Reproducing Context-sensitive Crashes of Mobile Apps using Crowdsourced Monitoring

María Gómez, Romain Rouvoy, Bram Adams, Lionel Seinturier

Reproducing Context-sensitive Crashes of Mobile Apps using Crowdsourced Monitoring

Maria Gomez, Romain Rouvoy, Lionel Seinturier
University of Lille & Inria, France
firstname.lastname@inria.fr

Bram Adams
MCIS, Polytechnique Montreal
bram.adams@polymtl.ca

ABSTRACT

While the number of mobile apps published by app stores keeps on increasing, the quality of these apps varies widely. Unfortunately, for many apps, end-users continue experiencing bugs and crashes once installed on their mobile device. Crashes are annoying for end-users, but they definitely are for app developers who need to reproduce the crashes as fast as possible before finding the root cause of the reported issues. Given the heterogeneity in hardware, mobile platform releases, and types of users, the reproduction step currently is one of the major challenges for app developers. This paper introduces MoTiF, a crowdsourced approach to support app developers in automatically reproducing context-sensitive crashes faced by end-users in the wild. In particular, by analyzing recurrent patterns in crash data, the shortest sequence of events reproducing a crash is derived, and turned into a test suite. We evaluate MoTiF on concrete crashes that were crowdsourced or randomly generated on 5 Android apps, showing that MoTiF can reproduce existing crashes effectively.

CCS Concepts

•**Software and its engineering** → *Software post-development issues;* •**Human-centered computing** → *Collaborative and social computing; Ubiquitous and mobile computing;*

Keywords

Mobile app crash reproduction, