges: (1) It must identify the patterns of library APIs misuse that lead to NPDs; (2) It needs to understand the app's customized network logic (primarily the retry logic) that does not use standard library APIs; (3) It should ease fixing of detected issues for developers with limited networking background.

To address the first challenge, NChecker follows the observations summarized in § 2. We discuss the NPD patterns in § 4.2 and how NChecker identifies these patterns in § 4.4. For the second challenge, NChecker recognizes the customized retry logics by catching retry loops (c.f., § 4.5), as we observe almost all the customized retry logics are implemented using loops. For the third challenge, NChecker generates informative, easy-to-understand warning reports to help developers understand and fix the detected NPDs (c.f., § 4.6).

4.2 Identifying API Misuse Patterns

Network operations are implemented by calling network library APIs (either Android native or third-party libraries). As such, NChecker maps the NPDs studied in § 2.3 to misused libraries APIs in the code. NChecker identifies four library API misuse patterns according to the NPD characteristics. Table 5 shows these pattens. Column 2 shows NPD root

API misuse pattern	NPD cause	Example of identifying misuse in the code
Miss request setting APIs	No connectivity check No retry on transient error No timeout	Do not call `getNetworkInfo` to check connectivity before sending a network request Do not call `setMaxRetries` to set retry times for sent network request Do not call `setReadTimeout` to set timeout for sent network request
Improper API parameters	Over retry	Set $retires \geq 0$ in `setMaxRetries` in Android Service or POST request
No/implicit error message in request callbacks	No failure notification	Do not call `Toast.show` to display a UI message in `onErrorResponse()` of a network request made by user
Miss resp. checking APIs	No invalid resp. check	Do not call `isSuccessful()` to check the response status before reading the response body

Table 5: API misuse patterns and examples. These patterns are derived from from our findings in Section 2.3. All API examples are from the top network libraries in Table 4.

causes corresponding to each pattern, and Column 3 gives examples of API misuses from network libraries in Table 4.

Note we do not check mishandling network switch (including "No reconnection on network switch" and "No automatic failure recovery") because there is no library APIs related to them.

Pattern 1: Missing request setting APIs. Besides APIs that perform the basic data transfer functions, there are also a number of APIs that guarantee the app reliability or help improve UX, but these are often overlooked by developers. Table 5 gives two examples of such APIs, one network connectivity checking API and one setting timeout API.

As mentioned before, developers should set API parameters according to their apps' demands, instead of blindly using the default values. So even if the library has a default value, e.g. 10 second timeout, we also give developers an alarm if they forget to invoke the corresponding setting API.

Pattern 2: Improper API parameters. We check if developers set improper retry API parameters. Table 5 shows an example of over retrying in a background service or POST request. Except for manually setting wrong parameters, not invoking retry APIs can also cause problems due to the default values. For example, Android Async HTTP library retries 5 times for all kinds of requests by default, causing energy waste and other undesirable effects (§ 2.3, Cause 2.2).

Pattern 3: No/Implicit error messages in request callbacks. Apps do not show explicit error messages after finishing the networking operations:

(1) No failure notification in the request callback. Generally, there are two ways to display a message : (a) Directly leverage the request callbacks provided by libraries to show messages in the UI thread. For example, the Volley library provides `onErrorResponse()`, which can be overwritten to display the UI notification on failure. Table 5 shows an example using Toast to show an error message. (b) If there is no such callback interface, Android provides a `Handler` class, which allows a background thread to communicate with the UI thread and define actions performed on the UI thread.

(2) Not leveraging the error types to pinpoint the error cause. For example, Volley library passes `Error` objects to the error callbacks. By examining the object, developers can extract the cause of the failure.This information is helpful for developers to handle different failures accordingly: For `NoNetworkError`, a "retry" button can be shown; for `ClientError` 400/401, developers should reason client-side anomalies and handle accordingly. NChecker checks if developers refer to these error types in the error callbacks.

Pattern 4: Missing response checking APIs. In the studied libraries, most libraries do not automatically check the validity of network responses (except Volley). Thus, the developers need to manually process the response to filter out invalid responses. Otherwise it can crash the application unexpectedly. Table 5 lists an example from OkHttp library. An invalid response body could be null so developers should call response checking APIs before using the response object.

4.3 Library API Annotation

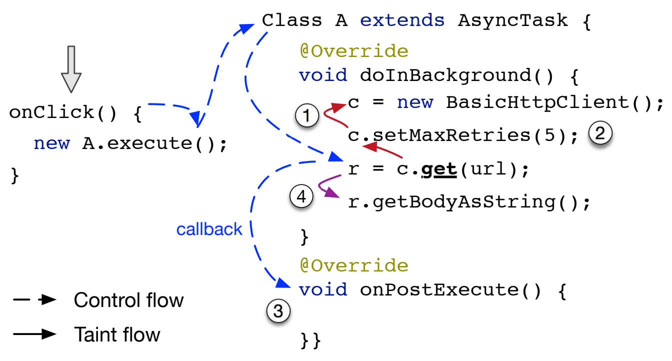
For our analyses, we annotate three kinds of APIs as follows:

- **Target APIs** are used to submit a network request, such as `BasicHttpClient.get()` for sending a HTTP GET request.
- **Config APIs** are used to configure the parameters of a network operation, and *have great impact on the reliability of the operation*. Combining with our characteristic study, we especially identify two kinds of config APIs: APIs that set timeouts and APIs that set retry policies.
- **Response checking APIs** are used to check the validity of the network response, e.g. OkHttp's `isSuccessful()`.

