Tooling and Language Support for Robust and Easy Network Programming of Mobile Applications

Permalink
https://escholarship.org/uc/item/51g3s6kr

Author
Jin, Xinxin

Publication Date
2017

Peer reviewed|Thesis/dissertation
Tooling and Language Support for Robust and Easy Network Programming of Mobile Applications

A dissertation submitted in partial satisfaction of the requirements for the degree of Doctor of Philosophy in Computer Science by Xinxin Jin

Committee in charge:
Professor Yuanyuan Zhou, Chair
Professor William G. Griswold
Professor Ramesh Rao
Professor Alex Snoeren
Professor Geoff Voelker

2017
The Dissertation of Xinxin Jin is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2017
TABLE OF CONTENTS

Signature Page .......................................................... iii
Table of Contents ....................................................... iv
List of Figures ............................................................ vi
List of Tables ............................................................. viii
Acknowledgements ....................................................... ix
Vita ........................................................................... xi
Abstract of the Dissertation ............................................ xii

Chapter 1 Introduction ................................................... 1
  1.1 The Rise of Network Programming Defects ......................... 1
  1.2 Challenges of Mobile Network Programming ....................... 4
  1.3 Contribution ................................................................ 5

Chapter 2 Characteristic Study of Network Programming Defects .... 8
  2.1 Methodology .......................................................... 8
  2.2 Impact on User Experience .......................................... 10
  2.3 Root Cause Analysis ................................................ 13

Chapter 3 The State-of-the-art ............................................. 18
  3.1 Fault-tolerant features provided by Network Libraries ......... 19
  3.2 Network libraries fail to prevent NPDs ......................... 19
  3.3 Summary and Implication ........................................... 23

Chapter 4 NChecker: Detecting Network Programming Defects ....... 25
  4.1 Overview .................................................................. 25
  4.2 Identifying API Misuse Patterns ................................... 26
  4.3 NChecker Design and Implementation ............................ 30
    4.3.1 Library API Annotation ....................................... 30
    4.3.2 Analysis Algorithm ............................................ 31
    4.3.3 Identifying Customized Retry Logic ....................... 35
    4.3.4 NChecker Report ................................................ 37
    4.3.5 Limitations ....................................................... 38
  4.4 Evaluation ............................................................ 39
    4.4.1 Methodology ..................................................... 39
    4.4.2 Effectiveness ..................................................... 39
    4.4.3 Accuracy ........................................................ 45
Chapter 4.4 Practicality ................................................................. 46
4.5 Guidelines Towards User-Friendly, Robust Design of Mobile Network Library .................................................. 49
4.5.1 What Should Be Abstracted away from APIs? ............... 49
4.5.2 What Should be Exposed via APIs? ............................... 51

Chapter 5 ANEL: Robust Mobile Network Programming Using a Declarative Language ................................................. 53
5.1 Motivation ................................................................. 53
5.2 ANEL Declarative Language ........................................... 56
5.2.1 Property Specifications ........................................... 58
5.2.2 Modifier Clauses ..................................................... 60
5.2.3 Annotation Scopes ................................................... 61
5.2.4 Discussion and Limitations ....................................... 64
5.3 ANEL Library ............................................................... 65
5.3.1 Limitation and Discussion ....................................... 69
5.4 ANEL Compiler ............................................................ 69
5.4.1 Overview ............................................................ 69
5.4.2 Annotation Parser .................................................. 71
5.4.3 Conflict Resolver .................................................... 72
5.4.4 Code Generator ..................................................... 74
5.5 Evaluation ............................................................... 77
5.5.1 Overview ............................................................ 77
5.5.2 Case Studies ........................................................ 77
5.5.3 System Overhead .................................................... 83

Chapter 6 Related Work ......................................................... 85
6.1 Mobile App Testing ...................................................... 85
6.2 Mobile App Bug Detection ........................................... 87
6.3 Declarative Programming ............................................ 88
6.4 System-level Support for Network Operations .................... 89

Chapter 7 Conclusion and Future Work ...................................... 91

Bibliography ................................................................. 95
| Figure 1.1. | A problematic patch of Android app ChatSecure. The patch fails to handle login() failure under intermittent network. | 2 |
| Figure 1.2. | A buggy patch of Telegram. The patch fails to handle the frequent re-connections under poor network conditions. | 3 |
| Figure 2.1. | Distribution of NPD impact on user experience | 10 |
| Figure 2.2. | Bug in XBMC Remote that fails to catch null response | 16 |
| Figure 3.1. | The sensitivity of default API parameters to different network conditions | 21 |
| Figure 4.1. | 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. | 31 |
| Figure 4.2. | Identify the customized retry logic | 35 |
| Figure 4.3. | An example of NChecker warning report for GPSLogger | 38 |
| Figure 4.4. | Cumulative distribution of apps as a function of the ratio of requests missing connectivity check | 41 |
| Figure 4.5. | Cumulative distribution of apps as a function of the ratio of requests missing timeout APIs | 42 |
| Figure 4.6. | For apps that set but partially miss failure notifications, the CDF of the ratio of requests missing failure notifications | 45 |
| Figure 4.7. | 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. | 48 |
| Figure 5.1. | ANEL overview | 55 |
| Figure 5.2. | Add fault-tolerant features to Twitter Lite using hand-written native APIs vs. using ANEL annotations | 57 |
| Figure 5.3. | Scopes supported by ANEL. post() is lowest-level API to send out a HTTP request. Each arrow flow from the annotation indicates its control scope. | 62 |
Figure 5.4. Scope chaining and overriding ........................................... 64
Figure 5.5. Anel’s implementation of the Async HTTP class .................. 66
Figure 5.6. App runtime before enhancement vs. after enhancement .......... 67
Figure 5.7. The enhanced code of the code input shown in Figure 5.2(a). The bolded line is generated by the compiler and runtime. .......... 68
Figure 5.8. Annotated code as compiler input and enhanced code as compiler output. The output is written in Java for convenience. The real generated code is Jimple .......................................................... 70
Figure 5.9. We improve GpsLogger by applying two method-scoped annotations: “UserReq” is for user-initiated request and “!UserReq, SucceedEventually” is for background request. .................................................. 80
Figure 5.10. We improve Twitter client by applying a class-scoped annotation. 82
LIST OF TABLES

Table 1.1. The average age (time since first release) of the top 1,000 apps in two platforms, and the percent. of independent developers for these apps. ................................................................. 5

Table 2.1. 21 Android apps used in the study .................. 9

Table 2.2. Representative NPDs found in real world mobile apps ........... 11

Table 2.3. Root causes of studied NPDs ............................. 13

Table 3.1. Top libraries and their abilities in tolerating NPDs. ★ indicates tolerating the NPD automatically while ○ indicates providing APIs but requiring developers to set explicitly. .......................... 20

Table 4.1. API misuse patterns and examples. These patterns are derived from our characteristic study of NPD issues. All API examples are from the top network libraries in Table 3.1. ......................... 27

Table 4.2. Evaluated apps and their libraries ....................... 39

Table 4.3. Percent. of buggy apps detected by NChecker categorized by NPD causes................................................................. 40

Table 4.4. Timeout API invoke rate for different libraries .............. 43

Table 4.5. 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. ......................... 44

Table 4.6. NChecker results of open source apps. 61 NPDs were confirmed and fixed by developers. ........................................ 46

Table 4.7. Real world app NPDs used in our user study ................. 47

Table 4.8. Observations of applying NChecker in a large scale and the derived guidelines for designing mobile network libraries. ......................... 50

Table 5.1. Conflicted pairs of annotation specifications and fault-tolerant API settings for a given fault-tolerant (FT) primitive .................. 73

Table 5.2. Evaluated Android apps and applied ANEL annotations. ........ 78

Table 5.3. Latency to download 1K data for different HTTP clients ........ 84
ACKNOWLEDGEMENTS

Words cannot express my appreciation to people who have continuously supported me and helped me over the past six years.

It is my life honor to be advised of my advisor Yuanyuan Zhou. For the past six years, YY has kept inspiring me to step out the comfort zone, to dream big, to think deeply and to push the boundary of myself. When I was content with some superficial knowledge of one thing, YY always simulates me to go extra miles and dig deeper and deeper. When the world becomes dark and I feel so small, YY is always the most supportive one and encourage me not to give up. Her timely feedbacks and advices during the past years have kept making me become a better version of myself. Her spirit of questing the truth and perfection has driven me to go further on the road of exploration. She raises me up to more than I can be, and also sets up a role model for me to learn in my whole life.

I would like to thank Professor William Griswold who has guided me along the way in finishing the programming language part of the thesis. Bill broadens my views on the elegance of language designs. Without his efforts, I would never have achieved the milestone.

I would like to thank Professors Geoff Voelker, Alex Snoeren and Stefan Savrage, who have made SysNet an amazing group for every PhD student to exchange ideas and practice presentation skills.

Thanks to my thesis committee members Professors William Griswold, Geoff Voelker, Alex Snoeren, Ramesh Rao for their insightful feedbacks on my research work.

Graduate study is often fraught with self-doubt and helplessness, and I could not get through without my dear Opera friends. I cannot remember how many times they offer generous help to me. Working and having fun with them is the most memorable things of my days at UCSD. We have fought against countless obstacles and we share
joys with each other. When I was a junior students, their passion stimulated my interests on research. I give my heartfelt thanks to my dear colleagues: Tianyin Xu, Peng Huang, Xiao Ma, Soyeon Park, Tianwei Sheng, Ding Yuan, Jiaqi Zhang, Weiwei Xiong, Rishan Chen, and Xuepeng Fan.

Last but not least, I am thankful to my family for their care and love. I am so proud of inheriting my parents’ gene, integrity and perseverance, which allows me not to be afraid of every difficulty. I am so grateful that my husband has been my dearest soulmate and strongest supporter over these years.

Chapter 2, in part, is a reprint of the material as it appears in Proceedings of the Eleventh European Conference on Computer Systems. Jin, Xinxin; Huang, Peng; Xu, Tianyin; Zhou, Yuanyuan 2016. The dissertation author was the primary investigator and author of this material.

Chapter 3, in part, is a reprint of the material as it appears in Proceedings of the Eleventh European Conference on Computer Systems. Jin, Xinxin; Huang, Peng; Xu, Tianyin; Zhou, Yuanyuan 2016. The dissertation author was the primary investigator and author of this material.

Chapter 4, in part, is a reprint of the material as it appears in Proceedings of the Eleventh European Conference on Computer Systems. Jin, Xinxin; Huang, Peng; Xu, Tianyin; Zhou, Yuanyuan 2016. The dissertation author was the primary investigator and author of this material.

Chapter 5, in part is currently being prepared for submission for publication of the material. Jin, Xinxin; Griswold, William G; Zhou, Yuanyuan. The dissertation author was the primary investigator and author of this material.
VITA

2004 Bachelor of Engineering, Beijing University of Posts and Telecommunications
2004–2008 Master of Science, Peking University
2011–2016 Research Assistant, University of California, San Diego
2017 Doctor of Philosophy, University of California, San Diego

PUBLICATIONS


FIELDS OF STUDY

Major Field: Computer Science

Studies in Computer Systems
Professor Yuanyuan Zhou

Studies in Software Engineering
Professor William G. Griswold
ABSTRACT OF THE DISSERTATION

Tooling and Language Support for Robust and Easy Network Programming of Mobile Applications

by

Xinxin Jin

Doctor of Philosophy in Computer Science

University of California, San Diego, 2017

Professor Yuanyuan Zhou, Chair

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 thesis, 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 by providing flexible fault-tolerant APIs, we observe that many app developers often ignore such APIs or lack the capability to use them correctly and therefore still introduce NPDs.

In this thesis we make three contributions towards robust network programming for mobile apps. In the beginning, we study the characteristics of 90 real-world NPDs in Android apps towards a deep understanding of their impacts, root causes, and code patterns. This study exposes the common mistakes and pitfalls made by app developers when dealing with intermittent network. It also reflects that NPDs are pervasive despite the availability of fault-tolerant API from third-party network libraries.

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. Shockingly, NChecker detects network bugs in 99% of the evaluated apps. 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.

Besides NPD detection, a more fundamental way to eliminate NPDs is to provide developers easy-to-use interfaces for network programmings. We argue that the difficulty of avoiding NPDs can be mitigated through an annotation language that allows developers to declaratively state desired and actual properties of the application, largely without reference to fault-tolerant concepts, much less implementation. A pre-compiler can process these annotations, replacing calls to standard networking libraries with customized calls to a specialized library that enhances the reliability. The last component of the thesis presents ANEL, a declarative language and middleware for Android that enables non-experts. We demonstrate the expressiveness and practicability of ANEL.
annotations through cases studies on real-world networked mobile apps. We also show that the ANEL middleware introduces negligible runtime performance overhead.
Chapter 1

Introduction

1.1 The Rise of Network Programming Defects

With mobile applications rapidly growing in the last years [59], it is critical for an app to provide good user experience in order to win the brutally competitive marketplace. Bad user experience is reported to be the leading cause for users to uninstall the app [28], and even for small companies to fail [1]. One of the major contributors to mobile UX is the app’s ability to 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 [111]. For mobile users who heavily rely on networked apps in the life, they usually have a high expectation on the app despite mobility. Apps that fail to handle network problems well are likely to receive low star-rating reviews [58, 57, 84]. Below shows some 1-star reviews of frequently downloaded apps in the Google Play Store because the apps fails to deliver expected functionalities when network problems occur.

