# Reproducing Context-sensitive Crashes in Mobile Apps using Crowdsourced Debugging

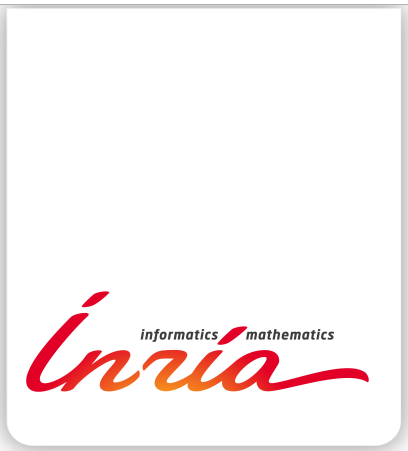
Maria Gomez, Romain Rouvoy, Lionel Seinturier


# Reproducing Context-sensitive Crashes in Mobile Apps using Crowdsourced Debugging

**Maria Gomez, Romain Rouvoy, Lionel Seinturier**

**RESEARCH REPORT**
**N° 8731**
May 2015
Team Spirals

ISSN 0249-6399 ISRN INRIA/RR--8731--FR+ENG




# Reproducing Context-sensitive Crashes in Mobile Apps using Crowdsourced Debugging

Maria Gomez, Romain Rouvoy, Lionel Seinturier

Team Spirals

Research Report n° 8731 — May 2015 — 13 pages

**Abstract:** While the number of mobile apps published by app stores keeps increasing, the quality of these apps greatly varies. Unfortunately, end-users continue experiencing bugs and crashes for some of the apps installed on their mobile devices. Although developers heavily test their apps before release, context-sensitive crashes might still emerge after deployment. This paper therefore introduces MoTiF, a crowdsourced approach to support developers in automatically reproducing context-sensitive crashes faced by end-users in the wild. The goal of MoTiF is to complement existing testing solutions with mechanisms to monitor and debug apps after their deployment. We demonstrate that MoTiF can effectively reproduce existing crashes in Android apps with a low overhead.

**Key-words:** Context-sensitive bugs; Crowdsourced debugging; Mobile apps

RESEARCH CENTRE
LILLE – NORD EUROPE

Parc scientifique de la Haute-Borne
40 avenue Halley - Bât A - Park Plaza
59650 Villeneuve d'Ascq

# Reproduction de *crashes* contextuels dans les applications mobiles grâce au débogage externalisé

**Résumé :** Alors que le nombre d'applications mobiles publiés par les magasins d'applications ne cesse d'augmenter, la qualité de ces applications varie grandement. Malheureusement, les utilisateurs continuent à subir des *bugs* et des *crashes* pour certaines des applications installées sur leurs appareils mobiles. Bien que les développeurs testent leurs applications largement avant la publication, les *crashes* contextuels peuvent encore apparaître après le déploiement. Cet article présente donc MoTiF, une approche de *crowdsourcing* pour soutenir les développeurs dans la reproduction automatique de *crashes* contextuels rencontrés par les utilisateurs après le deployment. Le but du MoTiF est de compléter les solutions de test existantes avec des mécanismes pour surveiller et déboguer des applications mobiles après leur déploiement. Nous démontrons que MoTiF peut reproduire efficacement les *crashes* existants dans les applications Android avec un faible surcoût.

**Mots-clés :** Bogues contextuels; Débogage externalisé; Applications mobiles

# Reproducing Context-sensitive Crashes in Mobile Apps using Crowdsourced Debugging

María Gómez, Romain Rouvoy and Lionel Seinturier
University of Lille / Inria
firstname.lastname@inria.fr

***Abstract*—While the number of mobile apps published by app stores keeps increasing, the quality of these apps greatly varies. Unfortunately, end-users continue experiencing bugs and crashes for some of the apps installed on their mobile devices. Although developers heavily test their apps before release, context-sensitive crashes might still emerge after deployment. This paper therefore introduces MOTIF, a crowdsourced approach to support developers in automatically reproducing context-sensitive crashes faced by end-users in the wild. The goal of MOTIF is to complement existing testing solutions with mechanisms to monitor and debug apps after their deployment. We demonstrate that MOTIF can effectively reproduce existing crashes in Android apps with a low overhead.**

## I. INTRODUCTION

With the proliferation of mobile devices and app stores (*e.g.*, Google Play, Apple App Store, Amazon Appstore), the development of mobile applications (*apps* for short) is experiencing an unprecedented popularity. For example, the Google Play Store reached over 50 billion app downloads in 2013 [39].

Despite the high number of mobile apps available, the quality of these apps greatly varies. Unfortunately, end-users continue experiencing crashes and errors for some apps installed on their devices. For instance, we have already identified $10,658$ suspicions of bugs in a dataset of $46,644$ apps collected from Google Play Store [20].

To fix these bugs, app developers can use a wide range of testing tools for mobile apps [5], [15], [23], [30]. However, even if apps are tested extensively *in vitro* prior to release, many bugs may still emerge once deployed *in vivo*. In fact, the rapid evolution of the mobile ecosystem (OS, SDK, devices, etc.) makes difficult to guarantee the proper functioning of the developed apps along time.

When an issue is reported by users, developers must quickly fix their apps in order to stay competitive in the ever-growing mobile computing landscape. Either on desktop, server, or mobile applications, the first task to efficiently fix a bug is to *reproduce* the problem [44]. However, any software developer knows that faithfully reproducing failures that users experience *in vivo* is a major challenge. In particular, the failure reproduction task becomes even harder in mobile environments, where developers have to deal with device fragmentation and diverse operating conditions [12].

To overcome this issue, we present MOTIF,[1] a crowdsourced approach to support developers in reproducing mobile app context-sensitive crashes faced by end-users in the wild. In particular, the key idea is that by exploiting the experience faced by a multitude of individuals, it is possible to assist developers in isolating and reproducing such crashes in an automatic and effective manner. MOTIF therefore aims to complement existing testing solutions with novel mechanisms to monitor and debug apps *in vivo*.

Beyond existing crash reporting systems for mobile apps (*e.g.*, *Google Analytics* [4], *SPLUNK* [7]), which collect raw analytics on the execution of apps, MOTIF automatically generates *in vivo crash test suites* to reproduce faithfully crashes experienced by users. These test suites reproduce the shortest sequence of user interactions that lead to the crash of the app, together with the execution context under which such crashes arise.

As an illustration, users recently experienced crashes with the Android *Wikipedia app*.[2] This app crashed when the user pressed the *menu* button. However, this crash only emerged on LG devices running Android 4.1. Thus, for developers knowing the *user interactions* and the *execution context* which lead to crashes are crucial informations to be collected in order to faithfully reproduce bugs.

In particular, MOTIF exploits machine learning techniques atop of crowdsourced data collected from real devices in order to automatically identify recurrent crash patterns among user actions and contexts. From the identified patterns, MOTIF generates candidate *in vivo crash test suites* that potentially reproduce the observed crashes. Finally, MOTIF uses the crowd of devices to assess if the generated test suites truly reproduce the observed crashes and can generalizes to other contexts or not. For example, some failures only emerge in specific device models or in devices running a specific configuration (*e.g.*, low memory, network connection unavailable). The devices successfully reproducing the crowdsourced crashes will be qualified as candidate devices to assess the quality of future fixes, while other devices will be used to check that the future fixes do not produce any side-effect.

Our current implementation of MOTIF focuses on Android because, according to a recent study, the $70\%$ of mobile app developers are targeting the Android platform [40]. We therefore evaluate MOTIF on a set of buggy Android apps and demonstrate that it effectively reproduces crashes with a low overhead.

