2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)

Learning Performance Optimization from Code Changes for Android Apps

Ruitao Feng
SCSE
Nanyang Technological University
Singapore
Email: feng0082@e.ntu.edu.sg

Guozhu Meng
SCSE
Nanyang Technological University
Singapore
Email: gzmeng@ntu.edu.sg

Xiaofei Xie
SCSE
Nanyang Technological University
Singapore
Email: xiexiaofei@tju.edu.cn

Ting Su
SCSE
Nanyang Technological University
Singapore
Email: tsuletgo@gmail.com

Yang Liu
SCSE
Nanyang Technological University
Singapore
Email: yangliu@ntu.edu.sg

Shang-Wei Lin
SCSE
Nanyang Technological University
Singapore
Email: shang-wei.lin@ntu.edu.sg

***Abstract*—Performance issues of Android apps can tangibly degrade user experience. However, it is challenging for Android developers, especially a novice to develop high-performance apps. It is primarily attributed to the lack of consolidated and abundant programmatic guides for performance optimization. To address this challenge, we propose a data-based approach to obtain performance optimization practices from historical code changes. We first elicit performance-aware Android APIs of which invocations could affect app performance to a large extent, identify historical code changes that produce impact on app performance, and further determine whether they are optimization practices. We have implemented this approach with a tool PERFOPTIMIZER and evaluated its effectiveness in 2 open source well-maintained projects. The experimental results found 83 changes relevant to performance optimization. Last, we summarize and explain 5 optimization rules to facilitate the development of high-performance apps.**

***Keywords*-Android App; Change Abstraction; Performance Optimization;**

I. INTRODUCTION

Performance is an important metric to assess the quality of Android apps. High-quality apps are expected to use power sparingly and respond quickly [5]. However, Android apps are suffering from serious performance issues such as energy depletion, memory bloating, and GUI lagging [8]. These issues are attributed manifoldly from the app view: the wide presence of demanding high-consuming resources for Android apps. In this paper, we aim to find those improper programming practices and proposing optimization practices.

Android development documentation has offered a number of performance tips to assist app development [5]. These tips are intuitively easy-to-follow, whereas there are still many performance optimization practice are unknown. For example, relocating some tasks from background threads to the main thread sometimes can improve app performance to some extent, which is found by our analysis in Section V-A.

A number of work has been proposed to enable performance bug detection and optimization. Pathak *et al.* [9] proposed to detect energy hotspots by profiling the energy consumption of resources. Liu *et al.* [8] conducted an empirical study on resource leak and subsequently developed an approach to detect pertinent performance bugs. Firtman [4] summarized many best performance practices based on developers' programming experience. Different from other work, we aim at distilling performance programming optimization practices from historical code changes.

To achieve this goal, we have to address two challenges. First, how to identify the code changes related to performance optimization (*e.g.*, code refactoring). Second, how to abstract the optimization practices effectively from code changes.

In this paper, we propose a systematic approach, termed as PERFOPTIMIZER, to solve the aforementioned challenges. We address the first challenge in three steps. First, we specify our searching clues in the massive amount of code changes for fast localization. We start with performance-aware APIs (PAAs) (see Section II-A) that can significantly affect app performance. Second, we extract code changes from one open-source Android project (see Section IV-A). Third, we conduct an impact analysis by portraying API usages in these changes, and determine the performance impact to APIs (see Section IV-B). To overcome the second challenge, we define notations to abstract Java code and primitives, and then summarize these optimization practices for the benefits of future research (see Section IV-D).

Contributions. We make the following contributions.

- We propose a proof-of-concept approach, PERFOPTIMIZER, to identify performance optimizations from code changes of Android projects.
- We conduct impact analysis based on the three heuristics (i.e., add/deletion, substitution, and usage change), which can determine how the code changes affect the usage of

performance-aware APIs.
- We have evaluated the effectiveness of our approach on 2 open-source projects, and successfully identified 83 changes for performance, and summarized 5 optimization patterns.

