

# FILO: FIx-LOcus Recommendation for Problems Caused by Android Framework Upgrade

Marco Mobilio

Dept. of Informatics, Systems and Communication  
University of Milano - Bicocca  
Milan, Italy  
marco.mobilio@unimib.it

Oliviero Riganelli

Dept. of Informatics, Systems and Communication  
University of Milano - Bicocca  
Milan, Italy  
oliviero.riganelli@unimib.it

Daniela Micucci

Dept. of Informatics, Systems and Communication  
University of Milano - Bicocca  
Milan, Italy  
daniela.micucci@unimib.it

Leonardo Mariani

Dept. of Informatics, Systems and Communication  
University of Milano - Bicocca  
Milan, Italy  
leonardo.mariani@unimib.it

**Abstract**—Dealing with the evolution of operating systems is challenging for developers of mobile apps, who have to deal with frequent upgrades that often include backward *incompatible* changes of the underlying API framework. As a consequence of framework upgrades, apps may show misbehaviours and unexpected crashes once executed within an evolved environment.

Identifying the portion of the app that must be modified to correctly execute on a newly released operating system can be challenging. Although incompatibilities are visible at the level of the interactions between the app and its execution environment, the actual methods to be changed are often located in classes that do not directly interact with any external element.

To facilitate debugging activities for problems introduced by *backward incompatible upgrades of the operating system*, this paper presents FILO, a technique that can recommend the *method* that must be changed to implement the fix from the analysis of a single failing execution. FILO can also select *key symptomatic anomalous events* that can help the developer understanding the reason of the failure and facilitate the implementation of the fix.

Our evaluation with multiple known compatibility problems introduced by Android upgrades shows that FILO can effectively and efficiently identify the faulty methods in the apps.

**Index Terms**—Debugging, Android, API upgrades.

## I. INTRODUCTION

To continuously release new and advanced features that exploit the latest hardware and software upgrades, the operating systems of mobile devices must evolve at a dramatic speed. For instance, the Android API framework evolves at the average rate of 115 API updates per month [1] and a new release is produced every two months in average [2]. Such a fast *evolution is not problem-free*. For example, in their study Wei et al. found that *more than one third* of the compatibility issues affecting popular Android apps are due to API evolution [3]. Note that these problems rarely consist of faults in the framework, but they rather consist of *backward incompatible changes* that require the apps to be fixed to run correctly. This is also confirmed in the study by Mostafa et al. who found that the large majority of backward compatibility problems are fixed in the client code of the apps [4].

Migrating an app to a new API can be *painful*. Once a problem in the app is discovered, developers have to investigate the behaviour of the app to understand the cause of the problem, identify a suitable location for the fix, and implement it. This whole process is demanding and makes developers reluctant to adopt new APIs. For instance, McDonnell et al. [1] reported an average migration time of 16 months, in contrast with an API release interval of few months only.

Simplifying the migration process is thus extremely important to let developers quickly adapt their apps to newly released operating systems. In this paper, we focus on the challenge of assisting the problem resolution task by *automating the identification* of the code region that must be modified to fix an app that is incompatible with a newer version of the underlying framework. This can be seen as an instance of spectrum-based fault localization (SBFL) [5], [6], but contrarily to SBFL that requires a full test suite with passing and failing test cases, our approach, namely *FILO*, requires only a *single* failed test case to be applied. This has three important benefits: (i) it is applicable to the many cases where an extensive unit test suite is not available, (ii) it can be straightforwardly applied to those cases where the failure is exposed with a system-level interaction, such as an automatic system test derived from a bug report entered by a user, and (iii) it is extremely cheap to execute since it avoids the execution of large test suites.

In contrast with SBFL techniques that can only localize suspicious code regions, FILO also isolates information about the *anomalous events* that are the consequence of the incompatibility between the app and the newly released framework, providing further information potentially useful to the developers to understand the failure and implement a proper fix.

The intuition behind FILO is twofold:

*Interactions between the framework and the app are likely to contain an evidence of the failure*: Since the incompatibility is between an app and its API framework, the problem should intuitively be visible by observing their interactions (i.e.,

calls from the app to the framework, and vice versa). The comparison of the interactions observed when the app interacts with the compatible (older) and the incompatible (newer) versions of the framework can be used to identify *suspicious interactions* that are in turn useful to identify the *part of the behavior of the app that must be modified* to obtain the fix.

*The method that must be fixed is likely responsible for a large and coherent set of suspicious interactions:* The detailed analysis of the full failed execution, not only limited to interactions between the app and its framework, can reveal the methods *internal to the app* that control the execution of a significantly *large and coherent* set of suspicious interactions. These methods should include the one that must be changed to fix the program. Based on this strategy, FILO generates a ranking that can be exploited by the developers to identify the fix location. Each method can also be associated with the *suspicious interactions under its influence* to provide insights about the rationale of the selection.

We empirically assessed FILO with multiple incompatibilities between Android apps and various versions of the Android framework. Results show that FILO can efficiently identify the method where the fix should be implemented from the analysis of a single failing test case: it ranked the method that must be modified in the top 5 positions in the large majority of the cases, with several cases where the method occurred at the top of the ranking. We also compared FILO to SBFL, confirming its higher effectiveness in addition to its higher applicability.

In a nutshell, the main contributions of this paper are:

- FILO, a technique that can address incompatibilities between an app and an updated framework by producing a ranked list of suspicious methods that must be changed, associated with supporting evidence about their selection, from a single system-level failed execution;
- the empirical evidence that FILO can operate efficiently and effectively;
- a freely available implementation of our tool and a replication package (<https://gitlab.com/learnERC/filo>) that can be used to replicate the results reported in the paper.

The rest of the paper is organized as follows. Section II presents a running example that is used throughout the paper to illustrate our approach. Sections III and IV describe and empirically evaluate FILO, respectively. Section V discusses related work. Finally, Section VI provides finale remarks.

## II. RUNNING EXAMPLE

In this section, we describe an actual Android app that suffers from an upgrade issue common to many others Android apps. The issue is due to the backward incompatible change implemented between the Android API 22 and the Android API 23 that forces apps to explicitly request for a permission when accessing resources for the first time. The opensource *Good Weather* application [7] is one of the applications affected by this backward incompatibility problem, due to an incomplete implementation of support to API 23. The application exploits the location services to produce a weather

forecast about the current location of the user. This application has been installed by more than 5,000 Android users according to Google Play. We used this app both in the empirical evaluation and in Section III to illustrate how FILO works.

The method whose implementation is incompatible with API 23 can be obtained from Listing 1 by ignoring the red code, which is the code added to obtain the fix, and including the code with the strikethrough font, which is the code removed to obtain the fix. In the faulty implementation, when the `gpsRequestLocation` method is invoked, the permissions to access to the fine and coarse grained locations are checked. If the permissions are granted (as it happens with API 22), the location is regularly updated executing the method `requestLocationUpdates()`. Otherwise, if the permissions are not granted (as it happens with API 23), the method returns without updating the location, resulting in the application hanging forever. The hang is due to the code responsible for the elimination of the progress bar (not shown in the listing) that is executed only once the location has been updated.