The goal of MOTIF is therefore to drastically improve the quality of mobile apps by contributing along the following

[1] MOTIF stands for MObile Testing In-the-Field. A *Motif* means a repeated image or design forming a pattern, both in French and in English.

[2] https://play.google.com/store/apps/details?id=org.wikipedia&hl=en

axes:

- We propose a *crowdsourced approach* to support developers to reproduce crashes faced by end-users in the wild;
- We propose *crowd crash graphs* as a novel mechanism to aggregate, in a meaningful way, data collected from a multitude of devices;
- We propose an algorithm to extract relevant *crash patterns* of user interactions and contexts to reproduce crashes;
- We propose a *crowd-validation* mechanism to assess the crash test suites generated from consolidated patterns in the crowd;
- We conduct an *empirical evaluation* to demonstrate the feasibility of our approach.

The remainder of this paper is organized as follows. Section II provides an overview of the proposed approach. Section III describes the monitoring strategy followed by MOTIF. Section IV introduces *Crowd Crash Graphs*, a technique to aggregate in a meaningful manner traces collected from a multitude of devices. Section V presents the algorithms to extract consolidated steps and contexts to reproduce crashes. Section VI illustrates the test case generation technique from patterns extracted from the crowd. Section VII provides implementation details. Section VIII evaluates the approach. Section IX summarizes the related work. Finally, Section X concludes the paper and outlines future work.

## II. OVERVIEW

Figure 1 depicts an overview of the proposed approach. In particular, the four key phases of MOTIF are:

1) *Collect execution traces from devices in the wild.* MOTIF collects user interaction events and context data during the execution of a subject app in mobile devices. If the app crashes, the collected traces are submitted to the MOTIF server (cf. Section III);
2) *Identify crash patterns across mobile app executions.* First, MOTIF identifies *crash patterns* among app execution traces collected in the wild. These patterns will be used to automatically extract the minimum set of steps to reproduce crashes and characterize the operating context under which failures arise (cf. Sections IV and V);
3) *Synthesize crash test suites.* Based on the crash patterns collected in the wild and the operating conditions identified in step 2, MOTIF synthesizes a *crash test suite* to reproduce faithfully a category of crashes experienced by users. This test suite will replay a sequence of user interactions that lead to a crash of the application, while taking care not to disclose any sensitive information—*e.g.*, password, login, address (cf. Section VI);
4) *Assess operating contexts that reproduce crashes.* Taking as input the crash patterns identified in step 2, MOTIF learns the contexts where the crash test suites truly reproduce the observed crashes and determine if they can generalize to other contexts or not. Then, it selects candidate devices in the crowd that should be able to reproduce the crashes through the execution of the crash test suites generated in step 3 (*cf.* Section VI-B). Once a test suite is validated, MOTIF notifies the app developers.

The MOTIF architecture includes a cloud server component and an Android client library, which runs on mobile

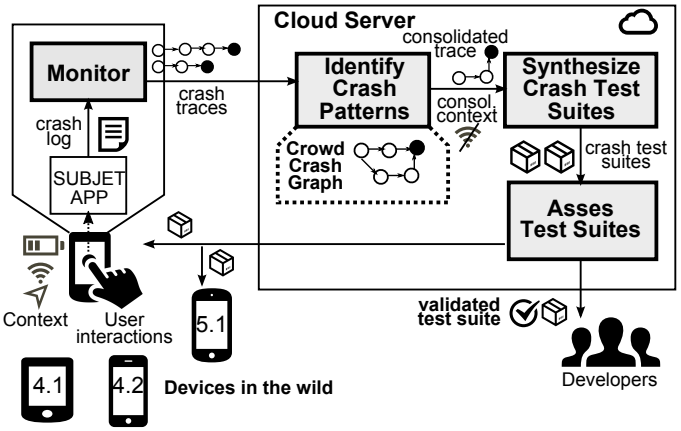



Fig. 1. Overview of the MOTIF proposal

devices. To monitor a subject app, MOTIF does not require accessing the source code of apps neither instrumenting the monitored apps. The only requirement is that apps must be flagged as *debuggable*. Our approach is transparent to users who use their apps normally. Users only have to give their consent to send debugging information when a failure happens in their devices, like current error reporting systems do.

## III. MONITOR THE CROWD

In this section, we first discuss the most popular classes of Android app crashes, and then we present the monitoring strategy used by MOTIF.

### A. Causes of App Crashes

There are many causes that induce app failures. If the failures are handled inadequately in the source code, then the app throws an unhandled exception and the operating system terminates the app. In this paper, we focus on bugs that manifest with crashes. Kechagia *et al.* [25] study a dataset of stack traces collected from real devices and identify 7 causes of Android App crashes. In addition, Liang *et al.* [27] identify *context-related bugs* in mobile apps: *network conditions*, *device heterogeneity* and *sensor input*. We further classify such failures into two groups: *permanent* and *conditional* bugs. The former refers to failures that always arise in the apps (*e.g.*, division by 0). The latter refers to failures that only emerge under specific circumstances or configurations (*e.g.*, GPS unavailable in indoor locations). Table I summarizes the categories of Android app crashes together with a sample app.

Specially, the crashes that depend on context are more challenging to isolate and to reproduce by developers in lab. We aim to complement existing in-house testing solutions with a collaborative approach that monitors apps after their deployment in the wild. Our main goal is to help developers to reproduce crashes by automatically generating a test suite that reproduces the crashes faced by users.

### B. What Context Information to Monitor from the Crowd?

In order to reproduce a crash, information regarding to the actions that the user performed with the app, and the context under which the crash arise are crucial. Thus, during

TABLE I. CATEGORIES OF ANDROID APP CRASHES

| Cause | Permanent/Conditional | Sample app | Problem identification |
|---|---|---|---|
| Missing or corrupted resource | P | PocketTool | *App crashes if the Minecraft game is not installed on the device* |
| Indexing problem | P | Ermete SMS | *App crashes when deleting a phone number taken from the address book* |
| Insufficient permission | P | ACV | *App crashes when long-pressing a folder* |
| Memory exhaustion | C | Le Chti | *App crashes after some navigation steps in the app* |
| Race condition or deadlock | C | Titanium | *App crashes if the back button is clicked during the app launching* |
| Invalid format or syntax | C | PasswdSafe | *App crashes when opening a password that contains Norwegian characters* |
| Network conditions | C | Wikipedia | *App crashes when attempting to save a page without network connectivity* |
| Device heterogeneity | C | Wikipedia | *App crashes when pressing the Menu button on LG Devices* |
| Sensor input | C | MyTracks | *App crashes if the GPS is unavailable* |

the execution of a subject app, MOTIF tracks input events (*e.g.*, user interaction events) and unhandled exceptions thrown by the app. To contextualize events, MOTIF records metadata and context information. Specifically, we have identified the following relevant information to confine crashes:

- *Event metadata*: timestamp, method name, implementation class, thread id, and view unique id,
- *Exception metadata*: timestamp, location, and exception trace,
- *Context data*: information related to the operating context, which we further classify as:
  - *Static properties*. Properties that remain invariable during the whole execution—*e.g.*, device manufacturer, device model and SDK version,
  - *Dynamic properties.* Properties that change along execution—*e.g.*, memory state, battery level, network state, and state of sensors.

