Automated API-Usage Update for Android Apps

Mattia Fazzini
Georgia Institute of Technology
Atlanta, GA, USA
mfazzini@cc.gatech.edu

Qi Xin
Georgia Institute of Technology
Atlanta, GA, USA
qxin6@gatech.edu

Alessandro Orso
Georgia Institute of Technology
Atlanta, GA, USA
orso@cc.gatech.edu

ABSTRACT

Mobile apps rely heavily on the application programming interface (API) provided by their underlying operating system (OS). Because OS and API can change frequently, developers must quickly update their apps to ensure that the apps behave as intended with new API and OS versions. To help developers with this tedious, error prone, and time consuming task, we developed a technique that can automatically perform app updates for API changes based on examples of how other developers evolved their apps for the same changes. Given a target app to be updated and information about the changes in the API, our technique performs four main steps. First, it analyzes the target app to identify code affected by API changes. Second, it searches existing code bases for examples of updates to the new version of the API. Third, it analyzes, ranks, and transforms into generic patches the update examples found in the previous step. Finally, it applies the generated patches to the target app in order of ranking, while performing differential testing to validate the update. We implemented our technique and performed an empirical evaluation on 15 real-world apps with promising results. Overall, our technique was able to update 85% of the API changes considered and automatically validate 68% of the updates performed.

CCS CONCEPTS

• **Software and its engineering** → **Software evolution**.

KEYWORDS

Mobile apps, automated update, API analysis

ACM Reference Format:
Mattia Fazzini, Qi Xin, and Alessandro Orso. 2019. Automated API-Usage Update for Android Apps. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3293882.3330571

1 INTRODUCTION

Mobile apps are increasingly widespread, and we use them daily for a range of activities. One common trait of these apps is that they rely heavily on the underlying operating system (OS), which provides a number of fundamental services to the apps through its application programming interface (API).

Unfortunately, OSs and their APIs get updated frequently [6, 19, 31, 35, 63]. Although OS developers try to maintain backward compatibility between API versions, API methods are routinely deprecated, eliminated, added, and changed as OSs evolve. For an app to keep working with the newest versions of its underlying OS, the usages of the API within the app must be suitably updated.

Unfortunately, evolving an app to adapt it to changes in the API is a tedious, error prone, and time consuming task, especially when the app must work both on the latest OS and on earlier versions of it (e.g., for fragmented ecosystems [22, 29]). Moreover, although API changes are usually discussed in the API documentation, in most cases this documentation does not provide relevant examples of how to update software that uses the changed API. Developers must therefore discover how to update their code on their own. Finally, these updates can be extensive, due to the typically widespread use of the API within apps.

The intuition behind this work is that evolving an app from an old to a new version of its underlying API can be automated to a large degree by leveraging existing code bases; that is, by finding and analyzing the code of existing apps that already went through the same update. Based on this intuition, we developed AppEvolve. AppEvolve takes as inputs (1) a target app A to be updated and (2) information about the changes in the API between two given versions V and $V + 1$. (In this work, we assume that the information about the API changes is manually provided by the developer.) Given these inputs, the technique operates in four main phases. The *API-usage analysis* phase analyzes A to identify code that is affected by the changes in the API and should therefore be modified. The *update examples search* phase searches existing code bases for examples of relevant updates (i.e., updates from version V to $V + 1$ that involve usages of the same API methods used in A). The *update examples analysis* phase analyzes the identified examples to generalize the changes therein, transform them into generic code patches, and rank the resulting patches. Finally, the *API-usage update* phase applies the generated patches to A one at a time, in order of ranking, and stops when a patch is successfully validated (using differential testing). When the technique terminates successfully, it produces an updated version of A as well as an API-usage update report, which developers can use to double check the update.

Although the problem of performing code updates based on examples has been studied before, most existing techniques (e.g., [8, 39, 52]) mainly target repetitive updates within the same code base. Moreover, these techniques typically assume that update examples are provided as inputs. AppEvolve, conversely, automatically searches for and applies examples across different code bases, while handling the challenges involved in this process. In particular, AppEvolve can handle examples that differ in the way the update was performed by distilling a common core of update operations and ranking the examples based on how closely they capture the essence of the update. In addition, AppEvolve can validate updates using differential testing, thus providing more confidence in the update.

To assess the usefulness of AppEvolve, we implemented it for Android apps and performed an empirical evaluation.[1] In our evaluation, we assessed the effectiveness and efficiency of our technique by applying it to 15 real-world apps and 20 real API changes, As far as effectiveness is concerned, AppEvolve was able to update a large majority (85%) of the API changes considered and automatically validate many (68%) of the updates performed. As for efficiency, the cost of running AppEvolve is dominated by the search for examples, which can however be run overnight. Overall, we believe that our results are promising and provide initial, but clear evidence of the usefulness of our technique.

The main contributions of this paper are:

- AppEvolve, a new technique for automatically updating an app when its underlying API evolves, based on distilling existing examples of analogous updates of other apps.
- A tool that implements AppEvolve and that is publicly available, together with our experiment data and infrastructure (https://sites.google.com/view/appevolve).
- An empirical evaluation of AppEvolve, performed on 15 real-world apps, that shows the effectiveness, efficiency, and overall usefulness of our technique.

2 TERMINOLOGY & MOTIVATING EXAMPLE

Consider two API versions: *old API* = $[m_1, ..., m_k]$ and *new API* = $[m'_1, ..., m'_l]$. We define an *API usage* as any call of one or more methods of either the old or the new API versions. Intuitively, an API-usage change (*AU change*) between the old and the new versions of the API can typically be described as a mapping (*AU change mapping*) between one or more methods in the old version and one or more methods in the new version: $[m_1, ..., m_p] \rightarrow [m'_1, ..., m'_q]$.

To illustrate, consider the change in the Android API version 23 described at [15]: method `getAllNetworkInfo()`, which returns connection information for all network types supported by a device, was deprecated in version 23 of the Android API in favor of the two new methods `getAllNetworks()` and `getNetworkInfo(Network)`. The corresponding AU change mapping would be from method `getAllNetworkInfo()` to methods `getAllNetworks()` and `getNetworkInfo(Network)`. Given an AU change , we define an *old API usage* (resp., *new API usage*) for that AU change as a sequence of one or more method invocations that match the left-hand side (resp., right hand side) of the AU change mapping. For the AU change we just considered, an old API usage would be an invocation of `getAllNetworkInfo()`, whereas a new API usage would be an invocation of `getAllNetworks()` followed by an invocation of `getNetworkInfo(Network)`. Considering an app A and an AU change, we use the term *update* to indicate the operation of updating an old API usage in A to a new API usage, and call the updated app A'. Finally, we use the term *update example* to refer to existing updates (e.g., in a code base).

[1]Although we defined and implemented AppEvolve for the Android ecosystem, due to its popularity and the many apps available together with their source code, our technique can be generalized to other mobile platforms and to evolving APIs in general.

To motivate our work (and help illustrate our technique in later sections), Figures 2–5 present two different updates for the AU change described above. Both updates, which we indicate as *updateV1* and *updateV2*, are derived from actual updates of real apps. In the figures, lines starting with - indicate code removed by the update while + indicate added code.

Figure 2 shows the code before updateV1. The code invokes `getAllNetworkInfo` at line 4 and iterates over the returned array of `NetworkInfo` objects at lines 6-9 to check whether the device is connected to a network. In the version of the code after updateV1, shown is in Figure 3, the developer introduced a check at line 4 to determine the version of the platform on which the app is running. If the app is running on the old version of the platform (lines 13-18), the code checks whether the device is connected to a network similarly to how it was performed in the old version of the code. Conversely, if the app is running on the new version of the platform (lines 5-11), the code invokes `getAllNetworks` (from the new API) at line 5 and iterates over the array of `Network` objects to determine whether the device is connected to a network (lines 7-11). To determine the connection status, the developer retrieves network information from a `Network` object invoking `getNetworkInfo`. This update example is characterized by two fragments of code (lines 13-18 and 5-11), each executing on different versions of the platform (and having access to different APIs). As pointed out in related work [22, 29], this coding practice is frequent in Android apps to account for issues generated by the fragmentation of the ecosystem [22, 29, 47]. This example also shows that updating API usages might involve using new methods, handling new parameters, and manipulating new return values.

As it is possible to observe from Figures 4 and 5, updateV2 is similar to updateV2, in that both updates (1) introduce a check to determine the version of the platform on which the app is running and (2) invoke `getAllNetworks` and `getNetworkInfo` from the new API in one branch of the condition and `getAllNetworkInfo` in the other. However, the two updates also have some differences. UpdateV1 checks the connection status of a `NetworkInfo` object by comparing the result obtained by invoking `getState` against `CONNECTED` (line 7 in Figure 2 and lines 9 and 16 in Figure 3), whereas updateV2 uses `isConnected` instead. Moreover, updatev2 also logs information about the type of network being connected (lines 10 and 18 in Figure 5) and makes changes that are not related to the API usage (line 11 in Figure 4 and line 22 in Figure 5).

This motivating example highlights the potential of using update examples to update API usages, while also presenting some of the challenges involved in doing so automatically. In particular, the example shows that different updates can have commonalities (e.g., the check on the version of the API) but also differences (e.g., the use of `getState` and `CONNECTED` versus the use of `isConnected` or the presence of additional statements unrelated to the update).

3 TECHNIQUE

In this section, we present AppEvolve, our technique for automatically performing API-usage updates based on update examples. The technique targets updates related to changes in the API of the Android platform. The basic idea behind AppEvolve is to update

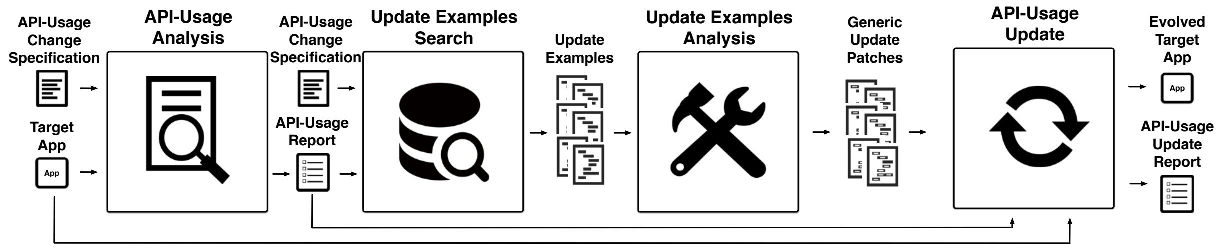

Figure 1: High-level overview of AppEvolve.

```
  public boolean isNetworkAvailable(Context ctx) {
    ConnectivityManager ctv = (ConnectivityManager)
      ctx.getSystemService(Context.CONNECTIVITY_SERVICE);
-   NetworkInfo[] info = ctv.getAllNetworkInfo();
-   if (info != null) {
-     for (int i = 0; i < info.length; i++) {
-       if (info[i].getState() == NetworkInfo.State.CONNECTED) {
          return true;
        } }
    }
    return false;
  }
```

Figure 2: API-usage example before updateV1.

```
  public boolean isNetworkAvailable(Context ctx) {
    ConnectivityManager ctv = (ConnectivityManager)
      ctx.getSystemService(Context.CONNECTIVITY_SERVICE);
+   if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
+     Network[] networks = ctv.getAllNetworks();
+     NetworkInfo networkInfo;
+     for (Network mNetwork : networks) {
+       networkInfo = ctv.getNetworkInfo(mNetwork);
+       if (networkInfo.getState().equals(NetworkInfo.State.CONNECTED)) {
          return true;
        }  }
+   } else {
+     NetworkInfo[] info = ctv.getAllNetworkInfo();
+     if (info != null) {
+       for (NetworkInfo anInfo : info) {
+         if (anInfo.getState() == NetworkInfo.State.CONNECTED) {
+           return true;
+         }  }  }
    }
    return false;
  }
```

Figure 3: API-usage example after updateV1.

```
  public boolean isConnected(Context cont) {
    ConnectivityManager conn = (ConnectivityManager)
      cont.getSystemService(Context.CONNECTIVITY_SERVICE);
-   NetworkInfo[] info = conn.getAllNetworkInfo();
-   if (info != null) {
-     for (int i = 0; i < info.length; i++) {
-         if(info[i].isConnected()) {
            return true;
          }  }
    }
-   Toast.makeText(cont, R.s.noNet, Toast.L_S).show();
    return false;
  }
```

Figure 4: API-usage example before updateV2.

```
  public boolean isConnected(Context cont) {
    ConnectivityManager conn = (ConnectivityManager)
      cont.getSystemService(Context.CONNECTIVITY_SERVICE);
+   if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
+     Network[] networks = conn.getAllNetworks();
+     NetworkInfo networkInfo;
+     for (Network mNetwork : networks) {
+       networkInfo = conn.getNetworkInfo(mNetwork);
+       if(networkInfo.isConnected()))) {
+         Log.d("Net","NAME:"+networkInfo.getTypeName());
          return true;
        }  }
+   } else {
+     NetworkInfo[] info = conn.getAllNetworkInfo();
+     if (info != null) {
+       for (NetworkInfo anInfo : info) {
+         if(anInfo.isConnected()) {
+           Log.d("Net","NAME:"+anInfo.getTypeName());
+           return true;
+         }  }  }
    }
+   Toast.makeText(cont, cont.getString(R.s.noNet), Toast.L_S).show();
    return false;
  }
```

Figure 5: API-usage example after updateV2.

API usages in a target (Android) app by leveraging how developers of other apps updated the same API usage in their apps.

Figure 1 provides an overview of AppEvolve and shows its four main phases. Given a target app and a specification of the AU changes as inputs, in its *API-usage analysis* phase, the technique (1) analyzes the source code of the app to identify API usages that should be changed and (2) stores this information in the *API-usage report*. The *update examples search* phase uses this information together with the AU changes specification to look for API-usage updates in existing code bases. The *update examples analysis* phase processes the examples identified to generalize them, compute their common core, and rank them based on the proximity to the common core. In this process, the examples are transformed into *generic update patches*. Finally, the *API-usage update* phase leverages the generic update patches to change the API-usage locations reported by AppEvolve's first phase and validates them using differential testing. The final outputs of the technique are an evolved target app and an API-usage update report documenting the changes in the app. We now discuss the different phases in detail.

3.1 API-Usage Analysis

This phase takes as inputs the source code of a target app (*TA*) and a set of AU changes (AUC_1, ..., AUC_n) and creates an API usage report that contains the location in *TA* of old API usages that should be updated. Each AU change consists in a mapping between methods in the old and new versions of the API, as described in Section 2.

AppEvolve identifies old API usages that should be updated by checking if they can execute while *TA* is running on the new version of the the API. (When an app can run on multiple API versions, some of its code may be programmatically prevented from running on the some versions of the API [22, 29].) The technique computes this information by statically analyzing *TA*. First, AppEvolve computes the set of API versions (S_{API}) on which the statements in *TA* can execute by leveraging an inter-procedural data-flow analysis defined in related work [22]. Second, the technique performs an intra-procedural analysis to identify old API usages and check whether method calls therein can execute on the new version of the API (based on S_{API}). If this is the case, such API usages require update, so AppEvolve stores them in the API-usage report together with the location of the corresponding API calls.

Algorithm 1: Search for update examples.

```
   Input  : $AU$: API usage in old version of API
            $AU'$: API usage in new version of API
            $chi$: code hosting infrastructure based on version-control system
   Output : $ues_{AU\ change}$: update examples for $AU\ change = AU \rightarrow AU'$
1  begin
2      $ues_{AU\ change} = \emptyset$
3      $kws$ = Compute-Keywords($AU'$)
4      $files$ = Find-Files($kws$, $chi$.Get-Index())
5      foreach $f \in files$ do
6          $cb = f$.Get-Code-Base()
7          $versions_f = cb$.Get-Versions($f$)
8          foreach $v_f \in versions_f$ do
9              $v_n = v_f$
10             $v_o$ = Find-Previous-Version($versions_f$, $v_n$)
11             if $v_o$ == null then
12                 continue
13             $diff$ = Compute-Differences($v_o$, $v_n$)
14             $lines_r = diff$.Get-Removed()
15             $lines_a = diff$.Get-Added()
16             if $\neg(v_o.lines$.Uses($AU'$)) $\wedge lines_r$.Uses($AU$))
17              $\wedge$ ($lines_a$.Uses-With-Check($AU$, $AU'$))
18              $\wedge$ Same-Containing-Method($v_o$, $AU$, $v_n$, $AU'$) then
19                 $sig_o = v_o$.Get-Containing-Method-Signature($AU$)
20                 $sig_n = v_n$.Get-Containing-Method-Signature($AU'$)
21                 $ue_{AU\ change}$ = Up-Ex($cb$, $v_o$, $sig_o$, $v_n$, $sig_n$, $AU$, $AU'$)
22                 $ues_{AU\ change}$.Add($ue_{AU\ change}$)
23     return $ues_{AU\ change}$
```

3.2 Update Examples Search

This phase identifies update examples for each old API usage reported in the previous phase by looking at how other developers updated the corresponding API usages in their apps. To do so, the technique analyzes the version control history of other apps. The final output of this phase is a set of update examples for each old API usage that requires update in the target app.

Algorithm 1 describes how AppEvolve automatically identifies update examples. The algorithm takes as inputs an old API usage (AU), the corresponding new API usage (AU'), and the location of a code hosting infrastructure (chi) where code bases for other apps are publicly accessible. The output of the algorithm is a set of update examples (ues_{AU}) for AU.

The algorithm starts with an empty set of examples (line 2) and identifies code bases that could contain examples by performing a textual search in the files in chi that use AU' (lines 3-4). The search (Find-Files) is based on a set of keywords extracted from AU' and on an index built on the content of the files in chi. This step enables AppEvolve to efficiently consider large code bases and quickly discard irrelevant code (based on the intuition that if a developer performed an update, the update should be present in the latest version of a file). The set of keywords (kws) used in the search contains (1) the name, (2) parameter types, and (3) the declaring class for each method in AU'. For our motivating example, kws = {`getAllNetworks`, `ConnectivityManager`, `getNetworkInfo`, `Network` }.

At this point, the algorithm processes each file (f) resulting from the search to identify those whose history contains an update of AU to AU' (lines 5-22). To do so, the algorithm compares each version (v_n) of f with its previous one (v_o), starting from the most recent version. Given these two versions, the algorithm computes their differences [41] and extracts removed lines ($lines_r$) from v_o and added lines ($lines_a$) in v_n.

The core part of the algorithm checks whether an update of AU to AU' is present between v_o and v_n (lines 16-22). Specifically, the algorithm checks that v_o contains no use of AU' and searches for (1) a method in v_o whose removed lines are using AU and (2) a corresponding method in v_n that added new lines using AU and AU' (in different branches of a condition that checks the API version). The algorithm searches for methods in v_n also using AU so as to find examples that perform backward compatible updates, which is common practice when updating Android apps [22]. If the search is successful, the algorithm adds the update between the method in v_o (sig_o) and v_n (sig_n) to the list of update example for AU (line 21). For our motivating example, both updateV1 and updateV2 would be considered valid update examples, as (1) they do not use AU' ([`getAllNetworks`, `getNetworkInfo`]) in their old version, (2) removed lines in their old version contain AU ([`getAllNetworkInfo`]), and (3) added lines in their new version contain AU and AU' in different branches of an API-version check.

The algorithm stores the following information for each update example it finds: (1) the code base (cb) and its version history information, (2) the version of f before the update (v_o), (3) the method signature in v_o that contains AU (sig_o), (4) the version of f after the update (v_n), (5) the method signature in v_n that contains AU and AU' (sig_n), and (6) the API usages associated with this update (AU and AU'). The algorithm terminates by returning the list of update examples ($ues_{AU\ change}$) found.

3.3 Update Examples Analysis

This phase takes as inputs the update examples identified in the previous phase and translates them into *generic update patches* that can be applied to the target app. The algorithm also orders the patches based on how closely related they are to the common core of the update that is shared across examples, so as to prioritize patches that best capture the essence of the update. Algorithm 2 illustrates how AppEvolve performs these two tasks.

3.3.1 *Generic Update Patch Generation.*

The algorithm generalizes update examples related to an AU change ($ues_{AU\ change}$) at lines 3-18. It first creates an empty list of generic update patches ($gups_{AU\ change}$) and then processes each example ($ue_{AU\ change}$). AppEvolve starts the task of generalizing $ue_{AU\ change}$ by identifying a list of edit operations that transforms the method in the old version of the code in $ue_{AU\ change}$ (sig_o) to the method in the new version of the code (sig_n). The algorithm encodes these edits ($edits$) as tree operations performed on the abstract syntax trees (ASTs) built for the methods (ast_o and ast_n). Specifically, the algorithm computes these edits based on an existing technique that generates an ordered list of tree operations for transforming an AST into another AST by operating on the nodes of the two ASTs [10, 38]. The technique consider four types of edit operations: (1) $\underline{\text{I}}$nsert(sn_1, sn_2, i), which adds statement node sn_1 as the i^{th} child of statement node sn_2; (2) $\underline{\text{M}}$ove(sn_1, sn_2, i), which moves statement node sn_1 from its current position and adds it as the i^{th} child of statement node sn_2; (3) $\underline{\text{U}}$pdate(sn_1, sn_2), which replaces statement node sn_1 with statement node sn_2; and (4) $\underline{\text{D}}$elete(sn_1), which deletes statement node sn_1. For each type of edit, we refer to sn_1 as the statement node *affected* by the edit and denote the corresponding statement with as. Figures 6 and 7 show a summarized version (for space limitations) of the edits for updateV1 and updateV2 in our motivating example.

At this point, the algorithm identifies edits that are related to AU and AU' ($redits$) using an intra-procedural dependency analysis

Algorithm 2: Analysis of update examples.

```
   Input  : $ues_{AU\ change}$: update examples for $AU\ change = AU \rightarrow AU'$
   Output : $gups_{AU\ change}$:: ordered list of generic update patches
1  begin
2      //create generic update patches
3      $gups_{AU\ change}$ = []
4      foreach $ue_{AU\ change} \in ues_{AU\ change}$ do
5          $ast_o$ = Build-AST($ue_{AU\ change}.sig_o$)
6          $ast_n$ = Build-AST($ue_{AU\ change}.sig_n$)
7          $edits$ = Compute-Edits($ast_o$, $ast_n$)
8          $redits$ = Find-Related-Edits($edits$, $ast_o$, $ue_{AU\ change}.AU$, $ast_n$,
             $ue_{AU\ change}.AU'$)
9          $gedits$ = []
10         foreach $redit \in redits$ do
11             $pstn$, $pstn_2$= Compute-Placement($redit$, $redits$)
12             $abstr$ = Compute-Abstraction($redit$)
13             $gedit$ = Generic-Edit($redit$, $pstn$, $pstn_2$, $abstr$)
14             $gedits$.add($gedit$)
15         $cvars$ = Find-Context-Variables($ast_n$, $gedits$)
16         Annotate-Context-Variables($cvars$, $ast_o$, $AU$)
17         $gup_{AU\ change}$ = Generic-Update-Patch($gedits$, $cvars$)
18         $gups_{AU\ change}$.Add($gup_{AU\ change}$)
19     //sort generic update patches based on proximity to core of update
20     $caseq$ = Compute-Core($gups_{AU\ change}$)
21     foreach $gup_{AU\ change} \in gups_{AU\ change}$ do
22         $cprox$ = Compute-Core-Proximity($gup_{AU\ change}$, $caseq$)
23         $gup_{AU\ change}$.Set-Core-Proximity($cprox$)
24     $gups_{AU\ change}$ = Order-By-Core-Proximity($gups_{AU\ change}$)
25     return $gups_{AU\ change}$
```

(Find-Related-Edits at line 8). Specifically, it uses backward and forward dependency analysis to select the edits that are influencing or are influenced by the values involved in method calls from AU and AU'. This analysis uses both ast_o and ast_n to compute dependencies, as certain edits might operate only on statements in one of the two trees (e.g., `D for int i=0 i<info.length i++` from Figures 6 and 7). The algorithm considers all the edits not in *redits* at the end of the analysis as unrelated to the update and discards them. In our motivating example, AppEvolve determines that all edits in Figure 6 and all edits except for `U Toast.makeText...` in Figure 7 are related to the update.

The edits identified so far are dependent on the specific example from which they were computed. At this point, the algorithm generalizes them so that they can be used to update AU in the methods of the target app. To do so, the algorithm iterates over *redits* (lines 10-14) and translates each edit into a generic edit (*gedit*). A generic edit consists of three elements, which we describe below: (1) the original edit (*redit*), (2) the position (*pstn*) of *as*, expressed in terms of an ancestor (*ancst*) and a predecessor (*pred*) statements, and (3) an abstraction (*abstr*) of *as*.

ancst and *pred* of *as* are computed by analyzing *predits*, the set of statements that are affected by edits in *redits* and occur before the edit under analysis. *ancst* is computed by analyzing parent relations in the AST and corresponds to the first statement in the relation chain starting at *as* that is affected by an edit in *predits*. *pred* is the latest statement appearing before *as* in a postorder traversal of the AST that (1) is not present in the subtree rooted at *as* and (2) is affected by and edit in *predits*. In the case of Move edits, the technique also computes $pstn_2$, which contains *ancst* and *pred* for the new position of *as*. Computing *ancst* and *pred* is necessary because the original position of a statement affected by an edit is based on the AST on which it was computed and will not be meaningful in different ASTs. Consider, as an example, the edits in Figure 6. The *ancst* and *pred* for edit `I NetworkInfo networkInfo` would be edits `I if Build.VERSION.SDK_INT>= Build.VERSION_CODES.M` and `I Network[] networks = ctv.getAllNetworks()`, respectively. *abstr* is a representation of a statement in which variables are replaced by their types. For instance the abstraction for statement `if info!=null`, where the type of info is `NetworkInfo`, would be `if NetworkInfo!=null`. AppEvolve uses this abstraction in a later stage of the algorithm to compute commonalities of the update across examples.

At this point (line 15), the algorithm has extracted an ordered list of generic edits from an update example. It then proceeds by identifying the variables (and their types) that are used by statements in ast_n that are affected by the edits but are not defined in any of these statements. We refer to such variables as *context variables* (*cvars*), as they provide the context for the update. For the motivating example, the edits associated with updateV1 have only one context variable: `ctv`, of type `ConnectivityManager`. The algorithm further processes the context variables (Annotate-Context-Variables) to identify those directly used by method invocations in AU. For example, variable `ctv` in updateV1 would be identified because it is used by method `getAllNetworkInfo`. AppEvolve uses this information when applying a patch to the target app.

Context variables and generic edits define the content of a generic update patch ($gup_{AU\ change}$), which the algorithm adds to the list of patches ($gup_{AU\ change}$).

3.3.2 *Generic Update Patch Prioritization.*

After the algorithm has translated all update examples into generic update patches, it orders them based on how related they are to their common core—the parts of the update shared across all patches (lines 20-24). We define the common *core* in terms of the longest subsequence of edits that is shared across all patches, which the algorithm computes by solving a multiple longest common subsequence problem (MLCS) [18]. Because patches from different examples might operate on variables with different names, our technique computes an abstraction of the edit in the patches before encoding them into the MLCS problem. The abstraction for an edit consists in a string composed of (1) the type of the edit followed by (2) the abstraction *abstr* of the statement involved in the edit. Figures 8 and 9 show the abstract edits computed for the patches from updateV1 and updateV2 (there types are replaced by `$T` due to space limitations). Figure 10 shows the longest subsequence shared across the two patches.

After computing the core as we just described (Compute-Core at line 20), the algorithm computes the core proximity of each patch p, which indicates how closely related is p to the core and is computed by dividing the number of edits in the core by the number of edits in p. It then order the patches accordingly (Order-By-Core-Proximity). For our motivating example, the core proximity values associated with the patches computed for updateV1 and updateV2 are 0.77 and 0.67, which shows that the patch for updateV1 is closer to the core (due to the fact that this patch does not contain the additional logging statement `Log.d`). However, the patch for updeteV1 does not have maximum value (1) because the two patches use different statements (`getState` and `CONNECTED` vs. `isConnected`), which affects the computation of the core. Note that we decide to rank patches instead of merging them to avoid disregarding statements that are necessary to perform the update. The ordered list of patches $gups_{AU\ change}$ is the final output of the algorithm.

```
I if Build.VERSION.SDK_INT>=Build.VERSION_CODES.M
I Network[] networks=ctv.getAllNetworks()
I NetworkInfo networkInfo
I for Network mNetwork:networks
M NetworkInfo[] info=ctv.getAllNetworkInfo()
M if info!=null
I networkInfo=ctv.getNetworkInfo(mNetwork)
U if info[i].getState()==CONNECTED
  if networkInfo.getState().equals(CONNECTED)
M if networkInfo.getState().equals(CONNECTED)
I for NetworkInfo anInfo:info
I if anInfo.getState()==CONNECTED
I return true
D for int i=0 i<info.length i++
```

Figure 6: Edits for updateV1 in our motivating example.

```
I if Build.VERSION.SDK_INT>=Build.VERSION_CODES.M
U Toast.makeText(cont,noNet,L_S).show()
  Toast.makeText(cont,cont.getString(noNet),L_S).show()
I Network[] networks=conn.getAllNetworks()
I NetworkInfo networkInfo
I for Network mNetwork:networks
M NetworkInfo[] info=conn.getAllNetworkInfo()
M if info!=null
I networkInfo=conn.getNetworkInfo(mNetwork)
U if info[i].isConnected()
  if networkInfo.isConnected()
M if networkInfo.isConnected()
I Log.d("Net","NAME:"+networkInfo.getTypeName())
I for NetworkInfo anInfo:info
I if anInfo.isConnected()
I Log.d("Net","NAME:"+anInfo.getTypeName())
I return true
D for int i=0 i<info.length i++
```

Figure 7: Edits for updateV2 in our motivating example.

```
I if SDK_INT>=M
I Network[] $T=$T.getAllNetworks()
I NetworkInfo $T
I for Network $T:$T
M NetworkInfo[] $T=$T.getAllNetworkInfo
M if $T!=null
I $T=$T.getNetworkInfo($T)
U if $T[$T].getState()==CONNECTED
M if $T.getState().equals(CONNECTED)
I for NetworkInfo $T:$T
I if $T.getState()==CONNECTED
I return true
D for int $T=0 $T<$T $T++
```

Figure 8: Edit abstractions for updateV1 in our motivating example.

```
I if SDK_INT>=M
I Network[] $T=$T.getAllNetworks()
I NetworkInfo $T
I for Network $T:$T
M NetworkInfo[] $T=$T.getAllNetworkInfo
M if $T!=null
I $T=$T.getNetworkInfo($T)
U if $T[$T].isConnected()
M if $T.isConnected()
I Log.d("Net","NAME:"+$T.getTypeName())
I for NetworkInfo $T:$T
I if $T.isConnected()
I Log.d("Net","NAME:"+$T.getTypeName())
I return true
D for int $T=0 $T<$T $T++
```

Figure 9: Edit abstractions for updateV2 in our motivating example.

```
I if SDK_INT>=M
I Network[] $T=$T.getAllNetworks()
I NetworkInfo $T
I for Network $T:$T
M NetworkInfo[] $T=$T.getAllNetworkInfo
M if $T!=null
I $T=$T.getNetworkInfo($T)
I for NetworkInfo $T:$T
I return true
D for int $T=0 $T<$T $T++
```

Figure 10: Common edit abstraction subsequence between updateV1 and updateV2.

3.4 API-Usage Update

The last phase of our technique applies generic update patches to old API usages in the target app and validates them through differential testing. The technique updates old API usages detailed in the API-usage report one at a time to ensure that, during the validation process, updates for different API usages do not interfere with one other. Algorithm 3 describes this phase. The algorithm takes as inputs (1) the target app (*TA*), (2) the API usage that should be updated (*AU*), (3) the location of *AU* in *TA* (loc_{AU}), (4) the generic update patches ($gups_{AU\ change}$) for the AU change at hand, and (5) a test suite (*TS*) for validating the update target app. It produces as outputs an updated target app *TA′* and an API-usage update report ($ur_{AU\ change}$), in which changes (for the specific occurrence of *AU*) are documented for the developer of *TA*.

After creating an empty $ur_{AU\ change}$, the algorithm processes each patch starting from the one at the top of the list—the one closest to the core of the update (lines 3-20). Given a patch, the algorithm first determines whether the patch ($gup_{AU\ change}$) is applicable to *AU* at loc_{AU} in *TA*. Applicability of a patch is determined by two factors. First, the context variables associated with the patch must be in scope in the program point in which the patch is supposed to be applied. Second, it must be possible to successfully apply the generic edits (in their entirety) to the method in *TA* using *AU*.

When determining the applicability of a generic update patch $gup_{AU\ change}$, the algorithm first builds the AST (ast_t) of the method containing *AU* based on loc_{AU}. It then iterates over the statement nodes (*stmt*) in ast_t to find the point in which to apply $gup_{AU\ change}$ (lines 6-20). It does so by (1) computing *svars*, the set of variables in scope at *stmt* (Find-Variables-In-Scope) and (2) identifying whether there is a mapping between variables in *svars* and context variables ($gup_{AU\ change}.cvars$). The algorithm tries to find such mapping by translating this task into an instance of the assignment problem [43] and considering as mappings all solutions of minimum cost. Specifically, AppEvolve uses the algorithm by Murty [44] by assigning a zero cost to the mapping of a context variable to a scope variable of the same type and a cost of 1 to all other mappings. In this way, all solutions with cost zero can be considered plausible mappings between $gup_{AU\ change}.cvars$ and *svars*. AppEvolve adds additional constraints to the assignment problem by forcing the mapping between (1) the variables in $gup_{AU\ change}.cvars$ that are marked as being used by method invocations from *AU* and (2) the corresponding variables in method invocations from *AU* in *TA* that are part of *svars*. In this way, AppEvolve can limit the number of mappings when the number of compatible variables in scope is high. If no mapping is possible, the algorithm analyzes the variables in scope at the next location in the AST.

After finding a successful mapping (*mpng*), the algorithm applies the patch to ast_t by first replacing its context variables with variables in scope according to *mpng*. It then performs the edits in the patch $gup_{AU\ change}.gedits$ starting at the location in ast_t identified by *stmt* (line 10). For each edit type, the algorithm identifies statements *pstmts* in ast_t to which the edit can be applied by performing a preorder traversal of ast_t starting at *stmt*. If an edit operation adds a statement at the beginning of *pstmts*, *stmt* is updated to be the new initial statement. The algorithm operates differently based on the edit type. For Insert edits, it adds the affected statement *as* at the earliest point across *pstmts*, based on position *pstn* (see Section 3.3). For Move edits, it uses *abstr* to find a matching statement *s* in *pstmts*, starting at *pstn*, and moves *s* to its new position $pstn_2$. For Update edits, it uses *abstr* to find a matching statement *s* in *pstmts*, starting at *pstn*, and updates *s* with the new statement from

Algorithm 3: API-usage update.

```
   Input  : $TA$: target app
            $AU$: old API-usage that requires update
            $loc_{AU}$: location of the old API-usage in $TA$
            $gups_{AU\ change}$: generic update patches for $AU\ change = AU \rightarrow AU'$
            $TS$: test suite to validate the updated target app
   Output : $TA'$: target app with performed AU change $AU\ change$
            $ur_{AU\ change}$: API-usage update report for $AU\ change$
1  begin
2      $ur_{AU\ change}$ = []
3      foreach $gup_{AU\ change} \in gups_{AU\ change}$ do
4          $sig_t$ = $TA$.Get-Containing-Method-Signature($loc_{AU}$)
5          $ast_t$ = Build-AST($sig_t$)
6          foreach $stmt \in ast_t$.Preorder() do
7              $svars$ = Find-Variables-In-Scope($ast_t$, $stmt$)
8              $mpngs$ = Compute-Mappings($svars$, $gup_{AU\ change}.cvars$, $ast_t$, $AU$)
9              foreach $mpng \in mpngs$ do
10                 $ast'_t$ = Apply-Update-Patch($ast_t$, $stmt$, $mpng$, $gup_{AU\ change}$)
11                 if $ast'_t$ != null then
12                     $TA'$ = $TA$.Replace(Code($ast_t$), Code($ast'_t$))
13                     if $TS$ != null then
14                         if Validate($TA'$, $TS$) then
15                             $ur_{AU\ change}$ .Add(Val-Update(Code($ast_t$), Code($ast'_t$)))
16                             return $TA'$, $ur_{AU\ change}$
17                         else
18                             continue
19                     else
20                         $ur_{AU\ change}$ .Add(Appl-Update(Code($ast_t$), Code($ast'_t$)))
21     $TA'$ = $TA$
22     return $TA'$, $ur_{AU\ change}$
```

the edit. Finally, for Delete edits, it uses *abstr* to find a matching statement *s* in *pstmts*, starting at *pstn*, and deletes *s*.

If any of the edit operations fails, the algorithm returns a `null` value to indicate that the update was unsuccessful. Conversely, if all edit operations succeed, function Apply-Update-Patch returns a new AST (ast'_t) that encodes the update of *AU*. At this point, the algorithm considers the patch as applicable to *TA* and updates the code of ast_t with the code of ast'_t in *TA*, thus generating the updated app *TA′*. It then validates *TA′* using differential testing (lines 13-20).

If a test suite for *TA* is already available and exercises *AU* without failures, AppEvolve uses this test suite (passing it as parameter *TS* to Algorithm 3). Otherwise, if a test suite is not available, AppEvolve tries to create one using random input generation while running *TA* on the old version of the API. (Note that any input generation strategy could be used.) If the created test suite exercises *AU* without failures, AppEvolve uses it as *TS*. Otherwise, AppEvolve passes a `null` *TS* to the algorithm, which would not validate the update (line 20) and document the code changes in $ur_{AU\ change}$, marking them as applicable but not validated.

Conversely, if Algorithm 3 receives an actual test suite *TS* as input, it uses it to validate the update. Procedure *Validate* (line 14) executes *TS* on both (1) *TA′* running on the new version of the API and (2) *TA′* running on the old version of the API (to check for backward compatibility). If no failures occur, the update is considered to be valid: the algorithm documents the code changes in report $ur_{AU\ change}$, marking them as validated, and moves to the next API usage in the API-usage report. Otherwise, the algorithm keeps validating updates until it either is able to validate one or it had analyzed all possible updates.

After processing all API usages in the API-usage report, this phase produces as output an evolved target app and an API-usage update report, where validated and applicable updates are documented. These two artifacts are the final output of AppEvolve.

Table 1: Benchmarks used in the empirical evaluation.

ID_A	*Name*	*Category*	*App Ver*	*API Ver*	*LOC (K)*
A01	BipolAlarm	Entertainment	0.1.1	22	4
A02	Conversations	Communication	1.8.0	22	53.1
A03	ParkenDD	Navigation	1.2.3	22	18
A04	Clean SB	Tools	1.1.4	22	16.4
A05	OpenSudoku	Game	2.5.2	22	24.3
A06	WiGLE WIFI	Tools	2.10.0	23	35.7
A07	Footguy	Lifestyle	1.5.0	23	3.4
A08	Calendar IE	Productivity	2.4.0	23	8.2
A09	Diolinux	News	2.2.2	23	13
A10	Solar Compass	Navigation	1.0.0	23	14.4
A11	Symphony	Entertainment	1.1.9	25	15
A12	SysLog	Tools	2.1.1	25	27.1
A13	Muzei	Personalization	2.4.0	25	64.4
A14	Notes	News	1.0.1	25	25.2
A15	OneTwo	Tools	1.1.6	25	20.7

4 EMPIRICAL EVALUATION

To determine the effectiveness and efficiency of AppEvolve, we implemented it in a tool built in Java and Python. We then evaluated the performance of the tool on a set of real-world apps, using GitHub as the code base where to search for update examples. We also compared the effectiveness of AppEvolve with that of LASE [39]. We selected LASE as a baseline because it also distills edit scripts from examples, is the technique most closely related to AppEvolve, handles Java programs, and is publicly available [40]. However, it is fair to note that LASE has different goals from AppEvolve. In particular, LASE was designed to work with one application at a time, and thus with fairly homogeneous examples. In our context, conversely, examples are collected from different apps and can therefore be quite diverse.

Specifically, we targeted the following research questions:
RQ1: Can AppEvolve successfully update API usages in real apps?
RQ2: For the update examples identified by AppEvolve, how do AppEvolve and LASE compare in terms of effectiveness?
RQ3: What is the cost of running AppEvolve?

4.1 Benchmarks

As benchmarks, we used 15 real-world apps from the F-droid repository [9]. We chose F-droid because the apps therein are categorized based on the latest API version they support. We selected three sets of five apps with two characteristics. First each set contained apps with the same latest supported API version: 22, 23, and 25.[2] For each API version considered, we then manually generated an API-change specification by studying the corresponding API documentation [11–14]. Second, using the generated API-change specifications, we made sure that each app contained at least one API usage that (1) was different from those in the other apps and (2) had to be updated in the subsequent API version. Table 1 provides information on the benchmarks considered: name (*Name*), category (*Category*), version (*App Ver*), latest supported API version (*API Ver*), and size (*LOC*).

Note that generating API-change specifications took us less than an hour. Moreover, these specifications must be computed only once for each new version of the API. Nevertheless, in future work we plan to compute the specification automatically (see Section 7).

[2]We chose these three versions because their subsequent version is a major version release and their adoption rate is above 10% [16] at the moment of this writing.

Table 2: Results of the update examples search phase.

ID_U	*Old API Usage*	*API Ver*	*Files*	*PFiles*	*Time*	*UE*
U01	`[addAction]`	23	17139	11862	*timeout*	1
U02	`[getAllNetworkInfo]`	23	6502	6502	5h41m	15
U03	`[getCurrentHour]`	23	6950	6950	9h16m	19
U04	`[getCurrentMinute]`	23	6532	6532	8h55m	18
U05	`[setCurrentHour]`	23	3988	3988	5h27m	17
U06	`[setCurrentMinute]`	23	2977	2977	4h48m	17
U07	`[setTextAppearance]`	23	35914	26151	*timeout*	21
U08	`[addGpsStatusListener]`	24	587	587	58m	1
U09	`[fromHtml]`	24	49070	13597	*timeout*	64
U10	`[release]`	24	25420	25420	21h50m	2
U11	`[removeGpsStatusListener]`	24	467	467	52m	1
U12	`[shouldOverrideUrlLoading]`	24	7842	7842	12h22m	0
U13	`[startDrag]`	24	2804	2804	3h3m	2
U14	`[abandonAudioFocus]`	26	138	138	1h03m	5
U15	`[getDeviceId]`	26	21144	21144	16h51m	7
U16	`[requestAudioFocus]`	26	443	443	1h30m	4
U17	`[saveLayer]`	26	19532	19532	11h02m	1
U18	`[setAudioStreamType]`	26	3122	3122	4h05m	16
U19	`[vibrate(long)]`	26	3018	3018	14h42m	7
U20	`[vibrate(long[],int)]`	26	2930	2930	14h43m	3

Table 3: Characteristics of the update examples considered.

ID_U	*CUE*	*Edits*			*Relevant Edits*			*Core Proximity Value*			*Time*
		Min	*Max*	*Avg*	*Min*	*Max*	*Avg*	*Min*	*Max*	*Avg*	
U01	1	6	6	6	6	6	6	1	1	1	52s
U02	5	16	22	19	16	21	18.8	0.43	0.56	0.48	14s
U03	5	6	17	9.8	5	16	8.8	0.31	1	0.68	14s
U04	5	7	13	9.2	5	13	8.2	0.38	1	0.7	15s
U05	5	7	19	11.4	5	5	5	1	1	1	15s
U06	5	7	19	9.4	5	5	5	1	1	1	15s
U07	5	5	13	8	5	6	5.2	0.83	1	0.97	15s
U08	1	5	5	5	5	5	5	1	1	1	3s
U09	5	5	14	8.6	5	13	8.2	0.31	0.8	0.54	14s
U10	2	5	26	15.5	5	5	5	1	1	1	4s
U11	1	5	5	5	5	5	5	1	1	1	11s
U12	-	-	-	-	-	-	-	-	-	-	-
U13	2	5	5	5	5	5	5	1	1	1	8s
U14	5	5	17	8.8	5	8	6.2	0.63	1	0.85	26s
U15	5	7	17	12.4	7	17	11.6	0.24	0.57	0.5	26s
U16	4	6	18	11.5	6	15	10.5	0.2	0.5	0.43	26s
U17	1	5	5	5	5	5	5	1	1	1	30s
U18	5	8	11	9	8	10	8.8	0.6	0.75	0.69	29s
U19	5	7	47	22.6	7	9	7.6	0.56	0.71	0.67	25s
U20	3	6	47	33.33	6	13	10.33	0.46	1	0.65	29s

4.2 Results

4.2.1 RQ1 (Effectiveness). To answer RQ1, we applied AppEvolve to the 15 benchmarks considered. Tables 2, 3, and 4 report the results of the evaluation. Overall, AppEvolve was able to update 17 out of 20 API usages (85% success rate) and 37 out of 41 of their occurrences across benchmarks.

Table 2 shows the results of the Update Examples Search phase. Lines in the table correspond to API usages that require update and are present in at least one of the benchmarks. The number of API usages (20) is greater than the number of benchmarks (15) because some benchmarks contained multiple API usages to be updated. For each API usage, the table shows: its ID (ID_U); the API method call to be updated (*Old API Usage*); the new version of the Android API (*API Ver*); the number of files in the code base matching the keywords in the code search (*Files*); how many of these files were processed by AppEvolve (*PFiles*); the time it took AppEvolve to process them (*Time*); and the number of actual update examples finally identified by the search (*UE*). (The values in Columns *Files* and *PFiles* are the same unless AppEvolve reached the 24 hours timeout before processing all the files returned by the search.) As the table shows, the technique was able to automatically find at least one update example for all API usages but U12.

Table 3 shows, for each of the 20 API usages considered, the properties of the corresponding update examples. Column *CUE* represents the number of update examples we considered. This number differs from the one in column *UE* of Table 2 for practical reasons: in our current implementation, update examples must be manually encoded as Eclipse projects, which is extremely time consuming. We therefore randomly selected up to five update examples resulting from the search for the Update Examples Analysis phase. Columns under the *Edits* header show the minimum (*Min*), maximum (*Max*), and average (*Avg*) number of edits needed to transform the old method ASTs into the new method ASTs for the considered examples. Columns under the *Relevant Edits* header show the minimum, maximum, and average number of these AST edits that are actually related to the API usage through a chain of dependencies. As the table shows, in 14 out of 19 cases, the average number of relevant edits is lower than the average number of AST edits. This shows that it is common for developers to perform additional changes that are unrelated to the update and that excluding these changes from the generated patches is important. The table also shows that the relevant edits still involve multiple statements, which highlights the need for a technique that performs updates automatically. Columns under the *Core Proximity Value* header show the minimum, maximum, and average value computed to measure the proximity of a patch to the update core. For 11 out of 19 cases, the average core proximity value is different from its minimum and maximum values, which shows that the examples considered perform the update differently.

Table 4 shows the results of the API-Usage Update phase, together with the details of its validation process. These are the main results for AppEvolve, as they illustrate the updates that the technique could successfully generate. In this case, each row in the table corresponds to the first patch validated for a specific occurrence of a given API usage (ID_U) in a specific application (ID_A).

For each patch p, Column *Appl GUP* shows the total number of applicable patches (including p) computed for the same API usage. In four cases AppEvolve was unable to identify an applicable patch. For U12, in A09, the technique did not find any update examples. For two occurrences of U11 and one of U08, in A06, the available patches required a context variable that was not in scope for the methods containing the usages in the benchmark. In all other cases, the validated patch was also the first applicable one.

Columns *Edits*, *REdits*, and *CPV* show the number of edits, the number of relevant edits, and the core proximity value for p. In three cases (two instances of U14 in A11, and the instance of U15 in A12), p does not have the maximum core proximity value among all examples (see maximum values in Table 3). In these cases, the additional statements in p lowered its core proximity value but also reduced the number of context variables (e.g., by defining some of these variables), thus making the patch applicable. This shows that only selecting statements from the core of the update can prevent successful updates in some cases.

Columns *SVars* and *CVars* report the number of scope and context variables in p. The columns under header *Validated* show whether

Table 4: Details on validated patches for all API usages.

ID$_A$	*ID$_U$*	*Appl GUP*	*Edits*	*REdits*	*CPV*	*SVars*	*CVars*	*Validated*		*Time*
								Auto	*Man*	
A01	U01	1	6	6	1	8	5	✗	✓	1s052ms
	U01	1	6	6	1	8	5	✗	✓	899ms
	U01	1	6	6	1	8	5	✗	✓	783ms
A02	U02	2	16	16	0.56	3	1	✓	✓	2s690ms
	U03	2	7	5	1	6	2	✗	✓	3s964ms
	U04	2	7	5	1	4	2	✗	✓	4s396ms
	U05	5	17	5	1	5	2	✓	✓	5s964ms
	U06	5	7	5	1	7	2	✓	✓	4s760ms
A03	U03	2	7	5	1	12	2	✓	✓	3s186ms
A04	U03	2	7	5	1	2	2	✓	✓	1s558ms
	U04	2	7	5	1	3	2	✓	✓	2s323ms
	U05	5	17	5	1	3	2	✓	✓	1s964ms
	U06	5	7	5	1	4	2	✓	✓	1s636ms
A05	U07	4	11	5	1	16	3	✓	✓	5s278ms
	U07	4	11	5	1	16	3	✓	✓	5s403ms
A06	U08	-	-	-	-	-	-	-	-	3s195ms
	U09	3	5	5	0.8	31	2	✗	✓	9s802ms
	U11	-	-	-	-	-	-	-	-	2s616ms
	U11	-	-	-	-	-	-	-	-	3s414ms
A07	U09	3	5	5	0.8	9	2	✓	✓	809ms
A08	U09	1	5	5	0.8	5	2	✓	✓	1s595ms
	U09	3	5	5	0.8	4	2	✗	✓	1s311ms
	U10	2	26	5	1	3	1	✓	✓	524ms
A09	U12	-	-	-	-	-	-	-	-	-
A10	U13	2	5	5	1	8	4	✗	✓	1s630ms
A11	U14	2	8	8	0.63	8	2	✓	✓	1s060ms
	U14	2	8	8	0.63	7	2	✓	✓	1s030ms
	U16	1	15	15	0.2	11	5	✓	✓	724ms
	U18	3	8	8	0.75	3	1	✓	✓	710ms
A12	U15	1	7	7	0.57	8	2	✓	✓	1s479ms
A13	U17	1	5	5	1	14	7	✓	✓	1s520ms
A14	U18	3	8	8	0.75	2	1	✗	✓	1s211ms
A15	U19	3	47	7	0.71	3	2	✗	✓	1s897ms
	U19	3	47	7	0.71	4	2	✓	✓	1s815ms
	U19	3	47	7	0.71	3	2	✓	✓	825ms
	U19	3	47	7	0.71	3	2	✓	✓	1s066ms
	U19	3	47	7	0.71	10	2	✓	✓	770ms
	U19	3	47	7	0.71	11	2	✓	✓	1s054ms
	U20	1	6	6	1	6	3	✓	✓	927ms
	U20	1	6	6	1	5	3	✗	✓	770ms
	U20	1	6	6	1	12	3	✗	✓	560ms

the update was automatically validated (✓in column *Auto*) or manually validated (✓in column *Man*). Since none of the benchmarks had an associated test suite, our technique generated random inputs to automatically validate the patches, as described in Section 3.4. AppEvolve could automatically validate 25 patches out of the 37 generated. For the remaining 12 patches, the generated inputs did not cover the code of the update, so we validated the patches by generating inputs manually. We also manually analyzed the code of all 37 patches and confirmed that it was following the changes described in the API documentation.

In summary, we believe that the results for RQ1 show that AppEvolve can be effective in automatically updating API usages. Overall, AppEvolve was able to automatically update 85% of the cases considered and automatically validate 68% of these updates.

4.2.2 RQ2 (Effectiveness Comparison). To answer RQ2, we applied LASE to the same set of 15 benchmarks considered for RQ1. As inputs to LASE, we used the update examples in Table 3, that is, the examples automatically retrieved by AppEvolve and used to perform the update tasks in Table 4. Unfortunately, LASE was unable to (fully) perform the update tasks considered, for different reasons. For nine tasks, LASE could not identify where to apply the edit script in the target app (Reason #1). For another eight tasks, the generated edit script was incomplete (Reason #2). For one task, LASE applied the edit script at the wrong program location (Reason #3). The remaining update tasks failed for multiple reasons: 10 tasks for #1 + #2 and 12 tasks for #2 + #3. (In one case, AppEvolve did not find update examples.)

After further analyzing the results, we believe that the main reason for LASE's performance is the fact that the tool was designed to operate in a different context. As we mentioned in Section 4, LASE was designed to work with update examples extracted from a *single code base.* When presented with the update examples automatically extracted by AppEvolve from multiple code bases, LASE had problems handling the diversity in terms of (1) operations used to perform the updates and (2) locations where the updates are performed. In fact, these possible limitations are mentioned in the LASE paper itself [39].

Based on the above results and analysis, we conclude that AppEvolve is more effective than LASE when used to automatically perform app updates for API changes.

4.2.3 RQ3 (Efficiency). To answer RQ3, we measured the time taken by each phase of the technique to process the benchmarks when running on a workstation with 64GB of RAM, one Intel Xeon i7-6700K Skylake 4.0GHz processor, and Ubuntu 16.04. The API-Usage Analysis phase took 28 seconds on average. As for the other phases, the columns labeled *Time* in Tables 2, 3, and 4 report the time taken by the Update Examples Search, Update Examples Analysis, and API-Usage Update, respectively. (Note that the time shown in Table 4 does not include the time to validate a patch, as it is too dependent on either the characteristics of an existing test suite or the specific input generation tool used.)

As these results show, the Update Examples Search phase completely dominates the other phases, with 10 hours and 27 minutes on average to complete (and reaching a timeout of 24 hours in three cases). The most expensive operation in this phase is transferring a remote code base to analyze it locally. However, indexing repositories offline could speed-up this phase, which could also be parallelized and run overnight (and the results shared across developers). All the other phases can run fairly efficiently.

5 LIMITATIONS AND THREATS TO VALIDITY

Our technique analyzes updates within a method's boundary and does not currently handle updates that span across multiple methods, a situation that occurred for two of the 20 cases we considered (see Table 3). We leave this improvement as future work. AppEvolve cannot currently automatically search for examples if there are no methods in the new API usage (i.e., the API usage was removed). This situation should rarely occur, as APIs tend to follow a *deprecate-replace-remove* cycle [30], and we did not experienced it in our evaluation.

Threats to external validity. Our results might not generalize to other apps or API usages. To mitigate this threat, we used randomly selected real-world apps from different categories and considered different versions of the API.

Threats to construct validity. There might be errors in the implementation of our technique. To mitigate this threat, we extensively tested our tools and inspected our results.

6 RELATED WORK

Example-Based Program Update. LASE [39] performs repetitive edits in a program by computing an edit script from update examples. It extends SYDIT [38], an earlier approach by the same authors, with support for multiple examples to improve generalization of edit scripts. Given two edits, LASE identifies the common edit operations between them, abstracts them as an edit script, and identifies possible locations where to apply the edit script. Unlike LASE, which relies on developers to provide example edits, AppEvolve leverages API change specifications to locate update examples automatically. Moreover, AppEvolve computes and ranks patches by considering the differences between the update examples, rather than simply taking their common edit operations, which avoids creating incomplete edit scripts when multiple, possibly heterogeneous changes are considered. Finally, AppEvolve can automatically validate updates using differential testing, rather than just providing them to the developers. RASE [37] extends LASE to target clone removal. Meditor [61], developed concurrently with AppEvolve, shares similar goals with our technique, but uses a different approach that performs updates by looking at examples in isolation.

REFAZER [52] learns code transformations from examples by leveraging a domain-specific language that describes common program transformations. Code transformations consist of a location expression, used to determine applicability of a rewrite rule, and edit operations to be performed. Because the location expression is designed to target repetitive edits, it can be difficult to use these code transformations at structurally different locations (which is needed in our context). This issue is also present for three other techniques: ARES [8], which computes update patterns using an example ordering process, a set of adjustment rules, and determines update applicability based on example locations; VuRLE [34], which uses update examples and their code location to detect and repair security vulnerabilities; and the approach by Santos and colleagues [53], which uses different techniques to identify possible locations where to apply a change. AppEvolve is more general in determining such location because it considers the variables in scope within the methods that contain API usages. In addition, none of these techniques uses differential testing to validate updates automatically.

A4 [27] learns API migration patterns from update examples and applies these patterns to the source code of Android apps. AppEvolve differs from A4 in that it identifies update examples in remote repositories, handles changes in return values, and is able to prioritize examples that are closely related to the core of the update shared across examples. Update examples have also been used by HireBuild [21] to generate updates for build scripts, which is a related but different problem.

Coccinelle [48] uses manually defined semantic patches to perform repetitive edits. Twinning [46] uses programmer specified mappings to adapt programs to alternative APIs. Balaban and colleagues [4] also leverage manually specified mappings, but to support class library migration. Unlike these techniques, AppEvolve searches and computes edit scripts automatically. textscspfind [1] extends Coccinelle by computing edit scripts from manually provided examples that contain term-replacement operations. These operations are less expressive than the ones computed by AppEvolve, as they cannot express ordering between terms.

Edit Suggestions. Other techniques suggest code edits but do not modify target code (e.g., [3, 7, 42, 45, 57, 60]). In particular, LibSync identifies changes between two versions of an API and analyzes updated API client applications to identify usage adaptation patterns and suggest edit examples. However, it cannot abstract suggested edits and developers need to update the code manually. Compared to AppEvolve, most of the other techniques either target a different problem (e.g., suggesting parameters for method calls [3] or recommending code changes for back-porting of device drivers [57]) or do not automate and validate the update process.

Code Transfer. Our technique is also tangentially related to approaches that transfer code and functionality across software systems for various purposes, such as adding new features to a system (e.g., [5, 20, 54]), improving performance (e.g., [51]), or repairing functionality (e.g., [55]). AppEvolve share with these techniques the idea of transferring code between software systems but focuses on software evolution and uses the version history of existing code bases to automatically identify the code to be transferred.

Program Repair. Program repair techniques use various approaches to modify statements in a faulty program and repair it (e.g., [2, 17, 23–26, 32, 33, 36, 49, 50, 56, 58, 59, 62]). AppEvolve is related to some of these techniques (e.g., [24, 28, 32, 33]) that also use examples to find fixes. However, besides having a different goal, these techniques use examples mostly to improve the effectiveness of their search for a fix.

7 CONCLUSION

Mobile apps must be updated when the API of their underlying OS platform changes. This task is not only time consuming, but also error prone. To address this problem, we proposed AppEvolve, a technique that leverages existing code bases to distill examples of app updates and create patches that can be applied to other apps. In our empirical evaluation, we applied AppEvolve to 15 real-world apps. Our results provide initial but strong evidence of the usefulness of our technique, as AppEvolve was able to automatically update 85% of the API usages considered and automatically validate 68% of these updates.

In future work, we plan to perform a more extensive evaluation of the approach to confirm our initial results. We will also extend the technique in different ways. First, we plan to handle updates that span across multiple methods. Second, we plan to automatically compute API change specifications based on a mix of program analysis and natural language processing techniques. Third, and more on the engineering side, we plan to add to the tool the ability to handle app binaries. Finally, we will investigate the use of our technique in other contexts (e.g., web applications).

ACKNOWLEDGMENTS

This work was partially supported by NSF, under grants CCF-1161821 and CCF-1563991, DARPA, under contracts FA8650-15-C-7556 and FA8650-16-C-7620, ONR, under contract N00014-17-1-2895, and gifts from Google, IBM Research, and Microsoft Research. Na Meng, Miryung Kim, and Kathryn McKinley helped us with the comparison between AppEvolve and LASE.

REFERENCES

[1] J. Andersen and J. L. Lawall. 2008. Generic Patch Inference. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 337–346.

[2] A. Arcuri and Xin Yao. 2008. A novel co-evolutionary approach to automatic software bug fixing. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*. 162–168.

[3] M. Asaduzzaman, C. K. Roy, S. Monir, and K. A. Schneider. 2015. Exploring API method parameter recommendations. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 271–280.

[4] Ittai Balaban, Frank Tip, and Robert Fuhrer. [n.d.]. Refactoring Support for Class Library Migration. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, New York, NY, USA, 265–279.

[5] Earl T Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, Baltimore, MD, USA, 257–269.

[6] Gabriele Bavota, Mario Linares-Vásquez, Carlos Eduardo Bernal-Cardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2015. The Impact of API Change- and Fault-proneness on the User Ratings of Android Apps. *IEEE Transactions on Software Engineering* 41 (2015), 384–407.