Listing 1. The Fix for the Good Weather application.

```
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.main_menu_detect_location:
            requestLocation();
            gpsRequestLocation();
            ...
    }
}

private void requestLocation() {
    detectLocation();
    ...
}

private void detectLocation() {
    gpsRequestLocation();
    ...
}

public void gpsRequestLocation() {
    if (checkSelfPermission(this, ACCESS_FINE_LOCATION) ==
        PERMISSION_GRANTED) {
        if (VERSION.SDK_INT >= VERSION_CODES.M) {
            if (checkSelfPermission(this, ACCESS_FINE_LOCATION) !=
                PERMISSION_GRANTED && checkSelfPermission(this,
                    ACCESS_COARSE_LOCATION) != PERMISSION_GRANTED) {
                return;
            }
            locationManager.requestLocationUpdates(
                locationManager.GPS_PROVIDER, 0, 0,
                mLocationListener);
        }
    }
}
```

In order to fix the program it is necessary to ask the user to grant the access to the location information before invoking the `requestLocationUpdates` method. The developers obtained the fix by modifying the `onOptionsItemSelected` and the `gpsRequestLocation` methods, making them to invoke new methods designed to acquire the required permissions. A proper analysis of a failing interaction with the app should report the `gpsRequestLocation` and the `onOptionsItemSelected` methods as the methods to be modified to obtain the fix.

In this case, FILO successfully reported the `gpsRequestLocation` and `onOptionsItemSelected` methods at the top of the ranking. In addition FILO can isolate and report anomalous interactions that happened in the failed execution about both permission checking and access to the

location service to the developers. These anomalous events are well representative of what the problem is, that is, the app lacks the permission to access the location service.

Finally, note that the failure per se is not explicative of the fact that the problem is about permissions: the user can only see the hang and the app does not log any error message. This is why the output produced by FILO can be extremely helpful to quickly fix this app.

### III. FILO

The purpose of FILO is to automatically recommend the likely locations of fixes for problems caused by upgrades of the Android framework to developers. FILO requires three inputs: an *automatic test case*, which is used to reproduce the problem, and the access to two Android environments, one running the app with the *compatible API* and the other running the app with the *incompatible API*. The output produced by FILO is a ranked list of methods corresponding to the possible fix locations, associated with a set of suspicious interactions that motivate the ranking and that can be used by the developer to investigate the problem and implement a fix.

FILO works in three main phases as shown in Fig. 1: the *Test Execution* phase runs the test case that reproduces the failure and collects the interactions between the app and the framework from the two available environments; the *Anomaly Detection* phase identifies blocks with suspicious interactions by comparing the collected traces; the *Fix Locus Candidates Identification* phase identifies and ranks the places where a fix is likely to be implemented and associates the corresponding evidence. We describe these three phases in details below.

#### A. Test Execution

The test execution phase collects the interactions between the app and the framework for both the *base environment*, which runs the target app with a *compatible* version of the framework API, namely *v1*; and the *upgraded environment*, which runs the same target app with an incompatible version of the framework API, namely *v2*. The collected interactions consist of all the calls to methods of the framework produced by the app, and all the calls to methods of the app produced by the framework. While the rest of the calls, that is, calls internal to the framework and calls internal to the app, are ignored. This is because incompatibilities must be observable while looking at the interactions between the app and its framework.

More formally, a *framework method* is a method defined in the Android framework or in a standard library (e.g., `java.*`); an *application method* is a method implemented in the app. An *API call* is a call to a framework method originated by an application method. Vice versa a *callback* is a call to a method of the app originated by a framework method. All the other cases are internal method calls that are ignored in this step (note that calls internal to the app are on the contrary relevant to the third step for the identification of the fix locus).

The trace files recorded by FILO only include API calls and callbacks. We refer to the union of the API calls and callbacks as the *boundary calls*. To collect this information

FILO instruments the app and executes the automatic test case on both the base and upgraded environments.

The output of this phase consists of two traces containing an interleaved sequence of API calls and callbacks. The *baseline trace* is obtained by running the test case within the base environment and represents how the app and the framework interact when the execution is correct. The *failure trace* is obtained by running the test case within the upgraded environment and represents how the app and the upgraded framework interact when the app fails.

Listing 2. Excerpt of the Good Weather baseline trace.

```
MainActivity.onCreate()#b
AppCompatActivity.onCreate()#b
AppCompatActivity.onCreate()#e
AppCompatActivity.getSupportActionBar()#b
AppCompatActivity.getSupportActionBar()#e
AppCompatActivity setContentView(...)#b
AppCompatActivity setContentView(...)#e
AppCompatActivity.findViewById(...)#b
AppCompatActivity.findViewById(), returnValue:...#e
AppCompatActivity.getSupportActionBar(...)#b
AppCompatActivity.getSupportActionBar(...)#e
MainActivity.onCreate()#e
```

An excerpt of a baseline trace is shown in Listing 2. Note that FILO traces when the execution of every boundary call both starts and ends (marked with `#b` and `#e` in the sample trace). If a boundary call produces a return value, FILO also records it in the trace (such as for `findViewById()` in the sample file): if the return value is of primitive type FILO directly records it, if the return value is non-primitive FILO records the value returned by method `toString()` if overridden, otherwise FILO only records the dynamic type of the return value. In the example trace we indicate callbacks in italics: only the call to `MainActivity.onCreate()` is a callback since the method is invoked from a method of the framework (not shown in the trace). All the other calls are API calls, that is, they are calls to the framework (`AppCompatActivity` is a class of the framework) generated by the app (`MainActivity.onCreate()` is implemented in the app).

Executing *exactly* the same test in both environments and restricting the observations to boundary calls reduce the risk of having incidental differences in the traces, which implies that the large majority of the observed differences are relevant to the problem under analysis. If non-deterministic interactions are present in the traces, they can be filtered out by executing the same test multiple times and eliminating the changing portion of the trace from the analysis. We however never observed such case in our evaluation.

Note that although the scope of the observation is limited to boundary calls, the size of the traces might be significant. For instance, the baseline trace collected for the running example includes 14,245 method calls and has a size of  $\sim 41$ MB, while the size of the traces in the experiments reached 617,427 method calls and  $\sim 1.8$ GB for the MapDemo application [8].

#### B. Anomaly Detection

The anomaly detection phase compares the two traces produced in the test execution phase and isolates a number of invocation blocks that look suspicious. An *invocation block*

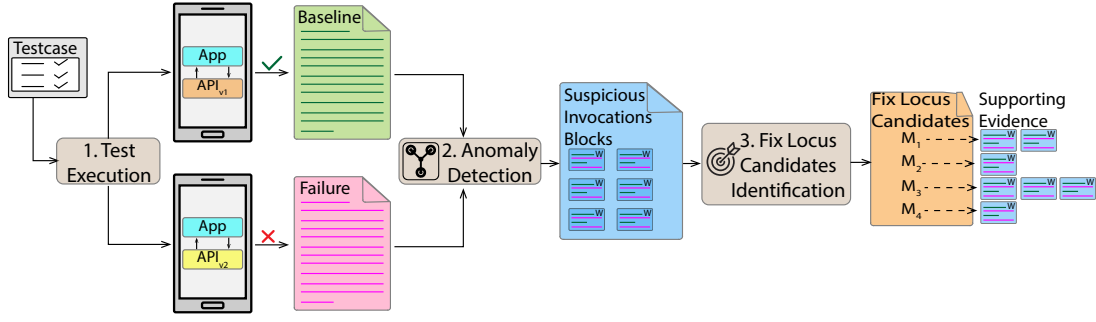


Fig. 1. Overview of the FILO technique.

is a contiguous sequence of boundary calls extracted from the traces generated during test execution. More formally, given a recorded trace  $e_1, \dots, e_n$ , an invocation block is any subsequence  $e_i, e_{i+1}, \dots, e_{i+k}$ , s.t.,  $i \geq 1$  and  $i + k \leq n$ .

