information

MDPI

Article

FILO: Automated FIx-LOcus Identification for Android Framework Compatibility Issues

Marco Mobilio, Oliviero Riganelli *, Daniela Micucci and Leonardo Mariani

Department of Informatics, Systems, and Communication, University of Milano-Bicocca, Viale Sarca 336, 20126 Milan, Italy; marco.mobilio@unimib.it (M.M.); daniela.micucci@unimib.it (D.M.); leonardo.mariani@unimib.it (L.M.)

* Correspondence: oliviero.riganelli@unimib.it

Abstract: Keeping up with the fast evolution of mobile operating systems is challenging for developers, who have to frequently adapt their apps to the upgrades and behavioral changes of the underlying API framework. Those changes often break backward compatibility. The consequence is that apps, if not updated, may misbehave and suffer unexpected crashes if executed within an evolved environment. Being able to quickly identify the portion of the app that should be modified to provide compatibility with new API versions can be challenging. To facilitate the debugging activities of problems caused by backward incompatible upgrades of the operating system, this paper presents FILO, a technique that is able to recommend the method that should be modified to implement the fix by analyzing a single failing execution. FILO can also provide additional information and key symptomatic anomalous events that can help developers understand the reason for the failure, therefore facilitating the implementation of the fix. We evaluated FILO against 18 real compatibility problems related to Android upgrades and compared it with Spectrum-Based Localization approaches. Results show that FILO is able to efficiently and effectively identify the fix-locus in the apps.

Keywords: Android fragmentation; fault localization; regression testing; software evolution

Citation: Mobilio, M.; Riganelli, O.; Micucci, D.; Mariani, L. FILO: Automated FIx-LOcus Identification for Android Framework Compatibility Issues. *Information* **2024**, *15*, 423. https://doi.org/10.3390/info15080423

Academic Editors: Sanjay Misra, Robertas Damaševičius and Bharti Suri

Received: 15 June 2024
Revised: 11 July 2024
Accepted: 16 July 2024
Published: 23 July 2024

1. Introduction

Mobile frameworks are continuously updated to release new features that exploit the latest hardware and software upgrades. Evolution is well-known to happen at a dramatic speed. For instance, the Android API framework evolved at the average rate of 115 API updates per month [1] between APIs at levels 3 to 15, and the rate even increased between APIs 16 to 30, reaching up to an average of 160 API changes when considering methods and field changes, additions, and removals [2].

Unfortunately, the speed of evolution of the Android framework can be a source of issues for the apps that must able to cope with frequent, and partially unexpected, changes. For example, 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].

Adapting apps to a changed API can be particularly expensive since it requires understanding how the changes in the underlying framework have impacted the app, and developing the logic necessary to deal with the updated implementation. The cost of addressing an evolving API is also confirmed in the study by McDonnell et al. [1] who report an average app migration time of 16 months, in contrast with an API release interval of a few months only. Mahmud et al. also confirmed that API field updates (that must be addressed in the app code) take more than three months to get fixed [2].

Reducing the cost and effort of migrating apps to updated versions of the underlying framework is extremely important to improve both the quality of the available apps and their degree of compatibility with the most recent version of the operative system. 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 becomes incompatible with 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 to be applied, our approach, namely FILO, requires only a single failed GUI 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, which is frequently true for mobile apps, (ii) it can be straightforwardly applied to those cases where the failure is exposed through GUI interaction, such as an automatic test case derived from a bug report entered by a user, and (iii) it is fast to execute since it avoids the execution of large test suites.

In contrast with SBFL techniques that can only localize suspicious code regions, FILO isolates information about the anomalous app behaviors 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 the definition of FILO is twofold:

- The interactions between the framework and the app must contain evidence of the failure: Since the incompatibility is between an app and its API framework, the problem must be intuitively 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 and the incompatible versions of the framework can be used to identify the suspicious interactions that are in turn useful to identify the code regions that originated them, and are thus likely responsible for the failure.
- The method that must be fixed is likely responsible for a large and coherent set of suspicious interactions: Since the faulty code in the app must be the source of the incorrect interactions between the app and the framework, the method to be fixed must be a method internal to the app that controls the execution of a significantly large and coherent set of suspicious interactions. Based on this intuition, FILO generates a ranked list of methods that the developers can exploit to ease fix location. Each method is also associated with the suspicious interactions under its influence to provide insights about the rationale of the selection.

We empirically assessed FILO with 18 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 five positions in the large majority of the cases, with several cases where the method occurred in the top part of the ranking. We also compared FILO to SBFL, confirming its higher effectiveness in addition to its higher applicability.

This paper extends our earlier conference paper [7] in several ways: (1) we generalize the analysis implemented in FILO by allowing a deeper exploration of the interactions between the apps and their framework (i.e., we consider not only direct interactions, but also the internal computations caused by the direct interactions); (2) we improve the presentation by better framing the three phases of FILO and presenting highlights of the underlying algorithms; (3) we extend the empirical evidence by increasing the number of apps used in our evaluation and studying the impact of FILO parameters in its performance, to demonstrate its resilience and generalization capability; and (4) we present a more detailed and extensive comparison to related work.

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

- FILO, a technique that addresses the incompatibilities between an app and an updated framework by producing a ranked list of suspicious methods, associated with supporting evidence about their selection, from a single GUI test case;
- 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 accessed on 10 June 2024) that can be used to replicate the results reported in the paper.

The rest of the paper is organized as follows. Section 2 presents a running example that is used throughout the paper to illustrate our approach. Sections 3 and 4 describe FILO and its prototype implementation to be used for validation, respectively. Section 5 describes the empirical evaluation of FILO. Section 6 discusses related work. Finally, Section 7 provides final remarks.

2. Running Example

In this section, we describe *Good Weather*, one of the Android apps that suffers from an upgrade issue from the list of subjects used in the evaluation phase. The issue is due to a backward incompatible behavioral change that happened between the Android API 22 and the Android API 23. This change forces apps to explicitly request permissions when accessing resources for the first time, while before it was enough to list them in the Manifest file. The Good Weather app, being a weather forecast app, exploits the location of the phone to give a contextualized forecast. The app has been installed by more than 10,000 Android users according to Google Play, demonstrating how relevant it is to address incompatibilities between framework versions. We used this app both in the empirical evaluation and in Section 3 to illustrate how FILO works.

The method with the incorrect implementation can be seen in Listing 1 by not considering the red code, which are instructions added to obtain the fix, but also considering the code with the strikethrough font, which represents the instructions removed to obtain the fix.

