ANEL: Robust Mobile Network Programming Using a Declarative Language

Xinxin Jin
University of California, San Diego
xinxin@cs.ucsd.edu

William G. Griswold
University of California, San Diego
wgg@cs.ucsd.edu

Yuanyuan Zhou
University of California, San Diego
yyzhou@cs.ucsd.edu

ABSTRACT

The dynamics of mobile networks make it difficult for mobile apps to deliver a seamless user experience. In particular, intermittent connections and weak signals pose challenges for app developers. While recent network libraries have simplified network programming, much expert knowledge is still required. However, most mobile app developers are relative novices and tend to assume a reliable network connection, paying little attention to handling network errors in programming until users complain and leave bad reviews.

We argue that the difficulty of avoiding such software defects 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. This paper presents ANEL, a declarative language and middleware for Android that enables non-experts. We demonstrate the expressiveness and practicability of ANEL annotation through case studies and usability studies on real-world networked mobile apps. We also show that the ANEL middleware introduces negligible runtime performance overhead.

1 INTRODUCTION

1.1 Motivation

More and more mobile apps rely on the internet to provide users key functionalities like messaging and online shopping [56]. As such, delivering a seamless internet experience is vital for a networked app to win in the competitive market. However, due to user mobility, apps experience many more dynamic network disruptions than desktop applications, such as network switches and very weak signal. Such disruptions can give rise to a specific class of software defects–network programming defects (NPDs), if app developers do not handle network disruptions properly, resulting in various app glitches, including freezing and crashes. According to a recent study, NPDs exist in over 98% of the examined mobile apps [36]. NPDs can impact user experience negatively and thus apps with NPDs often receive poor reviews [26, 27, 39, 50]. Here 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 [9]*

"Sometimes chat messages take forever to load." – *A review of Steam [21]*

"Wheter I am connected to wifi or using my own data bitstrips keeps saying network error? This sucks big time." – *A review of Bitstrips [5]*

```
// Load tweets on creating the UI view
View onCreateView() {
   …
   load();
}

void load() {
+  // 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;
+  }
   // Below is functional code to post a tweet
   AsyncHttpClient client=new AsyncHttpClient();
   RequestParams params = new RequestParams();
   client.get(apiUrl, params,
              new AsyncHttpResponseHandler());
}
```

Figure 1: The code snippet of Twitter client to load home timeline. The original code execute load() without any fault-tolerance mechanisms. The bold lines are the patched fault-tolerant code that checks network connectivity and notifies user if no network.

Making networked apps reliable is notoriously hard for many app developers [16]. Because many existing popular network libraries were originally designed for traditional desktop applications, they ignore the dynamics of network connections and therefore the burden of writing fault-tolerant code falls to app developers. From a software engineering point of view, the difficulties of robust network programming are mainly due to the following issues:

(1) Fault-tolerant networking code can be complex. For example, consider a code snippet from the Twitter Lite client [23] shown in Figure 1. The *unbolded* lines are the original code, where `load()` is called to get timeline when user enters the Twitter home page. Here `load()` uses the popular Android Async HTTP library to construct and send out the HTTP request. This original code will

MOBILESoft '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5712-8/18/05…$15.00
https://doi.org/10.1145/3197231.3197237

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 `load()` fails (so the user does not know that the new tweets will not be delivered). The code shown in Figure 1 in bold with plus signs at the front addresses these issues. It first checks if the network is connected and then sends out the request. 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., `ConnectivityManager`, `NetInfo`) to check availability.

(2) Many mobile app developers today are amateur and hobbyist developers [4, 20, 34], who typically lack rigorous software development training, especially with regards to fault-tolerant mobile networking. According to a survey of over 10,000 app developers in the UK, 83% of app developers are self-taught and only 7% have attended a Bootcamp or other training course [24]. Many app developers come from desktop application development and are uninformed about the distinct complexities of mobile networking.

(3) Developers tend to delay or ignore writing fault-tolerant code during software development [32, 38, 55, 58], so they can focus on developing an app's core functionality and its acceptability to users. Adding fault-tolerant features up front would slow down the release cycle, unwisely delaying critical user feedback during the early stages of development.

1.2 State of The Art

A number of in-house testing approaches have been proposed for mobile apps [35, 43, 46, 47, 54]. Specially, dynamic fault injections are used to inject web errors and simulate unreliable network connections into the apps under testing to find bugs [43, 54]. While the run-time tools can accurately report bugs that cause the app crash for a given run, many NPDs can hardly be triggered by these testing frameworks, because those NPDs are not manifested under a disabled network (but rather under intermittent network), so it requires strict timing fault model to trigger NPDs. Also, as the next section shows, many types of NPDs do not manifest in crashes, such as the "miss connectivity check" and "no error message" in Figure 1, and therefore cannot be exposed by fault injection.