The anomaly detection phase identifies, groups, and assigns weights to the differences between the baseline trace and the failure trace. Since the two traces show the boundary calls collected while running exactly the same test case, most of the differences are likely due to changes in the underlying framework and their impact on the app, specifically in the case of the failure. FILO identifies these differences by running *diff* [9], which returns the invocation blocks in the failure trace with no counterpart in the baseline trace. Since these blocks represent differences that may characterize the failure, we refer to them as the *Suspicious Invocation Blocks* (SIBs).

The set of differences contained in the SIBs can still be large. Indeed a different behavior of the framework may produce several differences in the return values of the API calls and on the generated callbacks. As a consequence the app may react differently than with the older version of the framework and produce unexpected boundary calls. For instance, in the running example there are thousands of differences between the baseline and failure traces.

To analyze the content of the SIBs, FILO performs two operations: it associates the block with its first boundary call, since that call is likely to be the cause of all the differences reported in the block; in addition, considering that spurious interactions not related to the failure are often due to short SIBs of length 1, FILO weights the relevance of each block based on the number of boundary calls that it contains.

For example, Listing 3 shows an excerpt of a SIB detected for the running example. The block starts with a call to method `MainActivity.onLocationChanged`, which is representative of the entire stream of anomalous interactions that follow. Since the block includes 30 boundary calls, this is also the weight of the block, which is shown on the first row of the listing.

The output of this phase is thus a number of weighted SIBs that FILO uses to identify how and where to change the app to fix the compatibility problem. Note that the anomaly detection step typically identifies several blocks. For instance, the number of SIBs in the running example are 98. However, not all of them have the same relevance based on our heuristics, indeed

many of them are short spurious anomalies. For example, 70 SIBs have a weight of 1 (i.e., they include a single boundary call) and only 23 SIBs have a weight higher than 3.

Note that the set of SIBs often does not contain the method that should be modified to fix the program but only the methods that misbehave due to the upgrade. For instance, one of the methods that must be modified in the running example is `gpsRequestLocation()` and this method never occurs in the baseline and failure traces. This is confirmed in our empirical evaluation, where for 50% of the cases the recorded traces do not include the method with the fix.

Listing 3. Excerpt of a Suspicious Invocation Block  
`MainActivity.onLocationChanged(...)#b Weight 30`  
`android.app.Dialog.cancel()#b`  
`android.app.Dialog.cancel#e`  
`android.location.Location.getLatitude()#b`  
`android.location.Location.getLatitude, returnValue: ...#e`  
`...`

### C. Fix Locus Candidates Identification

The *fix locus candidates identification* phase is the last phase of the FILO technique. It takes the set of weighted SIBs as input and produces a ranked list of methods that represent the locations where the fix should be likely implemented as output. Each method in the ranking is associated with supporting evidence that consists of the set of SIBs that might be affected by the method. Intuitively, the suspicious invocations associated with a method represent the symptoms of the failure that can be removed by implementing a proper fix in the method. This information can be useful for the developers who can benefit from contextual information and insights about the failure, in addition to information about the location where a fix should be implemented, when working on the app to produce a fix.

Since the fixes must be often implemented in methods that are not part of the SIBs, which in fact represent the symptoms of failures and not their causes, FILO considers any method executed in the failing execution as a potential target for the fix. In particular, FILO creates the *failure call tree*, which is a tree that includes all the methods present in the stack trace at the time each SIB has been detected (this information is collected when executing the failing test). Nodes represent the invoked methods and direct edges represent calls between methods. More formally, for each SIB  $sib_i$

with weight  $w_i$  and representative call  $c_i$ , FILO collects its stacktrace  $\langle m_1, \dots, m_{n_i}, c_i \rangle$ , where  $m_1$  is always the method `ZygoteInit.main`. The failure call tree is a triple  $(N, E, r)$ , where  $N = \{m_1, \dots, m_{n_i}, c_i | \forall sib_i\}$  is the set of nodes,  $E = \{(m_{n_i}, c_i), (m_i, m_{i+1}), i = 1, \dots, n_i - 1 | \forall sib_i\}$  is the set of edges, and  $r = \text{ZygoteInit.main}$  is the root of the tree.

Note that the root of the failure call tree is the `ZygoteInit.main` method, which is the initial method that handles the forking of every application launched in Android. The leafs are the representative methods of the SIBs, returned by the anomaly detection phase. Each leaf node has a weight corresponding to the weight of the SIB that originated the node. Leaf nodes with high weights are more relevant to the analyzed failure than the other leaf nodes.

FILO scores each node of the tree, that is, each method call, based on its degree of influence on the SIBs, represented by the weighted leaf nodes of the failure call tree. Intuitively a high score indicates a method that can directly affect the execution of several blocks of non-trivial weight. This score is the *suspiciousness* of the method and is used to produce the final ranking. More formally, we compute the suspiciousness of a method  $m$  as a linear combination of two attributes based on the following formula:

$$\text{Susp}(m) = k_1 \text{ImpBlocks}(m) + k_2 \text{Depth}(m)$$

The sum  $k_1 + k_2$  is 1. Each attribute measures a characteristic that should be taken into account in the attempt of identifying the method that is likely the locus of the fix and its value ranges between 0 and 1. We discuss these two attributes below.

$\text{ImpBlocks}(m)$  measures the number of SIBs that can be affected by a specific method. It is computed by summing the weights of the SIBs (leaf nodes) that can be reached from  $m$  following a path in the failure call tree. The value is normalized with respect to the sum of the weights of all the SIBs in the tree. Note that by definition the root of the failure call tree can affect all the SIBs, thus  $\text{ImpBlocks}(\text{rootnode}) = 1$ . Similarly a node  $sib$  representing a SIB with weight  $w$  can only influence itself and thus  $\text{ImpBlocks}(sib) = \frac{w}{W}$ , where  $W$  is the sum of the weights of all the SIBs in the tree.

$\text{Depth}(m)$  measures the position of the node with respect to the height of the failure call tree, that is, it measures how close a selected method is to the SIBs. The score is normalized with respect to the height of the tree. Thus,  $\text{Depth}(\text{rootnode}) = 0$  and  $\text{Depth}(sib) = 1$  for the deepest SIB  $sib$ .

These two attributes suitably interact one with the other to identify the methods that with the highest probability must be changed to fix the failure. The  $\text{ImpBlocks}(m)$  attribute privileges the choice of nodes at the top of the tree selecting methods that control the execution of a significant number of SIBs. This intuitively satisfies the criterion that the method that must be changed to implement the fix *must have an impact on a significant number of the SIBs* that have been detected.