NChecker's current implementation annotates APIs from the 6 top popular HTTP libraries mentioned in Section 3. In total, we annotate 14 target APIs, 77 config APIs, and 2 response checking APIs from the six libraries.

4.4 Analysis Algorithm

To detect the API misuse patterns in § 4.2, NChecker performs four different analyses. Figure 5 illustrates the analysis algorithm for detecting NPDs based on a real world



```
onClick() {
  new A.execute();
}
```

```
Class A extends AsyncTask {
  @Override
  void doInBackground() {
    c = new BasicHttpClient();
    c.setMaxRetries(5);
    r = c.get(url);
    r.getBodyAsString();
  }
  @Override
  void onPostExecute() {

  }}
```

Figure 5: Simplified excerpt of a read app as a running example to explain the analysis algorithm of NChecker. ① Checking request setting APIs; ② Checking improper API parameters, ③ Checking failure notification; ④ Checking invalid response.

case. NChecker first performs reachability analysis and determines if there exist a target API `get()` which can be reached by the entry point `onClick()` (the callback of a click event), and then analyzes the app. We use this example through this section to describe the analysis algorithm.

4.4.1 Checking Request Setting APIs

NChecker first performs control-flow analysis to check the connectivity APIs. For each path from the entry point to the target API, NChecker checks if there is connectivity checking API invoked on the path. For the requests not guarded by the connectivity checks, NChecker raises alarms. In Figure 5 (Step ①), NChecker reports an error because the connectivity check is missed from `onClick()` to `get()`.

For config APIs that are defined in network libraries, NChecker performs taint tracking [42, 48–50, 63, 65] of the network request statically to determine which request setting APIs are missed for the checked request. After the target API is identified, NChecker first taints the HTTP client object at the call site of this API and then performs backward propagation until reaching the call site of creating the HTTP client instance. Next, NChecker further propagates forward from the client instance to obtain a set of tainted data. In the propagation path between the client initialization method and call site of sending request, NChecker records all the methods referred by the tainted objects. In the end, NChecker matches the recorded methods with the set of config APIs corresponding to the target API, and raises alarms for unmatched config APIs. For instance, in Figure 5, the taint analysis taints the client object `c`, performing both backward and forward analysis, and getting it has config API `setMaxRetries()`.

4.4.2 Checking Improper API Parameters

A naive way to check an API parameter is to directly look at the argument at the API call site, but whether the setting is proper or not depends on different app contexts. Thus, NChecker needs to analyze the app context of the network request. Based on the over retry behaviors in § 2.3, NChecker considers three different app contexts: time-sensitive requests initiated by users, time-insensitive requests initiated by background services, and the POST request type.

NChecker determines the first two app contexts based on reachability analysis. In Android, the user triggered operations are called from an *Activity* class; the background operations are called from a *Service* class. All the Activity and Service classes of an app are declared in AndroidManifest.xml file. By checking the declaration class of each entry point (lifecycle or callback methods) of the path to the network request, NChecker knows whether the request is initiated by a user (Activity) or by a background service (Service). NChecker determines the (POST) request type via the target API or the API parameters, e.g., Android Async HTTP uses `post()` API for POST requests; Volley's target API's first parameter indicates the request method. Finally, NChecker infers the value of config APIs through constant propagation [40]. If the config API is not invoked along the path, NChecker uses its default value. In Figure 5 (Step ②), NChecker learns `onClick()` is triggered by a UI event, so the app behaves correctly with 5 retry attempts. NChecker also analyzes apps' customized retry behavior (§ 4.5).

4.4.3 Checking Failure Notification

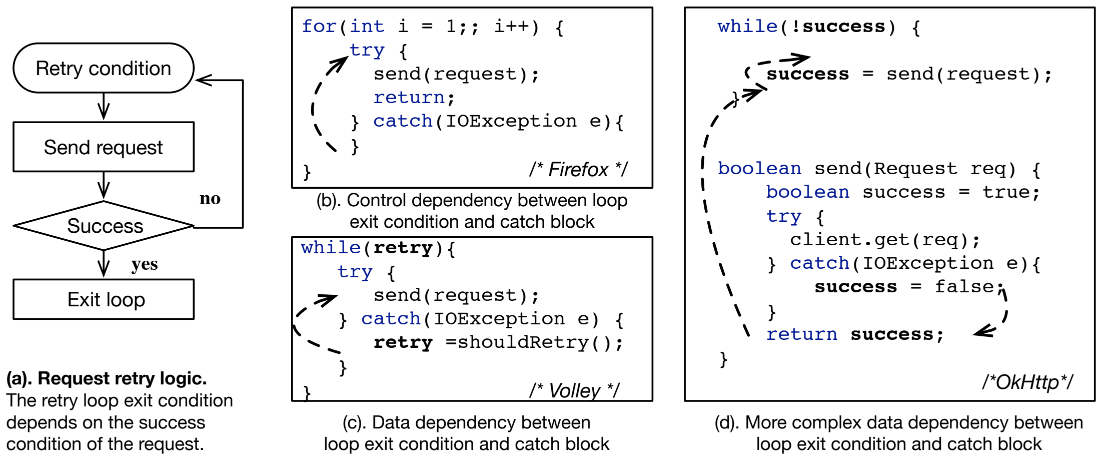
In Android, the UI notification must be done in some callbacks in the UI thread after the network operation completes. The mappings between the network request and its callback are defined in Android framework and network libraries as described in § 4.2. NChecker maintains such mappings and iterates the call graph to match the request with the callback method. Because the error message is only helpful when the user initiates the request, NChecker only checks callbacks whose corresponding network requests are initiated from an Activity. The process is similar as described in § 4.4.2.