### C. Track Input Events

Android apps are UI centric—*i.e.*, `View` is the base class for widgets. To intercept user interaction events, the `View` class provides different event listener interfaces which declare public event handler methods. The Android framework calls the event handler methods, when the respective event occurs [1]. For example, when a view (such as a button) is touched, the method `onTouchEvent` is invoked on that object. MOTIF therefore intercepts the execution of the event handler methods. Each time an event is executed, MOTIF logs both event metadata and context data. Table II reports on a subset of the handler methods intercepted by MOTIF.

TABLE II. EXAMPLES OF ANDROID VIEW TYPES WITH THEIR EVENT LISTENERS AND HANDLER METHODS.

| Type | Event listener interface | Event handler method |
|---|---|---|
| View | OnClickListener | onClick |
| | OnLongClickListener | onLongClick |
| | OnTouchListener | onTouch |
| | OnDragListener | onDrag |
| ActionMenuView | OnMenuItemClickListener | onMenuItemClick |
| AdapterView | OnItemClickListener | onItemClick |
| Navigation Tab | TabListener | onTabSelected |
| | OnTabChangedListener | onTabChange |
| Orientation | OrientationEventListener | onOrientationChanged |
| GestureDetector | OnGestureListener | onDown |
| | | onFling |
| | | onLongPress |
| | | onScroll |
| | | onShowPress |
| | | onSingleTapUp |

### D. Log Crash Traces

During the execution of an app, MOTIF keeps the observed events in memory. If the app crashes, then MOTIF saves the trace of events in a log file in the device. We define a *crash trace* ($ct$) as a sequence of events executed in an app before a crash arises, $ct = \{e_1, e_2, ..., e_n\}$. Events can be of two types: *interaction* and *exception* events. The last event of a trace ($e_n$) is always an exception event. The static context is only reported in exception events, since it remains invariable along the whole app execution. In contrast, the dynamic context is reported for each of the events. Figure 2 depicts an example of a crash trace with two events $e_1$, $e_2$, leading to a crash $crash1$. To minimize

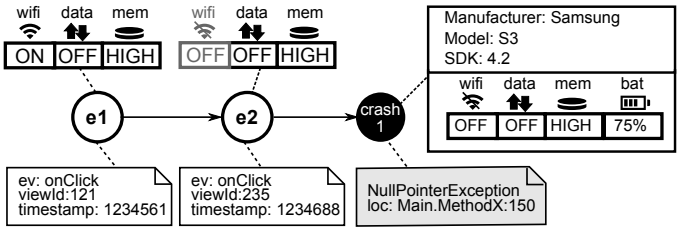



Fig. 2. Example of a crowdsourced crash trace

the impact on battery lifespan and the data subscription of end-users, MOTIF only reports the logs to the cloud server when the device is charging and connected to the Internet.

### E. Adaptive Logging

In order to minimize the runtime overhead, MOTIF performs an *adaptive logging* strategy. In other words, MOTIF logs more information when the suspicion is higher, or when developers request to do it. By default, MOTIF only monitors the raise of uncaught exceptions during the execution of apps. When the number of collected exceptions for a given app reaches a predefined threshold number $N$ of exceptions, then MOTIF flags the app as *buggy-suspicious* and increases the monitoring depth to track user interaction events, additionally. $N$ is a configuration parameter to be decided by app developers when using MOTIF.

## IV. AGGREGATE CROWD DATA

In the cloud, MOTIF aggregates the crash traces collected from a multitude of devices in the wild. We transform the collection of crash traces into a weighted directed graph that we denote as *Crowd Crash Graph*. The *Crowd Crash Graph* represents an aggregated view of all the events performed in an given app before a crash arises, with their frequencies. This model enables MOTIF to induce 1) the minimum sequence of steps to recreate a failure, and 2) the context under which failures arise.

### A. Definition: Crowd Crash Graph

A *Crowd Crash Graph* is a model to provide an aggregated view of all crash traces collected from a multitude of devices which run a given app. The crowd crash graph ($CCG$) consists of a collection of directed graphs: $CCG$=$\{G_1, G_2, ..., G_n\}$, where each $G_i$ is a *crash graph* for a different type of failure.

A *crash graph* aggregates together all crash traces that lead to the same exception. The *crash graph* is based on a Markov chain (1st order Markov model), which is a widely accepted formalism to capture sequential dependences [34]. In our *crash graph*, nodes represent events, and edges represent sequential flows between events. Nodes and edges have attributes to describe event metadata and transition probabilities, respectively. The transition probability between two events ($e_i$, $e_j$) is computed as the ratio of the number of times that $e_i$ and $e_j$ are fired consecutively and the number of times that $e_i$ is fired. In each node, the probability to execute the next event only depends on the current state, and does not take into consideration previous events.

Our crash graphs are based on the idea of graphs proposed by Kim et. al. [26] to aggregate multiple crashes together. Nevertheless, our crash graphs capture a different information. Their nodes represent functions and their edges represent call relationships between functions, extracted from crash reports. On the contrary, our nodes represent events, and edges represent sequential flows. Our crash graphs provide an aggregated view of event traces generated by mobile devices to capture how users interact with an app. In addition, we use crash graphs with a different purpose: to synthesize the most likely sequence of steps to reproduce a crash. We expand the crash graphs with attributes in nodes and edges to store event and context metadata, and represent the graph as a Markov model.

### B. Build the Crowd Crash Graph from Crash Traces

As illustration, we consider a version of the Wikipedia app which contained a bug[3]. The app crashes when the user tries to save a page and the network connectivity is unavailable. Table III shows an example of five traces generated by the subject app.

TABLE III. CRASH TRACES (FIRST COLUMN) AND SINGLE STEPS OF THE TRACES (SECOND COLUMN). IN BRACKETS, THE NUMBER OF OCCURRENCES OF EACH STEP AMONG THE TRACES.

| Crash traces | Single trace steps |
|---|---|
| e1→e2→crash1 | e1→e2(2) e2→crash1(3) |
| e1→e4→e5→e2→crash1 | e1→e4(1) e4→e5(1) |
| e3→e1→e2→crash1 | e5→e2(1) e3→e1(1) |
| e1→e5→crash2 | |
| e1→e6→e5→crash2 | |

Given a set of traces collected from a multitude of devices, MOTIF first aggregates the traces in a single graph. Then, the process to build the Crowd Crash Graph is applied as follows.

*1) Cluster traces by type of failure:* First, MOTIF clusters the traces leading to the same exception together. We can implement different heuristics to identify similar exceptions. For example, Dang et al. [17] propose a method for clustering crash reports based on call stack similarities. In MOTIF, we use an heuristic that considers that two exceptions are the same if they have the same *type* (*e.g.* `java.lang.NullPointerException`) and they raise in the same location—*i.e.*, same *class* and *line* number. For example, in Table III (first column), we identify two clusters of traces. The first cluster contains three traces leading to $crash1$; and the second cluster contains two traces leading to $crash2$.

*2) Merge traces in a crash graph:* Next, for each cluster of traces, we form a *crash graph* following the graph construction technique proposed by [26]. First, we decompose each trace into single steps—*i.e.* pairs of events executed in sequence in a trace. The trace $e_1 \rightarrow e_2 \rightarrow crash1$ contains two steps: $e_1 \rightarrow e_2$ and $e_2 \rightarrow crash1$.

Then, for each event in the step, we create a node in the graph. If the node already exists, we update its weight. Then, we add a directed edge to connect the two events executed in sequence in the step. If the edge already exists, we update its weight. In addition, we create a start node ($S$) that represents the launch of the app, and add edges to connect the start node with the first event node of each trace. Finally, we add the context metadata associated with each event, as attributes in the corresponding event nodes.