Listing 1. The Fix for the Good Weather app.

```
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 the faulty implementation, the `gpsRequestLocation` method invocation checks for permissions to access both the fine and coarse-grained locations. If those permissions are granted (as it happens with API versions ≤ 22), the location is regularly updated by the execution of the method `requestLocationUpdates()`. Otherwise, in case the permissions are not granted (as it happens with API versions ≥ 23), the method returns without updating the location, resulting in the app hanging forever waiting for the update. The hang is actually caused by the code responsible for removing the progress bar that is executed only once the location has been updated. The graphical result is that the progress bar is never dismissed and it blocks the user interface from any other interaction.

To correctly implement this feature and remove the fault that appears in API ≥ 23 it is necessary to explicitly ask the user to grant access to the location information before the invocation of the `requestLocationUpdates` method. The developers of Good Weather obtained a viable fix by modifying the `onOptionsItemSelected` and `gpsRequestLocation` methods, making them invoke the new methods designed to ask and acquire the permissions.

Given this description, proper analysis and fix-locus identification should report the `gpsRequestLocation` and the `onOptionsItemSelected` methods as the methods to be modified to obtain the fix. Methods that should be implemented but are not present in the app cannot be reported by design.

In this case, FILO successfully reports `gpsRequestLocation` and `onOptionsItemSelected` between the first and third method in the ranking in all the configurations that include interactions from the framework toward the app (see Section 5.3 for further details). In addition to identifying the fix-locus, 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 proper permission to access the location service. In fact, as ancillary information for those methods, FILO reports the `checkSelfPermission` method, which is the framework method invoked in `gpsRequestLocation` and part of the code that needs to be modified.

Finally, it is interesting to note that the failing execution per se is not explicative of the fact that the problem is about permissions: the user can only see the app loading view, without logging any error message. This is why the output of FILO can be helpful to ease the app fixing task.

3. FIx-LOcus

The main goal of FILO is to support developers in fixing problems caused by the upgrades to the Android Framework. FILO achieves this by automatically computing and recommending a list of potential fix locations ordered by their likelihood, annotated with supporting evidence that specifies the suspicious interactions between the app and the underlying framework that a faulty method may have originated.

FILO only requires a GUI test case that reproduces the problem to run the analysis. To analyze a failure, FILO compares the behavior obtained by running the input GUI test case with the Android API that causes the failure, to the behavior shown by running the same test case on a (close) version of the API that does not cause the problem. Since compatibility problems are typically introduced when upgrading the Android framework, the GUI test case is executed on both the current (problem-free) version of the Android API and the new (problematic) one.

Figure 1 shows the three main phases of FILO:

1. The *Test Execution* phase executes the GUI test case on the two Android APIs to collect the interactions between the app and the two Android frameworks. These interactions are collected as execution traces.
2. The *Anomaly Detection* phase identifies the blocks (i.e., the contiguous sequences) of suspicious interactions by comparing the two execution traces.
3. The *Fix Locus Candidates Identification* phase processes the suspicious interactions to return a list of likely faulty methods of the app, with the methods being associated with the corresponding suspicious blocks as supporting evidence of the suspicious interactions that motivate the selection.

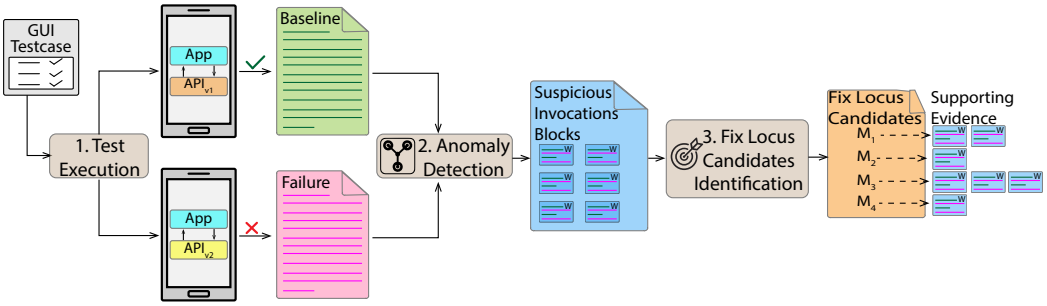

Figure 1. Overview of the FILO technique.

The next subsections provide details of each phase.

3.1. Test Execution

During the *Test Execution* phase, FILO collects the interactions between the app and the framework, both for the *base environment*, which runs the target app with the compatible version of the framework API (the test case passes), and the *failure environment*, which runs the same target app with an incompatible version of the framework API (the test case fails). From now on, we will refer to the *base environment* and the *failure environment* as *v1* and *v2*, respectively.

The underlying idea of FILO is that incompatibilities should be observable in the invocations of API methods from the app and the callbacks from the framework. For this reason, the test execution phase records every *interaction* between the framework API and the app. All the other invocations, that is, calls internal to the framework and calls internal to the app, are considered only to gain additional knowledge about the consequences of the interactions between the app and the framework.

We can define a *framework method* as a method defined within the Android framework or in a standard library (e.g., `java.*`); conversely an app method is a method implemented in the app's own source code. Therefore, an *API call* is a call to a framework method originated from an app method. Similarly, a *callback* is an invocation of an app method from a framework method. All other invocations are internal method calls.

Figure 2 represents the different kinds of invocations mentioned. For example, the invocation $am_1 \rightarrow fm_1$ represents an API call, while the invocation $fm_2 \rightarrow am_2$ represents a callback. Calls $am_1 \rightarrow am_3$ and $fm_1 \rightarrow fm_3$ represent internal calls within the app and the framework, respectively.

We refer to API calls and callbacks as *direct interactions*, while we refer to calls internal to either the framework or the app as *indirect interactions*. When an indirect interaction occurs, we define it as a *depth level x interaction* if it is the xth indirection interaction in a row after a direct interaction. More rigorously, given the invocation chain $\ldots \rightarrow m_k \rightarrow m_{k+1} \rightarrow m_{k+2} \rightarrow \ldots \rightarrow m_{k+n} \rightarrow \ldots$, the invocation m_j is an indirect interaction of depth level x if every invocation m_i with $i \in [j-x+1, j]$ is an indirect invocation and m_{j-x} is a direct invocation.

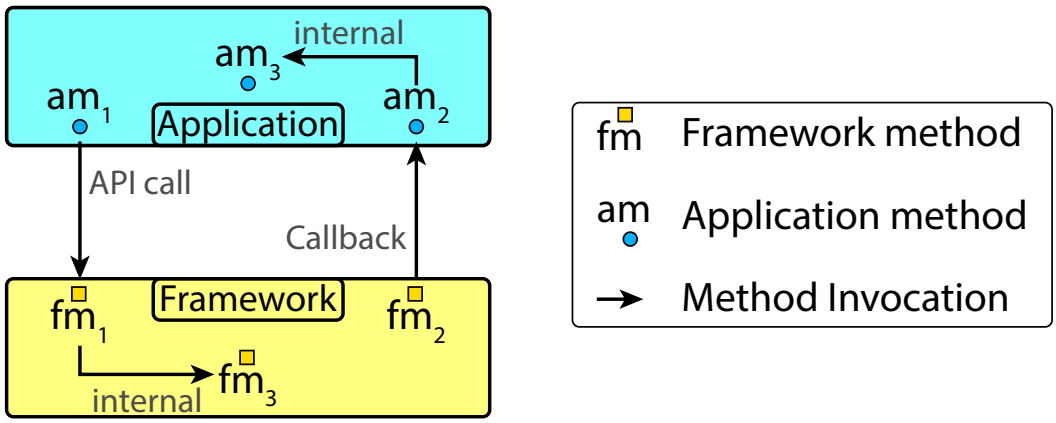

Figure 2. Interaction types.

The depth of an indirect interaction can be intuitively used to extend the monitoring of the interactions between the app and its framework to deeper levels, collecting additional evidence about the consequence of method calls. To this end, we define the concept of *boundary calls* as the set of all the method calls that occur near the boundary between the app and the framework. More rigorously, the boundary calls of level x consist of the union of the direct interactions with the indirect interactions of level $x-1$, where 0 means not recording the indirect interactions. FILO can monitor the software to collect every boundary call up to a given level x.

The output of the test execution phase consists of two traces containing all the boundary calls recorded during the execution of the test case in the base and failure environments. We refer to the trace obtained by running the test case within the base environment as *baseline trace*, and it represents how the framework and the app interact with each other when the execution is correct. Conversely, the *failure trace* is obtained by running the same test case with the same version of the app, but running it in the *failure* environment, and it represents how the app and the framework interact with each other when the execution fails.

Listing 2 shows an excerpt of a baseline trace. FILO traces when the invocation of a method both starts and ends (marked with `#b` and `#e`, respectively) regardless of the type of invocation. If the invocation has a return value, its value is appended at the exit invocation point, as it happens to the invocation of the method `findViewById()` in Listing 2. It is noteworthy that the actual value is returned in the case of primitive types, for non-primitive types FILO records the output of the method `toString()` if overridden, otherwise, only the dynamic type of the returned value is written in the trace. Furthermore, Listing 2 differentiates between callbacks and API calls, with the only callback present, namely `MainActivity.onCreate()`, written in italics. The rest of the invocations are API calls, which in the context of this example, means that the `MainActivity.onCreate()` method (implemented within the app) invokes all the other methods in the listing (which are implemented in the `AppCompatActivity` class, belonging to the Android Framework).

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.setSupportActionBar(...)#b
AppCompatActivity.setSupportActionBar(...)#e
MainActivity.onCreate()#e
```

FILO expects the *exact* same test to be executed in the two environments and restricts the observations to the chosen level of interaction. This setup reduces incidental differences between the two traces being analyzed, which implies that the vast majority of the differences that are actually detected are relevant to the fault under analysis. If large amounts of non-deterministic interactions are present in the traces, they can be filtered out by repeating the test execution multiple times within the same environment and then removing those parts of the traces from the analysis. We, however, never observed relevant changes in the traces collected for the repeated executions of the same test during our experiments.

It is important to report that, even limiting the scope of observation to boundary calls of depth 1, the size of the traces acquired and analyzed by FILO can be significant. As an example, the baseline trace for Good Weather (the running example), contains between 33 and 70,100 entries (configurations A1F0 and A2F2, respectively, see Section 5) with sizes going from 39 KB up to 231 MB. Within our experiment, the largest trace produced belongs to the Inventory Agent app [8], with up to 730,090 entries and ∼2.0 GB in terms of disk space.

Algorithm 1 summarizes the test execution phase. The core of this phase is represented by the `run_test()` function, which boots the required emulator, configures the monitor to collect the right set of boundary calls, and then runs the available test case, finally retrieving the generated trace. This process is repeated twice in order to acquire both the baseline and the failure traces.

Algorithm 1: Test Execution

Input:
t: failing testcase
$env_{baseline}$: the baseline environment
$env_{failure}$: the failure environment
k: depth level of interaction
Output:
baseline: trace generated by t in $env_{baseline}$
failure: trace generated by t in $env_{failure}$

```
1  Function test_execution(t, env_baseline, env_failure, k):
2      start_appium()
3      baseline = run_test(t, env_baseline, k)
4      failure = run_test(t, env_failure, k)
5      stop_appium()
6  Function run_test(t, env , k):
7      boot(env)
8      configureMonitor(k)
9      execute_junit_test(t)
10     trace = retrieve_trace()
11     shutdown(env)
12     return trace
```

3.2. Anomaly Detection

The *Anomaly Detection* phase compares the two traces produced by the test execution phase to isolate the most suspicious invocations. In particular, anomaly detection focuses on invocation blocks. An *Invocation Block* is a contiguous sequence of interactions extracted from a trace. More formally, given a recorded trace $e_1, \ldots e_n$ of boundary calls, we define an invocation block as *any sub-sequence* $e_i, e_{i+1}, \ldots e_{i+k}$, *s.t.*, $i \geq 1$ *and* $i + k \leq n$.

During the anomaly detection phase, FILO identifies the differences between the baseline trace and the failure trace. It also groups these differences and assigns them weights based on their size. Since the compared traces contain the interactions collected while running the same test case, the detected differences are very likely due to the changes that happened within the underlying framework and their impact on the app.

FILO firstly identifies these differences by using *diff* [9], which returns the invocation blocks in the failure trace that have no counterpart in the baseline trace. Since these blocks are the main manifestation of the problem caused by the updated framework, we refer to them as the *Suspicious Invocation Blocks* (SIBs).

The set of differences represented by SIBs is usually rather large. In fact, a single different behavior of the framework may produce several differences in the flow, parameters, and return values of boundary calls. For example, the traces generated from the running example contain hundreds of differences grouped in 61 SIBs.

To categorize and analyze the content of the SIBs, FILO undertakes two distinct operations: firstly, it designates a single method invocation within each SIB to serve as its representative, and secondly, it assigns a weight to each SIB according to the number of interactions it encompasses. Consequently, larger SIBs are deemed to be more significant and informative, as opposed to sporadic differences in the execution traces.

An example of SIB is shown in Listing 3. This SIB starts with an invocation to the method `MainActivity.onLocationChanged`, which is chosen as a representative for the whole block (shown in red). Since the SIB is composed of 30 invocations (omitted in the listing), FILO assigns a total weight of 30 to the SIB, which is also shown in Listing 3 near the representative invocation.