Many third-party network libraries have been built to facilitate network programming [1, 18, 19, 25]. These libraries expose some fault-tolerant APIs for developers to handle network errors. However, NPDs are still prevalent despite those APIs [36]. The major reason is that developers often cannot comprehensively think through the error handling of network errors and therefore cannot utilize these APIs correctly (more details in §2). NChecker statically detects NPDs caused by network library API misuses [36], however, fixing the problems found by the tool poses another level of challenge to novice developers: First, NChecker reports missed

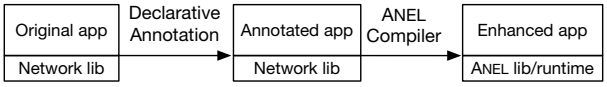

Figure 2: ANEL overview

fault-tolerant APIs or incorrectly configured API parameters under specific app context, but novice developers may lack the experience to leverage the information. According to NChecker user study, most developers cannot set proper retry mechanisms even though the tool accurately detects the error. Second, only part NPD types can be directly fixed by calling existing network library APIs. That is, for other NPDs, developers need to add fault-tolerant logic by themselves using complex native APIs, such as the code shown in Figure 1. A networked app has dozens of network operations, therefore the task is very time consuming.

1.3 Our Contributions

Because robust network programming requires a lot of expertise and many existing network libraries have been widely used by apps, in order to enable non-expert developers to add network fault-tolerance to the apps, we observe the need for two critical properties:

- The solution should be declarative, framed in familiar terminology and concepts, rather than in the alien language and concepts of fault-tolerant network implementation.
- The solution should enable the incremental augmentation of existing code rather than rewriting or refactoring it.

This paper presents ANEL, a declarative language and middleware for Android that meets these requirements. Figure 2 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 functoinality 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) [36] (See §2). Our declarative language is generalized from these NPDs, thus handling the vast majority of issues that would be encountered in the typical mobile app.

Figure 3 shows the Twitter Lite example using an ANEL annotation to achieve the same fault-tolerance as in Figure 1. The developer only needs to add a `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 annotation 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, the contributions of this paper are as follows:

```
View onCreateView() {
   …
   load();
}

void load() {
   @Anel_property({“UserReq”})
   AsyncHttpClient client=new AsyncHttpClient();
   RequestParams params = new RequestParams();
   client.get(apiUrl, params,
              new AsyncHttpResponseHandler());
}
```

Figure 3: Enhanced code using ANEL annotation to load tweets. The bold line is ANEL declarative annotation, which is translated by ANEL compiler into fault-tolerant code.

- A declarative annotation language for specifying network request semantics and enhancing network programming reliability, without reference to fault-tolerant concepts or their implementation (§3).
- A library designed for the unreliable mobile network and compatible with existing network libraries (§4).
- A complier translating the annotations into ANEL library fault-tolerant API calls (§5).
- 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 (§6).

2 BACKGROUND: NETWORK PROGRAMMING DEFECTS

To provide some background on why developers make mistakes in writing mobile networking code, this section describes different NPDs, and for each one, we discuss the corresponding fault-tolerant APIs of existing libraries and why it is hard for novice developers to utilize them. All NPDs types are characterized in previous work [36]. We will use the fault-tolerant APIs provided by the Android Async Http library (a top Android network library [2]) as an example. Other network libraries' APIs are similar.

No connectivity check. The developer does not check the network connectivity before sending a request. A connectivty check allows immediately returning back to the user if the phone is not connected to the network. It also can avoid consuming energy caused by attempting unnecessary network operations. Network operations tend to put (or keep) the wireless interface in a higher energy state.

To avoid these problems, Developers have to consider every network operation, yet non-experts often assume a reliable network.

No timeout. The developer does not set a timeout for network requests. Applying a timeout is important for catching errors caused by a slow or intermittent network. The Async Http library library has an API `setTimeout()` to set the timeout value.

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. The developer does not retry on a transient network error, but rather directly fails. Retry is a common practice for tolerating transient failures caused by an intermittent network. The Asynchronous Http library provides two methods for setting a retry policy: `allowRetryExceptionClass()`, which is used to accept retriable exception class types, and `setMaxRetriesAndTimeout()`, which is used to set retry times.

To use these methods, 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.

Over retry. Over retry happens for non-idempotent requests and background requests. HTTP POST requests generally have side-effects unless the server specifically deals with re-posting. Also, it is unnecessary to aggressively retry background requests because they are not time sensitive anyway.