Figure 3 shows the resulting *crash graph* from the cluster of traces leading to $crash1$ in Table III. For each step in the graph, we calculate the transition probabilities. For example, after executing event $e1$, the event $e2$ is executed 2 times; and the event $e3$ is executed 1 time. Therefore, the transition probabilities from node $e1$ to $e2$ and $e3$ are $P_{e1-e2} = 0.66$ and $P_{e1-e3} = 0.33$, respectively. The transition probabilities are normalized between 0 and 1. In addition, each node contains a weight indicating the number of occurrences of the event. The event $e2$ was executed 3 times.

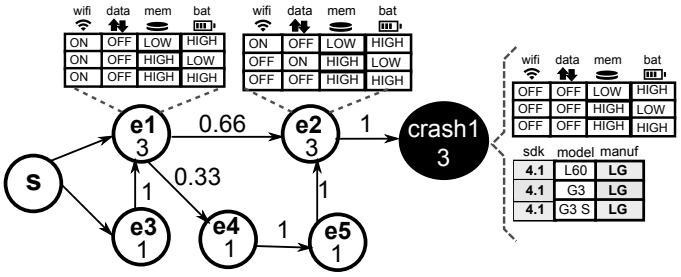



Fig. 3. Crash Graph derived from traces in Table III

*3) Aggregate crash graphs:* Finally, the set of crash graphs (one for each type of exception) is stored in a graph database to form the *Crowd Crash Graph* of a given app. This model provides an consolidated view of the most frequent actions among users before a crash arises together with the observed contexts.

## V. IDENTIFY CRASH PATTERNS

The *Crowd Crash Graph* captures the likelihood, observed from a multitude of devices, that a sequence of events leads to a crash. We assume that the most frequent events are the most relevant ones. Thus, MOTIF uses the *Crowd Crash Graph* to identify repeating patterns of events and contexts which

[3]Bug report: http://git.wikimedia.org/commit/apps\%2Fandroid\%2Fwikipedia.git/7c710ebf044504709148a964b86165189472b7da

appear frequently among crashes. While several data mining techniques can be used, MOTIF implements *Path Analysis* and *Sequential Patterns* algorithms to effectively induce the minimal sequence of steps that reproduces a crash and the context under which this crash arises.

### A. Synthesize Steps to Reproduce Failures

MOTIF applies graph traversal algorithms in order to effectively induce the shortest sequence of steps to reproduce a crash. Some of the collected crash traces can be long and contain irrelevant events to reproduce the failure. For example, consider the trace e1→e4→e5→e2→crash1 (cf. Table III), which includes five steps to crash the app. However, there is a two-steps trace, e1→e2→crash1, which results in the same crash. By exploiting the *Crowd Crash Graph*, MOTIF reduces the size of traces and filters the relevant steps to reproduce crashes. The goal of this phase is therefore to identify the shortest path from the starting node to the exception node which appear with high frequency in most traces.

In graph theory, *breadth-first search (BFS)* [31] is a widely known search algorithm to explore the nodes of a graph. We implement a variant of this algorithm, *weighted breadth first search* (WBFS) [42], which follows a similar strategy to BFS, but considering weights in the edges of a graph.

Figure 4 describes the pseudo-code of the algorithm. The algorithm begins at the start node (S) and traverses the graph until finding the exception node, following a depth-first search strategy. WBFS explores the most frequent nodes first. At each node, it chooses the next node with the highest transition probability. In case all the nodes have the same transition probability, it selects the node with the highest weight. The search continues until it finds the exception node, or reaches a node with all its neighbor nodes visited. In the latter case, it returns to the most recently visited node whose neighbor nodes are unexplored. The algorithm returns the sequence of events from the starting node to the exception node having the maximum likelihood. We denote the output path as the *consolidated trace* in the crowd, and it is promoted as the candidate trace to reproduce the crash.

**Require:** Starting node $S$, Exception node $E$, Crash Graph $G$
**Ensure:** $L_{final}$
  $L_o = \{\}$, $L_{final} = \{\}$
  $L_o \leftarrow S$
  **while** $L_o$ is not empty **do**
    **for all** node $n \in L_o$ **do**
      **for all** edge $e$ incident on $n$ **do**
        **if** $e$ has maximum transition probability **then**
          $n_d$ = endpoint node of $e$
          **if** $n_d$ is unexplored **then**
            insert $n$ in $L_{final}$
            insert $n_d$ in $L_o$
          **end if**
        **end if**
      **end for**
    **end for**
  **end while**

Fig. 4. Pseudo-code of the WBFS algorithm

In the crash graph of Figure 3, the *consolidated trace* to reproduce $crash1$ is *e1→e2→crash1*. The algorithm starts at node $S$ and selects node $e1$ since has the highest weight (3). After $e1$, it selects event $e2$ since has the highest transition probability (0.66). And finally, it finds the exception node $crash1$.

The algorithm can return $N$ different traces ordered by descending probability. If the trace does not reproduce the crash, then MOTIF tries with the next trace.

### B. Learn Error-prone Operating Contexts

Given the *consolidated trace*, MOTIF discovers frequent context properties and context changes along the trace. As previously mentioned, not all the devices suffer from same bugs and some crashes only raise under specific contextual conditions—*e.g.* network unavailable. Thus, MOTIF searches for recurrent context patterns among the observed traces. The context will help to 1) reproduce context-sensitive crashes, 2) select the candidate devices to assess the generated test suites, and 3) select devices to check that future fixes do not produce any side effects.

In order to learn frequent contexts, we use *Sequential Pattern Mining*, which is a well-known data mining technique to discover frequent subsequences in a sequence database [29]. A sequence is a list of itemsets, where each itemset is an unordered set of items. Figure 5 describes three sequences of context properties observed along the consolidated trace (synthesized from the graph in Figure 3). Each sequence contains 4 itemsets, one for each of the events in the trace and each item maps to a context property.

Specifically, we mine *frequent closed sequential patterns—i.e.*, the longest subsequence with a given support. The support of a sequential pattern is the number of sequences where the pattern occurs, divided by the total number of sequences. MOTIF searches for closed sequential patterns with support 100%—*i.e.*, patterns that appear in all the observed traces. To ensure that the context truly induces the crash, such context should be common to all the traces. Among the available algorithms to mine closed sequential patterns (*e.g.*, BIDE+, CloSpan, ClasSP), we choose *BIDE+* because of its efficiency in terms of execution time and memory usage [41]. In particular, we use the implementation of BIDE+ available in the SPMF tool [8].

In Figure 5, the algorithm identifies the following frequent context pattern is: {(wifiON, dataOFF), (wifiOFF, dataOFF), (sdk4.1, LG)}. That is to say, the crash affects LG devices, which run Android 4.1. In addition, when the crash arises when the network has been disconnected.

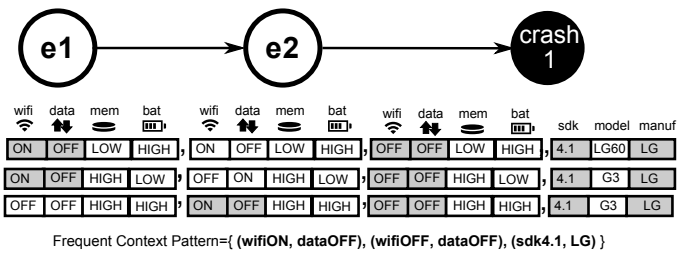



Fig. 5. Learning Error-prone context from a candidate trace

## VI. Synthesize Crash Test Suites

Based on the *trace* (cf. Section V-A) and the *error-prone context* (cf. Section V-B) consolidated in the crowd, MoTiF generates a test suite to faithfully reproduce the crash. These test suites recreate a sequence of user interactions that lead to the crash of the app, while taking care of not disclosing any sensitive information (*e.g.*, password, login, address). Then, the generated test suites are executed in the crowd of devices to assess if they truly reproduce the observed failure in the proper context.

### A. Generate Crowd-tests

In order to help developers to reproduce crashes faced by users in the wild, MoTiF generates black-box UI tests to automatically recreate the consolidated trace. We use *Robotium*, which is a test automation framework to write powerful and robust automatic black-box UI tests for Android applications [5]. *Robotium* extends the Android test framework to ease writing tests. We chose *Robotium* because has full support for native and hybrid applications, it does not require the source code of the application under test, and it provides fast test case execution.

*Robotium* provides full support to interact with the UI of Android apps. `Solo` is the main class to develop Robotium tests. The `Solo` class provides methods to interact with graphical elements (*e.g.*, `clickOnButton`, `enterText`, `clickOnMenuItem`), set the orientation (`setActivityOrientation`), and set the context (*e.g.*, `setWiFiData`, `setMobileData`).

We propose mapping rules between the Android event handler methods (cf. Section III-C) and the methods provided by the Robotium API.[4] For example, the Android event `onClick` in a view of type `Button` is mapped with the *Robotium* method `clickOnButton`. Table IV shows a subset of the mapping rules identified.

TABLE IV. Examples of mappings between Android event handler methods and Robotium methods

| Element | Android event handler method | Robotium method |
|---|---|---|
| View | onClick | clickOnView |
| | onLongClick | clickLongOnView |
| Button | onClick | clickOnButton |
| ActionMenu | onMenuItemClick | clickOnMenuItem |
| Orientation | onOrientationChange | setActivityOrientation |

These rules guide the automatic generation of test cases. MoTiF defines a base template for a Robotium test case (cf. Figure 6). First, MoTiF adds the error-prone context as an annotation in the test case (A). Second, MoTiF sets the launcher activity of the subject app (B). Finally, it generates a test method to recreate the steps of the candidate trace (C). Using the mapping rules, MoTiF translates each event in the trace into a Robotium method invocation.

Figure 6 shows the test case generated for the Wikipedia app. The test method `testRun` recreates the consolidated trace. Lines 2 and 6 correspond to the events $e1$ and $e2$ in the trace, respectively. Lines 1 and 3 represent delays between events. MoTiF calculates the delay between two

[4]http://robotium.googlecode.com/svn/doc

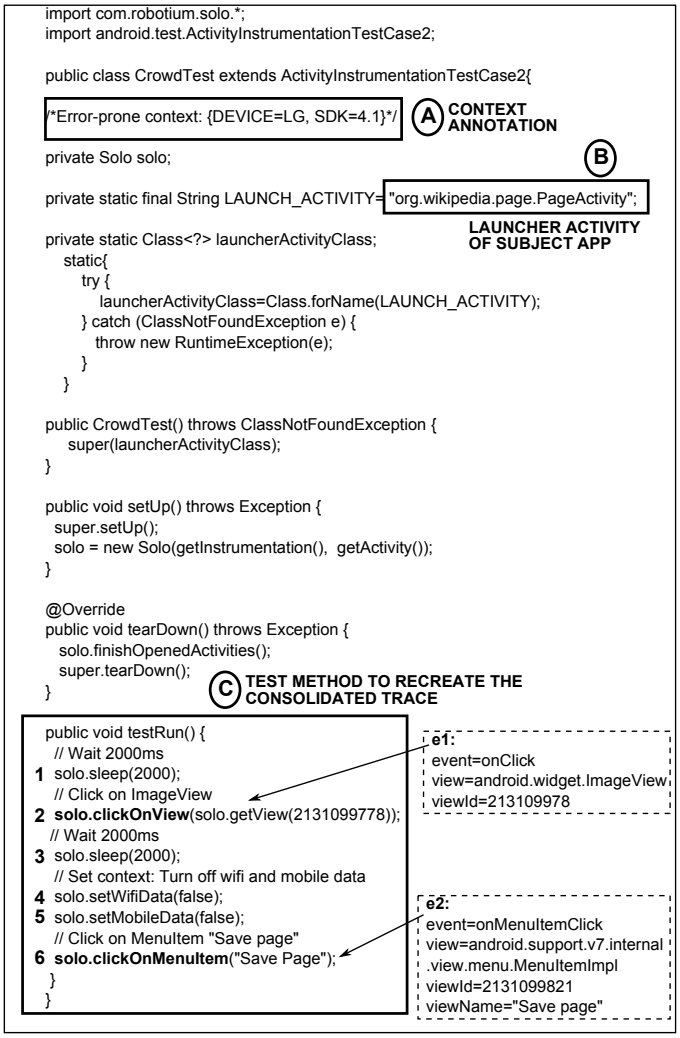



Fig. 6. Generated Robotium test case for the Wikipedia app.

events as the average of all the observed delays between those events. Finally, lines 4 and 5 set the network context. Network-related contexts can be automatically induced in the test cases because Robotium provides dedicated methods (`setWiFiData`, `setMobileData`) for this purpose. For the reminder context properties, like *OutOfMemory*, MoTiF adds the observed context as an annotation in the test case to help developers to isolate the cause of failures.

### B. Crowd-validation of Crash Test Suites

Before providing the generated test suites to developers, MoTiF executes the tests in the crowd of real devices to asses if: 1) they truly reproduce the observed crashes, and 2) they can generalize to other contexts/devices or not.

First, MoTiF uses the static context to select a sample of devices that match the context profile (*e.g.*, LG devices), and then checks if the test case reproduces the crash in those devices. MoTiF incorporates a heuristic to assess test cases: the test case execution fails and collects the same exception trace that the original failure observed in the wild.

Later, MoTiF selects a random sample of devices that do not match the context profile, and test if they reproduce the

crash. If the test case also reproduces the crash in a different context, MOTIF concludes that the context does not induce the crash. In this case, MOTIF adds the context in the test case as an informative note to developers about which are the most frequent devices running their apps. If on the contrary, the test case only reproduces the failure in the consolidated context, the context will be included as *critical* in an annotation in the test case. In addition, using the crowd of devices MOTIF can learn different rules. For example, crashes that only emerge in devices running in $sdk < 4.2$. The context rules will help developers to isolate bugs, and select devices to asses the quality of the posterior fixes.

To avoid any user disturbance, MOTIF executes the tests for validation only during periods of phone inactivity, for example during nights, and when the device is charging. Users should therefore give their consent to enable MOTIF to use their devices for crowd-validation.

### C. Privacy Issues

All approaches that record user inputs put privacy at a risk [44]. Our approach provides test suites to replay a sequence of user interactions that lead to a crash of the application, while taking care of not disclosing any sensitive information (*e.g.*, password, login, address). MOTIF exploits the crowd to mitigate privacy issues. Specifically, we incorporate two mechanisms: *anonymization* [13] and *input minimization* [45] techniques.

First, to ensure user anonymity, MOTIF assigns an anonymous hash value, which identifies each app execution in a specific device. Thus, different apps running in the same device produce different ids. The pseudo id cannot reveal the original device id (which can expose the identity of the user).

Since the collected information can contain personal and confidential information (*e.g.*, passwords, credit card data), MOTIF applies the input minimization approach proposed by Zeller and Hildebrandt [45] to simplify the input and include in tests only the relevant part. For example, consider the Android app *PasswdSafe*, which allows users to store all passwords in a single database. The app contained a bug[5] and crashed when opening a password that contained the Spanish character *ñ*. It is undesirable that MOTIF provides all the user's passwords to developers. Thus, MOTIF applies the minimization technique in all the user's inputs observed in the crowd and extracts the minimum relevant part that produces the crash. For example, consider three passwords from three different users that crash the PasswdSafe app: *"España"*, *"niño"*, and *"araña"*. MOTIF identifies 'ñ' as the minimum input to reproduce the crash, and include this input in the test suites instead of the original input that will reveal sensitive information.

## VII. IMPLEMENTATION DETAILS

This section provides details about the infrastructure that supports our approach. MOTIF does not instrument neither apps nor the operating system. MOTIF can monitor any *debuggable* app[6] running in devices, without requiring access to their source code. For the time being, our prototype implementation requires rooted devices and is composed of two parts: a mobile client library which runs on devices and a cloud service.

### A. Android client library.

The Android virtual machine (named *Dalvik*) implements two interfaces: the *Java Debug Interface (JDI)*[7] and the *Java Debug Wire Protocol (JDWP)*[8], which are part of the *Java Platform Debugger Architecture (JPDA)*[9]. This technology allows tools to communicate with the virtual machine. Android provides the `adb` tool (Android Debug Bridge) to communicate with Dalvik. The MOTIF's client app communicates with Dalvik via `adb` and the standard debugging interfaces JDWP and JDI over a network socket. Our tool extends and reuses part of the implementation provided by *GROPG* [33], an on-phone debugger. These techniques enable to monitor apps and to intercept user interaction and exception events. The *GROPG* implementation ensures low memory overhead and fast execution.

### B. Cloud service.

MOTIF sends the information collected in devices to a cloud service for aggregation and analysis. APISENSE[10] provides a distributed crowd-sensing platform to design and execute data collection experiments in the wild using smartphones [21]. MOTIF uses the APISENSE service for this purpose.

To store and aggregate the crash traces collected from the crowd, MOTIF creates a *graph database* with *Neo4J*[11]. Graph databases provide a powerful and scalable data modelling and querying technique capable of representing any kind of data in a highly accessible way [37]. We choose a graph database to store the crowd crash graphs. We can query the graph database using one of the available graph query languages (*e.g.*, Cypher, SPARQL, Gremlin). We chose *Cypher*[12], because is a widely used pattern matching language. Figure 7 shows an excerpt of the Crowd Crash Graph in Neo4J of the *Wikipedia* app.

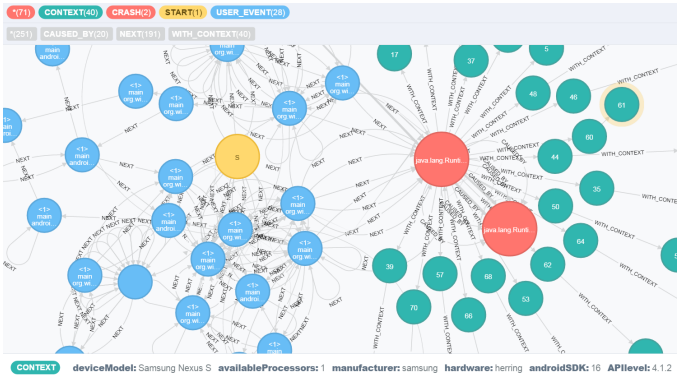



Fig. 7. Crowd crash graph in Neo4J

[5]Bug report PasswdSafe: http://sourceforge.net/p/passwdsafe/bugs/3
[6]Apps wich have the `android:debuggable` attribute in the manifest.
[7]http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/jdi/index.html
[8]http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/jdwp-spec.html
[9]http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/architecture.html
[10]APISENSE: http://www.apisense.com
[11]Neo4J: http://www.neo4j.org
[12]Cypher: http://docs.neo4j.org/refcard/2.0

## VIII. CASE STUDY

In this section, we report on experiments we performed to demonstrate the applicability of our approach.

We consider a case study with four existing buggy Android apps. We select buggy apps with different sizes, complexities, open-source, proprietary and which contain different types of bugs. Table V (left) lists the apps used in the study, with their version, size, type, and bug type. To perform the experiments we use 2 different devices *Samsung S3* (with Android 4.3): 1 *Google Nexus S* (running Android 4.1.2) and 1 *Wiko* (with Android 4.2). This study investigates the following research questions:

- **RQ1:** *How does* MOTIF *compare with the* Android Monkey tool*?*
- **RQ2:** *Can* MOTIF *synthesize test suites which effectively reproduce crashes?*
- **RQ3:** *What is the overhead of* MOTIF*?*

### A. Experimental Results

*1) Comparison with Monkey:* In in order to get an estimation about the difficulty to reproduce the bugs in the subject apps, first we run *Monkey* with our apps (**RQ1**). *Monkey* is a well-known testing tool (provided by Android), which generates pseudo-random user events (such as clicks, touches, or gestures, as well as system-level events) in apps running in device or emulators [10]. If the app crashes or receives an unhandled exception, *Monkey* stops and reports the error. We launch *Monkey* to send $50,000$ events to each of our apps. The apps *Bites* and *PocketTool* crash after executing $9,480$ events and 33 events, respectively. On the contrary, *Monkey* cannot find the crashes in the *Wikipedia* and *OpenSudoku* apps. Our approach can complement existing testing solutions, to discover crashes after deployment in the wild.

*2) Effectiveness:* Next, we study if MOTIF can generate crash test suites, which effectively reproduce the observed crashes (**RQ2**). We started by collecting crash traces from the subject apps running in our different devices. Table V (right) shows, for each subject app, the size of the crash log stored in the device, the number of crash traces collected, the average number of events in each trace, and the number of events in the consolidated trace obtained by MOTIF. We observe that the total number of events in the crowd-consolidated trace (generated by MOTIF) is smaller than the average size of original traces. For example, in the *Wikipedia* app, the average size of traces is $8.55$. However, MOTIF synthesizes from the aggregated crowd data 2 relevant events to reproduce the crash, together with a relevant context: *network disconnection*. Table VI lists the crowd-consolidated traces to reproduce the context-sensitive crashes.

From the consolidated traces, MOTIF generates Robotium test suites. The generated crowd crash graphs and test suites are available in the online appendix.[13]

We executed the test cases in devices and we observe that the test cases truly reproduces the bugs in the $4$ apps. That is to say, the execution of the test cases generates the same exception type in the same location that the original failures.

[13]Online appendix: https://sites.google.com/site/spiralsmotifase

TABLE VI. CROWD-CONSOLIDATED TRACES TO REPRODUCE CRASHES

| App | Events |
|---|---|
| Wikipedia | 1) Click on ImageView Menu<br>2) Disconnect WiFi and mobile data<br>3) Click on MenuItem "Save Page" |
| OpenSudoku | 1) Click on ListItem position 1<br>2) LongClick in ListItem position 1<br>3) Click on MenuItem "Edit note"<br>4) Click on Button "Save"<br>5) LongClick in ListItem position 1<br>6) Click on MenuItem "Delete puzzle"<br>7) Click on Button "Ok"<br>8) Change orientation |
| Bites | 1) Click on Tab "Method"<br>2) Click on context menu<br>3) Click on MenuItem "insert2"<br>4) Touch text field "Method"<br>5) Click Button "ok" |
| PocketTool | 1) Click on Button "Took Kit"<br>2) Click on Button "Change Textures/Skin" |

To further evaluate the synthesized test cases, we consider a patched version for each app. For the *Wikipedia* app, we use an existing updated version of the app that fixes this bug[14]. For *OpenSudoku* and *Bites*, since the crashes still exist in the apps and the source code is available, we manually add a patch. This patch wraps the code in the method that throws the exception with a try/catch block to capture the runtime exceptions that are not handled by the methods. We follow the same patching strategy with the *PocketTool* app. However, since it is a propietary app and the code is not available, we instrument the bytecode to inject the patch (cf. [19] for more details). Then, we execute the tests against the patched apps and we observe that the test suites pass in the patched versions.