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
...
```

The final output of this phase is thus a series of weighted SIBs that FILO exploits to identify the fix-locus in the next phases. Note that usually, the anomaly detection phase identifies several blocks. For instance, in the running example, FILO isolates between 1 and 61 different SIBs, depending on the depth level of interaction considered. In the Good Weather app, the vast majority of the SIBs are yielded by the interactions from the framework toward the app. While this behavior is common among most of the apps we analyzed, there are some cases in which the contribution of the invocations from the app toward the framework is more relevant than their counterparts, an example is the FilePicker app.

Finally, it is noteworthy that the actual fix-locus method is not usually representative of a SIB and, in fact, it often does not appear as an invocation at all nor in the baseline, or in the failure trace. For instance, one of the methods that should be modified in the running example is `gpsRequestLocation()` and this method never occurs in the traces. This is confirmed in our empirical evaluation where in about 50% of the cases the fix-locus is not included in any trace at all.

Algorithm 2 shows the detailed execution of the anomaly detection phase. It starts by running the Linux `diff` tool. Then, it identifies the SIBs from the diff (function `get_SIBs`) by creating a SIB for each difference occurring in the traces, assigning to it the first method of the block as representative and the length of the block as weight (lines 7 and 8, respectively).

Algorithm 2: Anomaly Detection

```
Input:
baseline: the baseline trace
failure: the failure trace
Output:
SIBs: the list of weighted SIBs
Function anomaly_detection(baseline, failure):
    diff_trace = linux_diff(baseline, failure)
    SIBs = get_SIBs(diff_trace)
    return SIBs
Function get_SIBs(diff_trace):
    for each diff_block in diff_trace do
        representative = first(diff_block)
        weight = len(diff_block)
        SIBs.append(new SIB(representative, weight, diff_block)
    return SIBs
```

3.3. Fix-Locus Candidates Identification

The *Fix-Locus Candidates Identification* phase processes the set of weighted SIBs and creates a ranked list of app methods that represent the fix-locus candidates, ordered according to the likelihood of pointing at the faulty method. The ranking of each method is explained by the set of SIBs that are under its influence. Intuitively, the SIBs associated with a specific method in the ranking represent the failure *symptoms* that resulted from the execution of that method. That is, these symptoms could be potentially eliminated by fixing the method, in case it is faulty. The SIBs associated with a method are also useful to the developers, since they add contextual information to the identification of fix-locus.

The ranked list of suspicious methods is created in two main steps from the SIBs. In the first step, *construction of the failure call tree*, the set of app methods that might be responsible for a SIB is determined by inspecting the call stack at the time the suspicious calls in the SIBs are generated. In the second step, *ranking*, the app methods are ranked according to their influence on the detected SIBs.

3.3.1. Construction of the Failure Call Tree

The fix-locus is not necessarily in the set of methods that belong to the SIBs. On the contrary, the SIBs include anomalous invocations originated by faulty methods, which are often placed elsewhere in the code of the app. To identify the set of methods that must be taken under consideration, FILO builds a *Failure Call Tree* (FCT). A FCT is a graph that is built from the SIBs representatives by considering their stack trace (this information is collected in the traces when executing the test case). Nodes in the FCT are method invocations and direct edges represent method invocations detected in the stack trace.

More rigorously, for each SIB sib_i with weight w_i and representative call c_i, FILO collects its stacktrace $\langle m_1, \ldots, m_{n_i}, c_i \rangle$, where m_1 is always the method `ZygoteInit.main`. The failure call tree is a triple (N, E, r), where $N = \{m_1, \ldots, m_{n_i}, c_i | \forall sib_i\}$ is the set of nodes, $E = \{(m_{n_i}, c_i), (m_i, m_{i+1}), i = 1, \ldots n_i - 1 | \forall sib_i\}$ is the set of edges, and r = `ZygoteInit.main` is the root of the tree. The root of the FCT is always the `ZygoteInit.main` method as it is the initial method that handles the fork of every app launched within the Android OS. The leaves of the tree are the SIBs representatives and FILO assigns a weight that corresponds to the weight of the corresponding SIBs to each leaf. The higher the weight is, the more relevant that leaf is.

After building the FCT, FILO scores each node in the tree, based on its degree of influence on the SIBs, its number of children, and its distance from the root. This score is the *suspiciousness* of that node and it is used by FILO to create the final ranking.

More formally, we compute the suspiciousness of a method m as a linear combination of three attributes as shown in Equation (1). The sum of $k_1 + k_2 + k_3$ is always 1.

$$Susp(m) = k_1 \times influence(m) + k_2 \times depth(m) + k_3 \times children\#(m) \quad (1)$$

Influence(m) measures the degree of influence of node m in terms of the weights of the leaves in the (sub-)tree rooted by m. It is computed by summing the weights of the leaves that can be reached from m normalized with respect to the sum of all the weights of all the nodes in the tree. Note that by definition, the *root* of the FCT influences all the SIBs, and therefore $influence(root) = 1$. Similarly, a leaf node *sib* that can only influence itself has $influence(sib) = \frac{w}{W}$, where w is the weight of the SIB and W is the sum of all the weights in the FCT.

Depth(m) measures the position of m with respect to the height of the FCT. The value is normalized with respect to the height of the tree itself. Therefore, $depth(root) = 0$ and $depth(sib_d) = 1$, where sib_d is the deepest SIB in the tree.

Children#(m) measure the number of direct children of m. The value is normalized with respect to the node with the higher number of children in FCT, which means that $children\#(m) = \#children_m / \#children_{Max}$ where Max is the node with the highest number of children.

The three scores *influence*, *depth*, and *children#* are designed to suitably interact with one another favoring different characteristics of the methods in the FCT, with the idea that the method that obtains the highest score considering all three dimensions has the highest probability of being the fix-locus.

The *influence(m)* score is defined to identify the nodes that control the execution of a large number of SIBs. Since SIBs are the symptoms of the changes in the execution of the test case, this criterion works under the assumption that the fix-locus *has an impact on a significant number of SIBs*.

The *depth(m)* score is designed to privilege the choice of nodes that are close to the leaves of the FCT, therefore discouraging the selection of methods higher up in the tree. The general idea is that *methods close to the SIBs are preferable to methods close to the root* as they are executed too early during the failure to be relevant.

The *children#(m)* score is designed to favor the nodes that have many direct children, in contrast with those that for instance have one. The intuition is that any node with a large number of children *is a node that acted as a differentiation point for multiple SIBs*, since it occurred in multiple stack traces of multiple SIBs.

The interaction of these three attributes should favor the selection of *nodes that influence the execution of a large and cohesive set of SIBs with relevant weights*. In fact, considering the leaf nodes, it is worth selecting an ancestor node, that is a method executed earlier in the failure, only if the loss in terms of *depth* is compensated by the gain in terms of *influence* and *number of children*.

Algorithm 3 shows the creation of the FCT graph. The process starts with the function `generate_FCT()` that takes the list of SIBs as input. For each SIB, its stack trace is retrieved and then recursively added to the FCT graph starting from the root node. The last node added (the leaf) shall represent the SIB itself. The recursion is handled by the `add_to_FCT()` function, which, for each stack trace method, checks if that method is already present in the set of children of the current FCT node. If the method is present, the function iterates the process with that node as a target. If the method is not present, the function creates a new child and continues by iterating the process with the newly added child. Once a new child is added, the rest of the stack trace will generate a new branch in the FCT.

Algorithm 3: Failure Call Tree Generation

```
Input:
SIBs: the list of weighted SIBs
influence, depth, children#: the weights for the suspiciousness score
Output:
fct: the failure call tree graph
Function generate_FCT(SIBs):
    fct = new FCT()
    for each sib in SIBs do
        stacktrace = sib.representative.get_stacktrace()
        if fct.get_root() == null then
            fct.set_root(stacktrace.pop())
        add_to_FCT(stacktrace, fct.get_root())