“When I download it recently it keep crashing when connected to internet.”
   – A review of Fun run 2 [29]

“Sometimes chat messages take forever to load.”
   – A review of Steam [50]

“Wheter I am connected to wifi or using my own data bitstrips keeps saying network error? This sucks big time.”
   – A review of Bitstrips [21]
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 1.1. A problematic patch of Android app ChatSecure. The patch fails to handle login() failure under intermittent network.

Mobile networks are fundamentally different from traditional desktop-/server-based networks for the dynamics incurred by mobility. In a desktop environment where the client application is connected with stable and high-speed network, network failures are rare and therefore have a low chance to influence broad users even if developers fail to handle them. On the contrary, 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) [65, 109]. 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 thesis, 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.
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 */
        %!
      }
    }
  });
}

Figure 1.2. A buggy patch of Telegram. The patch fails to handle the frequent re-connections under poor network conditions.

In fact, we detect NPDs in 98+% of the evaluated mobile apps (c.f., §4.4).

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 1.1 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. The root cause is that the buggy code did not verify whether the connection initialization is successful or not before login. When later the developers observe crash in offline mode, they add a patch which only enables login when network is available. At the first glance, this patch seems solve the problem perfectly, and the app will not crash in offline mode. 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. The correct fix should be adding error handling code when `login()` operation fails.

Figure 1.2 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 Challenges of Mobile Network Programming

The emerging pervasiveness of NPD issues is a collateral consequence of the mobile ecosystem changes:

(1) **Mobile apps are in general young.** We examined the top 1,000 apps in two platforms and found that the average age of these apps is 2.5 years for App Store, and 1.5 years for Google Play (Table 1.1), even though both app stores started in 2008. Therefore, many popular apps have not aged enough to be stable and well engineered. In comparison, popular desktop/server applications are much more mature: e.g., Firefox is 12 years old and MySQL is 19 years old.

(2) **Mobile app developers tend to be “young” too.** Unlike desktop and server applications that are typically developed and maintained by experienced teams, mobile apps are often written by novice developers or small teams (attributable to the mature SDK) [19, 49, 34, 55]. According to a survey of over 10,000 app developers in the
Table 1.1. The average age (time since first release) of the top 1,000 apps in two platforms, and the percent. of independent developers for these apps.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Avg. app age</th>
<th>% of inde developers</th>
</tr>
</thead>
<tbody>
<tr>
<td>iOS</td>
<td>2.5 years</td>
<td>&gt;12%</td>
</tr>
<tr>
<td>Android</td>
<td>1.5 years</td>
<td>&gt;5%</td>
</tr>
</tbody>
</table>

UK, 83% of app developers are self-taught and only 7% have attended a Bootcamp or other training course [55]. Many of these novice app developers lack professional programming training, and thus have trouble reasoning about potential network-related issues such as disruptions and re-connections, not to mention to handle these issues well. Some mobile developers come from the background of building desktop or server software and are uninformed about the distinct complexities of mobile networking.

(3) Individuals and small developer teams are usually limited in development and testing resources. A lot of mobile apps are developed by independent developers or small teams. From Table 1.1 we can see at least 12% of the developers for the top 1,000 apps in the App Store are individuals. 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 [27], and develop large-scale testing frameworks [14, 38, 45] to test the app under different environments. However, this is clearly not affordable for most app developers.

1.3 Contribution

The goals of this thesis are two-fold: (1) Understanding the common mistakes of mobile network programming. (2) Help non-expert mobile app developers avoid such mistakes.

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 characteristic study reveals that intermittent network significantly results in bad user experience, but many developers lack experience in handling it. We find that the common mistakes made by developers include: assuming stable network connection, mishandling transient network error, mishandling permanent error, and failing to react to network switch.

(2) We examine the library APIs of popular network libraries and find that even these libraries have some error handling features, they cannot prevent network bugs mostly because app developers cannot use library APIs correctly. We generalize a set of API misuse patterns that can cause network bugs.

(3) 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.

(4) Furthermore, to help app developers prevent NPDs at the first place, we develop ANEL, a declarative language and middleware that provides Android developers an easy-to-use interface for network programming. Programmers define high-level application requirements through Java annotations, which are translated into fault-tolerant code supported by the ANEL library and runtime. Key to making this solution tractable is our characteristic study of NPDs. The declarative language is generalized from these NPDs, thus handling the vast majority of issues that would be encountered in the typi-
cal mobile app. Our evaluation shows ANEL can improve the robustness of real-world networked apps without need for referencing complex fault-tolerant concepts or their implementation.
Chapter 2

Characteristic Study of Network Programming Defects

2.1 Methodology

As we need to understand the detailed root cause of each issue at the source-code level, not just user observable symptoms, we study mobile app projects with open issue tracking and version control systems, including the app projects hosted on GitHub [31], Google Code [32], Mozilla’s Bugzilla [22], and Android Framework [2]. 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. Some bugs selected by these keywords may not be closely related to mobile environment, such as concurrency bugs or wrong protocol implementation. 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 offline conditions). In total, we collected 90 NPDs from 21 open source Android app projects (Table 2.1).

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.
Table 2.1. 21 Android apps used in the study

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

For the app representativeness, the studied apps cover a wide range of categories including messaging, news, social, and audio, etc. Different network protocols can require different reliability level of network, so we also pay attention to collect apps using different protocols. The used protocols in our studied apps include HTTP, XMPP, XMLRPC, IMAP, POP3, WEBDAV, UDP, SSH, RTMP and SIP.

As for our collection method, we have carefully examined every piece of information related to the bug, including all the discussion threads, source code patches and code comments. For some code patches for unclear reasons, we manually reproduce the bug or check with developers to make sure we understand the root causes. 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 2.1 shows the four categories of common UX impacts and their distributions.

**Dysfunction (36%):** defects that impair app network operations and break the expected functionality. We do not claim that apps should be able to deliver every network request successfully. Here “Dysfunction” means the app developers did not try their best effort to improve the reliability of mobile apps against poor network. Dysfunction could be very severe. For example, the defect in Table 2.2(iii) causes users’ data losses. Other dysfunction NPDs bother users to redo certain operations, such as re-sending the message in Table 2.2(i)’s example; Or cause duplicated messages such as the one in Table 2.2(ii).

**Unfriendly UI (33%):** defects that do not deliver friendly or seamless UIs for...
### Table 2.2. Representative NPDs found in real world mobile apps

<table>
<thead>
<tr>
<th>ID</th>
<th>Category</th>
<th>App</th>
<th>NPD description</th>
<th>Developer’s fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i)</td>
<td>Dysfunction</td>
<td>AndStatus</td>
<td>The client repeats retrying post requests, resulting in duplicated posts</td>
<td>Increase timeout accordingly</td>
</tr>
<tr>
<td>(ii)</td>
<td>Dysfunction</td>
<td>Firefox</td>
<td>The download fails due to transient network errors</td>
<td>Add retry on conn. failures</td>
</tr>
<tr>
<td>(iii)</td>
<td>Dysfunction</td>
<td>Yaxim</td>
<td>The sent message is lost on network failure</td>
<td>Queue the message for re-sending</td>
</tr>
<tr>
<td>(iv)</td>
<td>Unfriendly UI</td>
<td>Hacker News</td>
<td>No indication if the feeds loading fails</td>
<td>Add error message</td>
</tr>
<tr>
<td>(v)</td>
<td>Unfriendly UI</td>
<td>gtalkSMS</td>
<td>Permanently display “Waiting for network to become available” every time losing signal</td>
<td>Remove unnecessary notification</td>
</tr>
<tr>
<td>(vi)</td>
<td>Crash</td>
<td>ChatSecure</td>
<td>Do not handle no connection exception on login</td>
<td>Add catch blocks</td>
</tr>
<tr>
<td>(vii)</td>
<td>Freeze</td>
<td>Chrome</td>
<td>Failed XMLHttpRequest on webpage freezes the WebView</td>
<td>Cancel the request on failure</td>
</tr>
<tr>
<td>(viii)</td>
<td>Battery drain</td>
<td>Kontalk</td>
<td>Frequent synchronizations in offline mode</td>
<td>Disable synchronization in offline</td>
</tr>
</tbody>
</table>
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 [3]. We observe that many patches have been added to display error messages upon network failures, such as the ones in Table 2.2(iv). Suppose there is no error message, when users fail to load new feeds, they do not know whether it is because there are no new feeds coming or because the network causes the problem. Similarly, when users send a message, they could suppose the sent message are always delivered without failure notification. The user interactive characteristic of mobile apps makes the UI messages super important. On the other hand, apps cannot over bother the user every time it fails, as in the defect in Table 2.2(v). All the reconnection or retry behaviors should be hidden from user interface.

**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 2.2(vi). 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 2.2(vii).

**Battery drain (10%)**: excessive network requests can cause heavy energy usage and drain the battery fast because WiFi and cellular network components are highly energy consuming. The example in Figure 1.2 belongs to this category, which aggressively reconnects on failure and causes battery drain. Table 2.2(viii) shows a similar example.
Table 2.3. Root causes of studied NPDs

<table>
<thead>
<tr>
<th>Root cause</th>
<th># Cases (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No connectivity checks</td>
<td>27 (30%)</td>
</tr>
<tr>
<td>Mishandling transient error</td>
<td>12 (13%)</td>
</tr>
<tr>
<td>Mishandling permanent error</td>
<td>24 (27%)</td>
</tr>
<tr>
<td>Mishandling network switch</td>
<td>27 (30%)</td>
</tr>
</tbody>
</table>

2.3 Root Cause Analysis

Table 2.3 categorizes the root causes of the studied NPDs and their distributions.

**Cause 1: No connectivity checks (30%).** In the desktop environment, once a connection is established, one can assume that it can last for a comparatively long time. 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 [3, 4] recommend the following practices for disconnected conditions: if the request is initiated by users, the app should display the network status information to users. Although the network request passing connectivity check could still fail and need further error handling, connectivity check renders responsive UX the early check can return to user fast and give user explicit information so they know how to fix it. If the request is initiated in background, the app should cache and stop the operation to avoid wasting energy and mobile data. Because every time the app initiates a connection, irrespective of the size of the associated data transferred, it potentially causes the radio to draw power for nearly 20 seconds when using a typical 3G wireless radio [5]. 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 to enhance the chance of a request being delivered on transient network errors. However, defining retry policies is non-trivial for mobile apps and should follow the following principles [3]: (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 [20]. As defined in HTTP/1.1: “A user agent MUST NOT automatically retry a request with a non-idempotent method.” So developers should be careful in retrying requests that have side effects.
**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 [56]. 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%).** This is one major reason for negative user experience, but often ignored by developers. Clear and easy-to-understand error message not only keep user aware of the problem, but also help them quickly troubleshooting and take actions, even for users with little technical background. Not notifying users upon network failures is not acceptable, as exemplified by Table 2.2(iii). What is worse is the misleading error message. In one bug in iOS Wordpress [6], 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. If the message correctly notifies users that the error is caused by a navigation, users can easily take actions to recover the error. Therefore, some patches are added to deliver the message according to different error types.

- **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.
```java
try {
    if (conn.getResponseCode() == HttpURLConnection.HTTP_UNAUTHORIZED) {
        e = new UnauthorizedException;
    } else if (conn != null) {
        if (conn.getResponseCode() == HttpURLConnection.HTTP_UNAUTHORIZED) {
            e = new UnauthorizedException;
        }
    }
} catch (IOException e) {
}
```

Figure 2.2. Bug in XBMC Remote that fails to catch null response

Figure 2.2 shows a code patch from Android app XMBC Remote. The code tries to check the HTTP status code from the HttpURLConnection client object `conn`. But when there is a network failure, the object `conn` can become null. So the code patch adds a null check before referring to the object.

**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. Although mobile users anticipate seamless operations for network apps, today’s operating system and network protocol provide little support in managing the host sporadic movement [108]. As such, mobile developers are supposed to handle these adverse conditions in their apps. 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. Every time the network changes, the old connection becomes stale, and can lead to buggy behaviors if the app does not explicitly build new connections. In a bug
report of the chat app GtalkSMS [35], 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%)** Unlike desktop users who get used to high quality network connection, Mobile app may frequently experience disconnection. Failing to automatically recover the failed request also frustrate users [7]. 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.