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. With this defect, the developer has not added a check for the validity of a response from a network request. This likely causes a null pointer exeception. Most existing libraries do not validate responses automatically, so developers always have to add extra checking code before using the response value.

No reconnection on network switch. This type of defect is particularly for applications with real-time requirements, such as Skype and streaming Twitter: When the network switches among different hotspots, the app does not establish a new connection but rather still uses the stale one.

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. With this defect, the developer has not written code to automatically resume a failed request but requires the user's intervention. 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 ANEL DECLARATIVE LANGUAGE

The ANEL declarative language describes high-level application properties or requirements instead of imperative implementations on low-level fault-tolerant APIs. Developers can easily add these specifications to existing code via annotations. 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

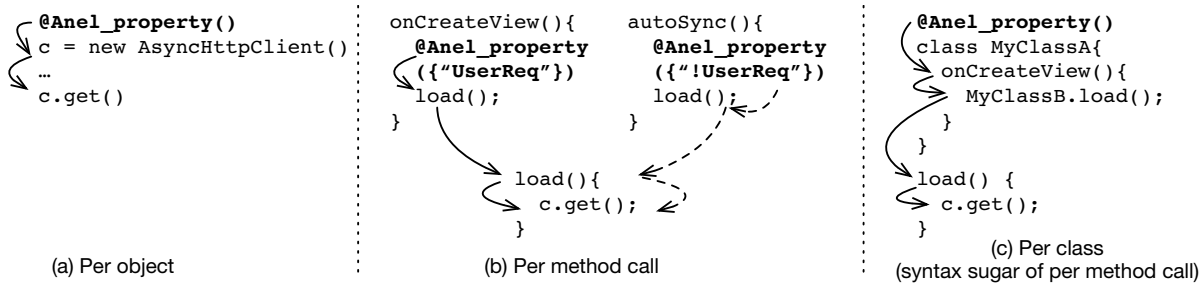

Figure 4: Scopes supported by ANEL. `post()` is lowest-level API to send out a HTTP request. Each arrow flow from the annotation indicates its control scope.

3.1 Property Specifications

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-seprated 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 §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 an 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 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 realtime data, e.g., displaying real-time tweats. 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.

3.2 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 features. It is useful when there exist conflicts between the the original code logic and the specifications defined in `@Anel_property` (discussed in §5.3). 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`.

3.3 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 3. A unique challenge is that network calls in utility code can be called in many situations, for example some being user

```
@Anel_property({"UserReq"})
View onCreateView {
   @Anel_property({"RepeatIsIdentical"})
   load(){}
}
                    ⇕ Equals
View onCreateView {
   @Anel_property({"UserReq","RepeatIsIdentical"})
   load(){}
}
```

(a) Scope chaining

```
@Anel_property({"UserReq","!RepeatIsIdentical"})
View onCreateView {
   @Anel_property({"RepeatIsIdentical"})
   load(){}
}
                    ⇕ Equals
View onCreateView {
   @Anel_property({"UserReq","RepeatIsIdentical"})
   load(){}
}
```

(b) Scope overriding

Figure 5: Scope chaining and overriding

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 [15]. Finally, some syntactic sugar is beneficial in lessening the number of annotations required. Figure 4 shows the essential features of ANEL annotation scopes.

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 4(a), the annotation's specification only influences operations on object `c`, like the `get()` request.

Method-call scoping: Figure 4(b) shows a simplified code snippet of a common coding pattern. Here `load()` is a shared utility called at two different call sites: one within `onCreateView()`, the callback for when a user loads an UI page; the other within `autoSync()`, called by a background service, which periodically sync with server. 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 4(b), the annotation for `onCreateView` influences the code path `onCreateView` → `load()` → `c = new ...`. The annotation for `autoSync()` influences the path `autoSync` → `load()` → `c = new ...`.

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 4(c) shows an example of the effects of a class annotation.

To cope with how annotations with overlapping scopes interact with each other, and allowing for a fine-grained annotation to override or extend a coarser one in exceptional situations, ANEL introduces two 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(b), the annotation spec `RepeatIsIdentical` of `load()` conflicts with the spec `!RepeatIsIdentical` of the outer method, so the inner annotation overrides the outer method'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(a), the annotation of method `load()` inherits the outer method's specification `UserReq`, so the effect is the same as applying `UserReq, RepeatIsIdentical` to `load()`.

3.4 Discussions

The ANEL annotations address the common NPDs described in §2. Advanced developers may want to only send requests under specific conditions, e.g., only use non-metered network, or only when the phone is plugged. Previous Android annotation framework APE [48], Tempus [49], Procastinator [53], and google's API JobScheduler [13] fit such demands well.