The  $\text{Depth}(m)$  attribute privileges the choice of nodes close to the leafs of the tree, discouraging the selection of methods that have been executed too early during the failure. The interaction of this attribute with the  $\text{ImpBlocks}(m)$  attribute favours the selection of *nodes that influence the execution*

TABLE I  
RANKING RETURNED BY FILO.

| MethodName                                                      | Susp        |
|-----------------------------------------------------------------|-------------|
| <b>org.asdtm.goodweather.MainActivity.gpsRequestLocation</b>    | <b>0.72</b> |
| <b>org.asdtm.goodweather.MainActivity.onOptionsItemSelected</b> | <b>0.69</b> |
| org.asdtm.goodweather.MainActivity\$1.onLocationChanged         | 0.53        |
| org.asdtm.goodweather.MainActivity.preLoadWeather               | 0.45        |
| org.asdtm.goodweather.MainActivity.onResume                     | 0.43        |
| org.asdtm.goodweather.BaseActivity.configureNavigationView      | 0.42        |
| org.asdtm.goodweather.MainActivity.onPause                      | 0.39        |
| org.asdtm.goodweather.BaseActivity.setupNavDrawer               | 0.39        |
| org.asdtm.goodweather.BaseActivity.onCreate                     | 0.36        |

of a cohesive set of SIBs with relevant weights. In fact, starting from the selection of a leaf node (i.e., a SIB), it is worth selecting an ancestor node, that is, a method executed earlier in the failure, only if the loss in the *Depth* attribute is compensated by the gain in the *ImpBlocks* attribute. In particular, it is convenient to consider methods that can control the execution of a higher number of SIBs only if these blocks have a relevant weight.

Since the failure call tree represents method calls and the same method might be invoked in multiple places, while FILO ranks methods based on their suspiciousness, the final ranking is obtained by considering each method only once using the occurrence with the highest suspiciousness for the ranking. Before returning the ranking to the user, all the methods belonging to the framework are removed because the adaptation must necessarily target the app. In the running example, the resulting ranking is shown in Table I. Note that the methods that require the fix, that is `gpsRequestLocation` and `onOptionsItemSelected`, are ranked at the top, they are thus the first methods that a developer would inspect.

Since SIBs with just one call, that is, blocks of weight 1, in the large majority of the cases correspond to noisy differences unrelated with the failure, FILO uses only blocks of minimal weight 2, unless all the blocks have weight 1.

Note that in the running example the top ranked method identified by FILO never occurs in the set of boundary calls and thus a trivial comparison of the log files could not produce a precise recommendation to the developer.

Finally, FILO cannot address faults that occur in methods that are not part of the stacktraces collected when the SIBs are detected. Although this may sometime happen, a failing execution produces several SIBs and thus a number of stacktraces, and consequently a number of methods are added to the failure call tree during the analysis. The possibility that the faulty method is not part of the tree is small, as also confirmed in our evaluation where this happened only in 1 case out of 12. We thus decided to design FILO to be fast and effective, at the cost of potentially missing a small percentage of faults.

#### IV. EMPIRICAL EVALUATION

The empirical evaluation answers to three research questions concerning the information captured in the SIBs, the quality of the ranking, and the comparison to *naive trace analysis* and *spectrum based fault localization* (SBFL).

**RQ1: Are the suspicious invocation blocks capturing information about the symptoms of failures?** This research question investigates if the SIBs really capture the symptoms of the failures and thus can be reliably used to build the rest of the analysis. To answer this research question we compare the content of the SIBs to the set of method calls actually related to the incompatibility between the framework and the app.

**RQ2: How well the ranking returned by FILO identifies the method that must be modified to implement the fix?** This research question investigates if FILO is able to produce rankings where the method that must be modified to fix the program occurs at the top positions of the ranking, possibly within the first 10 positions, and ideally within the first 5 [10].

**RQ3: How does FILO compare to both naive trace analysis and SBFL techniques?** This research question compares FILO to two competing approaches. Naive trace analysis consists of identifying the method that must be changed simply comparing the baseline and failure traces one to the other. This comparison has the objective to assess the importance of the third phase of the technique, which goes beyond the content of the trace files for the localization. We also compare FILO to SBFL techniques, Ochiai [5] in particular, in terms of the ability to identify the faulty method. Although FILO operates under weaker assumptions than SBFL, for instance FILO does not require the availability of a full test suite with passing test cases to be applied, we compared them to study how their effectiveness relate one to the other.

In addition to the three research questions, we qualitatively discuss the supporting evidence generated by FILO in a discussion section. In particular, we show how the SIBs isolated by FILO for the methods in the ranking may represent a useful support to understand the failure and possibly implement a fix, despite the presence of noisy method calls. We conclude the section presenting the threats to the validity of the experiment and discussing the limitations of the approach.

#### A. Prototype

Our prototype implementation consists of two main components: the tracer, which is responsible of recording the traces with boundary calls, and the analyzer, which is responsible of producing the ranking. Since Android represents the largest share of smartphone operative systems (86,8% of the units shipped in 2018Q3 are Android phones [11]), we implemented these two components targeting the Android ecosystem. Note that, although our implementation is specific to Android, FILO is general and could be implemented also for other ecosystems.

To make our results reproducible by third parties we implemented, for each app, the shortest interaction sequence that reveals the incompatibility between the app and the framework as an automatically executable Appium test case [12]. We packaged and made our prototype implementation, the apps studied in this paper, and the corresponding Appium test cases available online at the following url: <https://gitlab.com/learnERC/filo>.

The tracer collects the information about failures by running the Appium test case twice. The first time the tracer exploits

the native Android Tracer to obtain the list of executed methods. These methods are then used to configure our monitor implemented as an Xposed [13] module to selectively instrument the relevant methods only, collecting additional information that cannot be extracted with the Android Tracer, such as the return values, and the stack-trace of the invocations. The final trace with boundary calls only is obtained by filtering the collected calls based on the identity of the callers.

The analyzer is a Java component analyzing the traces as discussed in Section III. To configure the values of the two weights  $k_1$  and  $k_2$ , we empirically evaluated multiple combinations with a subset of the apps and we ended up using  $k_1 = 0.25$  and  $k_2 = 0.75$ . FILO demonstrated to be relatively sensitive to the choice of  $k_1$  and  $k_2$  since non-trivial variations of their values had little impact on the results. Specifically, values in the range  $0.01 < k_1 < 0.34$  and  $0.66 < k_2 < 0.99$  worsen the ranking by moving the target method only 1 place below in average.