    for each fct_node in fct do
        fct.set_susp_score(k1,k2,k3)
    return fct
Function add_to_FCT(stacktrace, current_FCT_node):
    current_method = stacktrace.pop()
    if current_method == null then
        return
    if current_method not in current_FCT_node.children then
        current_FCT_node.add_child(method)
    next_FCT_node = current_FCT_node.get_child(current_method)
    add_to_FCT(stacktrace, next_FCT_node)
```

Finally, the `generate_FCT()` loops on all the nodes in the graphs and using the three weights (*influence*, *depth*, and *children#*) assigns a suspiciousness score to each one.

3.3.2. Ranking

In this step, FILO generates the final ranking that shall include the list of methods ordered by suspiciousness, associated with the SIBs that motivate their ranking.

Since the FCT represents the method calls present in the stack trace at the time of the failure, the same method might be invoked multiple times and appear in multiple places in the tree. The choice here is to consider the most suspicious occurrence of a method call as the suspiciousness of the method. Indeed, a highly suspicious instance of a method is enough to justify its careful inspection, independent from the existence of other, less suspicious, occurrences of calls to the same methods. More rigorously, given a FCT and its set of nodes $nodes(FCT)$, the suspiciousness $susp(m)$ of a method m is $susp(m) = max\ susp(mc)$ with mc calls to m occurring in $nodes(FCT)$.

Since the goal of FILO is to suggest the fix-locus in the app, that is, the place where the app must be adapted to work with the updated framework, the location must necessarily be a method in the app. Thus, FILO discards every invocation to the framework from the final ranking.

The final ranking reported by FILO for the running example when analyzing level 2 interactions from the app toward the framework and level 1 interactions from the framework toward the app is shown in Table 1. 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 is supposed to inspect. As previously mentioned, it is important to note that the top-ranked method never occurred in the set of interactions in the traces, meaning that a trivial comparison of the traces could not have suggested it. We study the effectiveness of direct trace comparison in our empirical evaluation confirming its low effectiveness when supporting fault localization.

Table 1. Ranking returned by FILO (A2F1).

MethodName	Susp
org.asdtm.goodweather.MainActivity.onOptionsItemSelected	0.48
org.asdtm.goodweather.MainActivity.gpsRequestLocation	0.47
org.asdtm.goodweather.MainActivity$1.onLocationChanged	0.05

Of course, there are cases in which the returned ranking may miss the faulty method, for instance, because the fault is in a method that is not part of the stack traces found in the SIBs representatives. This, however, occurs rarely. For instance, in our evaluation, it happened 3 times out of 21 cases, if considering callback levels higher than zero.

4. Prototype

Our prototype implementation consists of two main components: the *Tracer* and the *Analyzer*. The former component records the execution of the app producing the traces with the boundary calls, while the latter component processes the collected traces to produce the final ranking.

To make our experiments and results reproducible we produced a failing test case for each app. Each test case represents the shortest interaction sequence that reveals the incompatibility between the framework and the app. These test cases are implemented as Appium [10] test cases for automatic execution.

Our prototype implements two interfaces: a command–line interface, to ease automation, and a GUI interface integrated within the Android Studio IDE [11], to provide an interface usable by developers. In particular, the integration with the Android Studio IDE is achieved by implementing a plug-in that also provides integrated views that support direct navigation from the ranking to the corresponding code element.

The *Tracer* is a platform-specific component that collects information about the failures exposed with the test case, by creating execution traces that log the entry and exit points of each boundary call. We implemented a specific tracer rather than using the Android Profiler [12], because it does not collect data about return values and generates huge traces that include method invocations that are not boundary calls.

We thus leveraged the XPosed Framework [13] and implemented the tracer as an Xposed Module. Our Tracer is able to produce traces with boundary calls only, according to a programmable depth. Moreover, it collects extra information such as parameter values, stack traces, and return values for each call. For primitive values, the Tracer reports the actual values in the trace, while for custom objects it provides a string representation, obtained by concatenating the result of the invocation of method `toString`, jointly with the value returned by method *isEmpty* if present. To reduce its impact on the system, the Tracer selectively instruments only the methods that may occur as boundary calls, identified from an initial run of the test case.

Analyzer

The *Analyzer* is a technology-agnostic component that takes as input the execution produced by the *Tracer* and is composed of two essential sub-components: the *Anomaly Detector* and the *Ranking Generator*. The *Anomaly Detector* is responsible for comparing the traces collected during the test execution and identifying SIBs along with their respective weights. The *Ranking Generator* constructs the failure call tree, executing the scoring algorithm, and generates a ranked list of potential fix locations, which includes supporting evidence.

To understand how the *Analyzer* works, we detail the operational workflow in the case of an application experiencing compatibility issues after a framework upgrade. The *Anomaly Detector* analyzes the execution traces from the base and failure environments, identifying deviations and anomalous patterns in the interactions between the app and the framework. The output of the *Anomaly Detector* is a set of SIBs, each assigned a weight based on the severity of THE anomaly. Then, *Ranking Generator* constructs a failure call tree

analyzing the set of SIBs. By running the scoring algorithm, the *Ranking Generator* produces a ranked list of methods that are likely to be the fix locations of the compatibility issue. This list not only prioritizes potential fix locations but also includes supporting evidence in the form of suspicious method invocations, which represent symptoms of failure that can be mitigated by implementing appropriate fixes in the identified methods.

The *Analyzer* implements additional features useful for our experiments. For example, it can perform the parameters space exploration presented in RQ4 and can generate the list of methods to be instrumented. For a demonstration of the *Analyser in action*, readers are invited to view the demo showcasing its functionalities and capabilities [14].

5. Empirical Evaluation

In our evaluation, we investigated five research questions.

RQ1: What is the sensitivity of FILO to the choice of the parameters? This RQ investigates how the choice of a configuration impacts the performance of FILO. To answer this RQ, we performed an exhaustive exploration of the configuration space and studied how the results changed with changing configurations.

RQ2: What is the quality of the ranking produced by FILO? This RQ investigates how well the ranking returned by FILO identifies the method that should be modified to implement the fix. To answer this RQ, we obtained the rankings for a number of different apps with different characteristics and checked whether FILO is able to place the methods that should be modified to produce the fix at the top positions of the ranking, specifically within the first 10 positions or, ideally, within the first five positions as recommended by Kochhar et al. [15].

RQ3: What is the effectiveness of FILO compared to both Naive Trace Analysis and SBFL techniques? This RQ compares the performances of FILO to two competing approaches. Naive Trace Analysis (NTA) represents the strategy of simply comparing the baseline trace with the failure one. This comparison is designed to demonstrate that the only comparison of the traces is not enough to localize the faulty method. We also compared FILO to Ochiai [1], which is an extensively used fault localization technique. This comparison is defined to demonstrate that a strategy specifically designed to localize problems introduced by framework changes is more effective than general-purpose fault localization. It is also noteworthy that FILO has *weaker assumptions* than Ochiai, since it does not require a full test suite with several passing tests to be applied, but it can be applied from a single failing test case.

RQ4: What is the relevance of the information that FILO associates with the methods in the ranking? This RQ investigates if the SIBs produced by FILO are really representative of the symptoms of the failures, and thus they can be used to obtain a better understanding of the failure root causes. To answer this RQ, we compared the content of the SIBs to the actual set of methods related to the incompatibility between the framework and the app.

RQ5: Is FILO actually useful to developers when fixing app issues? This RQ determines the perceived usefulness of FILO by developers. To answer this RQ we performed a human study with 24 developers, comparing the effort needed to debug apps with and without using FILO.

5.1. Subject Apps

To empirically evaluate FILO, we looked for Android apps that have issues caused by a framework upgrade. We searched for these apps on GitHub [16] with keywords such as *"after upgrade to Lollipop"* using API levels and names from 21 to 26 for an initial selection. Since our evaluation requires both the capability to reproduce the failures and the knowledge of the fix, we removed from the results the apps where it was not possible to replicate the upgrade-related issue (e.g., because specific hardware is required, or some of the backend services are not available anymore). When available, we used the official fixes for the issues, looking at the commits tagged with the closure of the issue, or subsequent commits that silently fixed it. When a fix was not available, we implemented it ourselves

following common design principles and Android development guidelines. We ended up with a total of 18 actual upgrade problems and corresponding Appium test cases that are able to replicate the problems.

Table 2 reports information about the apps, their failures, and the faults that caused the failures. Specifically, the *App*, *Ver*, and *Locs* columns indicate the name of the app, the version of the app (specified with the identifier of the commit), and the number of lines of code, respectively.

Table 2. Subject apps.

App	Locs	Inc. API	Failure	Fault
Activities	1.1 K	26	Crash on startup	Missing Oreo adaptive icon support
BossTransfer	2.6 K	23	Crash when opening the details about items in a list	Wrong permission logic
FakeGPS	2.4 K	23	Crash when opening the view to set the fake position	Missing permission logic
FilePicker	2.9 K	23	Folders erroneously shown as empty	Faulty support to the new api
FirefoxLite	102 K	22	Back button doesn't work when tracker popup window appears	Popup in background has to be explicit
FOSSBrowser	18 K	25	Crash on startup	Faulty support to the previous api