Second, NChecker performs control-flow analysis in these callbacks and checks if APIs related to alert messages get called. Android mostly uses 5 classes to show alert messages: `AlertDialog`, `DialogFragment`, `Toast`, `TextView` and `ImageView`. If none of these classes' methods appear in the callback, NChecker raises an alarm.

Finally, to check if the error message pinpoints the error cause, we perform taint analysis on the error object passed to the error callback, and check if developers refer to the object to get error types. Currently, only Volley's error callback specifies the error types, so this is only applied to Volley. In Figure 5 (Step ③), NChecker finds no failure notification in the callback `onPostExecute()`, so it raises an alarm.

4.4.4 Checking Invalid Response

NChecker also performs taint analysis to check invalid responses. NChecker first taints the response object, initially populated by the return value of network requests, or some input argument of the response callback for some libraries.



(a). Request retry logic. The retry loop exit condition depends on the success condition of the request.

```
for(int i = 1;; i++) {
    try {
      send(request);
      return;
    } catch(IOException e){
    }
}                              /* Firefox */
```

(b). Control dependency between loop exit condition and catch block

```
while(retry){
   try {
      send(request);
   } catch(IOException e) {
      retry =shouldRetry();
   }
}                            /* Volley */
```

(c). Data dependency between loop exit condition and catch block

```
while(!success) {

    success = send(request);
}


boolean send(Request req) {
    boolean success = true;
    try {
      client.get(req);
    } catch(IOException e){
        success = false;
    }
    return success;
}
                                /*OkHttp*/
```

(d). More complex data dependency between loop exit condition and catch block

Figure 6: Identify the customized retry logic

Then it propagates the tainted response forward and checks if necessary null checks or other necessary response validity checks are performed on the tainted data. If NChecker detects a path between the definition and the use of the response object, but the related validity check is missed on the path, an alarm is raised. For instance, in Figure 5 (Step ④), the response object `r` is tainted and propagated. NChecker will alarm that there is no validity check before the use of `r`.

4.5 Identifying Customized Retry Logic

In our studied apps, most of the customized retry logics are implemented by *retry loops*. While the loop can have multiple exit conditions, at least one of them depends on the *success state* of the network request. We refer to such exit conditions as *retry conditions* (Figure 6(a)). Here "success" means the network operation does not fail due to network failure, otherwise it will throw an `IOExcpetion`. The challenge of identifying retry loops is to differentiate retry loops from normal loops that send a sequence of network requests. We differentiate the two by identifying the two kinds of exit conditions of a retry loop: (1) At lease one of the *unconditional* exit conditions depends on the request success. In other words, the control flow can jump out of the condition as long as the network request succeeds. Figure 6(b) illustrates such an example. The loop will never exit unless `send()` succeeds and returns. (2) If the exit condition depends on some variable, then this variable must directly or indirectly depends on statements in the catch block. The intuition behind this is that a retry loop can only continue once a network failure happens and an `IOException` is caught. So there must exist some data dependency from the catch block to the exit condition. Figure 6(c) shows a retry example that the exit variable retry is dependent on the statement "`retry=shouldRetry()`" in the catch block. Figure 6(d) shows a more complex example of this data dependency, where the exit condition variable `success` is indirectly dependent on the "`success=false`" statement in the catch block of a callee method. Based on this observation, we detect retry loops by the following steps:

NPD Information
Missing network connectivity check before `HttpClient.get()` at `OpenGTSClient, line 115`
NPD impact
Bad UX, battery life
Network request context
Request made by user. Need to notify users if connection is unavailable.
Network request call stack

`<onSelection> (GpsMainActivity: 756)`
`-> <UploadFile> (OpenGTSHelper: 43)`
`  —> <sendHTTP> (OpenGTSClient:91)`
`   —> <httpClient.get> (OpenGTSClient: 115)`

Fix Suggestion
Use `getActiveNetworkInfo()` to check connectivity before `HttpClient.get()`. Show error message if no connection.

Figure 7: An example of NChecker warning report for GPSLogger

1. *Detect loops with loop bodies directly or indirectly containing the target APIs:* If the loop is implemented in some caller of the method including target API, we will recursively parse the caller to detect the loop.
2. *Extract loop exit conditions:*. The recognized exit conditions include conditional exits in the form of conditional branches (`if cond goto label`), and unconditional exits like `return` or `break` statements.
3. *Identify retry loops by exit conditions:* We identify retry loops if one of the following two conditions is satisfied: (a) the unconditional exit statement is unreachable from any other statement in the catch block; (b) the conditional exit conditions have control dependencies with other statements in the catch block. Backward slicing [51, 62, 68] is used to obtain the control dependency information.

NPD cause	Eval. condition	# Eval. apps	# Buggy apps (%)
Missed conn. checks	All apps	285	122 (43%)
Missed timeout APIs	Use libs that have timeout APIs	285	139 (49%)
Missed retry APIs	Use libs that have retry APIs	91	64 (70%)
Over retries	Use libs that have retry APIs	91	50 (55%)
Missed failutre notifications	Include user initiated requests	264	151 (57%)
Missed response checks	Use libs that have resp. check APIs	20	15 (75%)

Table 6: Percent. of buggy apps detected by NChecker categorized by NPD causes.

4.6 NChecker Report

NChecker generates a report for each detected NPD. The report aims at guiding inexperienced developers to understand and fix the defect. A warning report contains the following items: (1) *NPD information*: the error message including the problematic API usage and its code location. (2) *NPD impact*: the negative UX caused by this NPD. (3) *Request context*: if the request is made by users or a background services. (4) *Request call stack*: the call stack from an entry point to the reported buggy network request. (5) *Fix suggestion*: generated based on each type of NPD considering the context to facilitate inexperienced developers to fix it. Figure 7 shows a NChecker report of the GPSLogger app.