#### B. Subject Programs

To evaluate our approach we searched for Android apps that presented issues after a framework upgrade on GitHub. We used keywords such as "after upgrade to Lollipop" for the initial selection. Since our evaluation requires both the possibility to reproduce the failures and the knowledge of the location of the fix, we filtered out apps where it was impossible to replicate the upgrade problem (e.g., because it was impossible to compile the project or failure reproduction required a specific hardware). When available we used the official fixes, otherwise we implemented the fix ourselves. We ended up with 12 actual upgrade problems and corresponding Appium test cases that replicate the problem. Table II reports information about the apps, the failures, and the corresponding faults. Columns *Application*, *Ver*, and *Locs* indicate the name, the version of the app (specified with the identifier of the commit), and the number of lines of code of the app, respectively. BossTransfer is a game app [14]. FakeGPS is a GPS device simulator [15]. FilePicker is an app for selecting files and folders in a device [16]. GetBack GPS is an app for storing the location of points of interests [17]. GoodWeather is a weather app [7]. KanjiFix is an app to fix Japanese glyph rendering [18]. MapDemo is an app to test the setup of Google Play services [8]. PoGoIV is an IV calculator for Pokemon Go [19]. PrivacyPolice is an app that prevents leaking of sensitive information via Wi-Fi networks [20]. QuotoGraph is an app to create custom wallpapers [21]. SearchView is a persistent search and view library in material design [22]. ToneDef is a tone dialer application [23].

Column *Inc. API* reports the version of the framework API that is incompatible with the app. In most of the cases it is API 23 since it is also the most used version of the framework [24]. Column *Failure* provides a short description of the failure caused by the incompatibility with the framework API. Finally column *Fault* provides a short description of the fault that causes the failure of the app. The \* indicates that we implemented the fix since the official fix was not available.

TABLE II  
SUBJECT APPS.

| Application   | Ver  | Locs  | Inc. API | Failure                                                  | Fault                                      |
|---------------|------|-------|----------|----------------------------------------------------------|--------------------------------------------|
| BossTransfer  | 47   | 1.3K  | 23       | crash when opening the details about items in a list     | wrong permission logic*                    |
| FakeGPS       | 28   | 3.0K  | 23       | crash when opening the view to set the fake position     | missing permission logic*                  |
| FilePicker    | 115  | 5.8K  | 23       | folders erroneously shown as empty                       | faulty support to the new api              |
| GetBack GPS   | 2133 | 7.0K  | 23       | unable to retrieve current position                      | missing permission logic                   |
| GoodWeather   | 745  | 9.7K  | 23       | hang when refreshing meteo forecast                      | missing permission logic                   |
| KanjiFix      | 46   | 1.3K  | 21       | unable to fix Japanese glyph rendering                   | fonts require a new procedure to be loaded |
| MapDemo       | 5    | 0.6K  | 23       | crash when retrieving the current position               | missing permission logic*                  |
| PoGoIV        | 2328 | 19.2K | 24       | unable to perform the auto update                        | new api requires the use of FileProvider   |
| PrivacyPolice | 153  | 2.5K  | 23       | unable to connect to wifi networks                       | api methods with changed semantics         |
| QuotoGraph    | 289  | 8.4 K | 24       | crash on startup                                         | api methods with changed semantics         |
| SearchView    | 746  | 6.1K  | 21       | crash on startup                                         | api methods with changed semantics         |
| ToneDef       | 91   | 6.8K  | 23       | error message when dialling from the phone contacts list | missing permission logic*                  |

Note that some of these faults have been non trivial to debug, requiring from 1 day to several months to be fixed. Three of the faults also required multiple commits to be fixed (up to 23 commits in one case).

C. *RQ1: Are the suspicious invocation blocks capturing information about the symptoms of failures?*

To answer this research question, we analyzed the content of the SIBs measuring the completeness and soundness of the reported information. To *objectively* identify the set of anomalous boundary calls caused by a fault we exploited the fixed versions of the apps. We executed both the faulty and the fixed apps on the upgraded framework using the test case that reproduces the problem and collected the boundary calls. We then compared the two trace files. Differences correspond to suppressed or novel method calls introduced by the developer to fix the program. We refer to these calls as the *fault-related method calls (frmc)*. Ideally, the SIBs should be able to capture these calls, which would be then exploited to rank methods.

To measure the *soundness* of the content of the SIBs we compute the percentage of blocks that contain fault-related method calls, that is,  $soundness = \frac{\#SIB \text{ with } frmc}{\#SIB}$ . We perform this for both all the SIBs and for the blocks with a minimum length of 2, which are the ones used by FILO unless all the blocks have length 1. To measure the *completeness* of the content of the SIBs, we compute the percentage of fault-related method calls that are included in the SIBs, that is,  $completeness = \frac{\#frmc \text{ in } SIBs}{\#frmc}$ . Table III summarizes the results.

We can notice that the density of SIBs that include fault-related method calls is rather sparse when considering the full set of blocks (column *All SIBs*), ranging from 1% to 100%. However, if we exclude the blocks with a single call, which are the blocks that contribute the least to our analysis, we can see that the density of blocks incorporating information about the fault increases significantly (column *SIBs (length  $\geq 2$ )*): at least half of the blocks are always relevant, with a density of relevant blocks reaching 86% for FakeGPS and 100% for QuotoGraph. There is an exception to this that is KanjiFix. In that case all the SIBs have length 1, thus the analysis can only be performed with the full set of SIBs. Note again that half of the blocks carry relevant information.

TABLE III  
SUSPICIOUS INVOCATION BLOCKS.

| Application   | Soundness |                          | Completeness |
|---------------|-----------|--------------------------|--------------|
|               | All SIBs  | SIBs ( $length \geq 2$ ) |              |
| BossTransfer  | 50%       | 75%                      | 77%          |
| FakeGPS       | 60%       | 86%                      | 79%          |
| FilePicker    | 19%       | 52%                      | 64%          |
| GetBack GPS   | 45%       | 50%                      | 40%          |
| GoodWeather   | 55%       | 61%                      | 49%          |
| KanjiFix      | 50%       | -                        | 30%          |
| MapDemo       | 63%       | 69%                      | 45%          |
| PoGoIV        | 24%       | 63 %                     | 61%          |
| PrivacyPolice | 52%       | 67%                      | 32%          |
| QuotoGraph    | 100%      | 100%                     | 54%          |
| SearchView    | 48%       | 50%                      | 26%          |
| ToneDef       | 1%        | 67%                      | 22%          |

The performance of the SIBs is quite variable in terms of their completeness. For some apps, such as BossTransfer, FakeGPS, FilePicker, and GoodWeather, the SIBs include most of the behaviors related to the fault, while for other apps, such as KanjiFix, SearchView, and ToneDef, the blocks are representative of about one fourth of the behavioral differences introduced with the fix. It is important to emphasize two aspects. First, FILO requires to capture some relevant differences in the executions but does not need to capture all the differences. In fact, as discussed in RQ2, FILO managed to be effective with KanjiFix where only 30% of the relevant method calls have been captured. Second, low percentages do not imply that FILO is missing information present in the baseline and in the failure trace files (the diff procedure used in FILO is essentially complete by construction), but typically correspond to completely new behaviors introduced with the fix that were not present in the previous version of the app.

In summary, although the information contained in the blocks is not noise-free, the blocks, especially the ones with non-negligible weights, are confirmed to carry relevant information about the failure and can be realistically exploited to identify the method responsible for the fault, as shown in RQ2.