II. Background

A. Performance Issues in Android Apps

Android apps, the most popular GUI applications running on mobile devices, put more emphasize on performance optimization than traditional software (*e.g.*, desktop GUI applications) due to their constrained resources. Therefore, it is crucial for app developers to follow the best practices for performance. However, in reality, many apps are still suffering from performance issues.

In this paper, we identify performance optimization starting from considering inappropriate uses of performance-aware Android APIs. These APIs are resource-consuming, so that one inappropriate use of PPAs could lead to performance degradation. We obtain the list of PAAs primarily from [7]. In addition, we supplement this list by adding some methods from popular third party libraries like Realm[1].

B. Git Code Changes

Git is a widely-used version control system to track changes for computer files. The changes are organized in the *commit*s, and one commit records the involved developers, update time, the comment, and the changes.

In this paper, we concentrate on the commits that elevate app performance by altering the usage of specific PPAs. In Figure 1, the method getDrawable contains a stream input method which is a performance-critical API, so that this method will be added to our local pattern list. We can generate a predominator path that consists of condition node with the CFG of local method. By comparing the paths of these two versions, we can see the precondition if(isDrawableCached()) outside the target API was detected. With the context code , we can easily know that would be a performance optimization. As a validation, we reproduced it with a simple scenario on real devices, and it caused unnegligible performance problem.

In this paper, we aim at mining performance optimization practices from commits.

III. The Methodology

Figure 2 shows the overview of our approach. The approach proceeds in three stages:

Code Change Collection. Our approach starts from collecting open-source Android projects maintained by Git. It stems from two considerations: The well-documented change history makes their semantics easier to infer; the extraction of optimization practices at the code level is easier to follow.

[1]https://realm.io/blog/realm-for-android/

```
  public View display(Context context, int position, View
       v) {
  ...
+     if( isDrawableCached() )
+     {
-         contactIcon.setImageDrawable(getDrawable(context)
   );
+         contactIcon.setImageDrawable(getDrawable(
   contactIcon.getContext()));
+     }
  ...

  public Drawable getDrawable(Context context) {
  ...
        try {
           inputStream = context.getContentResolver().
                openInputStream(contactPojo.icon);
           return Drawable.createFromStream(inputStream,
                null);
  ...
  }

  boolean isDrawableCached()
  {
      return icon != null;
  }
```

Figure 1. Motivation Example of Performance Optimization

Performance Optimization Identification. After we obtain a number of change instances sensitive to performance, we employ static analysis to identify whether one instance is made for performance optimization. It consists of two key steps. We build control flow graphs for each change instance, and check whether the changes affect usage of performance-aware APIs. After that, we infer its impact (positive/negative) to the APIs by several rules.

Optimization Pattern Formalization. At last, we propose a notation system to represent optimization patterns by revealing the differences between two versions of code. We perform a graph traversal to elicit the usages of API and abstract arbitrary code without loss of generality. On the other hand, we also resort to manual analysis on the summarized patterns.

IV. Performance Optimization Identification & Formalization

In this section, we introduce the details about how to identify performance optimization.

A. Per-method Code Change

For each commit, Git performs a file-wise comparison between two versions, and therefore all changes are scattered in one Java file. The changes occur in varying components including class declaration, fields or methods. To facilitate the analysis, we first determine the location of the changes, and identify the corresponding components. Since the invocations of APIs generally take place in methods, we only consider the changes occurring in method.

After filtering the changes, we put forward a per-method model to describe code change.