Chapter 2, in part, is a reprint of the material as it appears in Proceedings of the Eleventh European Conference on Computer Systems. Jin, Xinxin; Huang, Peng; Xu, Tianyin; Zhou, Yuanyuan 2016. The dissertation author was the primary investigator and author of this material.
Chapter 3

The State-of-the-art

Big companies like Google and Facebook [14, 38, 45] have put major efforts to build testing infrastructures, but it is too costly and time-consuming for individual developers.

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, such as OkHttp [8] and Volley [9] for HTTP, aSmack [10] for XMPP, PJSIP for SIP [11], 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. Unfortunately, our investigation shows that these fault-tolerant APIs are insufficient to prevent NPDs. This chapter discusses the fault-tolerant features of those libraries, and answer the fundamental question, “why do these network libraries fail to prevent NPDs?”

We study the six most widely used mobile network libraries [15], including HttpURLConnection client, Apache HttpClient, Google Volley, OkHttp, Android Asyn-
chronous HTTP client, and Basic HTTP client. The first two are Android native libraries and the others are third-party libraries.

3.1 Fault-tolerant features provided by Network Libraries

Table 3.1 summarizes how the six libraries 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. For example, to deal with transient error, OkHttp will aggressively retry the failed requests on a slow/unreliable connection until it succeeds; Volley will retry one more time if the request fails. Although these two libraries adopt different retry policies, both of them internally tolerate the corresponding NPD. ○ 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. The library itself does not set any default value to tolerate “no timeout” defect.

3.2 Network libraries fail to prevent NPDs

While these libraries hide many low-level programming details and offer 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 [12]. The app developer complained about the default timeout value being too short that caused frequent timeout exceptions. But in the library developer’s opinion, different applications may have different requirements and therefore setting a timeout value that fits all apps is out
Table 3.1. Top libraries and their abilities in tolerating NPDs. ★ indicates tolerating the NPD automatically while ○ indicates providing APIs but requiring developers to set explicitly.

<table>
<thead>
<tr>
<th>NPD Causes</th>
<th>HttpURL Connection</th>
<th>Apache HttpClient</th>
<th>Volley Library</th>
<th>OkHttp Library</th>
<th>Android Async</th>
<th>Basic HTTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>No connectivity check</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>No retry on transient error</td>
<td>★</td>
<td>○</td>
<td>★</td>
<td>★</td>
<td>○</td>
<td>★</td>
</tr>
<tr>
<td>Over retry</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>No timeout</td>
<td>○</td>
<td>○</td>
<td>★</td>
<td>○</td>
<td>★</td>
<td>★</td>
</tr>
<tr>
<td>No/Misleading Failure notification</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>No invalid response check</td>
<td>○</td>
<td>○</td>
<td>★</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>No reconnetion on net switch</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>No auto failure recovery</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>
Figure 3.1. The sensitivity of default API parameters to different network conditions

**App developer:** “My requests to call this API often timeout when the default timeout is 2500ms. Should the API be re-designed?”

**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 [3] to control and throttle the network traffic. Figure 3.1 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. In Figure 3.1(a), we run the test in a simulator with standard 3G network. Until the file size increases to 1M, the majority of transmissions succeed. In Figure 3.1(b), we simulate the unreliable network by increasing the package loss rate of 3G to 10%. Under this poor condition, even downloading a 2K size file encounters random failures. When the file size increases to 64K and larger, the network requests can hardly get a response. The results indicate that app developers should tune the
API parameters carefully for 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. Because the main motivation for those developers to use these libraries is to avoid writing complex code in making a network request, such as making asynchronous requests out of UI thread, building request of different types, etc. they are even not aware of the existence of the reliability related APIs until encountering problems. 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 [13].

An analysis of the domain knowledge needed to handle NPDs (summarized in § 2.3) can give a better understanding of why it is hard for novice developers to cope with these APIs:

- **No connectivity check:** To avoid the problem, Developers have to consider every network operation, yet non-experts often assume a reliable network.

- **No timeout:** To set a timeout correctly, a developer needs to learn about the socket and possible exception types (e.g. `SocketTimeoutException, ConnectTimeoutException`) and set an appropriate timeout value. Also the developer needs to be aware that mobile network bandwidth can vary widely, which is quite different from the high-speed network available in the app’s development environment.

- **No retry on transient error:** To use the retry APIs, a developer not only needs to know what are the possible exceptions, but also needs to know what types are retriable. Yet, a novice developer may have never heard of a retry mechanism at
all. They also need to keep user experience in mind because the app cannot just retry indefinitely under the ground and the user has no clue what is happening.

- Over retry: To avoid over retry, developers need to have a firm grasp of the HTTP protocol and the particular server’s POST semantics; they also need to be able to differentiate time-sensitive requests with time-insensitive ones.

- No invalid response check: To avoid null response value, developers always have to add extra checking code before using the response value.

- No reconnection on network switch: To deal with this, a developer not only has to know that reconnection is necessary to reduce latency, but also needs to know how to detect network switches and reconnect using complex native APIs.

- No auto failure recovery: Failure recovery requires the developer to pay attention to the user experience when offline, which is very important for a mobile app. In addition, the fault-tolerant implementation is very complex, including saving failed requests, detecting resumed network, and resending the requests.

### 3.3 Summary and Implication

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, 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.

Based on this observation, we make two efforts in helping existing apps get rid of NPDs: In Chapter 4, we build a tool to automatically detect NPDs from the app’s binary code; In Chapter 5, we design a declarative annotation language for novice developers to easily specify fault-tolerant specifications.

Chapter 3, in part, is a reprint of the material as it appears in Proceedings of the Eleventh European Conference on Computer Systems. Jin, Xinxin; Huang, Peng; Xu, Tianyin; Zhou, Yuanyuan 2016. The dissertation author was the primary investigator and author of this material.
Chapter 4

NChecker: Detecting Network Programming Defects

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

4.1 Overview

NChecker detects NPDs in apps (as exemplified in Table 3.1). 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 [113]. Before doing the analysis, we transform the APK into an intermediate representation of a Java program Jimple [67] and build the program call graph by extending FlowDroid [62]. After finishing analysis, it generates NPD reports to developers.

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 and experiences.

To address the first challenge, NChecker follows the observations summarized in § 2.3. We discuss the NPD patterns in § 4.2 and how NChecker identifies these patterns in § 4.3.2. For the second challenge, NChecker recognizes the customized retry logics by catching retry loops (c.f., § 4.3.3), 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., § 4.3.4).

### 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 4.1 shows these patterns. Column 2 shows NPD root causes corresponding to each pattern, and Column 3 gives examples of API misuses from network libraries in Table 3.1.

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 4.1 gives two
Table 4.1. API misuse patterns and examples. These patterns are derived from our characteristic study of NPD issues. All API examples are from the top network libraries in Table 3.1.

<table>
<thead>
<tr>
<th>API misuse pattern</th>
<th>NPD cause</th>
<th>Example of identifying misuse in the code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miss request setting APIs</td>
<td>No connectivity check</td>
<td>Do not call <code>getNetworkInfo</code> to check connectivity before sending a network request</td>
</tr>
<tr>
<td></td>
<td>No retry on transient error</td>
<td>Do not call <code>setMaxRetries</code> to set retry times for sent network request</td>
</tr>
<tr>
<td></td>
<td>No timeout</td>
<td>Do not call <code>setReadTimeout</code> to set timeout for sent network request</td>
</tr>
<tr>
<td>Improper API parameters</td>
<td>Over retry</td>
<td>Set <code>retires \geq 0</code> in <code>setMaxRetries</code> in Android Service or POST request</td>
</tr>
<tr>
<td>No/implicit error message in request callbacks</td>
<td>No failure notification</td>
<td>Do not call <code>Toast.show</code> to display a UI message in <code>onErrorResponse()</code> of a network request made by user</td>
</tr>
<tr>
<td>Miss resp. checking APIs</td>
<td>No invalid resp. check</td>
<td>Do not call <code>isSuccessful()</code> to check the response status before reading the response body</td>
</tr>
</tbody>
</table>
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 4.1 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). Therefore it is also bad if developers do not invoke the APIs to set the retry times in Service or POST requests.

**Pattern 3: No/Implicit error messages in request callbacks.** Apps do not show explicit error messages after finishing the networking operations. In particular, we check two sub-patterns:

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 4.1 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, such as TimeoutError, NoConnectionError, AuthFailureError or ServerError. 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. For ServerError 5xx, developers can do retry or 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. For example, Basic HTTP client provide APIs in the format of `HttpResponse response = httpClient.get(url)`. During its execution, if some network disruption happens, the API internal logic will catch the exception, but return a null response. If the developers directly call `response.getBodyAsString()` afterwards without the null check, the code will generate a Null Pointer Exception in this situation. Similarly, some other APIs expose the responses of invalid HTTP status code (e.g. HTTP 404, HTTP 503). For example, using Basic HTTP client, developers need to call `response.getStatus()` and verify the obtained HTTP status code. Table 4.1 lists a response checking API from OkHttp library `response.isSuccessful()`, which enables developers to filter out the response of invalid status code.

Please notice that we do not deal with the situations where libraries do not provide configuration APIs. For example, if the server has multiple IP addresses, OkHttp silently retries all of them once encountering transient errors, but it does not provide any
interface for developers to configure the retry policies. Therefore it is out of developers’ control.

4.3 NChecker Design and Implementation

4.3.1 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, and `HttpClient.execute()` in Apache library.

- **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()`.

Please note that NChecker is designed to check the network library API usages, so these annotated APIs are identified in library level, according to which library NChecker check. For example, Volley’s target API encapsulates Android native API `HttpURLConnection`, but NChecker will not count the latter as a target API.

NChecker’s current implementation annotates APIs from the 6 most popular HTTP libraries mentioned in Chapter 3. In total, we annotate 14 target APIs, 77 config APIs, and 2 response checking APIs from the six libraries.
Class A extends AsyncTask {
    @Override
    void doInBackground() {
        c = new BasicHttpClient();
        c.setMaxRetries(5);
        r = c.get(url);
        r.getBodyAsString();
    }

    @Override
    void onPostExecute() {
    }
}

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

Figure 4.1. 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.

4.3.2 Analysis Algorithm

To detect the API misuse patterns in § 4.2, NChecker performs four different analyses. Figure 4.1 illustrates the analysis algorithm for detecting NPDs based on a real world 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.

4.3.2.1 Identifying Network Operations

Based on the call graph, NChecker iterates through all application classes and determines the call sites of target APIs. If there is no target API, the application will not be analyzed.

In Figure 4.1 step ①, NChecker explores the path from entry point onClick() (the callback of a click event) and find out the path can reach the target API get().
4.3.2.2 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. Because there could be many reachable paths from one app’s entry points to the same target API, NChecker explores all the paths and performs check on config APIs along each path. If there are multiple target APIs in one code path, there means there are multiple network requests. NChecker matches the connectivity checking API to the closet one. For the requests not guarded by the connectivity checks, NChecker raises alarms. In Figure 4.1 (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 [106, 62, 76, 72, 78, 110] 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 4.1, the taint analysis taints the client object c, performing both backward and forward analysis, and getting it has config API setMaxRetries(). As mentioned before, NChecker takes application’s ad-hoc retry logic into consideration in addition to the standard APIs. § 4.3.3 discusses how NChecker handles ad-hoc retries.
4.3.2.3 Checking Improper API Parameters

This phase attempts to infer the call context of a network request, e.g. called from an Activity or a Service, and analyzes if the app set inappropriate API parameters in this context. 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 [60]. If the config API is not invoked along the path, NChecker uses its default value.

In Figure 4.1 (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 (§ 4.3.3).
4.3.2.4 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. We parse the class hierarchy of the callback classes and extract the related 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 § 4.3.2.3.

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 4.1 (Step ③), NChecker finds no failure notification in the callback onPostExecute(), so it raises an alarm.

4.3.2.5 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. 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
Figure 4.2. Identify the customized retry logic

on the path, an alarm is raised. For instance, in Figure 4.1 (Step ④), the response object \( r \) is tainted and propagated. NChecker will alarm that there is no validity check before the use of \( r \).

### 4.3.3 Identifying Customized Retry Logic

The program may not rely on an external library’s retry logic, but implement their own. To improve the analysis accuracy, NChecker needs to identify the self-implemented 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 4.2(a)). Here “success” means the network operation does not fail due to network failure, otherwise it will throw an **IOException**. 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 4.2(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 4.2(c) shows a retry example that the exit variable retry is dependent on the statement “retry=shouldRetry()” in the catch block. Figure 4.2(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:

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 [114, 104, 81] is used to obtain the control dependency information.

   We do inter-procedural control flow and dataflow analysis, tracking the call chain backwards from the loop to the network target API. Along the call chain, we find out the data dependencies of the control variable. This process is similar to static backward
Finally, we check if there are exit dependencies in the catch block. And the existence of such dependency indicates a retry loop. Algorithm 25 shows the algorithm of retry loop identifier.