D. *RQ2: How well the ranking returned by FILO identifies the method that must be modified to implement the fix?*

The incompatibility problems introduced in our subject apps required the modification of a single method, with the excep-



TABLE IV  
RANKING.

| Application   | #SIB | Pos Ranking |
|---------------|------|-------------|
| BossTransfer  | 4    | <b>2*</b>   |
| FakeGPS       | 21   | <b>5</b>    |
| FilePicker    | 29   | <b>4</b>    |
| GetBack GPS   | 16   | 10*         |
| GoodWeather   | 28   | <b>1,2</b>  |
| KanjiFix      | 6    | <b>1</b>    |
| MapDemo       | 13   | 8           |
| PoGoIV        | 16   | 7           |
| PrivacyPolice | 12   | <b>1</b>    |
| QuotoGraph    | 1    | <b>1</b>    |
| SearchView    | 30   | 9           |
| ToneDef       | 3    | -           |

tion of GoodWeather that required changing two methods. We do not consider the new methods introduced by developers to organize the code of the fix since these methods were not present in the faulty version of the app.

Table IV shows the results. Column *#SIB* indicates the number of SIBs exploited by FILO in the localization, while column *Pos Ranking* indicates the position of the method(s) that must be fixed in the ranking (we never experienced tie positions in the evaluation). Values below or equal to 5 are reported in bold.

The number of SIBs may vary significantly based on the failure. However, FILO has been always able to report the method to be fixed within the top 10 positions, with the exception of ToneDef, where FILO was unable to include the fixed method in the ranking, since the method was not part of the collected stack traces. The BossTransfer and GetBack GPS apps are considered special cases. In the former case the exact method to be fixed is not part of the failure call tree and thus the final ranking does not include that method. However, the method is in an anonymous class that is defined inside another method that occurs at the second place of the ranking. In the latter case the developers placed the fix in a method within an abstract class, whereas FILO detected the override of the same method in the concrete class. Because of these special cases, we reported these results with a mark. In seven cases the method to be changed was ranked among the top 5 positions with four perfect results, that is, the method to be changed is ranked at the top place. This result is particularly good. In fact, practitioners have been reported to consider acceptable inspecting of up to 10 methods, with a preference for techniques that require inspecting 5 methods at most [10].

#### E. RQ3: How does FILO compare to both naive trace analysis and SBFL techniques?

This research question compares FILO to alternative approaches that can be used to localize the method to be fixed. We identified two main alternatives. The first one is what we called naive trace analysis, that is, simply comparing the baseline and failure traces and inspecting anomalous methods calls in the order of occurrence. This approach is included to confirm the need of analyzing the failure in a more sophisticated way, as FILO does, than simply comparing traces.

The second approach is a classic SBFL method: Ochiai [5]. Although there are several alternative formulas that can be used [25], we selected Ochiai because it is one of the most effective methods and a quick investigation based on alternative formulas has not revealed better results.

Note that SBFL techniques have stricter requirements than FILO to be applied and produce a more limited output. In fact, they require a test suite with passing test cases to compute the ranking and do not provide any additional information that can help the developer interpreting the result produced by the technique. On the contrary, FILO only requires a failing execution to be applied and augments the ranking with information about anomalous boundary calls that can help determining the reason of the failure. Since our apps are released without a test suite, in principle SBFL techniques would not be applicable to our cases. To overcome this issue, we generated a test suite of passing test cases with the Monkey automatic testing tool [26]. We used a setting favouring the generation of a rather extensive test suite: we generate test cases by configuring Monkey to emit 10,000 events per test case, which is 200 times the default value, and we run Monkey for 10 minutes per app, which is double than the time that has been empirically reported to produce advances in the coverage [27]. We setup the testing environment to prevent the generation of failures and inspected the test execution to make sure that failing test cases have not incidentally included in the test suite. We collected coverage data at the level of both methods, to be consistent with the granularity of the other approaches, and statements, to investigate the effectiveness at a finer granularity. We then computed the ranking using Ochiai.

Table V reports the position of the method that must be modified to implement the fix in the ranking returned by FILO, naive trace analysis, and Ochiai (since we have two methods to be modified, we have two positions for GoodWeather). Rows *Top-1*, *Top-5* and *Top-10* indicate the number of times each technique has ranked the target method in the top, top 5 and top 10 positions, respectively. Row *Not in the ranking* reports the number of times a technique has not included the target method in the ranking (marked with a “-” in the respective row). For each row, the best result is shown in bold.

Naive trace analysis achieved the worst results. In six cases the method with the fix was not present in the ranking and in the other cases the number of methods to be inspected before reaching the target method was incredibly high, consisting of hundreds of entries in several cases. This result confirms the unsuitability of simple trace analysis.

Ochiai performed better than FILO only twice, with the ToneDef and MapDemo apps. ToneDef escaped to FILO, while could still be addressed with Ochiai. The result obtained with MapDemo is due to the characteristic of the fault that is systematically activated every time a specific statement is executed. This case is favourable to SBFL techniques.

In all the other cases, FILO outperformed Ochiai. In three cases, GetBack GPS, QuotoGraph, and SearchView, Ochiai (both methods and statements) could not generate any ranking since the apps failed during startup and it was impossible to



produce a suite of passing test cases. On the contrary, FILO effectively ranked the method to be fixed at the 10<sup>st</sup>, 1<sup>st</sup> and 9<sup>th</sup> position of the ranking. In another case, FilePicker, Ochiai (statement) could not localize the fault because the fault is due to missing statements, while FILO effectively ranked the target method at the 4<sup>th</sup> position. In the remaining cases, FILO always ranked the target method significantly better than Ochiai (both methods and statements).

Overall, FILO was not able to rank the fix only once (1 out of 12). When successful, FILO always ranked the target method below position 10 (11 out of 12 cases), ranking it at the top in four cases and in the top five positions in seven cases. Ochiai (method) failed to include the target method in the ranking three times, achieving a perfect ranking only twice, and missed to rank the target method in the top 10 positions in 9 out of 12 cases. Ochiai (statement) failed to include the right statement in the ranking in five cases, achieved only one perfect ranking, and missed to include the statement in the top 10 places in all the rest of the cases.

Let us remark that FILO is cheaper to execute than SBFL. In fact it can process traces in few seconds and its cost mainly depends on the execution of the failing test case. In contrast, SBFL techniques require the execution of complete test suites with programs instrumented to collect full coverage data, which is order of magnitude more expensive.

In summary, the results show that FILO is more effective than naive trace analysis and SBFL with problems introduced by framework upgrades.

TABLE V  
COMPARISON BETWEEN FILO, NAIVE TRACE ANALYSIS AND OCHIAI.

| Application        | FILO       | Naive Trace Analysis | Ochiai (method) | Ochiai (statement) |
|--------------------|------------|----------------------|-----------------|--------------------|
| BossTransfer       | <b>2</b>   | 138                  | 4               | 32                 |
| FakeGPS            | <b>5</b>   | 328                  | 13              | 65                 |
| FilePicker         | <b>4</b>   | 203                  | 81              | -                  |
| GetBack GPS        | <b>10</b>  | -                    | -               | -                  |
| GoodWeather        | <b>1,2</b> | -                    | 1,32            | 5,5                |
| KanjiFix           | <b>1</b>   | -                    | 19              | 23                 |
| MapDemo            | 8          | 45                   | <b>1</b>        | <b>1</b>           |
| PoGoIV             | <b>7</b>   | -                    | 48              | 283                |
| PrivacyPolice      | <b>1</b>   | 26                   | 21              | 130                |
| QuotoGraph         | <b>1</b>   | -                    | -               | -                  |
| SearchView         | <b>9</b>   | 109                  | -               | -                  |
| ToneDef            | -          | -                    | 24              | <b>13</b>          |
| Top-1              | <b>4</b>   | 0                    | 2               | 1                  |
| Top-5              | <b>7</b>   | 0                    | 3               | 2                  |
| Top-10             | <b>11</b>  | 0                    | 3               | 2                  |
| Not in the ranking | <b>1</b>   | 6                    | 3               | 4                  |

## F. Discussion

A characteristic of FILO compared to other fault localization techniques is its capability to provide supporting evidence of the ranking in the form of anomalous calls that capture the symptoms of the failure. These symptoms can be conveniently used by the tester to better understand the failure and work on the implementation of a fix. In this section, we report

qualitative results by discussing the output produced by FILO for four of the faults analyzed in the empirical evaluation.