4.7 Limitations

NChecker accuracy can be affected by the following limitations: (1) The current implementation does not support inter-component and inter-application communication. In the future we plan to integrate NChecker with IccTA [58], an inter-component data flow analysis framework for Android Applications. (2) Because NChecker identifies the NPD patterns based on the annotations of pre-defined Android framework or library APIs, its usability is limited in apps using customized APIs (e.g. customized view of UI notification, annotation framework [3]).

5. Evaluation

The evaluation answers three major questions:

- Is NChecker effective in detecting new NPDs?
- How accurate is NChecker?
- Is NChecker practical in helping inexperienced developers fix NPDs?

5.1 Methodology

We applied NChecker to 285 Android apps, including 269 closed-source and 16 open-source ones. The apps are selected as follows. We crawled the top 1000 popular Android apps from Google Play Store. The majority of them use native libraries (HttpURLConnection and Apache HttpClient), and only a few use third-party libraries. To achieve better coverage, we kept all the apps that use the third-party libraries, and randomly sampled apps that use native libraries. 16 open-source apps are also chosen in order to diversify the data sources. The number of apps and their libraries are shown in Table 7.

Lib used	# Apps
Native	270
Volley	78
Android Async Http	25
Basic Http	18
OkHttp	11

Table 7: Evaluated apps and their libraries

5.2 Effectiveness

NChecker discovers 4180 *new* NPDs in nearly all (281) of the 285 apps (see details below). Table 6 summaries the percentage of buggy apps detected by NChecker corresponding to different NPD causes. Please note that we do not evaluate all 285 apps for *every* NPD type due to the different library APIs used by different apps. In the table, the second column explains what kind of apps are evaluated for this NPD cause and the third column shows the number of evaluated apps accordingly.

We have submitted patches to 6 open-source apps (some other open-source apps are not maintained by developers anymore so we did not report to them), including Popcorn Time [27] , F-Droid [12], Kontalk [20], Galaxy zoo [13], AnkiDroid [5] and GPS Logger [16], according to NChecker's reports. To date, 61 NPDs have been confirmed and fixed by developers. The result indicates that NChecker is effective in finding new NPDs, and developers are willing to fix them.

5.2.1 Missed Config APIs

A large number of apps *never* invoke the config APIs before any call site of the network requests, as shown in Table 6. We further look at apps that have invoked config APIs and check the coverage of config APIs. Figure 8 plots the cumulative distribution of apps as a function of the ratio of network requests that miss connectivity checks and timeouts.

43% of apps never check network connectivity. For the remaining 163 (57%) apps that have but partically miss connectivity checking, Figure 8 further shows 62% of apps miss connectivity checking in over half of their requests. This re-

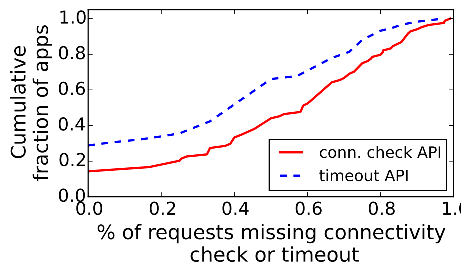


Figure 8: For apps that set but partially miss config APIs, the CDF of the ratio of requests missing connectivity check (red line) and timeout setting (blue line).

sult is consistent to the observation in our study: developers often do not pay attention to checking connectivity.

49% of apps never set timeout APIs. For the remaining 146 (51%) apps that set but partially miss timeout APIs, Figure 8 shows 58% of apps miss timeout setting in over half of the requests.

70% of apps never set retry APIs. We notice that one retry API `allowRetryExceptionClass()` of Async HTTP library is never set by any application. This API is used to set the exception class that should be retried on transient failures. One possible reason is that this API requires thorough understanding of the network exceptions, which is beyond many non-expert developers' capability.

10% of apps have customized retry logic. Without the library support, the majority of developers lack the experience in writing the retry code to handle transient network errors.

5.2.2 Improper Retry Parameters

We examine the improper retry parameters in 91 apps that use libraries with retry APIs. Table 8 shows the ratio of apps that have inappropriate retry behaviors caused by wrong retry API parameters. As the second column shows, 8% of apps do not retry for time-sensitive requests; 32% of apps over retry in background services and 25% of apps over retry in POST requests.

NPD cause	Apps(%)	Default behavior
No retry in Activities	8%	0
Over retry in Services	32%	76%
Over retry in POST requests	25%	98%

Table 8: Ratio of apps with inappropriate retry behaviors. The third column indicates the NPD is caused by library default behaviors. The total number of evaluated apps in this table is 91.

Interestingly, the majority of the over retry behaviors are caused by the library default behaviors. As shown in the third column of Table 8, 76% of over retries in Service and 98% of over retries in POST requests are caused by library default behaviors. That is, developers do not invoke the retry API to disable retries.

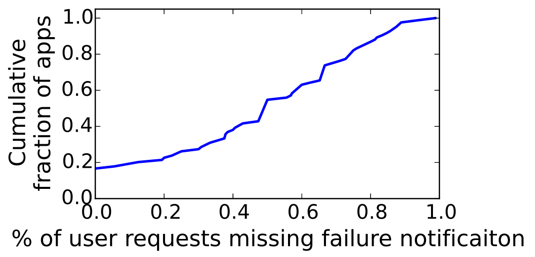


Figure 9: For apps that set but partially miss failure notifications, the CDF of the ratio of requests missing failure notifications

5.2.3 No/Implicit Failure Notification

57% of apps do not show any notifications for failures in any user-initiated network requests. For remaining 114 (43%) apps that show but partially miss error messages, Figure 9 shows the CDF of apps on the ratio of network requests that miss error messages. That means most developers do not pay attention to graceful UX on permanent network errors.