Algorithm 1. Algorithm of identifying retry loop

```plaintext
function RETRYLOOPIDENTIFIER(loop)
    isRetry ← false
    Su ← loop unconditional exit conditions
    Sc ← loop conditional exit conditions
    callStack ← call stack from the loop to the target API
    for each s ∈ Su do
        if s is not reached by catch block then
            isRetry ← true
        end if
    end for
    for each s ∈ Sc do
        V ← s's exit condition variables
        for each v ∈ V do
            while callStack is not empty do
                caller ← callStack.pop()
                if v is not dependent on caller then
                    break
                end if
                if v depends on catch block then
                    isRetry ← true
                end if
            end while
        end for
    end for
    return isRetry
```

4.3.4 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)
Use `getActiveNetworkInfo()` to check connectivity before `HttpClient.get()`. Show error message if no connection.

### 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 4.3.** An example of NChecker warning report for GPSLogger

**Request context:** if the request is made by users or a background service. (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 4.3 shows a NChecker report of the GPSLogger app.

### 4.3.5 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 [98], 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 [17]).
Table 4.2. Evaluated apps and their libraries

<table>
<thead>
<tr>
<th>Lib used</th>
<th># Apps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>270</td>
</tr>
<tr>
<td>Volley</td>
<td>78</td>
</tr>
<tr>
<td>Android Async Http</td>
<td>25</td>
</tr>
<tr>
<td>Basic Http</td>
<td>18</td>
</tr>
<tr>
<td>OkHttp</td>
<td>11</td>
</tr>
</tbody>
</table>

4.4 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?

4.4.1 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 4.2.

4.4.2 Effectiveness

NChecker discovers 4180 new NPDs in nearly all (281) of the 285 apps (see details below). Table 4.3 summaries the percentage of buggy apps detected by NChecker.
Table 4.3. Percent. of buggy apps detected by NChecker categorized by NPD causes.

<table>
<thead>
<tr>
<th>NPD cause</th>
<th>Eval. condition</th>
<th># Eval. apps</th>
<th># Buggy apps (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Missed conn. checks</td>
<td>All apps</td>
<td>285</td>
<td>122 (43%)</td>
</tr>
<tr>
<td>Missed timeout APIs</td>
<td>Use libs that have timeout APIs</td>
<td>285</td>
<td>139 (49%)</td>
</tr>
<tr>
<td>Missed retry APIs</td>
<td>Use libs that have retry APIs</td>
<td>91</td>
<td>64 (70%)</td>
</tr>
<tr>
<td>Over retries</td>
<td>Use libs that have retry APIs</td>
<td>91</td>
<td>50 (55%)</td>
</tr>
<tr>
<td>Missed failure notifications</td>
<td>Include user initiated requests</td>
<td>264</td>
<td>151 (57%)</td>
</tr>
<tr>
<td>Missed response checks</td>
<td>Use libs that have resp. check APIs</td>
<td>20</td>
<td>15 (75%)</td>
</tr>
</tbody>
</table>
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 [47], F-Droid [26], Kontalk [42], Galaxy zoo [30], AnkiDroid [18] and GPS Logger [33], 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.

4.4.2.1 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 4.3. We further look at apps that have invoked config APIs and find that the coverage of config APIs is low.
**Figure 4.5.** Cumulative distribution of apps as a function of the ratio of requests missing timeout APIs.

**43% of apps never check network connectivity.** For the remaining 163 (57%) apps that have but partially miss connectivity checking, Figure 4.4 further shows 62% of apps miss connectivity checking in over half of their requests. This result is consistent to the observation in our study: developers often do not pay attention to checking connectivity. Also, when the number of network requests is big, developers can hardly add complete checks before every request.

**49% of apps never set timeout APIs.** For the remaining 146 (51%) apps that set but partially miss timeout APIs, Figure 4.5 shows 58% of apps miss timeout setting in over half of the requests.

We notice that among all the invoked timeout APIs, developers set Volley library’s timeout API (invoked in 42.8% paths to Volley callsite APIs) more frequently than other libraries (Table 4.4). One possible reason is that Volley has very short default timeout value, which is only 2.5 seconds, and thus easily result in SocketTimeoutException in real world apps.
Table 4.4. Timeout API invoke rate for different libraries

<table>
<thead>
<tr>
<th>Lib</th>
<th>Default timeout(s)</th>
<th>Invoked rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>NA</td>
<td>32%</td>
</tr>
<tr>
<td>OkHttp</td>
<td>NA</td>
<td>33%</td>
</tr>
<tr>
<td>Volley</td>
<td>2.5</td>
<td>42%</td>
</tr>
<tr>
<td>Android Async</td>
<td>10</td>
<td>1%</td>
</tr>
<tr>
<td>Basic Http</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

70% of apps never set retry APIs. For 30% apps that set retry APIs, we noticed that 99% of them use Volley library. Different from other libraries which separate timeout and retry APIs, Volley uses the same API `DefaultRetryPolicy()` to set both timeout and retry. So when developers need to modify the timeout value, they pay attention to the retry policy too. We can infer that the reason for most developers not to set retry APIs is they are not aware of the existence of the 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.

4.4.2.2 Improper Retry Parameters

We examine the improper retry parameters in 91 apps that use libraries with retry APIs. Table 4.5 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
Table 4.5. 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.

<table>
<thead>
<tr>
<th>NPD cause</th>
<th>Apps(%)</th>
<th>Default behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>No retry in Activities</td>
<td>8%</td>
<td>0</td>
</tr>
<tr>
<td>Over retry in Services</td>
<td>32%</td>
<td>76%</td>
</tr>
<tr>
<td>Over retry in POST requests</td>
<td>25%</td>
<td>98%</td>
</tr>
</tbody>
</table>

apps over retry in POST requests.

Interestingly, the majority of the over retry behaviors are caused by the library default behaviors. As shown in the third column of Table 4.5, 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.

4.4.2.3 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 4.6 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.

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.

4.4.2.4 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.
Figure 4.6. For apps that set but partially miss failure notifications, the CDF of the ratio of requests missing failure notifications

### 4.4.3 Accuracy

Since it is hard to verify the accuracy on closed source apps, we measure the accuracy of NChecker using 16 open-source apps. We manually verify the detected results by examining the source code, and segregating the imprecise cases into two types:

1. **False positive (FP):** cases detected by the tool but we did not manually find them;
2. **False negative (FN):** cases we found manually but not detected by the tool. Table 4.6 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 apps because these two apps check connectivity before starting the activity that sends network requests, but NChecker does not handle Android inter-component communication.

\(^1\)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.
Table 4.6. NChecker results of open source apps. 61 NPDs were confirmed and fixed by developers.

<table>
<thead>
<tr>
<th>NPD cause</th>
<th># Correct warning</th>
<th># FP</th>
<th># Known FN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Missed conn. checks</td>
<td>31</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Missed timeout APIs</td>
<td>58</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Missed retry APIs</td>
<td>12</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Over retries</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Missed failure notifications</td>
<td>20</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Missed response checks</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>130</strong></td>
<td><strong>9</strong></td>
<td><strong>5</strong></td>
</tr>
</tbody>
</table>

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.

4.4.4 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 § 2.3. Table 4.7 shows the NPDs reported by NChecker that used in our user study. We recruited 20 undergrad volunteers who indicate they know Android programming 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 4.7. We measured the time they spent on fixing each NPD.

The potential biases should be considered when interpreting the results. Below we discuss some potential bases:

*Bias in case selection:* In bug selections, we avoid to choose bugs that need to be fixed by complex code, but only choose ones that can be fixed by a few lines of code.
Table 4.7. Real world app NPDs used in our user study

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

The reason is the latter can better measure the time they understand the problem, rather than learning time. If we give them complex cases, most of the time will be spent on learning, not bug fixing.

*Bias in methodology:* Unlike previous user studies [117, 112] which typically compare the result with a baseline, there is no baseline in NChecker user study. Without NChecker, programmers even are not aware of the symptoms, so it is unfair to measure the bug fixing time by telling them the potential symptoms detected by NChecker. However, as we will show in the result, the absolute time of fixing bugs is very short, which is enough to demonstrate the effectiveness of NChecker.

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 4.7 shows our study results. On average programmers take
Figure 4.7. 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.

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. That is why adding connectivity checks takes longer time than other cases because they need to apply Android native APIs for the purpose. 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.
4.5 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 before, 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 (Chapter 2) and evaluation results (§ 4.4), 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 [68]. Table 4.8 summarizes the design guidelines.

4.5.1 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.

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
Table 4.8. Observations of applying NChecker in a large scale and the derived guidelines for designing mobile network libraries.

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

### 4.5.2 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.

Chapter 4, in part, is a reprint of the material as it appears in Proceedings of the Eleventh European Conference on Computer Systems. Jin, Xinxin; Huang, Peng; Xu,
Tianyin; Zhou, Yuanyuan 2016. The dissertation author was the primary investigator and author of this material.
Chapter 5

ANEL: Robust Mobile Network Programming Using a Declarative Language

5.1 Motivation

After understanding and detecting network programming defects, the next fundamental research question we ask is: How to enforce the practice of robust network programming? Since NChecker can detect the problematic API usages and NPD patterns, a straightforward idea is: we can automatically patch the buggy code at the bytecode level once it is detected. However, after examining the app code we found that many applications’ semantics cannot be obtained from the binary code. For example, NChecker will raise an alert if a network request initiated from an Activity has no failure notification. But in reality, the developer may prefetch data from server in Activities too and this prefetching is totally transparent to user, even if it fails. Therefore, fixing NPDs cannot simply apply auto patch, but requires some knowledge of application semantics from developers.

From software engineering’s point of view, in addition to the lack of domain knowledge of non-expert developers, another important fact to consider in network programming is the app development cycle. Adding fault-tolerant features up front would
slow down the release cycle, unwisely delaying critical user feedback during the early stages of development. That is why developers tend to delay writing fault-tolerant code during software development [116, 79, 83, 107], so they can focus on developing an app’s core functionality and its acceptability to users.

In considering a programming solution that enables non-expert developers to add network fault-tolerance to their mobile apps after the fact, we observe the need for two critical properties:

- The solution should be declarative, enabling the incremental augmentation of existing code rather than rewriting or refactoring that code.

- The solution should be framed in familiar terminology and concepts, rather than in the alien language and concepts of fault-tolerant network implementation.

We will present ANEL, a declarative language and middleware for Android that meets these requirements. Figure 5.1 provides a system overview: Programmers define high-level application requirements through Java annotations, which are translated by the ANEL compiler into fault-tolerant API calls supported by the ANEL library and runtime. In addition to meeting the two requirements above, using annotations clearly delineates core functionality from the fault-tolerant aspects, even to the extent that the fault-tolerant features can be turned off (i.e., not compiled in), reverting to the original functionality. Key to making this solution tractable is prior work that identified and classified the most common network programming defects (NPDs). Our declarative language is generalized from these NPDs, thus handling the vast majority of issues that would be encountered in the typical mobile app.

We use a code snippet from Twitter Lite client [52] shown in Figure 5.2 to illustrate the power of ANEL annotation language. The original code (the unbolded code in Figure 5.2(a)) registers \texttt{tweet()} with a button click listener and sends out the tweet
Figure 5.1. ANEL overview

once user clicks the button. Here \texttt{tweet()} uses the popular Android Async HTTP library to construct and send out the HTTP post request. This original code will encounter no problems if the network is reliable. However, in the mobile context, where the network can fail intermittently, NPDs would manifest. For instance, the code initiates a network connection whether the network is available or not. Although it has little impact on a desktop computer, it can increase battery drain on a mobile device. Also, the original code does not notify the user if \texttt{tweet()} fails (so the user does not know that her tweet will not be delivered).

The code shown in Figure 5.2(a) in bold is the hand-written fault-tolerant code addressing the above issues. It first checks if the network is connected and then sends out the request, if so. Otherwise it displays a network error message to the user. Although the added code is not particularly long, it requires more domain knowledge to write, because the developers first need to keep in mind that mobile network failures are common, and they need to know proper fault-tolerant network programming practices (e.g., check connectivity, display a user error message). They also must know how to use multiple low-level Android networking classes (i.e., \texttt{ConnectivityManager}, \texttt{NetInfo}) to check availability.

Figure 5.2(b) shows the Twitter Lite example using an ANEL annotation to achieve the same fault-tolerance as in Figure 5.2(a). The developer only needs to add a \texttt{UserReq} annotation, which indicates it is a network request initiated by the user (not a background service). The ANEL compiler is responsible for reasoning about the an-
notation and mapping it to the appropriate fault-tolerant Java code: UserReq means this request is time-sensitive and user interactive, so the network implementation should automatically retry if the request fails and show an error message to the user when the network is unavailable. Note that ANEL annotations are not simply relieving the developer from writing lots of code; rather, it relieves her from reasoning about fault tolerance and its low-level implementation.