[7] Barthélémy Dagenais and Martin P. Robillard. 2008. Recommending Adaptive Changes for Framework Evolution. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA.

[8] Georg Dotzler, Marius Kamp, Patrick Kreutzer, and Michael Philippsen. 2017. More Accurate Recommendations for Method-level Changes. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. ACM, New York, NY, USA, 798–808.

[9] F-Droid 2018. F-Droid. Retrieved October 12, 2018 from https://f-droid.org

[10] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall. 2007. Change Distilling: Tree Differencing for Fine-grained Source Code Change Extraction. *IEEE Transactions on software engineering* 33 (2007), 725–743.

[11] Google 2018. Android API Differences Report (v23). Retrieved October 12, 2018 from https://developer.android.com/sdk/api_diff/23/changes

[12] Google 2018. Android API Differences Report (v24). Retrieved October 12, 2018 from https://developer.android.com/sdk/api_diff/24/changes

[13] Google 2018. Android API Differences Report (v26). Retrieved October 12, 2018 from https://developer.android.com/sdk/api_diff/26/changes

[14] Google 2018. API Reference. Retrieved October 12, 2018 from https://developer.android.com/reference

[15] Google 2018. ConnectivityManager. Retrieved October 12, 2018 from https://developer.android.com/reference/android/net/ConnectivityManager.html#getAllNetworkInfo()

[16] Google 2018. Distribution Dashboard. Retrieved October 12, 2018 from https://developer.android.com/about/dashboards

[17] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE, Zurich, Switzerland, 3–13.

[18] Koji Hakata and Hiroshi Imai. 1998. Algorithms for the Longest Common Subsequence Problem for Multiple Strings based on Geometric Maxima. *Optimization Methods and Software* 10 (1998), 233–260.

[19] Dan Han, Chenlei Zhang, Xiaochao Fan, Abram Hindle, Kenny Wong, and Eleni Stroulia. 2012. Understanding Android Fragmentation with Topic Analysis of Vendor-Specific Bugs. In *Proceedings of the 2012 Working Conference on Reverse Engineering*.

[20] M. Harman, W. B. Langdon, and W. Weimer. 2013. Genetic programming for Reverse Engineering. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. 1–10.

[21] Foyzul Hassan and Xiaoyin Wang. 2018. HireBuild: An Automatic Approach to History-driven Repair of Build Scripts. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 1078–1089.

[22] Dongjie He, Lian Li, Lei Wang, Hengjie Zheng, Guangwei Li, and Jingling Xue. 2018. Understanding and Detecting Evolution-induced Compatibility Issues in Android Apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, New York, NY, USA, 167–177.

[23] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards Practical Program Repair with On-demand Candidate Generation. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 12–23.

[24] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. ACM, New York, NY, USA.

[25] Yalin Ke, Kathryn T Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing Programs with Semantic Code Search (T). In *Proceedings of 30th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, Lincoln, NE, USA, 295–306.

[26] D. Kim, J. Nam, J. Song, and S. Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*. 802–811.

[27] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Chen. 2018. A4: Automatically Assisting Android API Migrations Using Code Examples. *CoRR* (2018).

[28] Xuan-Bach D Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution and Reengineering*. IEEE, Osaka, Japan, 213–224.

[29] Li Li, Tegawendé F. Bissyandé, Haoyu Wang, and Jacques Klein. 2018. CiD: Automating the Detection of API-related Compatibility Issues in Android Apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, New York, NY, USA, 153–163.

[30] Li Li, Jun Gao, Tegawendé F. Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. 2018. Characterising Deprecated Android APIs. In *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, New York, NY, USA, 254–264.

[31] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2013. API Change and Fault Proneness: A Threat to the Success of Android Apps. In *Proceedings of the 9th joint meeting on foundations of software engineering*. ACM, Saint Petersburg, Russia, 477–487.

[32] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic Inference of Code Transforms for Patch Generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, New York, NY, USA, 727–739.

[33] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, St. Petersburg, FL, USA, 298–312.

[34] Siqi Ma, Ferdian Thung, David Lo, Cong Sun, and Robert H. Deng. 2017. VuRLE: Automatic Vulnerability Detection and Repair by Learning from Examples, Simon N. Foley, Dieter Gollmann, and Einar Snekkenes (Eds.). Springer International Publishing, Cham, 229–246.

[35] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *Proceedings of the 29th IEEE International Conference on Software Maintenance*. IEEE, Eindhoven, Netherlands, 70–79.

[36] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering*. IEEE, Austin, TX, USA, 691–701.

[37] Na Meng, Lisa Hua, Miryung Kim, and Kathryn S. McKinley. 2015. Does Automated Refactoring Obviate Systematic Editing?. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 392–402.

[38] Na Meng, Miryung Kim, and Kathryn S Mckinley. 2011. Systematic Editing: Generating Program Transformations from an Example. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*. ACM, San Jose, CA, USA, 329–342.

[39] Na Meng, Miryung Kim, and Kathryn S McKinley. 2013. LASE: Locating and Applying Systematic Edits by Learning from Examples. In *Proceedings of the 35th International Conference on Software Engineering*. IEEE, San Francisco, CA, USA, 502–511.

[40] Na Meng, Miryung Kim, and Kathryn S McKinley. 2013. LASE Tool. Retrieved January 28, 2019 from http://people.cs.vt.edu/nm8247/research.html

[41] Webb Miller, Eugene, and W. Myers. 1985. A File Comparison Program. *Software: Practice and Experience* 15 (1985), 1025–1040.

[42] Tim Molderez, Reinout Stevens, and Coen De Roover. 2017. Mining Change Histories for Unknown Systematic Edits. In *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, Piscataway, NJ, USA, 248–256.

[43] J. Munkres. 1957. Algorithms for the Assignment and Transportation Problems. *Journal of the Society of Industrial and Applied Mathematics* 5 (1957), 32–38.

[44] Katta G. Murty. 1968. Letter to the Editor - An Algorithm for Ranking all the Assignments in Order of Increasing Cost. *Operations Research* 16 (1968), 682–687.

[45] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. 2010. A Graph-based Approach to API Usage Adaptation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, New York, NY, USA, 302–321.

[46] Marius Nita and David Notkin. 2010. Using twinning to adapt programs to alternative apis. In *Proceedings of the 32nd International Conference on Software Engineering*.

[47] OpenSignal 2015. Android Fragmentation. Retrieved October 12, 2018 from https://opensignal.com/reports/2015/08/android-fragmentation

[48] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and Automating Collateral Evolutions in Linux Device Drivers. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (Eurosys '08)*. ACM, New York, NY, USA.

[49] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. 2009. Automatically Patching Errors in Deployed Software. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. ACM, New York, NY, USA, 87–102.

[50] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward. 2018. Genetic Improvement of Software: A Comprehensive Survey. *IEEE Transactions on Evolutionary Computation* (2018), 415–432.

[51] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. 2014. Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class. In *Genetic Programming*, Miguel Nicolau, Krzysztof Krawiec, Malcolm I. Heywood, Mauro Castelli, Pablo García-Sánchez, Juan J. Merelo, Victor M. Rivas Santos, and Kevin Sim (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 137–149.

[52] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE, Buenos Aires, Argentina, 404–415.

[53] G. Santos, K. V. Paixao, N. Anquetil, A. Etien, M. de Almeida Maia, and S. Ducasse. 2017. Recommending source code locations for system specific transformations. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 160–170.

[54] Stelios Sidiroglou-Douskos, Eric Lahtinen, Anthony Eden, Fan Long, and Martin Rinard. 2017. CodeCarbonCopy. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 95–105.

[55] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic Error Elimination by Horizontal Code Transfer Across Multiple Applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 43–54.

[56] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. 2018. Repairing Crashes in Android Apps. In *Proceedings of the 40th International Conference on Software Engineering*. IEEE, Gothenburg, Sweden, 187–198.

[57] F. Thung, X. D. Le, D. Lo, and J. Lawall. 2016. Recommending Code Changes for Automatic Backporting of Linux Device Drivers. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 222–232.

[58] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, Washington, DC, USA, 364–374.

[59] Qi Xin and Steven P Reiss. 2017. Leveraging Syntax-related Code for Automated Program Repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, Urbana-Champaign, IL, USA, 660–670.

[60] Z. Xing and E. Stroulia. 2007. API-Evolution Support with Diff-CatchUp. *IEEE Transactions on Software Engineering* (2007), 818–836.

[61] Shengzhe Xu, Ziqi Dong, and Na Meng. 2019. Meditor: Inference and Application of API Migration Edits. In *Proceedings of the 27th International Conference on Program Comprehension*. IEEE Press, Piscataway, NJ, USA, 335–346.

[62] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* (2017), 34–55.

[63] Guowei Yang, Jeffrey Jones, Austin Moninger, and Meiru Che. 2018. How Do Android Operating System Updates Impact Apps?. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. ACM, New York, NY, USA, 156–160.