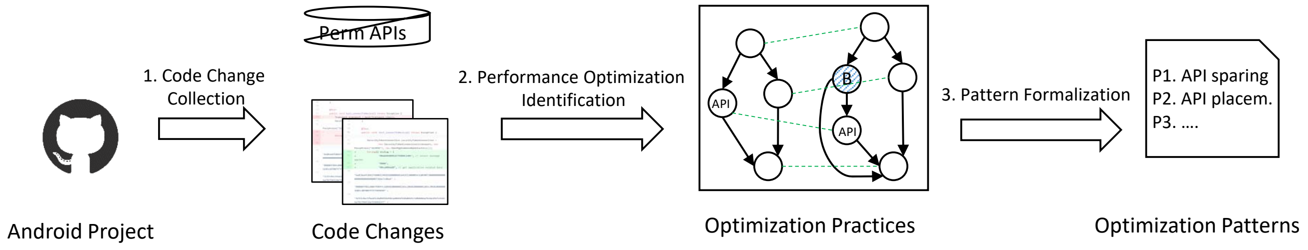

Figure 2. System overview of PerfOptimizer

Definition 1: In one commit, a per-method change instance M acts on one method and contains altered code Δ and constant code C i.e., $M = \{\Delta, C\}$ and $\Delta \neq \emptyset$. Altered code Δ denotes how developers modify code with removed code R, and added code A, i.e., $\Delta = \{R, A\}$ and $R \cup A \neq \emptyset$. **Rationale**. These changes may produce impacts to apps in performance. We conduct performance impact analysis within one method since it bounds the impact scope of change.

With code changes, we locate the methods where changes reside in and divide method body into *altered* and *constant* parts to recover the change impact.

B. Change Impact Analysis

In this paper, we rely on change impact analysis to identify performance optimization. Before introducing the technical details, we define the following notations:

Definition 2: Given one change instance M, we define Pr as a function to identify the changes of invoked performance-aware APIs in this instance, *i.e.* $Pr(M) = \{P_a, P_b\}$. P_a is the list of invoked PAAs before change. P_b is the list of invoked PPAs after change.

In order to determine whether one change produces performance impact, we propose three types of impact factors as follows.

I1. Addition/Removal of APIs. The utility of PAAs may suffer from performance issues. As a result, the changes of invoked PAAs could either optimize or degrade app performance. Given one change instance M and $Pr(M) = \{P_a, P_b\}$, if $P_a \subsetneq P_b$, then the changed code has added some PAAs. If $P_a \supsetneq P_b$, then the changed code has removed ones.

I2. Substitution of APIs. Along with the evolving Android system, some APIs may be deprecated or altered due to varying reasons (e.g., optimization, security and function extension). Developers may replace specific APIs with some substitutions which are functionally equivalent. Given the change instance M and $Pr(M) = \{P_a, P_b\}$ ($P_a \neq \emptyset \; and \; P_b \neq \emptyset$), substitutions occur when $P_a \neq P_b$ and $\exists p \in P_a \wedge p \notin P_b \wedge \exists p' \in P_b \wedge p' \notin P_a$ (Note that we regard overloaded methods as two methods here).

Besides the changes of API invocations, we also explore API usage changes. Assuming one change instance M, of which the original method is $M_a = \{R, C\}$ and the changed method is $M_b = \{A, C\}$. For these two methods, we construct their corresponding control flow graphs G_a, G_b, respectively, then compute dominators for each PAA in the method.

Definition 3: For one control flow graph $G = (V, E)$, one node a dominates another node b if every path from the entry to node b must pass though node a, which is denoted as $a >> b$. In addition, we define $\mathsf{dom}(v_o, G) = \{v_i | v_i >> v_o \wedge v_i \in G\}$ as the set of dominators of v_o in G.

I3. Usage Change of APIs. There exists many types of program transformations in code [13]. For simplicity, we only consider one program transformation, where the change has a control-flow influence on performance-aware APIs. More specifically, one change instance control-flow influences one API if the dominator set of this API has changed. Given two versions of control flow graph G_a and G_b of one change instance, if there exists one API a where $a \in P_a \; and \; a \in P_b$, but $dom(a, G_a) \neq dom(a, G_b)$, we determine that the change has influence on API a.

Figure 3 shows how the program (represented by its control flow) is changed in light of impact factors. S_* denotes one ordinary Java statement, and A_* denotes the invocation of one PPA. $Init$ shows the original program, indicating $S_1 \rightarrow A_1 \rightarrow S_2$. I_1 removes the PAA, I_2 replaces PAA A_1 with A_2, and I_3 changes the invocation manner to the PAA by adding one execution judgment.