Overall, ANEL makes the following contributions:

- A declarative annotation language for specifying network request semantics and enhancing network programming reliability, without reference to fault-tolerant concepts or their implementation.
- A library designed for the unreliable mobile network and compatible with existing network libraries.
- A complier translating the annotations into ANEL library fault-tolerant API calls.
- An evaluation of how ANEL can improve the robustness of real-world networked apps without need for referencing complex fault-tolerant concepts or their implementation.

### 5.2 ANEL Declarative Language

Our declarative language is motivated by the following analogy with racing cars versus automatic cars. Racing cars allow racers to control the speed more flexibly by providing many knobs like a stick shift and clutch. However, novice drivers still prefer automatic cars because controlling those knobs are beyond their expertise, and they don’t need to drive at peak speed. They have no idea about how to operate the complex stick shift and clutch, but they can easily learn that “pushing the pedal down” means
// Click button event handler
btn.setOnClickListener(new View.OnClickListener(){
    void onClick() {
        // Check connectivity before network request
        ConnectivityManager connMgr = 
            getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo netInfo = 
            connMgr.getActiveNetworkInfo();
        // Show error message if no connection
        if (netInfo == null || netInfo.isConnected()) {
            textView.setText("No network connection");
            return;
        }
        tweet();
    }
});

// Functional code to post a tweet
void tweet() {
    AsyncHttpClient client=new AsyncHttpClient();
    RequestParams params = new RequestParams();
    params.put("status", tweet);
    client.post(apiUrl, params,
        new AsyncHttpResponseHandler());
}

(a). The code snippet of Twitter client with hand-written fault-tolerant code (as shown in the bold lines). The code checks network connectivity and notifies user if no network.

btn.setOnClickListener(new View.OnClickListener(){
    void onClick() {
        tweet();
    }
});

void tweet() {
    @Anel_property(\"UserReq\")
    AsyncHttpClient client=new AsyncHttpClient();
    RequestParams params = new RequestParams();
    params.put("status", tweet);
    client.post(apiUrl, params,
        new AsyncHttpResponseHandler());
}

(b). The code snippet of Twitter client with ANEL annotation. The bold line is the declarative annotation achieving the same effect with the bolded code in (a).

Figure 5.2. Add fault-tolerant features to Twitter Lite using hand-written native APIs vs. using ANEL annotations.
speedup and “pressing on the brake” means slowing down. Automatic cars allow drivers to drive safely without worrying about how the mechanics work.

The ANEL declarative language describes high-level application properties or requirements instead of imperative implementations on low-level fault-tolerant APIs. After identifying the specifications, developers can easily add these specifications to existing code via annotations. The implementation strategies of the specifications are hidden from the developers. The declarative language enables robust network programming without forcing developers to reason about the fault-tolerant mechanisms.

An ANEL specification consists of three parts, each discussed in the next three subsections:

- A declaration of one or more properties

- An optional modifier that specifies how active properties should integrate with current specialized networking behavior.

- An implicit scope over which the modified property holds

### 5.2.1 Property Specifications

Declarative specifications indicate high-level application characteristics and demands by telling ANEL what to achieve instead of how to implement it. Properties are specified with the @Anel_property annotation. Its parameter is a list of one or more specifications. A specification is a statement of a boolean property, written as a quoted literal (i.e., "UserReq"), or possibly its negation by prepending it with an exclamation point (i.e., "!UserReq"). Consistent with Java annotation syntax, the list is comma-separated and offset by curly braces (e.g., @Anel_property({"UserReq", "RepeatIsIdentical"})). These indirectly determine the fault-tolerant behaviors of the network requests within the declaration’s scope.
Currently we identify five specifications, derived from the common mistakes summarized in Chapter 2.

**UserReq** indicates a user-initiated request. User requests are usually time sensitive and need to be interactive even if the request fails, so ANEL will retry the failed request automatically on transient failure and show error message if the request fails permanently. **!UserReq** indicates that this request is not initiated by a user, but by a background service. In this case, ANEL disables automatic retry to save energy and disables error notification because users are unaware of the network requests.

**RepeatIsIdentical** specifies an idempotent HTTP POST request, which means repeating the same POST request will not change server state. By default ANEL does not retry a POST request to avoid unwanted side effects due to a (possibly repeated POST. But sometimes POST requests can be idempotent: for example, no matter how many times a user signs in to her Facebook account, the server matches only one authentication record to this user. In this situation, ANEL can safely retry the POST request on transient failures.

**RealtimeData** indicates that the network connection transfers real-time data, e.g., displaying real-time tweets. The ANEL runtime monitors the network status, and if a network switch is detected, ANEL immediately reconnects to minimize the latency caused by the network failure.

**SucceedEventually** means the request needs to succeed eventually, even though it might experience multiple transient failures. For example, if the app tries to download a large file, but fails after several retries due to transient failure, ANEL will save the failed request to a queue, and next time when the connection is restored, ANEL will automatically retry the queued requests. SucceedEventually cannot be used together with UserReq because the latter is supposed to be user-interactive.
5.2.2 Modifier Clauses

If the core networking functionality is already somewhat specialized, a developer may provide additional modifier clauses to specify how the functionality implied by a property specification should be integrated. ANEL handles this by allowing suppression of behaviors in either ANEL or in the developer’s application code. Using these features may require more sophistication than the typical mobile app developer possesses. However, these features are generally not required unless the developer has already applied some fault-tolerant features in her code. Thus, these features generally only need to be applied by developers with greater sophistication.

@Anel_suppressAnel: ANEL allows a developer to suppress some aspects of ANEL’s fault-tolerant primitives in case they intentionally do not want them. Consider a situation in which the developer’s code fires a prefetching request silently, which should not show an error message if it fails. The developer can annotate the prefetch with @Anel_property("UserReq"). together with @Anel_suppressAnel("errorMsg"). The two annotations disable the error message but keep the other properties of UserReq such as checking connectivity and retry.

Suppression is also useful if the compiler detects conflicts between the specifications defined in @Anel_property and the original code logic (discussed in §5.4.3). For example, if the code is annotated with UserReq, which indicates retrying on failure, but for some reason the code sets setMaxRetriesAndTimeout(0, 0) for the same request, the compiler will report a conflict error. To resolve the conflict, the developer can use @Anel_suppressAnel("retry") to suppress the retry element of UserReq.

Based on the fault-tolerance techniques implied by ANEL property specifications, the suppressed elements could be one of the following items:
- **timeout** for timeout settings
- **retry** for retry settings
- **errorMsg** for showing network error messages
- **checkConn** for checking the network connectivity
- **checkResp** for validating the network response

@Anel\_suppressMine: In the opposite case in which the developer wishes to suppress features in the original code so that the specified ANEL property can be applied without conflicts, the @Anel\_suppressMine modifier is used. It accepts the same set of fault-tolerant primitives as @Anel\_suppressAnel.

### 5.2.3 Annotation Scopes

A developer requires individual control over each network object in her app. This requires being able to annotate an individual *object*, as seen in Figure 5.2(b). A unique challenge is that network calls in utility code can be called in many situations, for example some being user requests and some not. This could not be supported by placing a single property annotation in the utility code. To handle this situation, we allow annotating a method *call*, which means that the declared property is inherited by all the code executed by that call. This is in essence dynamic scoping [43]. Finally, some syntactic sugar is beneficial in lessening the number of annotations required. The essential features of ANEL annotation scopes are shown in Figure 5.3.

**Object scoping:** This is the finest scope for controlling each network request’s behavior. When an annotation is applied to a network connection object constructor, the annotation applies only to that object. In Figure 5.3(a), the annotation’s specification only influences operations on object *c*, like the **post**() request.
Method-call scoping: Figure 5.3(b) shows a simplified code snippet of a common coding pattern. Here `post()` is a shared utility called at two different call sites: one within `onClick()`, the callback for when a user clicks a button to upload the file; the other within `autoSend()`, called by a background service, which periodically uploads the file. Obviously, the two call sites require opposing ANEL specifications: the former is `UserReq` while the latter is `!UserReq`. If we apply an object-scoped annotation to object `c`, the annotation cannot realize the distinct requirements of the two execution flows. This code pattern is widely adopted because the network routine can be reused by many other components. To deal with this case, ANEL introduces method-call scoping of annotations. If an annotation is applied to a method call, it applies to all network objects that are constructed in the `reach` of this method call. For
each such constructed network object, the annotation applies to the object just as in object scoping. In Figure 5.3(b), the annotation for onClick influences the code path

\[\text{onClick} \rightarrow \text{upload()} \rightarrow c = \text{new} \ldots\]

The annotation for autoSend() influences the path

\[\text{autoSend} \rightarrow \text{upload()} \rightarrow c = \text{new} \ldots\]

**Method- and class-declaration scoping:** Sometimes the same annotation should be applicable across an entire method or even class, not just a call within a class. To avoid clutter and repetitive annotations, ANEL supports annotation of a method or class declaration, which acts as syntactic sugar for annotating every network object constructor, as well as every method call, in the method or class. Figure 5.3(c) shows an example of the effects of a class annotation.

The class and method call scopes provide coarser granularity of control and thus can help in reducing annotation effort if network requests within the same scope own the same ANEL specifications. However, one method or class may reach multiple network requests, some of which share the same specifications while some others do not. To cope with how annotations with overlapping scopes interact with each other, as well as allowing for a fine-grained annotation to override or extend a coarser one in exceptional situations, ANEL introduces two other scope control rules: scope overriding and scope chaining.

**Scope overriding:** ANEL annotations follow similar scope-overriding rule as programming languages: the local specification overrides the global specification if they conflict. In Figure 5.4(b), the annotation spec RepeatIsIdentical of getHomeTimeline conflicts with the spec !RepeatIsIdentical of the outer class, so the method’s annotation overrides the class’s.

**Scope chaining:** If there an outer annotation is not overridden, an inner annotation inherits an outer annotation’s specifications. For example, in Figure 5.4(a), the
@Anel_property({"UserReq"})
class TwitterHelper {
    @Anel_property({"RepeatIsIdentical"})
    void getHomeTimeline(){}
}

class TwitterHelper {
    @Anel_property({"UserReq","RepeatIsIdentical"})
    void getHomeTimeline(){}
}

(a) Scope chaining

@Anel_property({"UserReq","!RepeatIsIdentical"})
class TwitterHelper {
    @Anel_property({"RepeatIsIdentical"})
    void getHomeTimeline(){}
}

class TwitterHelper {
    @Anel_property({"UserReq","RepeatIsIdentical"})
    void getHomeTimeline(){}
}

(b) Scope overriding

---

**Figure 5.4.** Scope chaining and overriding

The annotation of method `getHomeTimeline()` inherits the outer class’s specification `UserReq`, so the effect is the same as applying `UserReq, RepeatIsIdentical` to `getHomeTimeline()`.

### 5.2.4 Discussion and Limitations

The ANEL annotations address the common NPDs described in Chapter 2. For advanced developers, the supported annotations may not be sufficient to cover their scenarios, prioritizing a particular network type (e.g., WiFi or LTE) for sending a request. As discussed previously, ANEL’s primary target audience is novice developers who typically pay little attention to network reliability and have little background in implementing fault tolerance. With ANEL, those developers can easily avoid pitfalls in mobile network programming.

Some mature applications may want to only send requests under specific con-
ditions in order to save energy or mobile data, e.g., only use non-metered network, or only when the phone is plugged. Previous Android annotation framework APE [96], Tempus [97], Procastinator [102], and google’s API JobScheduler [40] fit such demands well.

For big applications, writing many annotations in separate files can cause maintenance problems. One option would be to apply the idea of Aspect-oriented programming (AOP) [87, 69, 64], using pointcuts and advice to define what properties apply at specific program points. ANEL uses annotations because the semantics is more straightforward and easier for novice developers to learn. Also, for the vast majority of applications we examined, the amount of required annotation is small (c.f. §5.5.2).

5.3 ANEL Library

Given an application annotated with ANEL property specifications, the compiler interprets the annotations and translates them into fault-tolerant code that implements the specifications. A straw-man approach to generating the fault-tolerant code is to directly utilize the native Android APIs. For example, as shown in Figure 5.2(a), to check connectivity, the compiler needs to insert the checking code (in intermediate representation form) into the original code. This approach would unnecessarily complicate the compiler design. To simplify the translation, we introduce a new abstraction layer for the compiler to use. The new layer sits between the Android framework and the application, including an ANEL library and runtime.