In GoodWeather the fix must be implemented in `gpsRequestLocation` and `onOptionsItemSelected`, which are the top ranked methods. This ranking is supported by two anomalous SIBs. Since each block is represented by the first call of the block, the information provided on the first place to the developer consists of two calls: a call to `ContextCompat.checkSelfPermission` and a call to `LocationManager.requestLocationUpdates`. The provided information clearly points at a permission issue (based on the anomalous call to `checkSelfPermission`) with the location services (based on the anomalous call to `requestLocationUpdates`). Notably the fix exactly consists of adding the code to grant the permission to access the location services from the `gpsRequestLocation` method.

A similar case is MapDemo where FILO associates the method that must be modified to implement the fix with an anomalous call to `Location.getLatitude`, which returns `null` because of the lack of permissions and the fix consists again of adding the code necessary to obtain this permission.

Another interesting case is PrivacyPolice. The fix is implemented in method `ScanResultsChecker.onReceive`, which is ranked at the top. The SIBs associated with these methods share the presence of the `WifiManager.getScanResults`, which is the framework method that experienced the behavioural change (always returns `null` if the GPS is disabled) and leads the app to a malfunction. The fix requires handling the result produced by this method properly.

A particular case is represented by QuotoGraph, which crashes on startup without producing a trace. The result is the whole baseline trace treated as a single block of suspicious invocations clearly suggesting problems with the initialization of the app. FILO correctly pointed at the `LWQApplication.onCreate` method, which in fact has been fixed modifying the initialization procedure of the app.

## G. Threats to Validity

There are two main threats affecting the validity of our evaluation. One is an internal validity threat and is about the comparison to Ochiai. As discussed, FILO only requires a failing execution to be applied, while Ochiai, and other SBFL techniques, needs a test suite of passing test cases to be applied, and such a test suite was not available for the apps used in our evaluation. This case is quite frequent in practice, for instance the vast majority of the apps in GitHub are developed without having an associated test suite. This practical scenario confirms the value of FILO being independent on suites of passing test cases. To anyway obtain information about the comparison of FILO to SBFL we derived a quite extensive test suite of passing test cases for each app and then applied SBFL. In principle, since the outcome of the localization depends on the test suite, we cannot know if the results would be different using another test suite. However, we worked conservatively generating as many passing test cases as possible to obtain the best localization from Ochiai, so this is unlikely to happen.

The second threat is an external validity threat and concerns the generalization of the reported evidence. FILO performed consistently good in the studied incompatibilities and the steps of the analysis are based on general concepts, without including any ad-hoc optimization. This provides a good degree of confidence on the general validity of the results.

#### H. Limitations

Although FILO performed well with almost all the subjects, there are cases that cannot or can be extremely hard to address with our technique. We discuss three relevant cases below.

*Faulty method outside the set of collected stack traces:* In principle a faulty method might be missing from all the collected stack trace instances. FILO collects a number of these instances, one for each SIB, thus it is unlikely that the faulty method falls outside every stack trace instance. In our evaluation happened only in 1 case out of 12.

*Faulty configuration:* In some cases the fix might require changing a configuration file of the app rather than changing the app. These faults are outside the scope of FILO.

*New callback methods:* Problems caused by new callback methods, not existing in the base environment, cannot be detected by FILO. In some cases, these problems might be however introduced together with other faulty changes that FILO can detect, such as in the case of GoodWeather.

### V. RELATED WORK

Framework upgrades are frequent in mobile ecosystems and Android in particular [1], [2]. Many of these upgrades are intentionally *not* backward compatible and app developers struggle adapting their apps to newer versions of the framework, as witness by the many discussions opened on Stack Overflow every time a framework upgrade is released [28]. When these upgrades are not handled properly, apps might be affected by fragmentation-induced compatibility issues [3], as well as maintenance and security issues [29].

Many techniques focused on the *detection of the incompatibilities* introduced with these upgrades. For instance, CiD [30] builds API lifecycle models based on the revision history of the Android framework and uses these models to detect incompatibilities. In their work Mostafa et al. [4] detect behavioral backward incompatibilities in Java libraries, including Android, by running regression tests and checking bug reports. Similarly Mora et al. [31] defined an approach based on the lazy exploration of the behavioral space to assess if a library update may impact on the clients. DiffDroid detects inconsistent app behaviors across devices [32]. All these approaches focus on the *detection of misbehaviours*. On the contrary, FILO identifies *the place where the fix should be implemented* and *the symptoms* are used to assist the work of the developers who investigate the failures.

Some techniques can be used to reduce the compatibility problems. For instance, ReBA [33] is a technique for the development of libraries augmented with adapters to guarantee backward compatibility. While this might be an option for the developers who want to put extra effort on releasing backward

compatible components, in many practical cases developers intentionally release upgrades that are not backward compatible.

Instead of detecting failure symptoms, fault localization techniques can be used to attempt to localize faults. A number of them use the coverage profile of the passing and failing test cases to perform the localization. These techniques are usually referred to as *spectrum-based fault localization* (SBFL) [25]. Notable examples are Tarantula [6], Ochiai [5], and Zoltar [34]. These techniques can be potentially used to address a broader class of faults than FILO, in fact they are not limited to the case of framework upgrades. On the other hand, FILO is more effective than SBFL both in terms of its applicability, indeed FILO *does not require a test suite of passing test cases*, and its effectiveness, since it produces *failure symptoms* and *a more accurate localization*, based on our evaluation. Furthermore, SBFL has shown limitations [35]–[37] and can hardly satisfy the requirements that a practical fault localization approach should satisfy based on the opinion of practitioners [10]. On the contrary, FILO can be executed quickly, can produce accurate rankings, and can isolate symptoms that help understanding the failure and interpreting the ranking, which is a key requirement already highlighted in empirical studies [37]. Finally, API documentation could also be exploited to facilitate the API migration process [38].

Finally, FILO exploits *anomaly detection* in the analysis [39]. There exist a number of solutions that analyze and compare the behaviors of programs and components to identify anomalies that can be used to support the debugging activity. For instance, BCT [40], RADAR [41], and the technique by Pradel and Gross [42] perform this kind of analysis in the context of component-based systems, regression testing, and object-oriented software, respectively. Mimic performs a similar analysis in the attempt to analyze reproduced field failures [43]. Differently from these techniques, FILO originally combines fault-localization and anomaly detection, exploiting anomalies to both perform the localization and augment the ranking with symptomatic information about the failure.