GetBack GPS	9.0 K	23	Unable to retrieve current position	Missing permission logic
GoodWeather	6.3 K	23	Hang when refreshing meteo forecast	Missing permission logic
InventoryAgent	10.0 K	23	App stuck when loading inventory	Missing permission logic
KanjiFix	1.3 K	21	Unable to fix Japanese glyph rendering	Fonts require a new procedure to be loaded
MapDemo	0.6 K	23	Crash when retrieving the current position	Missing permission logic
OpenTraining	60.8 K	26	Notification does not show up	New library for notification
PoGoIV	14.3 K	24	Unable to perform the auto update	New api requires the use of FileProvider
PrivacyPolice	1.3 K	23	Unable to connect to wifi networks	Api methods with changed semantics
QuotoGraph	11.5 K	24	Crash on startup	Api methods with changed semantics
SearchView	3.9 K	21	Crash on startup	Api methods with changed semantics
ToneDef	3.4 K	23	Error message when dialling from the phone contacts list	Missing permission logic
TrebleShot	40.6 K	22	External storage cannot be written	Minimum version was erroneously setted

Activities is an app to launch hidden activities within an Android App and create shortcuts to them on the home screen [17]. BossTransfer is a game app [18]. FakeGPS is a GPS device simulator [19]. FilePicker is an app for selecting files and folders in a device [20]. FirefoxLite is a web browser [21] by Mozilla. FOSSBrowser is a privacy-friendly web browser [22]. GetBack GPS is an app for storing the location of points of interest [23]. GoodWeather is a weather app [24]. InventoryAgent is an app for corporations to handle corporate devices [8]. KanjiFix is an app to fix Japanese glyph rendering [25]. MapDemo is an app to test the setup of Google Play services [26]. PoGoIV is an IV calculator for Pokemon Go [27]. OpenTraining is a fitness app [28]. PrivacyPolice is an app that prevents the leaking of sensitive information via Wi-Fi networks [29]. QuotoGraph is an app to create custom wallpapers [30]. SearchView is a persistent search and view library in material design [31]. ToneDef is a tone dialer app [32]. TrebleShot is a peer-to-peer local file sharing app [33].

Column *Inc. API* indicates the version of the API that makes the app fail. This means that the specific version of the app in the column *Ver* works according to the app's intended behavior when executed in all prior API versions with respect to the one in the *Inc. API* column, but does not work with that specific API version.

The *Failure* column provides a short description of the failure caused by the fault described in column *Fault*. The * indicates the faults that do not have an official fix or a commit that resolves the fault. In these cases, we implemented the fix ourselves. It is interesting to report that some of the faults fixed by the developers have not been trivial to debug and fix, requiring from one day up to several months before a working version was

pushed in the repository. Three of the faults also required multiple commits and attempts before they were fixed (up to 23 commits in one of the apps).

5.2. RQ1: What Is the Sensitivity of FILO to the Choice of the Parameters?

We investigated the effectiveness of FILO for a range of configurations, concerning both the *amount of information collected* from the failing execution and the *values assigned to weights* k_1, k_2, *and* k_3 *in Formula 1*. While collecting more data may increase the chance of obtaining information useful to identify the faulty method, it may also introduce noisy and irrelevant information in the analyzed traces. For this reason, we experienced different combinations of data recording, distinguishing between direct and indirect invocations of different levels, and considering both independent and joint invocations from the app to the framework, and vice versa. In particular, we considered the following *eight configurations*:

A1F0: it records the level 1 interactions (direct method invocations) from the app to the framework, while it does not record any interaction from the framework to the app.
A0F1: it records the level 1 interactions (direct method invocations) from the framework to the app, while it does not record any interaction from the app to the framework.
A1F1: it records the level 1 interactions (direct method invocations) from the framework to the app and vice versa.
A2F0: it records the level 2 interactions (direct method invocations and indirect method invocations of depth 1) from the app to the framework, while it does not record any interaction from the framework to the app.
A0F2: it records the level 2 interactions (direct method invocations and indirect method invocations of depth 1) from the framework to the app, while it does not record any interaction from the app to the framework.
A1F2: it records the level 1 interactions (direct method invocations) from the app to the framework, and the level 2 interactions (direct method invocations and indirect method invocations of depth 1) from the framework to the app.
A2F1: it records the level 1 interactions (direct method invocations) from the framework to the app, and the level 2 interactions (direct method invocations and indirect method invocations of depth 1) from the app to the framework.
A2F2: it records the level 2 interactions (direct method invocations and indirect method invocations of depth 1) from the framework to the app and vice versa.

Concerning parameters k_1, k_2, and k_3 we simply considered every possible combination that can be obtained by varying the parameter value by 0.01, resulting in 5.151 configurations. The resulting size of the configuration space that is investigated is thus 41.208 configurations (5.151×8) for each app, with a total of 741.744 executions (41.208×18 apps).

To evaluate how good each configuration is we systematically executed every configuration with every subject app. To measure how good each ranking is, we used *AveP*, which is a metric commonly used in information retrieval [34]. The idea is to first compute the precision and recall of the ranking for every position p of the items in the returned raking. The value of *AveP* is obtained by computing the area under the precision–recall curve obtained by considering the items in the order they are in the ranking. The best result is obtained by returning all and only the relevant items ($AveP = 1$). Moreover, a ranking with the relevant items at the top will have a higher AveP than a ranking with the relevant items ranked at lower positions. In details, *AveP* is computed as follows:

$$AveP = \sum_{k=1}^{n} P(k)\Delta r(k) \tag{2}$$

where n represents the number of items in the ranking, k is the position of an item in the ranking, $P(k)$ is the precision calculated only considering the first k results, and $\Delta r(k)$ is the change in recall between $k-1$ and k.

AveP gives us an evaluation of the rankings obtained by FILO with a given configuration for each app individually. To combine those values and obtain a comprehensive

score for each set of parameters, we use the *Mean Average Precision* (*MAP*). MAP is another common metric that can score the ordering of the results of a set of queries and it is based on the AveP we already calculated [34].

$$MAP = \frac{\sum_{q=1}^{Q} AveP(app_q)}{Q} \quad (3)$$

As shown in Equation (3), MAP is given by the sum of all the $AveP(app_q)$ over the total number of apps Q, where $AveP(app_q)$ is the $AveP$ value for a specific app $app_q \in Q$.

The first part of Table 3 shows the *min*, *max*, and *average* MAP values for each configuration throughout the 18 apps tested and the 5151 parameters combinations of weights, for a total of 741,744 rankings produced (5151 AveP values $\times$18 apps $\times$8 invocations levels). Results show that the most relevant contribution to the localization of the faulty method is obtained by observing the callbacks, that is, the interactions from the framework toward the app (e.g., configuration A0F1). Increasing the level of interactions or adding the calls from the app toward the framework is mostly a confounding effect, decreasing performance.

Table 3. MAP values and distributions for the considered interaction combinations.

	MAP Distribution			**Best MAP**		**MAP$\Delta \leq 0.05$**		**MAP$\Delta \leq 0.1$**		**MAP$\Delta > 0.1$**	
	Min	Max	Average	#	%	#	%	#	%	#	%
A0F1	**0.3991**	**0.6134**	**0.4750**	230	4.47%	679	13.18%	1418	27.53%	3733	72.47%
A0F2	0.3527	0.5407	0.4361	8	0.16%	680	13.20%	2951	57.29%	2200	42.71%
A1F0	0.1250	0.1667	0.1388	**660**	**12.81%**	**5151**	**100.00%**	**5151**	**100.00%**	**0**	**0.00%**
A1F1	0.3483	0.5755	0.4443	1	0.02%	996	19.34%	2054	39.88%	3097	60.12%
A1F2	0.3120	0.5019	0.3849	5	0.10%	568	11.03%	5091	98.84%	60	1.16%
A2F0	0.1528	0.2222	0.1918	**660**	**12.81%**	4675	90.76%	**5151**	**100.00%**	**0**	**0.00%**
A2F1	0.3761	0.5755	0.4579	1	0.02%	835	16.21%	2054	39.88%	3097	60.12%
A2F2	0.3119	0.5111	0.3885	5	0.10%	550	10.68%	4511	87.58%	640	12.42%

The remainder of Table 3 reports the percentage and number of configurations (i.e., set of parameters values) that return the best MAP value (column *Best MAP*) or that return MAP values differing at most by 0.05 (column MAP$\Delta \leq 0.05$) and 0.1 (column $MAP\Delta \leq 0.1$). Finally, we port the number of configurations whose MAP value differs from the best value by more than 0.1 (column $MAP\Delta > 0.1$).

It is interesting to note that in most cases, multiple combinations of parameters can yield the same best MAP. For example, A0F1, which is the overall best-performing configuration, has 230 different parameters distributions that give the max MAP value of $\sim$0.6134 and more than a quarter of all the possible distributions (1418, or 27.53%) give a MAP value that have a delta from the max that is $\leq$0.1. This suggests that the technique is not particularly sensitive to the choice of the parameters.

From Table 3 we can also see that A1F0 and A2F0 have the highest number of parameter distributions that give their best MAP values and their whole spaces have only values within the 0.1 delta boundary. However, they are not very good configurations for FILO as they perform poorly, with A1F0 being the worst configuration in terms of MAP values and A2F0 being the second worst.

To better visualize the resilience of FILO with different values of parameters, Figure 3 shows a 3D plot for each configuration. The three axes represent the contribution of each parameter, since they must sum to 1.0 to be a valid set of parameters, only the outer surface of the plot is considered. Each plot contains 5151 MAP values colored based on the four ranges of values (Best, $delta \leq 0.05$, $delta \leq 0.1$, and $delta > 0.1$).

We can see from Figure 3 that the most influential parameter is *Children* as most of the distributions of the best yielding parameters have a high value for this parameter. Conversely, the least impactful parameter is *Depth*, with the *Influence* parameter in between the other two.

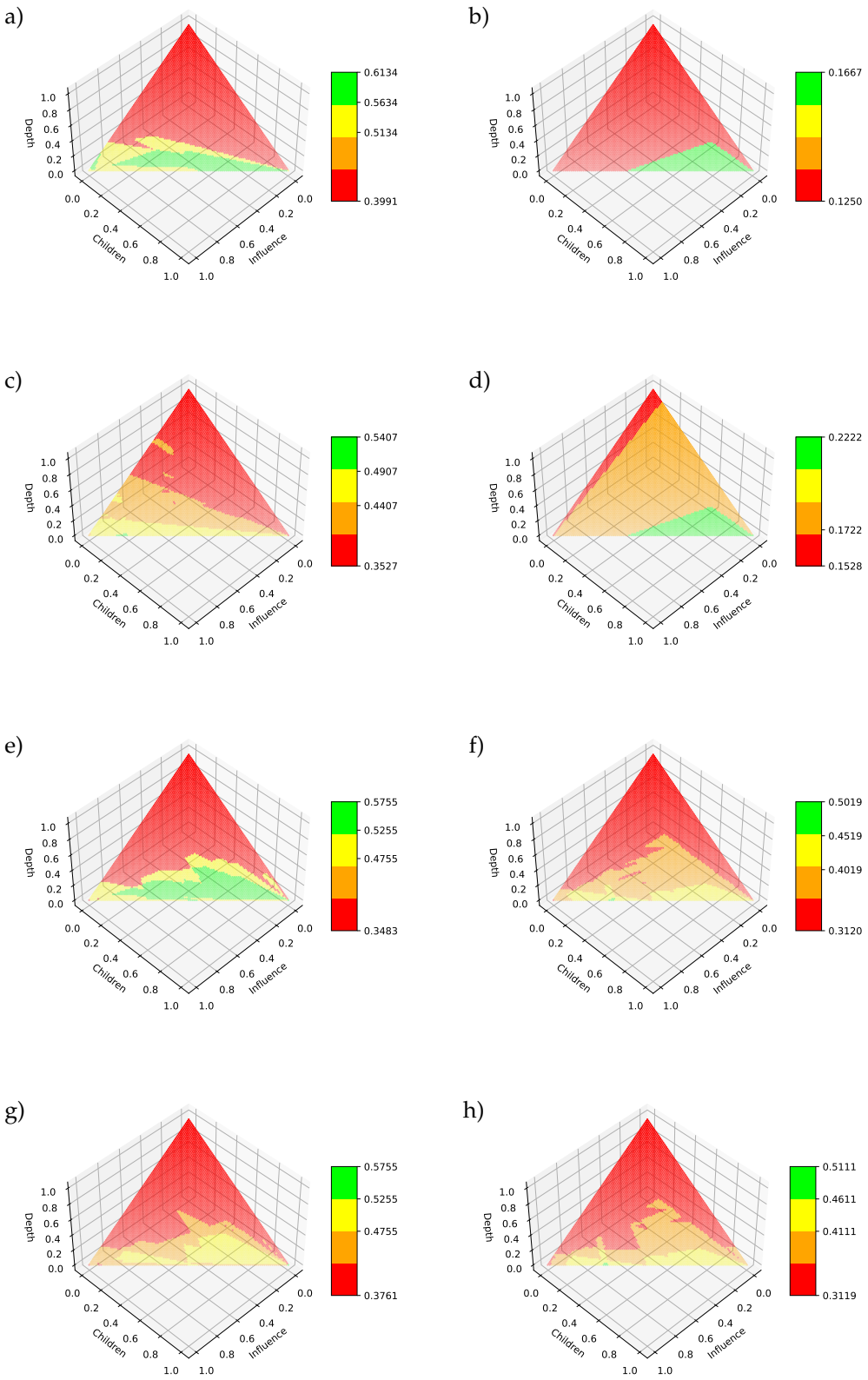

Figure 3. MAP values for the different invocations level: (**a**) A0F1 MAP values; (**b**) A1F0 MAP values; (**c**) A0F2 MAP values; (**d**) A2F0 MAP values; (**e**) A1F1 MAP values; (**f**) A1F2 MAP values; (**g**) A2F1 MAP values; and (**h**) A2F2 MAP values

5.3. RQ2: What Is the Quality of the Ranking Produced by FILO?