The ANEL library is designed for avoiding the NPDs described in Chapter 2. For example, it can set proper timeout times and retry counts for network requests and validate the returned response value; it also can automatically reestablish the connection or recover a failure on demand. The runtime includes a network status monitor that registers a BroadcastReceiver with the Android system and receives a CONNEC-
public class AnelAsyncHttpClient extends AsyncHttpClient {
// Anel is the underlying HTTP engine
protected AnelClient client = new AnelClient();

// Translate annotation specs
public AnelAsyncHttpClient(String[] specs,
String[] suppressAnel, string[] suppressMine) {
    if (specs.contains("SucceedEventually")) {
        client.setAutoResumeFailure(true);
        ...
    }
}

// Override Async HTTP library’s post()
@override
public void post(String url,
ResponseHandlerInterface handler) {

    // Execute post() of Anel library and then
    // translate the response back to the format of
    // Async HTTP library
    client.post(url, new AnelResponseHandler() {
        @Override
        public void onSuccess(AnelResponse response) {
            handler.onCompleted(respreponse.getBody());
        }
        @Override
        public void onError(Error e) {
            handler.onFailure(e.getError());
        }
    });
}
}

Figure 5.5. Anel’s implementation of the Async HTTP class

TIVITY_ACTION broadcast when the network status changes. After being notified, the
monitor calls the ConnectivityManager system service to determine whether the net-
work is online or offline, and then the library executes the fault-tolerant code based on
the network status.

There are two sub-layers within this abstraction layer. On the bottom sits a single
AnelClient instance that implements ANEL’s fault-tolerant features in a set of meth-
ods. Each property specification can be mapped to one or more fault-tolerant method
calls. For example, the specification SucceedEventually maps to the following
method call:

AnelClient.autoResumeFailure(True)
Figure 5.6. App runtime before enhancement vs. after enhancement.

When the runtime monitor detects that the network transitions from offline to online, the library will automatically resume the failed request.

Although this sub-layer provides all the necessary functionality, it is not syntactically compatible to the original code that it needs to replace. Thus the library provides a sub-layer on top, a set of classes that wrap (i.e., adapt) AnelClient in order to provide compatible subclasses (subtypes) of popular networking client classes. The only syntactic difference is the class name and a constructor that takes property specifications. Thus, the ANEL subclasses can directly replace references to these classes with no modifications to surrounding code. An advantage of this approach is that if such an object is stored, passed around the application, etc., it carries its behavior with it, behaving appropriately with no additional help from the compiler or runtime. Currently we implement ANEL subclasses for two network libraries: OkHttp and Android AsyncHttp, but this subclass pattern can be generalized to other libraries.

This transparent replacement strategy also works for the suppression of ANEL or original fault-tolerant features (@Anel::suppressAnel and @Anel::suppressMine). These two specifications are passed to the class constructor, just like the basic property specification. The passed suppress-ANEL properties alter the object’s initialization so that the named features are not turned on. The suppress-Mine properties serve to disable the associated public methods of the object with a simple conditional check, so that the calls present in the code serve as no-ops.
```java
void tweet() {
    AnelAsyncHttpClient client = new AnelAsyncHttpClient()
        .setUserReq();
    RequestParams params = new RequestParams();
    params.put("status", tweet);
    client.post(apiUrl, params,
        new AsyncHttpResponseHandler());
}
```

**Figure 5.7.** The enhanced code of the code input shown in Figure 5.2(a). The bolded line is generated by the compiler and runtime.

Figure 5.5 illustrates a code snippet of the ANEL implementation for the Android Async Http library. `AnelAsyncHttpClient` subclasses the `AsyncHttpClient` class of the Android Async Http library and overrides its methods to enable fault-tolerant behaviors. An `AnelAsyncHttpClient` instance initializes a `client` instance of `AnelClient` type, which powers the HTTP connection. The class constructor takes a list of specification strings as an optional parameter, which enables corresponding fault-tolerant method calls. In this example, the constructor parses `SucceedEventually` and sets `autoResumeFailure(True)` for `client` accordingly. To equip the network request `post()` with the fault-tolerant features of `AnelClient`, `AnelAsyncHttpClient` overrides it this way: it sends out the POST request using `AnelClient` (so it implements the `SucceedEventually` spec), but converts the response to the original response format of `AsyncHttpClient` (i.e., the callback method `handler` that is the parameter of `post()` in the code). The reimplemented `post()` is more robust, yet it guarantees that it never changes the original code’s functional behaviors because it returns the same output as the superclass’s method. Figure 5.6 compares runtime before translation versus after translation.

To add fault-tolerant behaviors to an annotated `AsyncHttpClient` constructor call, the compiler only needs to replace that call with a call to the `AnelAsyncHttpClient` constructor carrying the annotation’s property strings and suppressed prim-
itives. All the other AsyncHttpClient constructor calls in the original code are unchanged, unless similarly annotated. Thus, ANEL’s subclasses make the translation straightforward. For the annotated input code in Figure 5.2, the runtime executed code is shown in Figure 5.7.

5.3.1 Limitation and Discussion

To implement the subclass of the original libraries’ classes, the assumption is that the original class can be extended. This assumption does not work for final class. In this circumstance, the compiler cannot simply replace the constructor statement to achieve fault-tolerant purpose, but also need to insert or replace existing fault-tolerant APIs with ANEL library APIs.

5.4 ANEL Compiler

5.4.1 Overview

The ANEL compiler takes the annotated app as input, translating each annotation into corresponding ANEL library calls and generating an enhanced executable Android app. The design of the ANEL compiler had to address three challenges: (1) How to determine the annotations associated with a network request in all types of scopes. (2) How to guarantee the generated code does not conflict with any fault-tolerant code already present in the original app. (3) How to insert fault-tolerant code into the original app code.

Handling these issues requires significant compiler support. We use the Soot analysis and transformation framework [113]. We extended the FlowDroid Android app static analysis tool [62], itself based on Soot, to create the full app call graph, which is used for the ANEL compiler’s translation and analysis. The compiler initially translates the annotated Android code to Soot Jimple (a Java intermediate representation) and in
Figure 5.8. Annotated code as compiler input and enhanced code as compiler output. The output is written in Java for convenience. The real generated code is Jimple.
the end generates new .class files that can be assembled into an APK. The compiler’s current implementation can analyze and generate an APK for Android apps using the existing Android Async Http library and OkHttp library.

The following subsections describe the three components of the ANEL compiler that handle the above issues: the annotation parser, conflict resolver, and code generator. We will use the code in Figure 5.8 as a running example. This piece of code has a send utility that uses the Android Async Http library to send a network request. This method is called by two callers foo() and bar(). The calls to foo() and bar() are annotated with UserReq and !UserReq respectively.

5.4.2 Annotation Parser

The parser first identifies all the annotations, and the context of annotated objects, method-calls, methods, and classes, performing syntax checking to make sure all the specification strings are valid, and the properties named within one @Anel property do not conflict with each other. For example, SucceedEventually cannot be declared with UserReq, and !UserReq conflicts with UserReq. In our example, after the initial checking the parser obtains annotation sites and their specifications as follows: (foo : UserReq), (bar : !UserReq).

Next, for each annotation site, the compiler identifies all the network object constructors reachable by that annotation. If it is an object-scoped annotation, the compiler does static taint analysis [72, 76] starting from the annotated HTTP client object, and propagating forward until reaching the Http client object. For the other annotations, including method-call annotations, method- and class-declaration annotations, because the scope includes all the reachable statements starting from the annotation site, simply parsing a single method or object is insufficient. So ANEL performs an interprocedural reachability analysis to explore all the reachable paths from the annotation site to HTTP
client object.

Finally, the parser assigns annotation specifications to each target API. According to the method-call scope control, if method call \( A \) annotated with \( \{ p \} \) forward reaches method call \( B \), the specification of \( B \) is \( \{ p \} \). Additionally, the parser needs to consider scope chaining and overriding across methods: If a method call to a \( A \) annotated with \( \{ p, q \} \) reaches a call to method \( B \) annotated with \( \{ !p, r \} \), the specification of \( B \) is \( \{ !p, q, r \} \). The parser follows every method call along each reachable path to the target method, and applies the above rules to determine each method call’s annotation specifications. The search stops at the point of reaching a HTTP client, at which point the compilers knows the set of annotation specifications belonging to this client, as well as the call path associated with it.

At the end of the reachability analysis, the compiler knows the possible paths from the annotated method to all network object constructors that are candidates for replacement with ANEL fault-tolerant objects. In our example code, there are two paths to the constructor for client: foo → send → new AsyncHttpClient() and bar → send → new AsyncHttpClient().

### 5.4.3 Conflict Resolver

The compiler needs to guarantee that the generated code does not break the original code semantics. Because ANEL only changes fault-tolerant-related code behavior, it naturally ensures that the original functional code will not be impacted. However, if the annotated app already has some fault-tolerant logic, how can we make sure ANEL generated fault-tolerant code does not conflict with it?

Conflicts can arise when the original code contains fault-tolerant code, and an annotation refers to the same fault-tolerant primitive. Here primitive means some fault-tolerant mechanism, such as a retry or timeout. For example, both UserReq and
setTimeout() refer to the timeout primitive; both UserReq and setTimeout() refer to the retry primitive. If such conflicts exist, the compiler needs to figure out which fault-tolerant policy to follow: the declarative specification or the original API calls. By default, the original code takes precedent, as this guarantees preserving the developer’s explicit intent.

The compiler resolves the conflict in one of two ways, depending on the type of the conflict:

First, if the original fault-tolerant implementation potentially violates ANEL’s fault-tolerant semantics, the compiler will raise a compilation error. For example, in Figure 5.8, in the path bar → send → get, the network request is annotated as !UserReq, which indicates not to retry. But the method setMaxRetriesAndTimeout sets the number retry attempts to 3, which conflicts with the specification semantics. To suppress the conflict error, the developer has the choice of using either @Anel_suppressAnel or @Anel_suppressMine. The former means to ignore the fault-tolerant primitive defined in ANEL while the the latter means to ignore the primitives defined by the original code. The example code uses @Anel_suppressMine{"retry"} to suppress the retry settings in the original code. The compiler explicitly asks the developer to resolve the conflict in order to guarantee the compiler output will never change the original code semantics without the developer being aware of it. Table 5.1 defines all the conflict patterns of a given fault-tolerant primitive in which the specifications conflict with the fault-tolerant API settings.

Second, if the original fault-tolerant API just refers to the same primitive as the declarative specification, but does not violate the semantics, ANEL will not report an error, but rather keep the original setting, taking precedent over ANEL’s. In the example code, on the code path foo → send → get, there is an explicit call setting the number of retry attempts to three, and the UserReq specification also stipulates retries. Although
Table 5.1. Conflicted pairs of annotation specifications and fault-tolerant API settings for a given fault-tolerant (FT) primitive

<table>
<thead>
<tr>
<th>FT primitive</th>
<th>Specification</th>
<th>Conflicted API settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>retry</td>
<td>UseReq or RepeatIsIdentical</td>
<td>Set retry times 0</td>
</tr>
<tr>
<td>retry</td>
<td>!UseReq or !RepeatIsIdentical</td>
<td>Set retry times larger than 0</td>
</tr>
<tr>
<td>errorMsg</td>
<td>!UseReq</td>
<td>Call error message API</td>
</tr>
<tr>
<td>errorMsg</td>
<td>SucceedEventually</td>
<td>Call error message API</td>
</tr>
</tbody>
</table>

the ANEL library’s chosen number of retry attempts is different, the original code’s retry setting will not cause an NPD. Thus the call is allowed to remain, and it sets the number of retries in the ANEL library to three in the ANEL library.

To detect the above situations requires finding the fault-tolerant method calls associated with a network request. ANEL uses the data flow algorithm adapted from the NChecker.

The existing network library may have some default fault-tolerant settings (not exposed to developers), but our prior study shows that novice developers mostly ignore fault-tolerant features in networking APIs. As such, the compiler detects conflicts only when the developer explicitly calls and sets a fault-tolerant API, but automatically follow the semantics of the original APIs if developers do not call them in the code.

5.4.4 Code Generator

The code generator inserts the new fault-tolerant code into the original code and generates a new executable Android app. The essence is to replace the original network library objects with ANEL library objects, by replacing their respective constructors. This relies heavily on the ANEL-implemented subclasses of existing network libraries as introduced in §5.3, which makes the replacement almost transparent. As a reminder, the ANEL subclasses type compatible, allowing them to transparently stand in for the original classes. This allows them to be passed anywhere in the application. The primary
complexity that must be addressed is that the annotations governing the initialization of a backwards-compatible ANEL network object is determined by code execution flows, due to method-call scoping. The input in Figure 5.8 exhibits the challenge cited above, the scenario where the networking utility (send) is called on two different code paths, each with its own specification. At runtime, ANEL needs to recognize which path led to the invocation of send in order to pass the right specification to the constructor.

We now provide two alternatives of implementing the code generator. The first is a static approach that generate the final fault-tolerant code at bytecode level; the second is a dynamic approach that emit fault-tolerant code at runtime.