An alternative to specify the properties is to apply the idea of Aspect-oriented programming (AOP) [29, 30, 42], using pointcuts and advice to define what properties apply at specific program points. ANEL uses annotations because the semantics is more straightforwward 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. §6.1).

4 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 1, 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 §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 CONNECTIVITY_ACTION broadcast when the network status changes. After being notified, the monitor calls the ConnectivityManager system service to determine whether the network is online

```
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 get()
  @Override
  public void get(String url,
    ResponseHandlerInterface handler) {

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

Figure 6: Anel's implementation of the Async HTTP class. The subclass pattern can be generalized to other libraries.

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 methods. 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)
```

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 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.

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

```
void load() {
  AsyncHttpClient client=
    new AnelAsyncHttpClient({"UserReq"});
  RequestParams params = new RequestParams();
  client.get(apiUrl, params,
              new AsyncHttpResponseHandler());
}
```

Figure 7: The enhanced code of the code input shown in Figure 3. The bolded line is generated by the compiler and runtime.

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 suppressMine 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.

Figure 6 illustrates a code snippet of the ANEL implementation for Android Async Http library. `AnelAsyncHttpClient` subclasses `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, it parses `SucceedEventually` and sets `autoResumeFailure(True)` for `client` accordingly. To equip the network request `get()` 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 `get()` in the code). The reimplemented `get()` 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.

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 primitives. All the other `AsyncHttpClient` constructor calls in the original code do not need any change, unless similarly annotated. Thus, ANEL's subclasses make the translation straightforward. For the annotated code in Figure 3, the runtime executed code is shown in Figure 7.

5 ANEL COMPILER

5.1 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 [57]. We extended the FlowDroid Android app static analysis tool [28], 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 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 8 as a running example. This piece of code has a `load` utility that uses the Android Async Http library to send a network request. This method is called by two callers `OnCreateView()` and `autoSync()`. The calls to `OnCreateView()` and `autoSync()` are annotated with `UserReq` and `!UserReq` respectively.

5.2 Annotation Parser

The parser first identifies all the annotations, and the context of annotated objects, method-calls, methods, and classes, checking syntax 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`.

Next, for each annotation site, the compiler identifies all the network object constructors reachable by that annotation. This is trivial and precise for object annotations, but requires object-sensitive interprocedural reachability analysis for the other annotations, as enabled by Soot and FlowDroid, and is conservative. (The code generator generates code to make the identification precise at runtime (§5.4).) 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 replacment with ANEL fault-tolerant objects. In our example code, there are two paths to the constructor for `client`: *OnCreateView* → *load* → *get* and *autoSync* → *load* → *get*.

5.3 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 a developer annotates code that already has some fault-tolerant logic, how to make sure the generated fault-tolerant code does not conflict with the original one?

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 `setMaxRetries()` 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:

FT primitive	Specification	Conflicted API settings
retry	UseReq or RepeatIsIdentical	Set retry times 0
retry	!UseReq or !RepeatIsIdentical	Set retry times larger than 0
errorMsg	!UseReq	Call error message API
errorMsg	SucceedEventually	Call error message API

Table 1: Conflicted pairs of annotation specifications and fault-tolerant API settings for a given fault-tolerant (FT) primitive

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 8, in the path *autoSync* → *load* → *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 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 *OnCreateView* → *load* → *get*, there is an explicit call setting the number of retry attempts to three, and the `UserReq` specification also stipulates retries. Although 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 [36].

5.4 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 §4, 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 dynamically, due to method-call scoping.

The bottom of Figure 8 shows the code generated from the example input code. The input exhibits the first challenge cited above,

Original Input

```
@Anel_property({“UserReq”})
void onCreateView(){
  load();
}

@Anel_property({“!UserReq”})
@Anel_suppressMine({“retry”})
void autoSync(){
  load();
}

void load(){
  AsyncHttpClient client=new AsyncHttpClient();
  client.setMaxRetriesAndTimeout(3,0);
  client.get();
}
```

Compiler Output

```
void onCreateView(){
  Anel.pushProperty("UserReq");
  load();
  Anel.popProperty();
}

void autoSync(){
  Anel.pushProperty({"!UserReq"});
  Anel.pushSuppressMine({"retry"});
  load();
  Anel.popProperty();
  Anel.popSuppressMine();
}

void load(){
  AnelAsyncHttpClient client =
    new AnelAsyncHttpClient(Anel.topProperty(),
                            Anel.topSuppressAnel(),
                            Anel.topSuppressMine());
  client.setMaxRetriesAndTimeout(3,0);
  client.get();
}
```

Figure 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 scenario where the networking utility (`load`) is called on two different code paths, each with it's 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.