*3) Overhead:* The overhead introduced by MOTIF is imperceptible to the users when interacting with their apps. On the one hand, the runtime overhead to store exception events is $0$, since MOTIF logs the exception events after the app has crashed. On the other hand, the overhead to log a user interaction trace is $149\ ms$ on average. In addition, in order to minimize overhead, MOTIF performs an adaptive logging strategy—*i.e.,* only it logs user interactions when an app is buggy-suspicious (cf. Section III-E).

### B. Discussion and Threats

The preliminary results of our evaluation show that our approach is feasible and effective, for the subject apps we considered. The case study uses $4$ real buggy apps with different types of crashes, nonetheless further analyses are necessary to evaluate the efficacy of this approach with different types of crashes. Currently, we are working on experiments to further assess the approach.

The main benefits of using a *crowdsourced debugging* approach are the following. First, developers focus on real usages of applications; this enables them to tackle the most critical functionalities for users. Second, the crowd offers an high diversity of devices and contexts, which are difficult to simulate in lab. Third, by leveraging crowd feedback in a smarter way, it is possible to extract relevant information to reproduce context-sensitive failures, and to validate the extracted assumptions.

[14]Bug fix: http://git.wikimedia.org/commit/apps\%2Fandroid\%2Fwikipedia.git/7c710ebf044504709148a964b86165189472b7da

TABLE V. ANDROID APPS UNDER TEST. STATISTICS ON THE COLLECTED CRASH DATA

| Android App | Size (Kb) | Type | Crash type | Log size (Kb) | #Traces | Avg. events in crash traces | #Events in the crowd consolidated trace |
|---|---|---|---|---|---|---|---|
| Wikipedia (v2.0 − $alpha$) | 5650 | Open-source | Network conditions | 44.50 | 20 | 8.55 | 2 |
| OpenSudoku (v1.1.5) | 536 | Open-source | NullPointerException | 34.89 | 14 | 10.14 | 8 |
| Bites (v1.3) | 208 | Open-source | Invalid format | 44.40 | 21 | 7.43 | 5 |
| PocketTool (v1.6.12) | 1410 | Proprietary | Missing resource | 34.90 | 21 | 4.27 | 2 |

Our approach does not require any critical number of users to work. As soon as MOTIF collects one single trace, it can synthesize a test case to reproduce this trace. However, the bigger the number of users (with higher diversity), the more accurate the results that MOTIF produces. In future work, we will study mechanisms to provide incentives in order to motivate users to participate in crowdsourced debugging experiments.

## IX. RELATED WORK

This section summarizes the state of the art in the major disciplines that are related to this research.

*Mobile App Testing*. Currently, a wide range of testing tools for Android apps is available: *Monkey* [10], *Calabash* [2], *Robotium* [5], *Selendroid* [6]. In addition, previous researches have investigated GUI-based testing approaches for Android apps [15], [23], [30], [32]. The aforementioned approaches do not consider different operating contexts in the tests. Furthermore, Liang et al. [27] present CAIIPA, a cloud service for testing Windows phone apps over different operating contexts. Finally, several commercial solutions (*e.g.*, XAMARIN TEST CLOUD [11], TESTDROID [9]) exploit the cloud to test an app on hundreds of devices simultaneously. Despite the prolific research in this area, testing approaches cannot guarantee the absence of unexpected behaviors in the wild. Our approach aims to complement existing testing solutions, with a monitoring solution after deployment to help developers to quickly detect and fix bugs.

*Crash Reporting Systems*. Current crash reporting systems on mobile apps (*e.g.*, SPLUNK [7], GOOGLE ANALYTICS [4]) collect raw analytics on the execution of apps. Our approach goes beyond current crash reporting systems by exploiting crowd feedback in a smarter way. MOTIF provides developers *in vivo test suites*, which defines the steps to reproduce crashes and the context that induce the failures. The test suites are validated in the crowd before delivery to developers.

*Mobile Monitoring in the Wild.* Agarwal et al. [12] propose MOBIBUG, a collaborative debugging framework that monitors a multitude of phones to obtain relevant information about failures. This information can be used by developers to manually reproduce and solve failures. They do not consider privacy issues in their design. APPINSIGHT [35] is a system to monitor app performance in the wild for the Windows Phone platform. APPINSIGHT instruments mobile apps to automatically identify the critical path in user transactions, across asynchronous-call boundaries. On the contrary, they do not synthesize test cases.

*Monitoring User Interactions to Reproduce Bugs*. Monitoring user interactions for testing and bug reproduction purposes have been successfully applied in other domains, such as Web or desktop applications [22], [38]. *MonkeyLab* [28] is an approach to mine GUI-based models based on recorded executions of Android apps. The extracted models can be used to generate actionable execution scenarios for both natural and unnatural sequences of events. Our approach do not require to learn models of the GUI to generates steps to reproduce failures. In addition, our approach deals with context information, since the context is crucial to reproduce failures in mobile environments. We also incorporate a crowd-validation step, to assess the consolidated traces and contexts. Our approach is also related with record and and replay approaches. We do not use existing recording and replay tools (*e.g. RERAN* [18], *Android getevent tool* [3]) because in these approaches the recorded actions only fit one device at a fixed screen. Thus, the actions only can be reproduced in the same device in which were recorded. In order to reproduce context-related bugs, we need generic scripts that can be reproduced in a multitude of different devices in order to assess the validity of the consolidated contexts.

*Reproduce Field Failures*. The last group includes techniques to detect and reproduce crashes. Jin and Orso [24] introduce BUGREDUX to recreate field failures in the lab in desktop programs. STAR [16] provides a framework to automatically reproduce crashes from crash stack traces for object-oriented programs. Rö$\beta$ler et al. [36] introduce the approach BUGEX that leverages test case generation to systematically isolate failures and characterize when and how the failure occurs. Artzi et al. introduce RECRASH [14], a technique to generate unit tests that reproduce software failures. Nevertheless, all the aforementioned techniques are not available for mobile platforms.

For the mobile platform, *Crashdroid* [43] proposes an approach to automatically generate steps to reproduce bugs in Android apps, by translating the call stack from crash reports. They do not consider context information, then the approach cannot deal with context-sensitive crashes. In *Crashdroid*, first developers have to provide natural language descriptions of different scenarios of the apps under test. MOTIF can synthesize steps to reproduce crashes, without any preprocessing from developers.

## X. CONCLUSION

Due to the abundant competition in the mobile ecosystem, developers are challenged to rapidly identify, replicate and fix bugs, in order to avoid losing customers and credits.

This paper presents MOTIF, a crowdsourced debugging approach to help developers to detect and reproduce context-related bugs in mobile apps after their deployment in the wild. MOTIF leverages, in a smart way, crash and device feedback from the crowd to quickly detect crash patterns across devices. By using the crash patterns, MOTIF synthesizes in vivo crash test suites to reproduce the crashes. Then, MOTIF exploits the

crowd of devices to check the presence of such crashes and assess the tests in different contexts.

We evaluate the approach in a case study with 4 existing crashes in real apps with different characteristics. Our preliminary results demonstrate that MoTiF can effectively reproduce real crashes with low overhead. MoTiF takes into consideration privacy and energy issues in its design.