Developers are more likely to show error messages in explicit error callbacks. As discussed in Section 4, two types of interfaces can be leveraged to show alert messages: the first is to directly utilize the explicit error callback methods such as `onError()`; the second is to use the `Handler` class, which requires developers to write more complex code to define the UI actions. We notice that developers are more likely to show failure notifications in the former case: 30% of network requests with explicit error callbacks have failure notifications, while only 12% of network requests without explicit error callbacks have failure notifications.

93% of apps do not check the error types. That is because the error types are hidden from the error callback APIs, so most non-expert developers are not aware of it.

5.2.4 Missed Response Checking APIs

75% of total network responses miss validity checks before they are used. That means most app developers assume the received response to be valid.

5.3 Accuracy

Since it is hard to verify the accuracy on closed source apps, we measure NChecker's accuracy using 16 open-source apps. We manually verify the detected results by examining the source code. Table 9 gives the number of correct warnings detected by NChecker and false positives (FP) and false negatives (FN) [1].

NChecker correctly detects 130 *new* NPDs from the 16 apps in total. NChecker has 5 FNs of connectivity check from 1 app because of the path-insensitive analysis: the app invokes connectivity check APIs before the network requests, but these APIs are not the control conditions of the requests. NChecker has 4 FPs of connectivity checks from 2

[1] FNs means cases we found manually but not detected by the tool. The FNs could potentially be underestimated due to the lack of ground truth.

NPD cause	# Correct warning	# FP	# Known FN
Missed conn. checks	31	4	5
Missed timeout APIs	58	0	0
Missed retry APIs	12	0	0
Over retries	4	0	0
Missed failure notifications	20	5	0
Missed response checks	5	0	0
Total	**130**	**9**	**5**

Table 9: NChecker results of open source apps. 61 NPDs were confirmed and fixed by developers.

apps because these two apps check connectivity before starting the activity that sends network requests, but NChecker does not handle Android inter-component communication currently. There are 5 FPs of missed failure notification from 1 app, which is also caused by the inter-component communication: the app broadcasts the error code and shows error messages in another activity.

5.4 NChecker Practicality

We conduct a controlled user study to understand the practicality of NChecker in helping app developers fix 7 real-world NPDs. The NPDs are selected to cover diverse root causes described in § 4. Table 10 shows the NPDs reported by NChecker that used in our user study. We recruited 20 undergrad volunteers who indicate they know Android programing basics. The average amount of experience of Android programming among volunteers is only 6 months. We first explained the code structures of these apps to the volunteers, and then asked them to read the NChecker reports and to fix the defects. The correct fix is shown in Table 10. We measured the time they spent on fixing each NPD.

Note that this is a best-effort user study. While many factors (e.g., user or NPD representativeness) can affect the accuracy of a user study, below we will show the absolute time of fixing these NPDs is very short, which is a strong evidence to demonstrate the usefulness of NChecker.

Results Figure 10 shows our study results. On average programmers take 1.7 ± 0.14 minutes (at 95% confidence interval) fixing these NPDs based on the warning reports. Interestingly, some volunteers have no network programming experience, but can still fix the code correctly given the information provided by NChecker. One volunteer said: "*I don't have any experience in programming network operations before this, but the information provided allowed me to fix the bugs even with this limitation.*" From our observation, majority of the volunteers immediately realized the problem after reading the NChecker report. They spent most of the time in understanding the APIs and writing the patches. For the "GPSLogger retry (no retried exception)" case, only one volunteer correctly sets the exception class that should be retried. All others cannot reason about the network exception types. This result is consistent with our result of applying NChecker to a large range of existing Android apps.

Name (NPD)	Correct fix
AnkiDroid (no connectivity check)	Add connectivity check before the request. Show error message if not connected.
GPSLogger1 (no timeout)	Add timeout API to set timeout value
GPSLogger2 (no retry times)	Add retry API to set retry times
GPSLogger3 (no retried exception)	Add another retry API to set exception class that should be retried
DevFest 1 (no error mesg)	Add error message in callback according to the error status.
DevFest 2 (invalid resp.)	Add null check and status check on the response before reading the its body
Maoshishu (over retry)	Add retry API and set retry time to be 0

Table 10: Real world app NPDs used in our user study

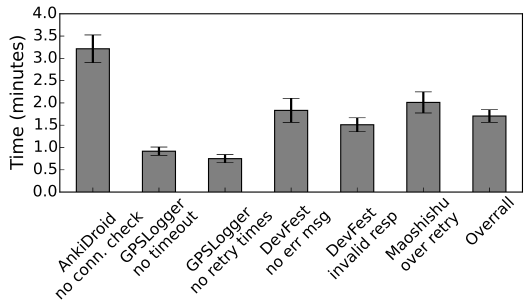


Figure 10: User study result, with error bars showing 95% confidence interval. GPSLogger (no retried exception) is excluded since majority users do not know the types of network exceptions.

6. Guidelines Towards User-Friendly, Robust Design of Mobile Network Library

The prevalence of NPDs detected in the hundreds of apps unveils that many app developers have trouble utilizing the library APIs in dealing with network disruptions. As discussed in § 3, most of these libraries are evolved from the ones designed for desktop apps with the assumption of experienced developers; thus, their APIs value flexibility more than ease-of-use (a.k.a. usability). However, this flexibility complicates the network programing for inexperienced app developers. This problem leads to rethinking the question of what a library should expose to mobile app developers.