As mentioned in §3.3, the semantics of ANEL's method-call scoping is fundamentally dynamic scoping [15], 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 §3.3. 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 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 `autoSync`'s call to `load`, the list `{"!UserReq"}` is pushed on the property stack, and `{"retry"}` is pushed on the suppress-mine stack. 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 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.

6 EVALUATION

6.1 Case Study: GPSLogger

As the importance of eliminating NPDs in mobile apps has been well studied [36], we focus on the expressiveness and practicality of using ANEL for eliminating NPDs. We present a case study that show how ANEL can ease developers' work in real-world programs by (1) enabling easy-to-understand specifications, and (2) handling exceptional situations, and (3) concisely specifying fault-tolerant features over varying code patterns.

GPSLogger is a mobile app that logs the GPS coordinates for the user's travels [10]. 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 network library. 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.

Figure 9 shows the relevant code snippet from GpsLogger. The network utility `UrlJob` is called on different two code paths: one starts from `GpsMainActivty.onMenuItemClick()`, a UI handler callback, the other starts from `GpsLoggerService.autoSendLogFile()` in a background service.

Annotation specifications: 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.

Annotation locations: To add annotations to the code, we needed to look only at the upper-level method calls. Their semantics – user call versus background call – were clearly communicated by the

```
public class GpsMainActivity extends
    AppCompactActivity {
  public boolean onMenuItemClick() {
    sendToOpenGTS();
  }

  @Anel_property("UserReq")
  private void sendToOpenGTS() {
    FileSender.uploadFile()
  }
}

public class GpsLoggingService extends Service {
  @Anel_property({"!UserReq",
    "SucceedEventually"})
  void autoSendLogFile() {
    FileSender.autoSendFiles();
  }
}

public class FileSender {
  public void autoSendFiles(){
    uploadFile();
  }

  public void uploadFile() {
    sendLocations();
  }

  public void sendLocations(){
    …prepare location data…
    (new Thread(new UrlJob())).start()

  }
}

public class UrlJob implement Runnable{
  public void run() {
    OkHttpClient client = new OkHttpClient();
    Request request = new
        Request.Builder().url(logUrl).build();
    Response resp =
        client.newCall(request).execute();
    resp.body().close();
  }
}
```

Figure 9: We improve GpsLogger by applying two method-scoped annotations: "UserReq" is for user-initiated request and "!UserReq, SucceedEventually" is for background request.

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). Therefore, we easily apply annotations to methods `sendToOpenGTS` and `autoSendLogFile`, as shown in Figure 9.

Discussion: 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. Despite these challenges, ANEL annotation specifications are easy to formulate based on basic cues in the app's design and implementation. Moreover, although there is a deep call stack from the upper-level methods to the lowest level HTTP requests, the scoping features of of the language makes the annotation very easy for developers: they can easily figure about the properties from the semantics of the entry point methods.

6.2 Other Apps

While ANEL annotation framework is generalized for existing network libraries, we have implemented ANEL library and compiler for two popular network libraries: Async Http library and OkHttp library. We apply ANEL to 6 open source Android applications originally using those two libraries, as shown in Table 2: column 3 indicates the number of network request call sites; column 4 is the number of annotation lines added; column 5 is the scope of applied annotations; column 6 is the corresponding annotation specifications. The annotating process is similar to what described in the GPSLogger case studies.

These apps present different code patterns: For example, in New York Times search, some network calls contained fault-tolerant code (show error message for user requests), but others not. So we need to use `suppressAnel` modifier to handle such exceptional conditions. In Twitter Lite, all the network utilities are encapsulated in a class, and we can see how class-level annotation helps save annotation effort by avoiding repetitive annotations.

6.3 Usability Study

We conduct a controlled usability study to understand if developers can easily and correctly apply ANEL annotations to the code. We recruited 5 Android programmers whose programming experiences are 1-10 months. They first read an instruction about the syntax, properties and basic scoping concepts of ANEL. Then we show them the source code of Twitter Lite [23], asking them to annotate two network operations: `getHomeTimeline()` for loading tweets in home page and `replyTweet()` for replying a tweet.

From the study we found that: (1) All developers can easily reason about the annotation properties according to their demands. For example, `getHomeTimeline()` is called either when user enters to home page or when user manually refreshes the timeline. Some programers annotates `UserReq` for both because both are UI interactions; but other programmers skip `UserReq` for the first scenario because they think users do not actively trigger the loading so they should not be notified about the network failure. (2) They can correctly apply method-scoping features for the shared network utility methods. (3) They can quickly decide the annotation properties after examining the entry points (the upper-level caller of the network operations).