**Static Approach:** The middle part of Figure 5.8 shows the generated code by the static approach. To pass the annotation specifications to the ANEL library, the code generator first initializes an ANEL library HTTP client carrying the declarative specifications. In the example, it utilizes ANEL implemented interfaces of Android Async Http library to create a new AnelAsyncHttpClient instance, which carries the specification strings as the parameter.

In order to handle the two code paths that have different annotation specifications, the compiler first creates two AnelAsyncHttpClient instances, which takes UserReq and !UserReq separately. To map each specification to the correct code path, the code generator assigns an identifier for each AnelAsyncHttpClient instance, as well as the code path where this instance should be called. Then for each code path reaching the target API, the compiler adds this identifier as an extra parameter to every method along the path. When the call flow reaches the target API, the identifier will control which instance to use.

As we will show in the evaluation, the static approach introduces unmeasurable runtime overhead. However, due to the imprecision of static analysis, it may not obtain accurate annotation properties on each path.
**Dynamic Approach:** The bottom of Figure 5.8 shows the code generated by the dynamic approach.

As mentioned in §5.2.3, the semantics of ANEL’s method-call scoping is fundamentally dynamic scoping [43], in which (intuitively speaking) a routine searches through the call stack to attempt to resolve a variable reference that is not defined in the local scope. Although not directly implementable in Java, the call-stack implementation suggests maintaining a separate stack of active ANEL specifications, actually three stacks, one for @Anel

property specifications, one for @Anel

suppressAnel, and one for @Anel

suppressMine. At each location that the compiler encounters an ANEL annotation on a method call (i.e., a method-call annotation, not an object annotation), it generates code to push the specification on its corresponding stack. If the control-flow analysis indicates that a method-call scope is not the top-level scope, then code must be generated to implement the scope chaining and scope overriding described in §5.2.3. The code generator emits code to call an ANEL runtime method that examines the new property specification and the one currently on the top of its stack to generate a modified specification that represents the scope resolution (not shown in Figure 5.8, as these are top-level method-call scopes). After the annotated method call, the code generator emits code to pop the specification off the stack to terminate or “close” the scope. In the example, at bar’s call to send, the list \{"!UserReq\}" is pushed on the property stack, and \{"retry\}" is pushed on the suppress-mine stack. In the actual generated Jimple, a try-catch block is also set up around the push/pop and the method call, in case an exception is thrown, so that the ANEL stacks are properly popped as the stack unwinds.

At the site of a networking object constructor that is reachable from a method-call annotation, code is generated to implement the fault-tolerant semantics as indicated by the elements on the top of the ANEL stacks. An ANEL subclass object constructor
call replaces the original one, and the specifications on the top of the property stack are passed to the constructor, as shown in the bottom half of Figure 5.8. Because the ANEL subclasses are interface-compatible with the originals, none of the existing calls need to be changed, and the ANEL object carries all its fault-tolerant enhancements, so it works anywhere that it’s referenced in the application.

Because it’s possible that there is no property in force at a given method-call-scoped network call, we did two things to keep the object code simple. One, each of the three stacks are initialized with the empty list as their bottom element. Two, all of the ANEL subclasses behave identically to their original counterparts if an empty ANEL property list is passed to their constructors. Together these provide the appropriate “no-op” behavior without any special-case code.

The specification stacks are maintained at runtime, so this approach overcomes the inaccuracy problem of the static approach. The tradeoff is the runtime performance overhead.

5.5 Evaluation

5.5.1 Overview

We have applied ANEL to 6 open source Android applications using Async Http library and OkHttp library. Table 5.2 lists the annotation efforts for each. We will describe the annotating process for GpsLogger and Twitter Lite in detail in § 5.5.2.

5.5.2 Case Studies

As the importance of eliminating NPDs in mobile apps has been well studied, we focus on the expressiveness and practicality of using ANEL for eliminating NPDs. We present two case studies that show how ANEL can ease developers’ work in real-world programs by (1) enabling easy-to-understand specifications, and (2) handling ex-
### Table 5.2. Evaluated Android apps and applied ANEL annotations.¹

<table>
<thead>
<tr>
<th>App</th>
<th>Lib used</th>
<th>#Net. req</th>
<th>#Anno. of lines</th>
<th>Scope</th>
<th>Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>GpsLogger [33]</td>
<td>okhttp</td>
<td>2</td>
<td>2</td>
<td>Method</td>
<td><code>anel.property(&quot;UserReq&quot;)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>anel.property(&quot;!UserReq&quot;, &quot;MustSucceed&quot;)</code></td>
</tr>
<tr>
<td>Twittler Lite [52]</td>
<td>async http</td>
<td>7</td>
<td>1</td>
<td>Class</td>
<td><code>anel.property(&quot;UserReq&quot;)</code></td>
</tr>
<tr>
<td>Kangaroo [41]</td>
<td>async http</td>
<td>4</td>
<td>2</td>
<td>Object</td>
<td><code>anel.property(&quot;UserReq&quot;, &quot;Idempotent&quot;)</code></td>
</tr>
<tr>
<td>Hacker news [37]</td>
<td>okhttp</td>
<td>4</td>
<td>1</td>
<td>Object</td>
<td><code>anel.property(&quot;UserReq&quot;)</code></td>
</tr>
<tr>
<td>Instagram photo viewer [39]</td>
<td>async http</td>
<td>2</td>
<td>1</td>
<td>Object</td>
<td><code>anel.property(&quot;UserReq&quot;)</code></td>
</tr>
<tr>
<td>New York Times search [46]</td>
<td>async http</td>
<td>2</td>
<td>2</td>
<td>Object</td>
<td><code>anel.suppressAnel(&quot;errorMsg&quot;)</code></td>
</tr>
</tbody>
</table>

¹ Clarifications about some annotated apps: (1) Kangaroo: The code uses HTTP POST for all downloading tasks, which is idempotent. (2) Hacker News: The code did connectivity check already so ANEL silently applies `anel.suppressAnel("checkConn")`. (3) New York Times Search: The code shows error message when network request fails so we suppress ANEL’s.
ceptional situations, and (3) concisely specifying fault-tolerant features over varying code patterns.

5.5.2.1 GpsLogger

GpsLogger is a mobile app that logs the GPS coordinates for the user’s travels \[?\]. It can send the logged route to various servers such as OpenGTS and OpenStreetMap. GpsLogger provides two ways to sync a log file to a server: one, the user can press the upload button on the app’s home page and select which server she wants to sync up; two, the user can change the app’s settings to periodically sync with a chosen server in the background.

The unmodified app uses the OkHttp library for network operations. It does not set a timeout; it does not check the network connectivity; it does not show any error messages when the network is not available or the upload fails. OkHttp does not provide an API to control a retry policy, but it keeps retrying on failure, so it can kill the battery.

To apply ANEL in fixing these problems, we first consider the application semantics for the two scenarios: if the user manually uploads the file, it is a user request (UserReq); if the service periodically syncs with a server, we can ignore the transient failure because it will sync again after an interval, so it is !UserReq. Furthermore, we can improve the background task's service quality by specifying SucceedEventually, so once the network resumes the app will retry without waiting for another next sync cycle.

Figure 5.9 shows the relevant code snippet from GpsLogger together with the annotations we apply to the code. The network utility UrlJob is called on different two code paths: one starts from GpsMainActivity.onMenuItemClick(), a UI handler callback, the other starts from GpsLoggerService.autoSendLogFile() in a background service. Since we need different annotations for the two call paths, we
Figure 5.9. We improve GpsLogger by applying two method-scoped annotations: “UserReq” is for user-initiated request and “!UserReq, SucceedEventually” is for background request.
cannot just annotate the OkHttpClient object. We apply annotations to the upper-level call methods `sendToOpenGTS` and `autoSendLogFile`.

In performing this case study, we found that although the annotated methods and the ultimate HTTP request are in different files, and there is a deep call sequence separating them, locating the right place to insert the annotations and deciding which specifications to apply were not difficult. We needed to look only at the upper-level method calls. Their semantics – user call versus background call – were clearly communicated by the method names (`onMenuItemClick` versus `autoSendLogFile`) and structural cues in the code (user interface code is declared in an Activity, background tasks are run in a Service).

### 5.5.2.2 Twitter Lite

This unofficial Twitter client fetches different types of tweets such home timeline feeds, my favorite feeds and feeds mentioned me, depending on which button user clicks [52]. In total, there are 7 different types of fetch requests. Each is put into a network helper method, all in one class, `TwitterHelper`, as shown in Figure 5.10. The unmodified app uses Android’s Async Http library to send out network requests. Each network helper method calls `getClient()` to get an `AsyncHttpClient` object. Interestingly, the original code has some fault-tolerant code. Among the seven methods, before calling `getHomeTimeline()` and `getMention()`, the code checks network connectivity and shows an error message to the user if the network request fails. There is no such checking for the other methods. This partial checking is common in other apps, too, perhaps because the process is error-prone, or the developer is responding narrowly to bug reports.

Because all the network requests are triggered by user actions, we can use `User-Req` to express the specification for all of them. If using object-scoped or method-call-
@Anel_property({"UserReq")
public class TwitterHelper {
    public void getHomeTimeline(){
        getClient.get();  //send Http GET
    }
    public void getMention(){
        getClient.get();
    }
    public void getFavorite(){
        getClient.get();
    }
    public void tweet(){
        getClient.post();  //send Http POST
    }
    ... three more network methods ...
}

Figure 5.10. We improve Twitter client by applying a class-scoped annotation.

scoped annotations, the developer would need to annotate seven lines of code for the seven network requests separately. Using class-level annotations, the developer only needs to add one annotation TwitterHelper. During conflict detection, the compiler detects the two methods with the connectivity checks and error messages, so it automatically passes @Anel SuppressAnel({'checkConn", "errorMsg")} on creating ANEL clients.

5.5.2.3 Discussion

GPSLogger and Twitter Lite presented different challenges in adding fault tolerant code after the fact.

GPSLogger presented the challenge that the same network utility code was called from very different contexts, requiring ANEL’s method-call scoping features to achieve the desired unique behavior on each path. The GPSLogger case study shows that, despite these challenges, ANEL annotation specifications are easy to formulate
based on basic cues in the app’s design and implementation. Moreover, it shows that the scoping features of the language makes the annotation process easy for developers, and is able to handle library code.

The Twitter Lite case study presented the challenge that some network calls contained fault-tolerant code, and others not. Here we see how ANEL’s compiler was able to turn off two of its fault-tolerant features where it found those features already present in the original app. Additionally, we saw how class-level annotation helps save annotation effort and provides clearer documentation of intended behavior by avoiding repetitive annotations.

### 5.5.3 System Overhead

To evaluate the runtime overhead associated with ANEL, we examine the additional *latency* and *energy consumption* induced by the ANEL middleware to complete a single network request.

To ensure measurement in a controlled environment, we wrote a small artificial Android app that repeatedly fetches one kilobyte of from a server using two implementations: (1) existing network library with hand-written fault-tolerant code; (2) existing network library with object-scoped annotation; (3) existing network library with method-call scooped annotation. We used Android’s Async Http library and OkHttp library for this evaluation. For the hand-written fault-tolerant code, we added the fault-tolerant code to check network connectivity before sending the request, and set retries for transient failure. In the ANEL versions, we added `@Anel_property({"UserReq"})` and `@Anel_suppressAnel("retry")` annotations at the designated locations for Http client object or method-call. Table 5.3 shows the latency of each approach. We can see that the performance of the ANEL-generated code is quite close to the hand-written code. The method-call annotations introduce 2ms - 6ms of latency, due to the
Table 5.3. Latency to download 1K data for different HTTP clients

<table>
<thead>
<tr>
<th>Http client</th>
<th>Latency (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android Async</td>
<td>0.161</td>
</tr>
<tr>
<td>Android Async + Annotation</td>
<td>0.173</td>
</tr>
<tr>
<td>Android Async + Annotation(method-call)</td>
<td>0.175</td>
</tr>
<tr>
<td>OkHttp</td>
<td>0.164</td>
</tr>
<tr>
<td>OkHttp + Annotation</td>
<td>0.164</td>
</tr>
<tr>
<td>OkHttp + Annotation(method-call)</td>
<td>0.169</td>
</tr>
</tbody>
</table>

overhead of dynamic scoping.

To measure energy overhead, we used the Qualcomm Trepn Profiler [51] to profile the power consumption of continuously sending 10 network requests by the above six approaches. The average power consumption was measured as 130mW/second for all six. So we conclude that ANEL middleware does not introduce measurable energy overhead.

The reason for ANEL’s negligible runtime overhead is that the compiler does much of the work at compile time. In real-world code, the relative costs would be even less than measured, since most apps do not continuously use the network like our test harness.