Based on our study (§ 2) and evaluation results (§ 5), we share our insights towards the design of user-friendly, robust mobile network libraries. The ultimate goal is to make the library hard to misuse, easy to understand and extend [46]. Table 11 summarizes the design guidelines.

6.1 What Should Be Abstracted away from APIs?

The features that are often ignored or misused by developers should be abstracted away from APIs and implemented as the library internals.

Observations	Guidelines for mobile network libs
43% apps never check network connectivity	Automatically check connectivity before each network request
70% apps ignore retry APIs; only 10% apps impl. customized retry	Automatically retry on transient network error
Over 76% of over retries are caused by default API values	Set default retries considering the request context
57% apps never show failure notifications for user-initiated requests	Pre-define error message on network failure
75% of network requests miss validity checks	Automatically put invalid response into error callbacks
More apps show error mesg. in explicit error callbacks than implicit ones	Explicitly separate success and error network callbacks
93% apps do not check error types	Expose important error types in addition to error callbacks

Table 11: Observations of applying NChecker in a large scale and the derived guidelines for designing mobile network libraries.

Connectivity checking before requests. Checking connectivity before every request is laborious and often overlooked by developers. Mobile network libraries should automate connectivity checking before sending network requests.

Retry on transient network errors. Both the evaluation and the user study indicate many developers either ignore retry APIs or lack the understanding of the retry policies and the types of retriable exceptions. Therefore, the library should automatically retry on transient disruptions.

Retry considering the app context. Since most developers are not aware of the library default behaviors, libraries should set default retries considering the request contexts. For example, disabling retry for POST requests and requests initiated by background services by default.

Predefined error messages on network failures. As the majority of apps do not pay attention to show failure notifications, libraries can predefine default actions on failure, such as displaying an error message when a request fails, or a default warning picture when downloading pictures fails.

Invalid response. Since many app developers do not know what kind of errors to handle and how to handle the errors correctly, libraries should not leave messy error handling logics to developers. Invalid responses should be put into error callbacks: when developers process the responses in success callbacks, the validity can be guaranteed.

6.2 What Should be Exposed via APIs?

The features that can help app developers pay special attention to the error handling should be exposed.

Explicit success/error network callbacks. As developers are likely to do error handling in explicit error callbacks, libraries should provide separate success/error response callbacks. This helps developers to easily identify the failure states and implement error handling logics in error callbacks.

Important error types. Error types are very useful in helping developers take different actions for different error conditions (c.f. § 4.2 pattern 3). In addition to error callbacks, library APIs should also expose the error types and give developers hints about how to handle each error type.

7. Related Work

A number of in-house testing approaches have been proposed for mobile apps, such as fuzz testing [22, 55, 61], symbolic testing [56], concolic testing [41], and model-based testing [18]. However, few of these look at the problems related to mobile network dynamics, probably because emulating different network environments with high fidelity is prohibitively difficult.

Many previous works dynamically instrument mobile apps to understand app behaviors [52, 59, 60]. Vanarsena [61] and Caiipa [54] dynamically inject environment related faults including Web errors and unreliable network into the apps under testing to find bugs, and file a crash report if the injected fault causes a crash. While the run-time approaches can generate accurate results for a given run, they are also restricted by the code coverage and run-time overhead. And for NPDs checked by NChecker, some can hardly be detected by above dynamic tools. For example, NPDs caused by “no timeout setting” requires additional timing fault model to be triggered. These code defects are also not manifested through crash behavior so cannot be observed by the dynamic tools. NChecker is complementary to these works in dealing with a broad range of NPDs.

The battery drain problem due to network operations has seen a large body of work in recent times. Prior work has investigated strategies [44, 47, 57], such as optimizing background tasks and prefetching, for energy-efficient network tasks. Our work tackles the NPDs causing battery drain from an app developer’s perspective.

The Android framework is continuously improved for network programming. From Android 4.4, Android’s native HTTP client HttpUrlConnection uses OkHttp as the underlying network library. For services with multiple IPs, it will try alternate addresses upon connection failures. From Android 3.0, NetworkOnMainThreadException would be thrown when apps attempt networking operation on the UI thread, in order to prevent the UI freezing. These improvements enhance the robustness and performance of mobile network operations, but are still too limited in terms of enforcing good practices. We hope our work will motivate more improvements in the Android framework.

8. Conclusion and Future Work

This paper studies and detects a class of software defects in mobile apps, network programming defects (NPDs), which are critical to apps' user experience but often overlooked by app developers. We studied the charactersitcs of NPDs from 90 real-world NPD cases and developed a static analysis tool NChecker to detect NPDs. NChecker effectively detected a large number of NPDs in 285 Android applications with 94% accuracy. Our controlled user study shows that NChecker can help inexperienced developers quickly fix NPDs. The findings of NChecker also provide insightful guidelines towards more user-friendly and robust design of mobile network library for mobile app developers.

This work mainly focus on NPDs in Android apps because of the availability of open-sourced Android apps. We leave studying and addressing NPDs in other platforms such as iOS and Windows phone for feature work. While NChecker detects NPDs caused by misusing existing network libraries' APIs, we will work on providing a better programming language support to eliminate NPDs.

Acknowledgments

We greatly appreciate Eurosys anonymous reviewers for their insightful feedback. We especially thank our shepherd, Dr. Oriana Riva, for her great effort to help us improve the paper. We also thank Jiaqi Zhang, Tianwei Sheng Weiwei Xiong, Yanqin Jin, Liqiong Yang, Congming Chen for their discussions and paper proof-reading. Last but not the least, we are grateful to the volunteers who participated our user study. This research is supported by NSF CNS-1017784 and NSF CNS-1321006.