To assess the quality of the rankings we consider the position of the faulty method in the rank. Table 4 shows the results obtained by FILO for every configuration when executed in one of its *Best MAP* set of parameters. When a fix requires changes to multiple methods (apps GoodWeather, InventoryAgent, and ToneDef), we consider the position of every method in the ranking (Column *Rank*). We do not consider any new method introduced by developers in order to organize the code of the fix, since these methods were not present in the faulty version of the app and thus cannot be localized. We also report the number of SIBs associated with the ranked methods (Column #*SIB*). In case a faulty method is not present in the returned ranking, we report a "-". The best results for each app are highlighted in bold.

Table 4. FILO rankings for multiple configurations of the recording depth.

	A0F1		A0F2		A1F0		A1F1		A1F2		A2F0		A2F1		A2F2	
Application	**#SIB**	**Rank**	**#SIB**	**Rank**	**#SIB**	**Rank**	**#SIB**	**Rank**	**#SIB**	**Rank**	**#SIB**	**Rank**	**#SIB**	**Rank**	**#SIB**	**Rank**
Activities	18	**1, 8**	18	**2, 6**	2	-	17	2, 8	17	**2, 6**	2	-	17	2, 8	17	**2, 6**
BossTransfer	33	**1**	37	**1**	18	-	36	**1**	40	**2**	18	-	36	**1**	40	2
FakeGPS	43	**1**	71	2	4	-	45	**1**	73	**1**	4	-	45	**1**	73	**1**
FilePicker	45	**1**	63	**1**	91	1	135	**1**	152	**1**	91	**1**	135	**1**	152	**1**
FirefoxLite	2170	**1**	2822	**1**	66	-	2630	**1**	3016	**1**	83	-	2649	**1**	3036	**1**
FOSSBrowser	131	**6**	218	-	2	-	131	-	218	-	3	-	132	-	219	-
GetBackGPS	35	3	41	3	7	-	40	**1**	48	3	9	-	43	**1**	49	2
GoodWeather	49	2, 3	61	2, 3	1	-	49	**1, 2**	61	2, 3	1	-	49	**1, 2**	61	2, 3
InventoryAgent	2456	6, 12	3134	2, 7	266	-	3010	7, 16	3149	2, 9	482	-	3486	6, 14	3842	2, 8
KanjiFix	1	**1**	1	-	1	**1**	1	**1**	1	-	1	**1**	1	**1**	1	-
MapDemo	5	2	257	**1**	0	-	5	**1**	262	5	0	-	5	**1**	262	5
OpenTraining	301	**1**	461	**1**	90	-	400	5	601	**1**	90	-	400	5	601	**1**
PoGoIV	72	**3**	105	5	24	-	90	5	155	6	24	-	90	5	155	6
PrivacyPolice	44	**1**	45	**1**	2	**1**	44	3	45	**1**	2	**1**	44	3	45	1
QuotoGraph	1	**1**	1	**1**	1	-	1	**1**	1	**1**	1	**1**	1	**1**	1	**1**
SearchView	1	-	1	-	1	-	1	-	1	-	1	-	1	-	1	-
ToneDef	178	-	182	5, 3	1	-	178	-	182	6, 3	1	-	178	-	182	6, 3
TrebleShot	41	**2**	91	3	4	-	47	4	100	**2**	10	-	52	4	110	**2**

While the number of SIBs returned by the anomaly detection phase varied greatly among the apps, FILO has been consistently able to report the method to be fixed in the first positions of the ranking. As anticipated by the results reported in Table 3, the ranking generated by A0F1 and A2F1 is better than the others, while A1F0 and A2F0 are worse. Indeed, A0F1 and A2F1 consistently reported the faulty methods in the top position of the ranking in most cases. This result can be considered rather good, since practitioners have been reported to consider acceptable inspecting up to 10 methods, with a preference for techniques that require inspecting 5 methods at most [15].

It is interesting to notice, however, how, for example, the fix-locus of ToneDef and one of the two fix-locus of Activities are always better identified when in the A*F2 configurations.

5.4. RQ3: How Does FILO Compare to Both Naive Trace Analysis and SBFL Techniques?

This RQ compares FILO to other common approaches for fault localization, namely Naive Trace Analysis (NTA) and Spectrum-Based Fault Localization (SBFL) [35]. NTA consists of a straight comparison between the baseline trace and failure trace to report to the user the suspicious method invocations that occur at the boundary between the app and the framework. This baseline method is considered to validate the hypothesis that a simple comparison between the two traces is not effective, and an analysis considering the computations internal to the app is necessary. Moreover, we considered Ochiai [1], one of the most commonly used and effective SBFL techniques (see Section 6 for further details on SBFL) to compare our solution for the localization of upgrade problems in Android to a general purpose fault localization solution.

It is important to remark that SBFL techniques have stricter requirements than FILO and produce a more limited output. In fact, they usually require a full test suite containing both passing and failing test cases to compute the ranking, and they do not offer any ancillary information that helps justifying the methods in the ranking. On the contrary, FILO only needs a single failing test case and augments the ranking with contextual information in the forms of SIBs that may help understanding the reason of the failure.

Since the apps used in this work are released without a full test suite, in principle the whole category of SBFL techniques would not be applicable at all. To overcome this issue, we generated a test suite of passing test cases exploiting the Monkey automatic testing tool [36]. We used a setting that favours the generation of an extensive test suite by configuring Monkey to emit 10,000 events per test case (200 times the default value) and we ran it for 10 min for each app (2 times the time needed to produce advances in coverage [37]).

Moreover, we set up the testing environment in order to prevent the generation of failures and inspected the logs to make sure that no failing test case was included in the suite (so to have only the same failing test case as with FILO). We collected coverage data at the method level to be consistent with the granularity offered by FILO and NTA, but also at the statement level, to investigate the effectiveness at a finer granularity. We then computed the rankings using Ochiai.

Table 5 shows the positions of the faulty methods in the rankings returned by the top two performing configurations of FILO, NTA exploiting the traces produced by the same two levels, and Ochiai, both at method and statement levels. To avoid issues due to overfitting, we also report two parameter distributions that fall within the MAP ≤ 0.1 area, which in the case of A2F1 covers almost 40% of the whole possible combinations of parameters. 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. We assessed up to the first 10 positions of the ranking, since these have been reported as useful positions for practitioners, with a preference for the top 5 positions [15]. *Not in the ranking* counts the number of times the techniques were not able to include the target method in the respective ranking (marked with a "-" in each row). Finally, values in bold represent the best-performing technique for each one of the 18 considered apps, and for the Top-x groups, in the case of *Not in the ranking* less is better, while more is better in every other row. In 3 out of the 18 apps, the methods to modify or add to perform the fix were 2 instead of 1, so we evaluated them individually as if they were two distinct cases, with a total of 21 values.

NTA generates the worst results, with the *NTA A0F1* configuration unable to identify the correct method in all 21 cases, while *NTA A2F1* performed only slightly better, being able to detect the faulty method only 5 times out of 21 and only once among the *top 5* results.

Ochiai is able to detect the fix-locus in 15 and 14 cases at method and statement levels, respectively, both levels outperforming or equating FILO in only 2 and 3 cases. The *Ochiai-method* was able to obtain a perfect result only twice, while *Ochiai-statement* only in one case.

In all the other cases at least one of the four configurations of FILO outperformed Ochiai at both levels. In fact, FILO A2F1 ranked 9 faulty methods at the top position, 15 among the top 5 positions, and 17 among the top 10. Even better, FILO A0F1 ranked 9 faulty methods at the top position, 15 among the top 5 positions, and 18 among the top 10.

If we consider non-optimal parameters distribution, FILO still outperforms SBFL techniques with 7 perfect scores and up to 15 ranked methods within the top 5.

It is also important to remark that FILO is cheaper to execute than SBFL. Its main cost is based on the execution time of the failing test case only. In contrast, SBFL techniques require the execution of a complete test suite with the instrumentation to collect full coverage data, which is orders of magnitude more expensive.

In summary, our empirical results show that FILO is more effective than NTA and SBFL in detecting the fix-locus in problems introduced by framework upgrades.

Table 5. Comparison between FILO, Naive trace analysis, and Ochiai.

Application	BestMAP A0F1	BestMAP A2F1	MAPΔ ≤ 0.1 A0F1	MAPΔ ≤ 0.1 A2F1	NTA A2F1	NTA A0F1	Ochiai (Method)	Ochiai (Statement)
Activities	**1**, 8	2, **8**	2, **2**	2, **2**	-, -	-, -	-	-
BossTransfer	**1**	**1**	**1**	2	-	-	4	32
FakeGPS	**1**	**1**	**1**	**1**	612	-	13	65
FilePicker	**1**	**1**	**1**	**1**	347	-	81	-
FirefoxLite	**1**	**1**	**1**	**1**	-	-	88	279
FOSSBrowser	**6**	-	**6**	-	-	-	-	-
GetBack GPS	3	**1**	3	3	-	-	-	-
GoodWeather	2, 3	**1, 2**	2, 3	**1, 2**	-, -	-, -	**1**, 32	5, 5
InventoryAgent	6, 12	6, 14	6, **11**	8, 17	-, -	-, -	**5**, 52	17, 199
KanjiFix	**1**	**1**	**1**	**1**	2	-	19	23
MapDemo	2	**1**	2	2	-	-	**1**	**1**
OpenTraining	**1**	5	3	6	-	-	195	1008
PoGoIV	**3**	5	**3**	**3**	-	-	48	283
PrivacyPolice	**1**	3	**1**	**1**	41	-	21	130
QuotoGraph	**1**	**1**	**1**	**1**	-	-	-	-
SearchView	-	-	-	-	311	-	-	-
ToneDef	-	-	-	-	-	-	24	13
TrebleShot	**2**	4	**2**	4	-	-	15	635
Top-1	**9**	**9**	7	7	0	0	2	1
Top-5	**15**	**15**	**15**	14	1	0	4	3
Top-10	**18**	17	**18**	17	1	0	4	3
Not in the ranking	**2**	3	**2**	3	17	22	5	6

5.5. RQ4: What Is the Relevance of the Information Captured in the SIBs Obtained by Comparing the Execution Traces?

To Answer RQ4, we analyzed the SIBs produced by the anomaly detection phase of FILO. During its normal execution, FILO exploits the comparison between the baseline trace and the failure trace to estimate the changes during the app execution that can be related to the fault. To be able to *objectively* identify the execution differences introduced by the fault and check if they are captured by the SIBs, we compared the failure trace with the *fix trace*, a different trace that is obtained by the (successful) execution of the test case on the fixed version of the app within the upgraded framework. The differences between the failure trace and the fix trace represent the suppressed or novel method invocations caused by the fix of the application. We define such differences as *Fault-Related Method Calls* (FRMCs). If the SIBs normally obtained by FILO are representative of the fault, they should contain the FRMCs (minus the invocations of methods added in the fix) and should be exploited to obtain the fix-locus. To measure how SIBs are related to FRMCs, we defined the *Soundness* of the content of the SIBs as follows:

$$Soundness = \frac{\#SIB\_with\_FRMCs}{\#SIB} \tag{4}$$

where *#SIB_with_FRMCs* represents the number of SIBs that contains at least one FRMCs. We computed this metric for all the applications considering the various depths configurations introduced in Section 5.2. Similarly, to measure the *Completeness* of information reported by the SIBs, we computed the ratio of FRMCs contained in the SIBs as

$$Completeness = \frac{\#FRMCs\_in\_SIBs}{\#FRMCs} \tag{5}$$

The average soundness and completeness for each application, as well as the overall values, are reported in Table 6.

Table 6 shows that the most of the SIBs carry information relevant to the fault, with six applications where every SIBs contain at least one FRMC. The only application with a soundness lower than 50% is TrebleShot (37.58%). The completeness values are more variegated, indicating that, in general, not all the FRMCs are reported in the SIBs produced by FILO. It is interesting to note, however, that even with low percentages of soundness