As future work, we plan to further evaluate our approach with different types of crashes and apps. We will also study mechanism to encourage users to collaborate in debugging experiments.

## References


[1] Android developers guide. Input events. http://developer.android.com/guide/topics/ui/ui-events.html. [Online; accessed May-2015].

[2] Calabash. http://calaba.sh/. [Online; accessed May-2015].

[3] Getevent tool. https://source.android.com/devices/input/getevent.html. [Online; accessed May-2015].

[4] Google Analytics. http://www.google.com/analytics. [Online; accessed May-2015].

[5] Robotium. https://code.google.com/p/robotium. [Online; accessed May-2015].

[6] Selendroid. http://selendroid.io/. [Online; accessed May-2015].

[7] SPLUNK. https://mint.splunk.com. [Online; accessed May-2015].

[8] SPMF: An open-source data mining library. http://www.philippe-fournier-viger.com/spmf/index.php. [Online; accessed May-2015].

[9] Testdroid. http://testdroid.com. [Online; accessed May-2015].

[10] UI/Application Exerciser Monkey. http://developer.android.com/tools/help/monkey.html. [Online; accessed May-2015].

[11] Xamarin Test Cloud. http://xamarin.com/test-cloud. [Online; accessed May-2015].

[12] S. Agarwal, R. Mahajan, A. Zheng, and V. Bahl. Diagnosing Mobile Applications in the Wild. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, HotNets, pages 22:1–22:6. ACM, 2010.

[13] C. C. Aggarwal and S. Y. Philip. *A general survey of privacy-preserving data mining models and algorithms*. Springer, 2008.

[14] S. Artzi, S. Kim, and M. Ernst. ReCrash: Making Software Failures Reproducible by Preserving Object States. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, ECOOP, pages 542–565. 2008.

[15] T. Azim and I. Neamtiu. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA, pages 641–660. ACM, 2013.

[16] N. Chen and S. Kim. STAR: Stack Trace based Automatic Crash Reproduction via Symbolic Execution. *IEEE Transactions on Software Engineering*, (99):1–1, 2014.

[17] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE'12, pages 1084–1093, Piscataway, NJ, USA, 2012. IEEE Press.

[18] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. RERAN: Timing- and touch-sensitive record and replay for Android. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 72–81, May 2013.

[19] M. Gomez, M. Martinez, M. Monperrus, and R. Rouvoy. When App Stores Listen to the Crowd to Fight Bugs in the Wild. In *37th International Conference on Software Engineering (ICSE), track on New Ideas and Emerging Results (NIER)*, Firenze, Italy, May 2015. IEEE.

[20] M. Gomez, R. Rouvoy, M. Monperrus, and L. Seinturier. A Recommender System of Buggy App Checkers for App Store Moderators. In D. Dig and Y. Dubinsky, editors, *Proceedings of the 2nd ACM International Conference on Mobile Software Engineering and Systems*, MobileSoft, Firenze, Italy, May 2015. IEEE.

[21] N. Haderer, R. Rouvoy, and L. Seinturier. Dynamic deployment of sensing experiments in the wild using smartphones. In *Proceedings of the International Conference on Distributed Applications and Interoperable Systems*, DAIS'13, pages 43–56, 2013.

[22] S. Herbold, J. Grabowski, S. Waack, and U. Bunting. Improved bug reporting and reproduction through non-intrusive gui usage monitoring and automated replaying. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 232–241. IEEE, 2011.

[23] C. Hu and I. Neamtiu. Automating GUI Testing for Android Applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST, pages 77–83. ACM, 2011.

[24] W. Jin and A. Orso. BugRedux: Reproducing Field Failures for In-house Debugging. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE, pages 474–484. IEEE Press, 2012.

[25] M. Kechagia, D. Mitropoulos, and D. Spinellis. Charting the API minefield using software telemetry data. *Empirical Software Engineering*, pages 1–46, 2014.

[26] S. Kim, T. Zimmermann, and N. Nagappan. Crash graphs: An aggregated view of multiple crashes to improve crash triage. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 486–493. IEEE, 2011.

[27] C.-J. M. Liang, N. D. Lane, N. Brouwers, L. Zhang, B. F. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, and Chandra. Caiipa: Automated Large-scale Mobile App Testing through Contextual Fuzzing. In *Proceedings of the 20th International Conference on Mobile Computing and Networking*, MobiCom. ACM, 2014.

[28] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, and D. Moran, K Poshyvanyk. Mining android app usages for generating actionable gui-based execution scenarios. In *12th IEEE Working Conference on Mining Software Repositories (MSR'15)*, May 2015.

[29] N. R. Mabroukeh and C. I. Ezeife. A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys (CSUR)*, 43(1):3, 2010.

[30] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, pages 224–234, New York, NY, USA, 2013. ACM.

[31] E. F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching, and Annals of the Computation Laboratory of Harvard University*, pages 285–292. Harvard University Press, 1959.

[32] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon. GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering*, 21(1):65–105, 2014.

[33] T. A. Nguyen, C. Csallner, and N. Tillmann. Gropg: A graphical on-phone debugger. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 1189–1192. IEEE, 2013.

[34] J. R. Norris. *Markov chains*. Number 2. Cambridge university press, 1998.

[35] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile App Performance Monitoring in the Wild. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, pages 107–120, 2012.

[36] J. Rö$\beta$ler, G. Fraser, A. Zeller, and A. Orso. Isolating failure causes through test case generation. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA, pages 309–319. ACM, 2012.

[37] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases*. O'Reilly, June 2013.

[38] T. Roehm, N. Gurbanova, B. Bruegge, C. Joubert, and W. Maalej. Monitoring user interactions for supporting failure reproduction. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 73–82. IEEE, 2013.

[39] Statista Inc. Cumulative number of apps downloaded from the Google Play Android app store as of July

2013 (in billions). http://www.statista.com/statistics/281106/number-of-android-app-downloads-from-google-play. [Online; accessed May-2015].

[40] VisionMobile. Developer Economics Q3 2014: State of the Developer Nation. Technical report, July 2014.

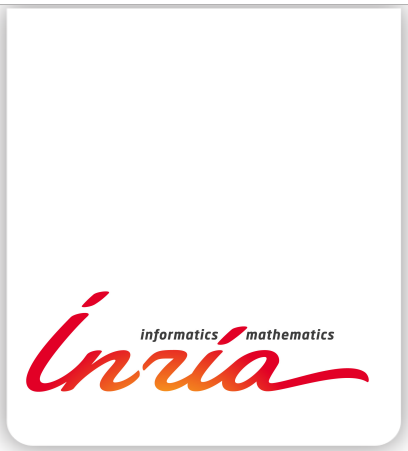
[41] J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 79–90. IEEE, 2004.

[42] Y. Wang, L. Li, and D. Xu. Pervasive qos routing in next generation networks. *Computer Communications*, 31(14):3485 – 3491, 2008.

[43] M. White, M. Linares-Vásquez, P. Johnson, C. Bernal-Cárdenas, and D. Poshyvanyk. Generating Reproducible and Replayable Bug Reports from Android Application Crashes. In *23rd IEEE International Conference on Program Comprehension (ICPC)*, May 2015.

[44] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[45] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on*, 28(2):183–200, 2002.




**RESEARCH CENTRE**
**LILLE – NORD EUROPE**

Parc scientifique de la Haute-Borne
40 avenue Halley - Bât A - Park Plaza
59650 Villeneuve d'Ascq

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399