C. Optimization Identification

Based on the aforementioned rules, we employ a lightweight static analysis to specify the instances that have performance impact. In particular, the first two rules can be achieved by lexically searching involved PAAs in instances. For the third rule, we implement the algorithm [6] to find dominators for each PAA. If the dominators are changed in a change instance, we regard it has performance influence.

Optimization Determination. With change impact analysis, we identify instances that have performance influence to target APIs. As a result, we propose three criteria to determine optimizations.

- **C1.** One PAA is removed.
- **C2.** One PAA a_1 is replaced with another one a_2, where a_2 is documented as performance better than a_1.

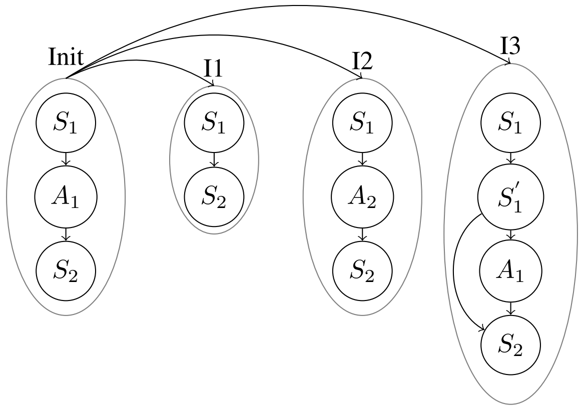

Figure 3. Demonstration for impact analysis on PAAs

- **C3.** Execution probability of one PAA a is reduced. In one control flow graph, the paths from the entry point to the exit point are denoted as T, of which the distinct paths passing through a are denoted as T_a. So the execution probability is $\frac{|T_a|}{|T|}$.

Algorithm 1 shows how to identify performance optimization from change instances. Given the set of change instances M, we first compute the corresponding PAA sets P_a and P_b for each instance (Line 4). The first criterion **C1** is checked at Line 5, and **C2** is checked from Line 7 to 11. It replies on manual check to determine whether P' are optimized counterparts of P at Line 10 in this paper (it will be automatized in our further work). The third criterion are implemented from Line 12 to 19, where we compute control flow graphs for two versions of methods, and determine the optimization as per execution probability.

D. Pattern Formalization

After identifying change instances in favor of performance optimization, we present specific patterns manually to formalize them. In accord with the three criteria in Section IV-C, the formal definitions for performance optimization could be represented in terms of change instances. If one PAA is removed, it can be represented as $PAA \rightarrow \emptyset$. If one PAA is replaced by another one, it can be represented as $PAA_i \rightarrow PPA_j$. The usage changes of PAAs are varying and thus elaborately discussed in Section V-A.

V. Tool Implementation & Evaluation

We develop a proof-of-concept tool named PerfOptimizer to extract optimization patterns from a large number of code changes. To make it automated, we implement a change analyzer to identify changes that somehow affect the usage of PAAs; then leverage JavaParser [10] to construct Abstract Syntax Tree (AST) from partial Java code. A lightweight static analyzer is implemented to derive control flow graphs and determine performance optimizations.

Algorithm 1: Performance Optimization Identification

```
Input: $M$: the set of change instances
       $M = \{M_1, M_2, ..., M_n\}$
Output: $M_o$: the set of change instances that
        optimize performance, where $M_o \subseteq M$
1  $M_o \leftarrow \emptyset$;
2  foreach $M_i \in M$ do
3      optimized = False;
4      cal $Pr(M_i) = \{P_a, P_b\}$;
5      if $P_a \supsetneq P_b$ then
6          optimized = True;
7      else if $P_a - P_b \neq \emptyset$ && $P_b - P_a \neq \emptyset$ then
8          $P = P_a - P_b$;
9          $P' = P_b - P_a$;
10         if $P'$ optimizes $P$ then
11             optimized = True;
12     else
13         get $M_a$ and $M_b$ from $M$;
14         $G_a = control\_flow\_graph(M_a)$;
15         $G_b = control\_flow\_graph(M_b)$;
16         foreach $p \in P_a \cap P_b$ do
17             calcuate execution probability of $p$ in $G_a$
               and $G_b$ as $ep_a, ep_b$;
18             if $ep_a > ep_b$ then
19                 optimized = True;
20     if optimized then
21         $M_o = M_o \cup M_i$;
22 return $M_o$
```

Evaluation Subject. We select two open-source projects hosted in Github for primary study. In detail, the first studied project is KISS[2], which is a blazingly fast launcher for Android. The second project is MiMangaNu[3], which is one app of reading and organizing mangas.