### VI. CONCLUSIONS

Timely fixing problems caused by framework upgrades is important to make mobile apps compatible with the latest releases of the operative systems. FILO can assist developers when performing this task by automatically identifying the faulty method that must be fixed to solve the compatibility issue, and reporting selected anomalous events observed in the failing execution, to facilitate the analysis of the problem.

The evaluation shows that FILO can be accurate and practical. Moreover it has weaker requirements and higher effectiveness than SBFL in the domain of faults caused by framework upgrades. As part of future work, we intend to investigate the possibility of applying FILO to other contexts, such as library evolution, to extend our approach with automatic program repair capabilities [44], and to experiment with fixes that span multiple methods and files.

*Acknowledgements:* This work has been partially supported by the H2020 ERC CoG Learn project (grant agreement n. 646867).

## REFERENCES

- [1] T. McDonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the android ecosystem," in *Proceedings of the International Conference on Software Maintenance (ICSM13)*, 2013.
- [2] "Android Version History," [https://en.wikipedia.org/wiki/Android\\_version\\_history](https://en.wikipedia.org/wiki/Android_version_history), [Online; accessed November 2018].
- [3] L. Wei, Y. Liu, and S.-C. Cheung, "Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps," in *Proceedings of the International Conference on Automated Software Engineering (ASE16)*, 2016.
- [4] S. Mostafa, R. Rodriguez, and X. Wang, "Experience Paper: A Study on Behavioral Backward Incompatibilities of Java Software Libraries," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA17)*, 2017.
- [5] R. Abreu, P. Zoetewij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION07)*, 2007.
- [6] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Proceedings of the International Conference on Automated Software Engineering (ASE05)*, 2005.
- [7] GitHub, "GoodWeather," <https://github.com/qqq3/good-weather>, [Online; accessed November 2018].
- [8] —, "MapDemo," <https://github.com/MiniPlaneterKUET/MapDemo>, [Online; accessed November 2018].
- [9] Linux, "Diff," <http://man7.org/linux/man-pages/man1/diff.1.html>, [Online; accessed March 2019].
- [10] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA16)*, 2016.
- [11] IDC, "Smartphone OS," <https://www.idc.com/promo/smartphone-market-share/os>, [Online; accessed November 2018].
- [12] J. Foundation, "Appium," <http://appium.io/>, [Online; accessed November 2018].
- [13] rovo89, "Xposed," <http://repo.xposed.info/>, [Online; accessed November 2018].
- [14] GitHub, "Boss Transfer," <https://github.com/AriaLyy/BlogDemo/tree/07f035fe0df6d2747795ef5c11f9120bd5a3b69d>, [Online; accessed November 2018].
- [15] —, "FakeGPS," <https://github.com/xiangtailiang/FakeGPS>, [Online; accessed November 2018].
- [16] —, "FilePicker," <https://github.com/DeveloperPaul123/FilePickerLibrary>, [Online; accessed November 2018].
- [17] —, "GetBack\_GPS," [https://github.com/ruleant/getback\\_gps](https://github.com/ruleant/getback_gps), [Online; accessed November 2018].
- [18] —, "KanjiFix," <https://github.com/ascendedguard/android-kanji-fix>, [Online; accessed November 2018].
- [19] —, "PoGoIV," <https://github.com/farkam135/GoIV>, [Online; accessed November 2018].
- [20] —, "PrivacyPolice," <https://github.com/BramBonne/privacypolice>, [Online; accessed November 2018].
- [21] —, "QuotoGraph," <https://github.com/stanidesis/quotograph>, [Online; accessed November 2018].
- [22] —, "SearchView," <https://github.com/lapism/SearchBar-SearchView>, [Online; accessed November 2018].
- [23] —, "ToneDef," <https://github.com/Fortyseven/ToneDef>, [Online; accessed November 2018].
- [24] Android, "Distribution Dashboard," <https://developer.android.com/about/dashboards/>, [Online; accessed November 2018].
- [25] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.
- [26] "Monkey," <https://developer.android.com/studio/test/monkey>, [Online; accessed November 2018].
- [27] S. R. Choudhary, A. Gorla, and A. Orso, "Automated Test Input Generation for Android: Are We There Yet?" in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE15)*, 2015.
- [28] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "How Do API Changes Trigger Stack Overflow Discussions? A Study on the Android SDK," in *Proceedings of the International Conference on Program Comprehension (ICPC14)*, 2014.
- [29] L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia, and J. Klein, "Characterising Deprecated Android APIs," in *Proceedings of the International Conference on Mining Software Repositories (MSR18)*, 2018.
- [30] L. Li, T. F. Bissyandé, H. Wang, and J. Klein, "CiD: Automating the Detection of API-Related Compatibility Issues in Android Apps," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA18)*, 2018.
- [31] F. Mora, Y. Li, J. Rubin, and M. Chechik, "Client-specific equivalence checking," in *Proceedings of the International Conference on Automated Software Engineering (ASE18)*, 2018.
- [32] M. Fazzini and A. Orso, "Automated cross-platform inconsistency detection for mobile apps," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE17)*, 2017.
- [33] D. Dig, S. Negara, V. Mohindra, and R. Johnson, "ReBA: A Tool for Generating Binary Adapters for Evolving Java Libraries," in *Proceedings of the Companion of the International Conference on Software Engineering (ICSE08)*, 2008.
- [34] A. Ribeiro and R. Abreu, "The GZoltar project: A graphical debugger interface," in *Proceedings of the Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC-PART10)*, 2010.
- [35] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Leveraging test generation and specification mining for automated bug detection without false positives," in *Proceedings of the International Conference on Software Engineering (ICSE17)*, 2017.
- [36] F. Steimann, M. Frenkel, and R. Abreu, "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA13)*, 2013.
- [37] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA11)*, 2011.
- [38] M. Lamothe and W. Shang, "Exploring the use of automated api migrating techniques in practice: an experience report on android," in *Proceedings of the International Conference on Mining Software Repositories (MSR18)*, 2018.
- [39] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys*, vol. 41, no. 3, pp. 15:1–15:58, 2009.
- [40] L. Mariani, F. Pastore, and M. Pezzè, "Dynamic analysis for diagnosing integration faults," *IEEE Transactions on Software Engineering*, vol. 37, no. 4, pp. 486–508, 2011.
- [41] F. Pastore, L. Mariani, A. Goffi, M. Oriol, and M. Wahler, "Dynamic analysis of upgrades in C/C++ software," in *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE12)*, 2012.
- [42] M. Pradel and T. R. Gross, "Leveraging test generation and specification mining for automated bug detection without false positives," in *Proceedings of the International Conference on Software Engineering (ICSE12)*, 2012.
- [43] D. Zuddas, W. Jin, F. Pastore, L. Mariani, and A. Orso, "MIMIC: locating and understanding bugs by analyzing mimicked executions," in *Proceedings of the International Conference on Automated Software Engineering (ASE14)*, 2014.
- [44] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering (TSE)*, vol. 45, no. 1, pp. 34–67, 2019.