and completeness, FILO is able to exploit the information available to produce meaningful rankings. For example, TrebleShot, which also has the lowest completeness value (18.17%) has the correct fix-locus identified within the top five elements six times over eight configurations. It must also be highlighted, however, how SearchView has both *soundness* = 100% and *completeness* = 69.57% and yet FILO is not able to detect the fix-locus. This might be related to the fact that the anomaly detection phase produces a single SIB, suggesting differences in the early stage of the application.

Table 6. Suspicious invocation blocks.

Application	Soundness	Completeness
Activities	100.00%	99.79%
BossTransfer	95.79%	97.78%
FakeGPS	100.00%	74.72%
FilePicker	91.56%	65.41%
FirefoxLite	88.37%	91.20%
FOSSBrowser	81.03%	27.32%
GetBack GPS	63.65%	49.82%
GoodWeather	87.50%	84.11%
InventoryAgent	98.29%	99.45%
KanjiFix	75.00%	75.00%
MapDemo	100.00%	49.71%
OpenTraining	94.18%	85.52%
PoGoIV	80.46%	55.57%
PrivacyPolice	100.00%	79,78%
QuotoGraph	100.00%	81.14%
SearchView	100.00%	69.57%
ToneDef	95.31%	34.54%
TrebleShot	37.58%	18.17%
Average	88.09%	68.57%

In summary, although the information contained in the SIBs is not noise-free and may be incomplete, SIBs are confirmed to carry relevant information about the failure and can be exploited to identify the methods that realistically need to be fixed in order to resolve the fault, as shown in RQ2.

5.6. RQ5: Is FILO Actually Useful to Developers When Fixing App Issues?

To answer RQ5, we designed an experiment with actual human subjects to determine if the results provided by FILO are useful and can ease debugging tasks. We selected two apps: GoodWeather and TrebleShot. We chose these two apps because they are relatively simple apps, which would give the developers a chance to understand their code base in the limited time available for experimentation. To support the study, we implemented an Android Studio plugin that allows users to compute the ranking with FILO and navigate to the implementation of methods in the ranking within the native editor.

We considered the availability of FILO as the treatment. We thus divided the subjects into two groups. Group 1 debugged the fault in GoodWeather exploiting the ranking returned by FILO, but they debugged TrebleShot without accessing the ranking. Vice versa, Group2 had access to the ranking for TrebleShot, but not for GoodWeather. This design of the experiment allows each subject to perform the same task with and without the treatment. Table 7 summarises the structure of the groups.

The debugging task performed by each subject consisted of identifying the fault and possibly fixing it. When a subject believes the fault is identified, he/she notes down the class(es) and method(s) where he/she thinks the fix-locus is and a brief description of

the cause. Then, he/she can continue the debugging activity until he/she believes the fault is fixed, or the time expires (we assigned a max of 30 min to each debugging task). Each subject was asked to fill out an entry questionnaire regarding their experience in development in general and specifically with Android apps and Android Studio (the IDE used in the experiment). They were also asked to fill out an exit questionnaire for each app, where they noted if they think they found the fault, specific details (classes and methods), a description of the cause, and if they were able to fix it, how the fix was produced. They also uploaded the modified version of the app. Furthermore, the exit questionnaires included questions regarding their perception of the usefulness of the ranking provided by FILO.

For this experiment, we recruited 24 students, most of them with a declared experience in Android development with at least 1–3 Android apps developed in the past. Five out of twenty-four subjects had previous professional experiences in mobile development and two out of twenty-four had developed more than four Android apps at the time of the experiment. They participated in the study in supervised lab sessions, after an initial warm-up training on the subject apps. We judged the capability to identify the cause of the fault and the fix-locus through their answers in the exit questionnaire, and we judged the correctness of the fix, when produced, by reviewing the answers in the questionnaire and manually checking the code. In case any subject identified the wrong fix-locus or produced a non-suitable fix, we marked it as if the fix-locus was not identified and the fix was not produced. Informed consent was obtained from all participants prior to their participation in the study.

Table 7. Design of the experiment to respond to RQ5.

Group	Ranking for GoodWeather	Ranking for TrebleShot	#Subjects
1	Yes	No	12
2	No	Yes	12

In general, the debugging tasks were not easy. Out of the total 48 reports collected (2 for each subject, 1 with the ranking and 1 without), only in 27.08% of the times the fix-locus has been identified within the time constraints (13 times). The availability of FILO's ranking helped to increase the success rate, as 33.33% of people with the ranking were globally able to determine the cause of the issue, as opposed to only 20.83% of the subjects without the ranking. As shown in Figure 4, FILO increased the chances of identifying the fix-locus by 12.5%, with an increase of 16.67% in the case of GoodWeather and 8.33% in the case of TrebleShot.

Some subjects reported that they might have needed more time to produce the fix.

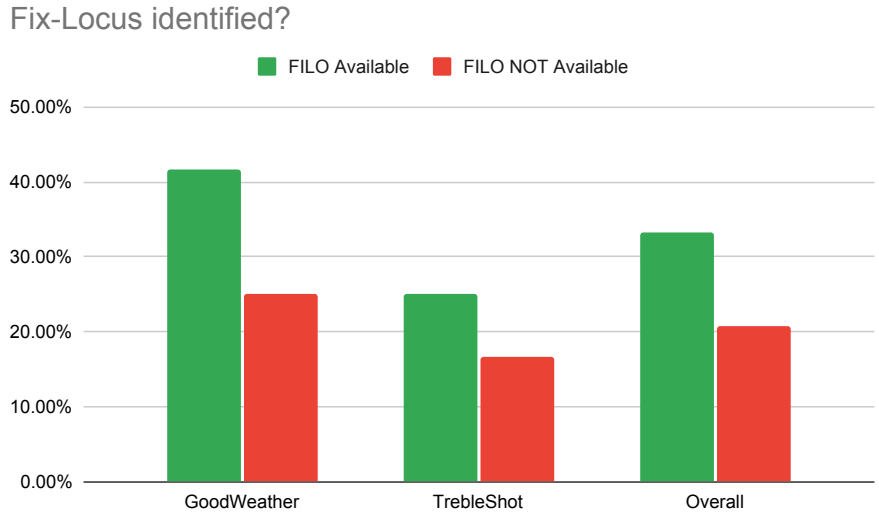

Figure 4. Percentage of subjects that correctly identified the fix-locus with and without FILO.

5.7. Threats to Validity

Our study is affected by internal and external threats to validity.

An internal threat to validity is about the comparison to Ochiai. As discussed, FILO only requires a failing execution to be applied, while Ochiai, like 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: 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 of test case suites. To 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.

External validity threats are concerned with the generalization of the collected evidence. FILO performed consistently well 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 in the general validity of the results.

The study with human subjects is based on computer science students, and professional developers may exploit the ranking and the SIBs generated by FILO differently. To mitigate this threat, we selected subjects with good experience with Android development and who have coded at least one Android app in the past.

5.8. 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 this happened only in 2 cases out of 21 for the A0F1 configuration and in 3 cases in all the other configurations that included any interaction from the framework toward the app.

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, however, be introduced together with other faulty changes that FILO can detect, such as in the case of GoodWeather.

6. Related Work

Mobile ecosystems are well-known for getting frequent framework upgrades [1,38]. In many cases, the upgrades are intentionally *not* backward compatible with previous versions and this results in developers struggling to continuously adapt their apps to the newer versions of the frameworks, as can be witnessed by the many discussion threads and requests for support opened on platforms, such as Stack Overflow, every time a framework upgrade is released [39]. When these upgrades are not handled properly, apps might be affected by fragmentation-induced compatibility issues [3], as well as maintenance and security issues [40]. A recent study identifies compatibility issues as one of the major root causes of Android app crashes [41]. Compatibility issues between apps and the Android framework have become a major concern for developers in the last years [3,42,43]. For these reasons, a number of techniques have emerged that focus on the detection of such incompatibilities.

Li et al. proposed CiD, an approach based on building API lifecycle models based on the revision history of the Android framework and then using these models to detect incompatibilities [44]. This approach is, however, limited to the detection of incompatible API calls, missing to detect bugs in the apps, or poor handling of different API versions within the app code.

In contrast, Huang et al. proposed CIDER, which is limited to the detection of incompatibilities related to callbacks [45]. It works by statically building graphs that model app control flow inconsistencies induced by the evolution of the API. Each graph is produced by analyzing the inconsistencies in the callback API invocation protocols across different API levels. CIDER statically combines the general graphs with the app API levels and callbacks to detect compatibility issues.

To combine the capabilities of both CiD and CIDER, Mahmud et al. introduced ACID [46], which is a tool that is able to detect incompatibilities both on API calls and in callbacks, as we defined them in Section 3.1. The detection is performed by applying static analysis techniques to the app code and therefore does not need to run the application. Furthermore, Mahmud et al. recently proposed FCID [2], a tool that does static analysis to determine backward incompatibilities related to API field changes.

The common drawbacks of CiD, CIDER, and therefore ACID and FCID is that they rely only on syntactic changes of the API, since the underlying graphs are statically built analyzing the changes in the API Reference and the framework source code. Conversely, FILO can provide suggestions in cases where the syntax of the callbacks is not changed, but their dynamic behavior is. For example, in PrivacyPolice the issue is due to a behavioral change in the API (no wifi networks are returned if location services are disabled), but the syntax of the framework showed no change. This also applies to QuotoGraph and SearchView and in general to the incompatibilities where the root cause is an API change that is semantic rather than syntactic.

Some incompatibility detection techniques are based on dynamic analysis. For instance, Mostafa et al. [4] present a technique for detecting behavioral backward incompatibilities in Java libraries, including Android, by running regression tests and checking bug reports. Similarly Mora et al. [47] 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 [48].

These approaches are more in line with the idea behind FILO, however, they provide no information on the *fix-locus*. Furthermore, conversely, to FILO, they do not provide any support to the debugging activity.

Fault localization techniques can instead be used to generate information about the possible fault location. These approaches are usually referred to as spectrum-based full localization (SBFL) [35]. Among the most known SBFL techniques there are Tarantula [6], Ochiai [5], and Zoltar [49]. All these techniques can potentially localize any fault and are not limited to framework upgrades as FILO. On the other hand, FILO can be implied from a single failing test, in contrast to SBFL which requires a full test suite with passing and failing test cases. Moreover FILO is able to produce contextual information and more accurate localization compared to SBFL techniques, as empirically reported in this paper. Furthermore, SBFL has been proven to show limitations [50–52] and can hardly satisfy the requirements that practical fault localization approaches should satisfy, according to the opinion of practitioners [15]. In contrast, FILO frequently ranked the faulty statements in the top positions and provided contextual information, as required by practitioners [51].