Detection Results. PerfOptimizer found 48 changes which improve app performance, amounting for 2.5% of all commits in project KISS, and 35 changes in project MiMangNu. All of these changes are confirmed to be performance optimization with manual analysis.

A. Case Study of Performance Optimization

In this section, we evaluate the usefulness of PerfOptimizer by summarizing some optimization patterns. We present 5 types of typical performance optimization which are elaborated as follows.

[2]https://github.com/Neamar/KISS
[3]https://github.com/raulhaag/MiMangaNu

Table I
OPEN-SOURCE PROJECTS IN THE EXPERIMENTS

Project	# Commits	# Optimizations	Ratio
KISS	1,928	48	2.5%
MiMangNu	1,395	35	2.5%
Total	3,323	83	2.5%

```
   void onFavoriteChange() {
   ...
     for (...) {
   ...
-        ImageView image = (ImageView) mainActivity.
     favorites.findViewById(favoritesIds[i]);
+        ImageView image = (ImageView) favoritesViews[i];
   ...
   ...
     }
     for (...) {
-        mainActivity.favorites.findViewById(favoritesIds[
     i]).setVisibility(View.GONE);
+        favoritesViews[i].setVisibility(View.GONE);
     }
 }
```

Figure 4. Caching resources

Patten 1: Caching resources to avoid multiple accessing. GUI elements are frequently accessed to display in a screen during GUI rendering. Unlimited accessing could degrade GUI responsiveness. As shown in Figure 4, in method onFavoriteChange(), mainActivity.favorites.findViewById() is called in two loops without any update on the target view. The optimized code preloads the view list into an array *favoritesViews* and traverses it directly in loops. Thus, the times of findViewById(), which is a performance-aware API, will be reduced a lot.

Patten 2: Reducing redundant computation. Some heavy components (e.g., DB) may be attached in an app for ease of use. However, multiple loading of these components can drastically degrade app performance. In the example given by Figure 5, the optimization adds a boolean judgment in method setListManga() to avoid repeatedly calling method getMangasCondition(). The optimized code will fetch the data from the database only if it is not attached to the app.

Patten 3: Asynchronizing UI events to increase responsiveness. Heavy tasks running in the main thread could reduce the responsiveness of one app. In Figure 6, one listener is registered to one EditText, and will accordingly execute a search task (method updateRecords()) once changed. The searching is time-consuming, so that it may produce GUI lagging. In such a case, the optimized code moves this tasks to a background thread, and make the EditText more responsive to users.

```
   public void setListManga(boolean force) {
+      if(attached) {
   ...
               mangaList = Database.getMangasCondition
                     (...);
   ...
   public static ArrayList<Manga> getMangasCondition(...)
          {
   ...
        Cursor cursor = getDatabase(c).query(...);
   ...
   }
```

Figure 5. Reducing redundant computation

```
   public void updateRecords(String query) {
   ...
+      Thread resultThread = new Thread(new Runnable() {
+          @Override
+          public void run() {
           ...
-              ArrayList<Record> records = dataHandler.
     getRecords(...);
+              final ArrayList<Record> records = dataHandler.
     getRecords(...);
               ...
           }
   });
   ....
 }
```

Figure 6. Thread optimization for UI actions

```
   @Override
   public View onCreateView(...) {
   ...
-      new Thread(new Runnable() {
-          @Override
-          public void run() {
   ...
-              getActivity().runOnUiThread(new Runnable() {
-                  @Override
-                  public void run() {
   ...
                       dayScheduleAdapter = new
                           DayScheduleAdapter(...);
   ...
```

Figure 7. Synchronizing UI events.