References

[1] https://github.com/loopj/android-async-http/issues/392.

[2] Airlock - facebook's mobile a/b testing framework. https://code.facebook.com/posts/520580318041111/airlock-facebook-s-mobile-a-b-testing-framework/.

[3] Androidannotations. http://androidannotations.org/, .

[4] https://android-review.googlesource.com, .

[5] Ankidroid. https://play.google.com/store/apps/details?id=com.ichi2.anki.

[6] https://github.com/Flowdalic/asmack.

[7] Insights into the testing and release processes for chrome for ios. http://www.infoq.com/articles/testing-releasing-chrome-ios.

[8] State of the developer nation q3 2014. http://www.visionmobile.com/product/developer-economics-q3-2014/, .

[9] Growing as a developer with no formal computer science background. http://www.talentbuddy.co/blog/growing-as-a-developer-with-no-formal-computer-science-background/, .

[10] Facebook launches facebook lite to resolve poor connectivity issues. http://thetechportal.in/2015/01/26/facebook-launches-facebook-lite-resolve-poor-connectivity-issues/, .

[11] New facebook tool helps developers test apps on slow networks. http://techcrunch.com/2015/03/23/new-facebook-tool-helps-developers-test-slow-pokes/, .

[12] F-droid. https://f-droid.org/.

[13] Galaxy zoo. http://www.galaxyzoo.org/.

[14] Github. https://github.com/.

[15] Google code. https://code.google.com/.

[16] Gps logger. http://code.mendhak.com/gpslogger/.

[17] gtalksms. https://code.google.com/p/gtalksms.

[18] Guitar a gui testing framework. http://guitar.sourceforge.net/index.shtml.

[19] https://developer.apple.com/library/ios/documentation/NetworkingInternetWeb/Conceptual/NetworkingOverview/WhyNetworkingIsHard/WhyNetworkingIsHard.html.

[20] Kontalk. https://kontalk.org/.

[21] http://developer.android.com/training/monitoring-device-state/connectivity-monitoring.html, .

[22] Ui/application exerciser monkey. http://developer.android.com/tools/help/monkey.html, .

[23] Bugzilla. https://bugzilla.mozilla.org.

[24] http://developer.android.com/training/basics/network-ops/managing.html.

[25] http://square.github.io/okhttp/.

[26] http://www.pjsip.org/.

[27] Popcorn time. http://popcorn-time.to/.

[28] Beware of http request automatic retries. http://www.ravellosystems.com/blog/beware-http-requests-automatic-retries/.

[29] Urlconnection timeouts. https://developer.android.com/reference/java/net/URLConnection.html#setConectTimeout(int).

[30] https://www.cbinsights.com/blog/startup-failure-reasons-top/.

[31] Android network libraries. http://www.appbrain.com/stats/libraries/tag/network/android-network-libraries.

[32] Uk app economy 2014. http://www.visionmobile.com/product/uk-app-economy-2014/.

[33] Five reasons usrs uninstall mobile apps. http://venturebeat.com/2013/06/03/five-reasons-users-uninstall-mobile-apps/.

[34] http://developer.android.com/training/volley/index.html, .

[35] https://groups.google.com/forum/topic/volley-users/sRnF7RVc2zI, .

[36] https://github.com/wordpress-mobile/WordPress-iOS/issues/522.

[37] App developers who are too young to drive. http://online.wsj.com/articles/SB10001424052702303410404577468670147772802.

[38] Ebuddy + weak signal = battery death. http://androidforums.com/threads/ebuddy-weak-signal-battery-death.85643/, Nov. 2010.

[39] Android app not seeing server over wifi. https://forums.plex.tv/discussion/103094/android-app-not-seeing-server-over-wifi, Mar. 2014.

[40] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers principles, techniques, and tools*. Addison-Wesley, 1986.

[41] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE'12, pages 59:1–59:11, Cary, North Carolina, 2012. ACM. ISBN 978-1-4503-1614-9. doi: 10.1145/2393596.2393666. URL http://doi.acm.org/10.1145/2393596.2393666.

[42] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'14, pages 259–269, Edinburgh, United Kingdom, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594299. URL http://doi.acm.org/10.1145/2594291.2594299.

[43] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting mobile 3g using wifi. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 209–222, San Francisco, California, USA, 2010. ACM. ISBN 978-1-60558-985-5. doi: 10.1145/1814433.1814456. URL http://doi.acm.org/10.1145/1814433.1814456.

[44] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, IMC'09, pages 280–293, Chicago, Illinois, USA, 2009. ACM. ISBN 978-1-60558-771-4. doi: 10.1145/1644893.1644927. URL http://doi.acm.org/10.1145/1644893.1644927.

[45] A. Bartel, J. Klein, M. Monperrus, and Y. L. Traon. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. *CoRR*, abs/1205.3576, 2012. URL http://arxiv.org/abs/1205.3576.

[46] J. Bloch. How to design a good api and why it matters. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA'06, pages 506–507, Portland, Oregon, USA, 2006. ACM. ISBN 1-59593-491-X. doi: 10.1145/1176617.1176622. URL http://doi.acm.org/10.1145/1176617.1176622.

[47] X. Chen, A. Jindal, N. Ding, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone background activities in the wild: Origin, energy drain, and optimization. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom'15, pages 40–52, Paris, France, 2015. ACM. ISBN 978-1-4503-3619-2. doi: 10.1145/2789168.2790107. URL http://doi.acm.org/10.1145/2789168.2790107.