Some approaches can be used to mitigate the compatibility problems. For instance, ReBA [53] 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.

Finally, there are some techniques based on dynamic analysis and *anomaly detection* [54] that rely on similar approaches to FILO. In fact, there are different solutions that analyze

and compare behaviours of applications and components to identify anomalies that can then be used to support the debugging activity.

For example, BCT [55], RADAR [56], and the technique by Pradel and Gross [52] 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 [57]. 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.

7. Conclusions

Timely fixes to 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 with 21 actual upgrade problems in 18 open-source apps 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 [58] and to experiment with fixes that span multiple methods and files.

Author Contributions: Conceptualization, M.M., O.R., D.M. and L.M.; Methodology, M.M., O.R., D.M. and L.M.; Software, M.M., O.R., D.M. and L.M.; Validation, M.M., O.R., D.M. and L.M.; Writing—original draft, M.M., O.R., D.M. and L.M.; Writing—review & editing, M.M., O.R., D.M. and L.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Informed Consent Statement: Informed consent was obtained from all subjects involved in the study.

Data Availability Statement: The data presented in this study are openly available in Gitlab [FILO] [https://gitlab.com/learnERC/filo] (accessed on 10 June 2024).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. McDonnell, T.; Ray, B.; Kim, M. An empirical study of API stability and adoption in the android ecosystem. In Proceedings of the IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, 22–28 September 2013; pp. 70–79.
2. Mahmud, T.; Che, M.; Yang, G. Android api field evolution and its induced compatibility issues. In Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Helsinki, Finland, 19–23 September 2022; pp. 34–44.
3. Wei, L.; Liu, Y.; Cheung, S.-C. Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Singapore, 3–7 September 2016; pp. 226–237.
4. Mostafa, S.; Rodriguez, R.; Wang, X. Experience Paper: A Study on Behavioral Backward Incompatibilities of Java Software Libraries. In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, 10–14 July 2017; pp. 215–225.
5. Abreu, R.; Zoeteweij, P.; van Gemund, A.J.C. On the accuracy of spectrum-based fault localization. In Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques—MUTATION (TAICPART-MUTATION 2007), Windsor, UK, 10–14 September 2007; pp. 89–98.
6. Jones, J.A.; Harrold, M.J. Empirical evaluation of the Tarantula automatic fault-localization technique. In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, Long Beach, CA, USA, 7 November 2005; pp. 273–282.
7. Mobilio, M.; Riganelli, O.; Micucci, D.; Mariani, L. FILO: FIx-LOcus Recommendation for Problems Caused by Android Framework Upgrade. In Proceedings of the 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), Berlin, Germany, 28–31 October 2019; pp. 358–368.
8. Teclib. GLPI Android Inventory Agent. Available online: https://github.com/glpi-project/android-inventory-agent (accessed on 10 June 2024).

9. Michael Kerrisk, Diff(1)? Linux Manual Page. Available online: http://man7.org/linux/man-pages/man1/diff.1.html (accessed on 10 June 2024).
10. Foundation, J. Appium. Available online: http://appium.io/ (accessed on 10 June 2024).
11. Android Studio. Available online: https://developer.android.com/studio (accessed on 10 June 2024).
12. The Android Profiler. Available online: https://developer.android.com/studio/profile/android-profiler (accessed on 10 June 2024).
13. XDA Developers, Xposed. Available online: https://forum.xda-developers.com/f/xposed-general.3094/ (accessed on 10 June 2024).
14. Mobilio, M.; Riganelli, O.; Micucci, D.; Mariani, L. Filo: Fix-locus localization for backward incompatibilities caused by android framework upgrades. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering Virtual Event, Melbourne, Australia, 21 September 2020; pp. 1292–1296.
15. Kochhar, P.S.; Xia, X.; Lo, D.; Li, S. Practitioners' expectations on automated fault localization. In Proceedings of the 25th International Symposium on Software Testing and Analysis, Saarbrucken, Germany, 18–20 July 2016; pp. 165–176.
16. GitHub, Inc. GitHub. Available online: https://github.com (accessed on 10 June 2024).
17. Cunningham, J. Activities. Available online: https://github.com/cunnj/Activities (accessed on 10 June 2024).
18. AriaLyy, Boss Transfer. Available online: https://github.com/shijunjzr/BlogDemo/ (accessed on 10 June 2024).
19. xiangtailiang, FakeGPS. Available online: https://github.com/xiangtailiang/FakeGPS (accessed on 10 June 2024).
20. Paul, T. FilePicker. Available online: https://github.com/DeveloperPaul123/FilePickerLibrary (accessed on 10 June 2024).
21. Mozilla, FirefoxLite. Available online: https://github.com/mozilla-mobile/FirefoxLite(accessed on 10 June 2024).
22. FOSS Browser. Available online: https://f-droid.org/packages/de.baumann.browser/ (accessed on 10 June 2024).
23. Dieter Adriaenssens, GetBack_GPS. Available online: https://github.com/ruleant/getback_gps (accessed on 10 June 2024).
24. Eugene Kislyakov, GoodWeather. Available online: https://github.com/qqq3/good-weather (accessed on 10 June 2024).
25. KanjiFix. Available online: https://github.com/ascendedguard/android-kanji-fix (accessed on 10 June 2024).
26. MapDemo. Available online: https://github.com/MiniPlaneterKUET/MapDemo (accessed on 10 June 2024).
27. Swanberg, J. PoGoIV. Available online: https://github.com/farkam135/GoIV (accessed on 10 June 2024).
28. OpenTraining. Available online: https://github.com/chaosbastler/opentraining (accessed on 10 June 2024).
29. PrivacyPolice. Available online: https://github.com/BramBonne/privacypolice (accessed on 10 June 2024).
30. Stanley Idesis, QuotoGraph. Available online: https://github.com/stanidesis/quotograph (accessed on 10 June 2024).
31. Martin Lapiš, SearchView. Available online: https://github.com/lapism/SearchBar-SearchView (accessed on 10 June 2024).
32. Network 47, ToneDef. Available online: https://github.com/Fortyseven/ToneDef (accessed on 10 June 2024).
33. TrebleShot. Available online: https://github.com/trebleshot/android (accessed on 10 June 2024).
34. Kishida, K. Property of Average Precision and Its Generalization: An Examination of Evaluation Indicator for Information Retrieval Experiments. National Institute of Informatics, Tokyo, Japan, 2005. Available online: https://www.nii.ac.jp/TechReports/public_html/05-014E.pdf (accessed on 10 June 2024).
35. Wong, W.E.; Gao, R.; Li, Y.; Abreu, R.; Wotawa, F. A survey on software fault localization. *IEEE Trans. Softw. Eng.* **2016**, *42*, 8. [CrossRef]
36. Google, Monkey. Available online: https://developer.android.com/studio/test/monkey (accessed on 10 June 2024).
37. Choudhary, S.R.; Gorla, A.; Orso, A. Automated Test Input Generation for Android: Are We There Yet?). In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, 9–13 November 2015; pp. 429–440.
38. Android Framework Classes and Services. Available online: https://android.googlesource.com/platform/frameworks/base.git (accessed on 10 June 2024).
39. Linares-Vásquez, M.; Bavota, G.; Penta, M.D.; Oliveto, R.; Poshyvanyk, D. How Do API Changes Trigger Stack Overflow Discussions? A Study on the Android SDK. In Proceedings of the 22nd International Conference on Program Comprehension, Hyderabad, India, 2–3 June 2014; pp. 83–94.
40. Li, L.; Gao, J.; Bissyandé, T.F.; Ma, L.; Xia, X.; Klein, J. Characterising Deprecated Android APIs. In Proceedings of the 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE), Gothenburg, Sweden, 14 May 2018; pp. 254–264.
41. Gong, L.; Lin, H.; Liu, D.; Yang, L.; Wang, H.; Qiu, J.; Li, Z.; Qian, F. Who Should We Blame for Android App Crashes? An In-Depth Study at Scale and Practical Resolutions. *ACM Trans. Sens. Networks* **2024**, *20*, 1–24. [CrossRef]
42. Xia, H.; Zhang, Y.; Zhou, Y.; Chen, X.; Wang, Y.; Zhang, X.; Cui, S.; Hong, G.; Zhang, X.; Yang, M.; Yan, Z. How Android developers handle evolution-induced API compatibility issues: A large-scale study. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June–19 July 2020; pp. 886–898.
43. Cai, H.; Zhang, Z.; Li, L.; Fu, X. A large-scale study of application incompatibilities in Android. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, Beijing, China, 15–19 July 2019; pp. 216–227.
44. Li, L.; Bissyandé, T.F.; Wang, H.; Klein, J. 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, Amsterdam, The Netherlands, 16–21 July 2018; pp. 153–163.

45. Huang, H.; Wei, L.; Liu, Y.; Cheung, S.-C. Understanding and detecting callback compatibility issues for android applications. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 532–542.
46. Mahmud, T.; Che, M.; Yang, G. Android compatibility issue detection using api differences. In Proceedings of the 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, 9–12 March 2021; pp. 480–490.
47. Mora, F.; Li, Y.; Rubin, J.; Chechik, M. Client-specific equivalence checking. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 441–451.
48. Fazzini, M.; Orso, A. Automated cross-platform inconsistency detection for mobile apps. In Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE) Urbana, IL, USA, 30 October 2017; pp. 308–318.
49. Riboira, A.; Abreu, R. The GZoltar project: A graphical debugger interface. In Proceedings of the International Academic and Industrial Conference on Practice and Research Techniques, Windsor, UK, 4–6 September 2010; pp. 215–218.
50. Steimann, F.; Frenkel, M.; Abreu, R. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In Proceedings of the 2013 International Symposium on Software Testing and Analysis, Lugano, Switzerland, 15–20 July 2013; pp. 314–324.
51. Parnin, C.; Orso, A. Are automated debugging techniques actually helping programmers? In Proceedings of the 2011 International Symposium on Software Testing and Analysis, Toronto, ON, Canada, 17–21 July 2011; pp. 199–209.
52. Pradel, M.; Gross, T.R. Leveraging test generation and specification mining for automated bug detection without false positives. In Proceedings of the 2012 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2–9 June 2012; pp. 288–298.
53. Dig, D.; Negara, S.; Mohindra, V.; Johnson, R. ReBA: A Tool for Generating Binary Adapters for Evolving Java Libraries. In Proceedings of the Companion of the 30th International Conference on Software Engineering, Leipzig, Germany, 10–18 May 2008; pp. 963–964.
54. Chandola, V.; Banerjee, A.; Kumar, V. Anomaly detection: A survey. *ACM Comput. Surv.* **2016**, *41*, 15. [CrossRef]
55. Mariani, L.; Pastore, F.; Pezzè, M. Dynamic analysis for diagnosing integration faults. *IEEE Trans. Softw. Eng.* **2011**, *37*, 4. [CrossRef]
56. Pastore, F.; Mariani, L.; Goffi, A.; Oriol, M.; Wahler, M. Dynamic analysis of upgrades in C/C++ software. In Proceedings of the 2012 IEEE 23rd International Symposium on Software Reliability Engineering Dallas, TX, USA, 27–30 November 2012; pp. 91–100.
57. Zuddas, D.; Jin, W.; Pastore, F.; Mariani, L.; Orso, A. MIMIC: Locating and understanding bugs by analyzing mimicked executions. In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, Vasteras, Sweden, 15–19 September 2014; pp. 815–826.
58. Gazzola, L.; Micucci, D.; Mariani, L. Automatic software repair: A survey. *IEEE Trans. Softw. Eng.* **2019**, *45*, 34–67. [CrossRef]