Chapter 5, in part is currently being prepared for submission for publication of the material. Jin, Xinxin; Griswold, William G; Zhou, Yuanyuan. The dissertation author was the primary investigator and author of this material.
Chapter 6

Related Work

6.1 Mobile App Testing

A number of in-house testing approaches have been proposed for mobile apps.

There are a large number of testing tools available in the market for functionality and UI testing. MonkeyRunner [44] is an API provided by Android SDK that enables developers to write python scripts to control an Android device. Developers can connect to a device, execute commands, start an app, send keystroke, or simulate touch event, and take snapshot of the screen. Robotium [48] is an Android testing framework for both black and white box tests. It can discover the UI elements, fired UI events specified by the programmer, and navigate from one activity to another activity. UIAutomator [54] is a Java library containing APIs to create customized functional UI tests, and an execution engine to run these tests. To use these tools, scripts must be written by skilled developers, and thus are time consuming for inexperienced developers. In addition, manual scripts cannot automatically explore app states and therefore cannot guarantee the coverage. Lastly, none of these automation tools take environment (e.g. network, sensors, SD card, etc.) into consideration.

Fuzz testing [91, 103, 63, 53] generates random inputs to apps and evaluates the app’s robustness. It is mostly used for stress testing, which are not thought about
in normal tests, so it can discover some corner bugs neglected by developers. The Android platform provides an UI automation tool Monkey \[53\] that can generate random UI events to Android apps running on Android device emulator. Dynodroid \[91\] selectively generates UI events and system events. It obtains the UI elements related with a particular page from parsing the app view hierarchy, and generates system events based on the registered system callbacks. VanarSena \[103\] automates UI by instrumenting app binary to explore the execution paths. Azim et al. \[63\] proposed targeted exploration combing static and dynamic analysis to explore app activities in a more comprehensive way.

Symbolic testing \[93\] and its variation concolic testing \[61\] tests an app by symbolically executing it to achieve high testing coverage. Instead of taking real values, symbolic execution takes symbolic values as inputs. An interpreter follows the program flow, arriving at expressions in terms of symbolic values and gathering constrains on those values along the path. Finally a solver can generate program inputs covering all the paths.

Few of these fuzz testing and symbolic testing works look at the problems related to mobile network dynamics, probably because emulating different network environments with high fidelity is prohibitively difficult.

Some works use fault injection to test apps under certain adverse external conditions. AppDoctor \[82\] tests mobile apps under various system and user actions. Its purpose is not to stress test apps by emulating user inputs, but to test how well the apps handle these events under normal execution. Vanarsena \[103\] injects environment related faults at runtime to find bugs. The authors studied and app crashes from Windows App Store and summarized five kinds of errors dominating the top crashes: web errors, unreliable network, sensor errors, invalid text input and impatient user behavior. Based on the study they proposed a fault inducer to inject the above faults. For example, to
inject network errors, the fault inducer instruments the app such that before sending a
network request, the inducer emulate different network conditions such as no network,
long latency, etc. It can also intercept the HTTP call and returns HTTP error codes such
as 404 or 401, triggering WebExceptions. The tool will report errors if the tested app
 crashes under the injected faults.

Vanarsena’s followup work Caiipa [88] uses similar injection approach, but pri-
oritizes the test cases that are more relevant with the targeted app. For instance, a mes-
saging app mostly relies on the internet rather than memory and CPU usage. So if
emulating different network conditions in the test, it will change the messaging apps be-
behavior a lot. Caiipa groups apps of similar resource usage patterns with the messaging
app and prioritizes network testing for this type of apps.

Dynamic fault injection can find out certain network programming defects that
may cause app crashes. However, according to our characteristic study, a large number
of NPDs do not manifest themselves through crashes, and therefore hard to be discov-
ered by fault injection. In addition, some of NPDs are triggered by specific conditions
which can be hardly triggered by the emulator. For example, a bug can only happen
under intermittent network, when some network requests succeed but some fail. This
requires a complex timing model of the fault injector design.

6.2 Mobile App Bug Detection

Static analysis analyzes the computer software without actually executing pro-
grams. By examining the app binaries, previous work have applied static analysis into
uncovering energy bugs, performance bugs and security problems of mobile apps.

Prior work [99] detects a class of energy bug called no-sleep bug, which are
casted by mishandling power management APIs in mobile apps or systems. The reason
for majority no-sleep energy bugs is that a turn-on API call is not matched with a cor-
responding turn-off API call before the end of program execution. Their bug detection tool applies Def-use dataflow analysis to detect such misuse pattern.

PerfChecker [89] identifies and detects performance bugs from Android apps, including GUI lagging, energy leak and memory bloat.

Because smartphones have become a ubiquitous source of users’ confidential data, there have been seen a rich body of works that apply static taint analysis to detect privacy leaks [75, 77, 62]. The basic idea is to check if there exist code path where an app first reads the sensitive data and later transmits this information to the network without user permission. One challenge of analyzing Android app is to create a comprehensive model of Android app lifecycle, including multiple entry points (UI event handlers), asynchronously executed functions, and system callbacks. FlowDroid [62] extracts entry points from Android’s Manifest file, dex files and layout xml files. and generates a dummy main method that connects the possible entry points for each app. Once the dummy main method is constructed, it builds a call graph to model the app’s lifecycle. Our static analysis implementation takes advantages of FlowDroid’s framework in constructing the call graphs.

6.3 Declarative Programming

ANEL is motivated by many previous projects in declarative programming. APE and Tempus used declarative annotations to specify mobile power management policies to postpone the execution of delay-insensitive code segments in order to save energy [96, 97]. Indeed, annotation programming have been used in many systems to facilitate programming: AndroidAnnotation, ButterKnife and Guice use annotation to generate code [16, 23, 36]. Defensive programming uses annotation to build DoS-resistant software so that developers do not need to write defensive code on their own, but just tell the compiler the resource usage information [100]. AAL use annotation to specify
the invariant that is used for program verification [85]. Quinlan et al. use annotations to describe abstraction properties for compiler optimization [101]. Guyer et al. propose an annotation language that enables compiler to optimize the library performance [80]. EnerJ use type qualifiers to declare approximate variables and map them to low-power storage [105]. JFlow adds statically-checked information flow annotations to protect privacy and integrity of sensitive data [95]. Cooprider et al. use safety annotations to enforce memory safety for TinyOS [73]. Zhou et al use type annotations to detect and recovering from type safety violations in software extensions [118]. In complement to these systems, ANEL is the first annotation language for aiding good network programming practices and enhancing mobile networked app reliability.

More broadly, Open Implementation allows a client of a component to determine its implementation strategy by describing performance requirements [90, 86, 92]. Open Implementation exposes some implementation details of a submodule, so that developers can control the implementation decisions when necessary. ANEL is an example of Open Implementation, as its annotations enable customizing fault-tolerance orthogonally from core behaviors, without specialized knowledge of fault tolerance.

### 6.4 System-level Support for Network Operations

The Android framework is being 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. This can enhance the reliability for services hosted in redundant data centers. From Android 3.0, NetworkOnMainThreadException would be thrown when apps attempt networking operation on the UI thread, which forces the developer to implement the related operations in a background thread and therefore prevent UI freezing. Android 5.0 provides the JobScheduler API can schedule a task
to execute when the criteria declared are met [40]. This is useful for non-user-waiting network operations that require a particular connection type. For example, do the periodic synchronization under unmetered network. SyncAdapter syncs data transfers only when network is available, and batches network operations to save energy [24]. DownloadManager monitors network status and restores long-running downloading tasks automatically [25]. However, although these systems’ APIs provide important functionalities, they are still non-trivial to use. The developer needs to learn a lot of system mechanisms in order to use them correctly. Complementary to the above APIs, ANEL is unique in that it offers a language for non-experts to express their application behaviors, and ANEL inserts the necessary fault-tolerant code. We hope our findings of network programming defects will motivate more improvements in the Android framework.

The battery drain problem due to network operations has seen a large body of work in recent times. Some work [70, 66, 74, 71] have characterized the energy consumption of different network conditions. Prior work has investigated strategies [70, 66, 94], such as optimizing background tasks and prefetching, for energy-efficient network tasks. Our study of the network defect impact is consistent with these work, but we tackles the NPDs causing battery drain mostly from an app developer’s perspective, which is the network programming interfaces.
Chapter 7

Conclusion and Future Work

With the rise of “young” apps and young app developers, many mobile apps have not carefully designed and implemented by considering the dynamic mobile environment because of the developers’ inexperience and insufficient resources, and/or the lack of time to evolve into mature apps. Specially, inexperienced developers often make mistakes in handling network dynamics; they unconsciously assume that the mobile network is stable and fast like desktop environment, and therefore fail to harden the code to handle network failures, causing network programming defects (NPDs).

The main contribution of this thesis is to understand NPDs and their root causes, and assist mobile app developers prevent NPDs in developing networked mobile apps.

- We studied the characteristics of NPDs from 90 real-world NPD cases. The study provides a deep understanding of NPD characteristics, including their impact on user experience, root causes, and code patterns. NPDs cause crash/ freeze, dysfunction, unfriendly UI and battery drain issues which negatively impact user experience. The root cause analysis reveals the common mistakes and pitfalls made by app developers, including no connectivity checks, mishandling transient error, mishandling permanent error and mishandling network switch. Despite fault-tolerant features provided by modern network libraries, they cannot help inexpe-
rienced app developers prevent NPDs, because most 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.

- We developed a static analysis tool NChecker to detect NPDs. NChecker effectively identifies the NPD code patterns obtained from the characteristic study. It has 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.

- We further developed ANEL, a novel system for eliminating network programming defects and improving the robustness of networked apps. Its declarative language is generalized from common network programming defects. It allows declaring what app behaviors are expected instead of how to implement it, and therefore can be easily adopted by non-experts who have difficulty in reasoning about the complicated fault-tolerant mechanisms. Together with the annotation language, we designed and implemented a compiler and runtime library. The library encapsulates the boilerplate fault-tolerant implementation, and provides a compatible interface with existing network libraries. The compiler determines the scopes of annotations and translates the annotations into ANEL library APIs. The case studies of real-world Android apps show that the ANEL approach can, at low run-time cost, meet our goals for incrementality and avoiding need for mastery of fault-tolerant networking concepts.

We plan to implement ANEL for more popular network libraries to demonstrate the generality of our approach. Because the code pattern of constructing and sending
out network requests is similar among different network libraries, we foresee that ANEL annotation scopes and the subclass implementation for existing libraries’ compatibility work for other libraries as well.

While we have applied several case studies to evaluate the practicability of ANEL annotations, a user study of inexperienced app developers will be conducted to examine how well developers new to ANEL can adapt to using annotations to express fault-tolerant requirements.

We can make ANEL library more intelligent and adaptive to environment. All existing libraries reply on static values (timeout, retry counts) for fault-tolerant feature configurations, which make dynamics difficult to accommodate. These values should be customized for application scenarios and real-time network latency. For example: the library can profile the network latency of several network request, and estimate the proper timeout; it can also measure the size of the data to be transferred, and make adjustment based on the data size.

This thesis mainly addresses the reliability problems induced by mobile networks. Another critical concern of networked app is the performance under poor network. During the characteristic study, we also found some common issues regarding caching, prefetching, and network-type aware optimization (e.g. only use high-quality encoding under wifi). Help developers achieve high-performance programing during app development would benefits lots of networked apps.

We mainly focus on NPDs in Android apps because of the availability of open-sourced Android apps. We have spotted similar NPD issues in other mobile platforms such as iOS and Windows phone too, and we leave comprehensively studying and addressing NPDs for other platforms as feature work.

Broadly speaking, except for API misuses in network related APIs, we also aim to understanding API misuse patterns of other types of APIs in the wild. A recent An-
droid framework includes over 5,000 new APIs, while the iOS framework covers over 4,000 new APIs. Many of these APIs directly talk with OS resource management features, e.g. the wake lock mechanism, the notification mechanism. These mobile specific schemes and enormous set of interfaces inevitably creates burden for app developers to correctly write applications. This situation is exacerbated by the rise of many young, individual mobile app developers who may not have considerable programming experiences and insufficient development and testing infrastructures and tooling support. We believe that understanding the API usages would provide a chance to improve of the whole mobile framework.
Bibliography


577468670147772802.


[38] Insights into the testing and release processes for chrome for ios. 


JobScheduler.html/.


09wi/general-concepts/scoping.html.


visionmobile.com/product/developer-economics-q3-2014/.


android.com/tools/help/monkey.html.


[65] Aruna Balasubramanian, Ratul Mahajan, and Arun Venkataramani. Augmenting mobile 3g using wifi. In *Proceedings of the 8th International Conference on*


[74] Ning Ding, Daniel Wagner, Xiaomeng Chen, Abhinav Pathak, Y. Charlie Hu, and Andrew Rice. Characterizing and modeling the impact of wireless signal


Conference on Mining Software Repositories, MSR ’16, pages 484–487, New York, NY, USA, 2016. ACM.