6.4 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 three 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 scoped annotation. We used Android's Async Http library and OkHttp library for this evaluation. For the hand-written

App	Lib used	#Net. req	#Anno. lines	Scope	Annotations
GpsLogger [10]	OkHttp	2	2	Method	`Anel_property("UserReq")` `Anel_property("!UseReq", "MustSucceed")`
Twittler Lite [23]	Async Http	7	1	Class	`Anel_property("UserReq")`
Kangaroo [14]	Async Http	4	4	Object	`Anel_property("UserReq", "Idempotent")`
Hacker news [11]	OkHttp	4	4	Method	`Anel_property("UserReq")`
Instagram photo viewer [12]	Async Http	2	1	Object	`Anel_property("UserReq")`
New York Times search [17]	Async Http	2	2	Object	`Anel_property("UserReq")` `Anel_suppressAnel("errorMsg")`

Table 2: Evaluated Android apps and applied ANEL annotations

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 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 overhead of dynamic scoping.

Http client	Latency (s)
Android Async	0.161
Android Async + Annotation(object)	0.173
Android Async + Annotation(method-call)	0.175
OkHttp	0.164
OkHttp + Annotation(object)	0.164
OkHttp + Annotation(method-call)	0.169

Table 3: Latency to download 1K data for different HTTP clients

To measure energy overhead, we used the Qualcomm Trepn Profiler [22] 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 measureable energy overhead.

The reason for ANEL's negligible runtime overhead is that the compiler does much of the work at compile time. The runtime costs of ANEL are induced at just three points: entry to a scope, exit from a scope, and on networking calls within a scope, mostly simple stack operations. 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.

7 RELATED WORK

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-insensitve code segments in order to save energy [48, 49]. Indeed, annotations have been used for code injection/generation [3, 6], DoS resistant programming [51], program verification [40], library and application performance optimization [33, 52], and bug detection [31, 37, 59]. 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 [41, 44, 45]. ANEL is an example of Open Implementation, as its annotations enable customizing fault-tolerance orthogonally from core behaviors, without specialized knowledge of fault tolerance.

The Android framework is being continuously improved for network programming. For example, SyncAdapter syncs data transfers only when network is available, and batches network operations to save energy [7]. DownloadManager monitors network and restores long-running downloading tasks automatically [8]. However, although these systems' APIs provide important functionalities, they are non-trivial to use. The developer needs to learn a lot of system mechanisms in order to use them correctly. ANEL is unique in that it offers a language for non-experts to express their application beavhiors, and ANEL inserts the necessary fault-tolerant code.

8 CONCLUSION

Most mobile developers are novices in fault-tolerant networking, and in any event need to solidify an app's core features before introducing fault-tolerance. ANEL is 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 evaluation on 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.

ACKNOWLEDGMENTS

We greatly appreciate MobileSoft anonymous reviewers for their insightful feedback. This research is supported by NSF CNS-1017784, NSF CCF-1719155 and NSF CNS-1321006.

REFERENCES

[1] Android Asynchrounous Http Library. http://loopj.com/android-async-http/.

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

[3] AndroidAnnotaions. http://androidannotations.org/

[4] App Developers Who Are Too Young to Drive. http://online.wsj.com/articles/SB10001424052702303410404577468670147772802.

[5] Bitstrips. http://www.bitstrips.com/.

[6] ButterKnife. http://jakewharton.github.io/butterknife/

[7] Creating a Sync Adapter. https://developer.android.com/training/sync-adapters/creating-sync-adapter.html

[8] DownloadManager. https://developer.android.com/reference/android/app/DownloadManager.html

[9] Fun Run 2. http://play.google.com/store/apps/details?id=com.dirtybit.funrun2.

[10] GpsLogger. http://code.mendhak.com/gpslogger/

[11] Hacker news. https://github.com/manmal/hn-android

[12] Instagram photo viewer. https://github.com/tanlnm512/InstagramPhotoViewer

[13] JobScheduler. https://developer.android.com/reference/android/app/job/JobScheduler.html/.

[14] Kangaroo. https://github.com/mehikmat/Kangaroo

[15] Lexical and Dynamic Scoping. https://courses.cs.washington.edu/courses/cse341/09wi/general-concepts/scoping.html

[16] Mobile development is tougher than people think-A brief look at what makes mobile app development so tricky. http://www.itworld.com/article/2701225/mobile/mobile-development-is-tougher-than-people-think.html.

[17] New York Times search. https://github.com/tanlnm512/NewYorkTimesNewsSearch

[18] OkHttp. http://square.github.io/okhttp/.

[19] Retrofit. http://square.github.io/retrofit/.

[20] State of The Developer Nation Q3 2014. http://www.visionmobile.com/product/developer-economics-q3-2014/.