[48] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA'07, pages 196–206, London, United Kingdom, 2007. ACM. ISBN 978-1-59593-734-6. doi: 10.1145/1273463.1273490. URL http://doi.acm.org/10.1145/1273463.1273490.

[49] 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 *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Vancouver, BC, Canada, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924943.1924971.

[50] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung. Vision: Automated security validation of mobile apps at app markets. In *Proceedings of the Second International Workshop on Mobile Cloud Computing and Services*, MCS'11, pages 21–26, Bethesda, Maryland, USA, 2011. ACM. ISBN 978-1-4503-0738-3. doi: 10.1145/1999732.1999740. URL http://doi.acm.org/10.1145/1999732.1999740.

[51] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI'88, pages 35–46, Atlanta, Georgia, USA, 1988. ACM. ISBN 0-89791-269-1. doi: 10.1145/53990.53994. URL http://doi.acm.org/10.1145/53990.53994.

[52] G. Hu, X. Yuan, Y. Tang, and J. Yang. Efficiently, effectively detecting mobile app bugs with appdoctor. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys'14, pages 18:1–18:15, Amsterdam, The Netherlands, 2014. ACM. ISBN 978-1-4503-2704-6. doi: 10.1145/2592798.2592813. URL http://doi.acm.org/10.1145/2592798.2592813.

[53] H. Khalid. On identifying user complaints of iOS apps. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE'13, pages 1474–1476, San Francisco, CA, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL http://dl.acm.org/citation.cfm?id=2486788.2487044.

[54] C.-J. M. Liang, N. D. Lane, N. Brouwers, L. Zhang, B. F. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra, and F. Zhao. Caiipa: Automated large-scale mobile app testing through contextual fuzzing. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*, MobiCom '14, pages 519–530, Maui, Hawaii, USA, 2014. ACM. ISBN 978-1-4503-2783-1. doi: 10.1145/2639108.2639131. URL http://doi.acm.org/10.1145/2639108.2639131.

[55] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE'13, pages 224–234, Saint Petersburg, Russia, 2013. ACM. ISBN 978-1-4503-2237-9. doi: 10.1145/2491411.2491450. URL http://doi.acm.org/10.1145/2491411.2491450.

[56] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing android apps through symbolic execution. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, Nov. 2012. ISSN 0163-5948. doi: 10.1145/2382756.2382798. URL http://doi.acm.org/10.1145/2382756.2382798.

[57] P. Mohan, S. Nath, and O. Riva. Prefetching mobile ads: Can advertising systems afford it? In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys'13, pages 267–280, Prague, Czech Republic, 2013. ACM. ISBN 978-1-4503-1994-2. doi: 10.1145/2465351.2465378. URL http://doi.acm.org/10.1145/2465351.2465378.

[58] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 543–558, Berkeley, CA, USA, 2013. USENIX Association. ISBN 978-1-931971-03-4. URL http://dl.acm.org/citation.cfm?id=2534766.2534813.

[59] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 107–120. USENIX, 2012. ISBN 978-1-931971-96-6. URL https://www.usenix.org/conference/osdi12/technical-sessions/presentation/ravindranath.

[60] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and scalable fault detection for mobile applications. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys'14, pages 190–203, Bretton Woods, New Hampshire, USA, 2014. ACM. ISBN 978-1-4503-2793-0. doi: 10.1145/2594368.2594377. URL http://doi.acm.org/10.1145/2594368.2594377.

[61] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and scalable fault detection for mobile applications. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys'14, pages 190–203, Bretton Woods, New Hampshire, USA, 2014. ACM. ISBN 978-1-4503-2793-0. doi: 10.1145/2594368.2594377. URL http://doi.acm.org/10.1145/2594368.2594377.

[62] T. Reps and G. Rosay. Precise interprocedural chopping. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, FSE'95, pages 41–52, Washington, D.C., USA, 1995. ACM. ISBN 0-89791-716-2. doi: 10.1145/222124.222138. URL http://doi.acm.org/10.1145/222124.222138.

[63] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP'10, pages 317–331. IEEE Computer Society, 2010. ISBN 978-0-7695-4035-1. doi: 10.1109/SP.2010.26. URL http://dx.doi.org/10.1109/SP.2010.26.

[64] H. Soroush, P. Gilbert, N. Banerjee, B. N. Levine, M. Corner, and L. Cox. Concurrent wi-fi for mobile users: Analysis and measurements. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT'11, pages 4:1–4:12, Tokyo, Japan, 2011. ACM. ISBN 978-1-4503-1041-3. doi: 10.1145/2079296.2079300. URL http://doi.acm.org/10.1145/2079296.2079300.

[65] R. Spahn, J. Bell, M. Lee, S. Bhamidipati, R. Geambasu, and G. Kaiser. Pebbles: Fine-grained data management abstractions for modern operating systems. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'14, pages 113–129. USENIX Association, Oct. 2014. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/spahn.

[66] A. Tongaonkar, S. Dai, A. Nucci, and D. Song. Understanding mobile app usage patterns using in-app advertisements. In *Proceedings of the 14th International Conference on Passive and Active Measurement*, PAM'13, pages 63–72, Hong Kong, China, 2013. Springer-Verlag. ISBN 978-3-642-36515-7. doi: 10.1007/978-3-642-36516-4_7. URL http://dx.doi.org/10.1007/978-3-642-36516-4_7.

[67] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON'99, pages 13–, Mississauga, Ontario, Canada, 1999. IBM Press. URL http://dl.acm.org/citation.cfm?id=781995.782008.

[68] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE'81, pages 439–449, San Diego, California, USA, 1981. IEEE Press. ISBN 0-89791-146-6. URL http://dl.acm.org/citation.cfm?id=800078.802557.