Patten 4: Synchronizing UI events to avoid repeatedly triggering. Generally, using a background thread to execute tasks instead of the main thread can improve app performance. However, if GUI events can induce many background operations that occupy enormous resources, it is better to make these operations executed in the main thread. In Figure 7, the view was refreshed by a UiThread running in a child thread in method onCreateView(). As shown in the code example, once the fragment is refreshed, the main thread forks an asynchronous thread to update its data model. If the user keeps refreshing the fragment repeatedly, the main thread will forks as many threads as requests. As a result, it will drastically increase the computation resource with continuously refreshing. The optimized code relocates the updating of data model into the main thread. As such, the GUI will not accept any user requests unless the current task is not fully finished.

Patten 5: Reducing thread creation. Thread creation oc-

```
  public synchronized void showOnLoad(final Segment
      segment) {
+     if (segment.isVisible()) {
          mHandler.post(new Runnable() {
              @Override
              public void run() {
-                 if (Page.this.isVisible()) {
  ...
```

Figure 8. Reducing thread creation

cupies lots of computing resources, and thereby degrades app performance. Therefore, it is recommended to limit the frequency of such kind of operations. In Reader.java of project MiMangaNu, method showOnLoad() as shown in Figure 8 creates a new thread to refresh the animation page. However, in the old version, the condition, which is used to avoid redundant rendering, is located in the run() method of new thread. In other word, it will be reached after running this thread. So that change in new version moves the if condition outside the run() function, will surely avoid creating nonsense threads.

VI. Discussion

Besides the changes that are influential to the usage of PAAs, there exist a number of optimization practices beyond one method. More specifically, developers may optimize the code of higher level, e.g., architecture. In such cases, our approach cannot obtain them effectively. One viable solution is to expand the analysis scope by considering higher level, and then identify the usage changes.

In addition, the impact analysis suffers from some other factors which are not currently covered in the paper. The changes may affect the usage of APIs from the view of data dependency. This could be mitigated by computing the data dependency between branch conditions and PAAs.

VII. Conclusion and Future Work

In this paper, we propose an approach PerfOptimizer to learn performance-oriented optimization practices from code changes. And we evaluated on two projects and summarized several optimization patterns by manual analysis to support the effectiveness and usefulness of PerfOptimizer.

Due to PerfOptimizer's limitations, its scalability still need to be evaluated. Moreover, we are considering to incorporate data flow analysis to increase the accuracy. Last, we attempt to take into account the code hierarchy and find out more performance practices of higher level.

References

[1] S. Chen, T. Su, L. Fan, G. Meng, M. Xue, Y. Liu, and L. Xu. Are mobile banking apps secure? what can be improved? In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 797–802, 2018.

[2] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, and G. Pu. Efficiently manifesting asynchronous programming errors in android apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 486–497, 2018.

[3] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su. Large-scale analysis of framework-specific exceptions in android apps. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 408–419, 2018.

[4] M. Firtman. *High Performance Mobile Web: Best Practices for Optimizing Mobile Web Apps*. O'Reilly Media, 2016.

[5] G. Inc. Performance tips. https://developer.android.com/training/articles/perf-tips. Online; accessed 1 May 2018.

[6] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, Jan. 1979.

[7] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Mining energy-greedy api usage patterns in android apps: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 2–11, New York, NY, USA, 2014. ACM.

[8] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1013–1024, New York, NY, USA, 2014. ACM.

[9] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM.

[10] N. Smith, D. van Bruggen, and F. Tomassetti. JavaParser: For processing Java code. https://javaparser.org/. Online; accessed 4 April 2018.

[11] T. Su. Fsmdroid: guided GUI testing of android apps. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, pages 689–691, 2016.

[12] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su. Guided, stochastic model-based GUI testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 245–256, 2017.

[13] M. Ward. Proving Program Refinements and Transformations. *DDPhil Thesis, Oxford University*, 1989.