[21] Steam. https://play.google.com/store/apps/details?id=com.valvesoftware.android.steam.community.

[22] Trepn Power Profiler. https://developer.qualcomm.com/software/trepn-power-profiler

[23] Twitter Lite. https://github.com/tanlnm512/Twitter

[24] UK App Economy 2014. http://www.visionmobile.com/product/uk-app-economy-2014/.

[25] Volley. http://developer.android.com/training/volley/index.html.

[26] Ebuddy + weak signal = battery death. http://androidforums.com/threads/ebuddy-weak-signal-battery-death.85643/.

[27] Android App not seeing Server over wifi. https://forums.plex.tv/index.php/topic/103094-android-app-not-seeing-server-over-wifi.

[28] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. ACM, Edinburgh, United Kingdom, 259–269. DOI: http://dx.doi.org/10.1145/2594291.2594299

[29] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. 2013. AppGuard: Enforcing User Requirements on Android Apps. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)*. Springer-Verlag, Berlin, Heidelberg, 543–548. DOI: http://dx.doi.org/10.1007/978-3-642-36742-7$_3$9

[30] Eric Bodden. 2013. Easily Instrumenting Android Applications for Security Purposes. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. ACM, New York, NY, USA, 1499–1502. DOI: http://dx.doi.org/10.1145/2508859.2516759

[31] Nathan Cooprider, Will Archer, Eric Eide, David Gay, and John Regehr. 2007. Efficient Memory Safety for TinyOS. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems (SenSys '07)*. ACM, New York, NY, USA, 205–218. DOI: http://dx.doi.org/10.1145/1322263.1322283

[32] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dussea, and Ben Liblit. 2008. EIO: Error Handling is Occasionally Correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*. USENIX Association, Berkeley, CA, USA, Article 14, 16 pages. http://dl.acm.org/citation.cfm?id=1364813.1364827

[33] Samuel Z. Guyer and Calvin Lin. 1999. An Annotation Language for Optimizing Software Libraries. In *Proceedings of the 2Nd Conference on Domain-specific Languages (DSL '99)*. ACM, New York, NY, USA, 39–52. DOI: http://dx.doi.org/10.1145/331960.331970

[34] Peng Huang, Tianyin Xu, Xinxin Jin, and Yuanyuan Zhou. 2016. DefDroid: Towards a More Defensive Mobile OS Against Disruptive App Behavior. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '16)*. ACM, New York, NY, USA, 221–234. DOI: http://dx.doi.org/10.1145/2906388.2906419

[35] Konrad Jamrozik and Andreas Zeller. 2016. DroidMate: A Robust and Extensible Test Generator for Android. In *Proceedings of the International Conference on Mobile Software Engineering and Systems (MOBILESoft '16)*. ACM, New York, NY, USA, 293–294. DOI: http://dx.doi.org/10.1145/2897073.2897716

[36] Xinxin Jin, Peng Huang, Tianyin Xu, and Yuanyuan Zhou. 2016. NChecker: Saving Mobile App Developers from Network Disruptions. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 22, 16 pages. DOI: http://dx.doi.org/10.1145/2901318.2901353

[37] Rob Johnson and David Wagner. 2004. Finding User/Kernel Pointer Bugs with Type Inference. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13 (SSYM'04)*. USENIX Association, Berkeley, CA, USA, 9–9. http://dl.acm.org/citation.cfm?id=1251375.1251384

[38] Mary Beth Kery, Claire Le Goues, and Brad A. Myers. 2016. Examining Programmer Practices for Locally Handling Exceptions. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 484–487. DOI: http://dx.doi.org/10.1145/2901739.2903497

[39] Hammad Khalid. 2013. On Identifying User Complaints of iOS Apps. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE'13)*. IEEE Press, San Francisco, CA, USA, 1474–1476. http://dl.acm.org/citation.cfm?id=2486788.2487044

[40] Sarfraz Khurshid, Darko Marinov, and Daniel Jackson. 2002. An Analyzable Annotation Language. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*. ACM, New York, NY, USA, 231–245. DOI: http://dx.doi.org/10.1145/582419.582441

[41] G. Kiczales. 1996. Beyond the Black Box: Open Implementation. *IEEE Softw.* 13, 1 (Jan. 1996), 8, 10–11. DOI: http://dx.doi.org/10.1109/52.476280

[42] Gregor Kiczales and Erik Hilsdale. 2001. Aspect-oriented Programming. *SIGSOFT Softw. Eng. Notes* 26, 5 (Sept. 2001), 313–. DOI: http://dx.doi.org/10.1145/503271.503260

[43] Chieh-Jan Mike Liang, Nicholas D. Lane, Niels Brouwers, Li Zhang, Börje F. Karlsson, Hao Liu, Yan Liu, Jun Tang, Xiang Shan, Ranveer Chandra, and Feng Zhao. 2014. Caiipa: Automated Large-scale Mobile App Testing Through Contextual Fuzzing. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking (MobiCom '14)*. ACM, Maui, Hawaii, USA, 519–530. DOI: http://dx.doi.org/10.1145/2639108.2639131

[44] Victor B. Lortz and Kang G. Shin. 1994. Combining Contracts and Exemplar-based Programming for Class Hiding and Customization. In *Proceedings of the Ninth Annual Conference on Object-oriented Programming Systems, Language, and Applications (OOPSLA '94)*. ACM, New York, NY, USA, 453–467. DOI: http://dx.doi.org/10.1145/191080.191150

[45] Chris Maeda, Arthur Lee, Gail Murphy, and Gregor Kiczales. 1997. Open Implementation Analysis and Design. In *Proceedings of the 1997 Symposium on Software Reusability (SSR '97)*. ACM, New York, NY, USA, 44–52. DOI: http://dx.doi.org/10.1145/258366.258383

[46] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 94–105. DOI: http://dx.doi.org/10.1145/2931037.2931054

[47] K. Moran, M. Linares-VÃąsquez, C. Bernal-CÃąrdenas, C. Vendome, and D. Poshyvanyk. 2016. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 33–44. DOI: http://dx.doi.org/10.1109/ICST.2016.34

[48] Nima Nikzad, Octav Chipara, and William G. Griswold. 2014. APE: An Annotation Language and Middleware for Energy-efficient Mobile Application Development. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 515–526. DOI: http://dx.doi.org/10.1145/2568225.2568288

[49] Nima Nikzad, Marjan Radi, Octav Chipara, and William G. Griswold. 2015. Managing the Energy-Delay Tradeoff in Mobile Applications with Tempus. In *Proceedings of the 16th Annual Middleware Conference (Middleware '15)*. ACM, New York, NY, USA, 259–270. DOI: http://dx.doi.org/10.1145/2814576.2814803

[50] F. Palomba, M. Linares-VÃąsquez, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. 2015. User reviews matter! Tracking crowdsourced reviews to support evolution of successful apps. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 291–300. DOI: http://dx.doi.org/10.1109/ICSM.2015.7332475

[51] Xiaohu Qie, Ruoming Pang, and Larry Peterson. 2002. Defensive Programming: Using an Annotation Toolkit to Build DoS-resistant Software. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 45–60. DOI: http://dx.doi.org/10.1145/844128.844134

[52] D. Quinlan, M. Schordan, R. Vuduc, and Qing Yi. 2006. Annotating user-defined abstractions for optimization. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*. 8 pp.–. DOI: http://dx.doi.org/10.1109/IPDPS.2006.1639722

[53] Lenin Ravindranath, Sharad Agarwal, Jitendra Padhye, and Chris Riederer. 2014. Procrastinator: Pacing Mobile Apps' Usage of the Network. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '14)*. ACM, New York, NY, USA, 232–244. DOI: http://dx.doi.org/10.1145/2594368.2594387

[54] Lenin Ravindranath, Suman Nath, Jitendra Padhye, and Hari Balakrishnan. 2014. Automatic and Scalable Fault Detection for Mobile Applications. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'14)*. ACM, Bretton Woods, New Hampshire, USA, 190–203. DOI: http://dx.doi.org/10.1145/2594368.2594377

[55] Hina Shah, Carsten Görg, and Mary Jean Harrold. 2008. Why Do Developers Neglect Exception Handling?. In *Proceedings of the 4th International Workshop on Exception Handling (WEH '08)*. ACM, New York, NY, USA, 62–68. DOI: http://dx.doi.org/10.1145/1454268.1454277

[56] Alok Tongaonkar, Shuaifu Dai, Antonio Nucci, and Dawn Song. 2013. Understanding Mobile App Usage Patterns Using In-app Advertisements. In *Proceedings of the 14th International Conference on Passive and Active Measurement (PAM'13)*. Springer-Verlag, Hong Kong, China, 63–72. DOI: http://dx.doi.org/10.1007/978-3-642-36516-4$_7$

[57] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'99)*. IBM Press, Mississauga, Ontario, Canada, 13–. http://dl.acm.org/citation.cfm?id=781995.782008

[58] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 249–265. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan

[59] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. 2006. SafeDrive: Safe and Recoverable Extensions Using Language-based Techniques. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7 (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 4–4. http://dl.acm.org/citation.cfm?id=1267308.1267312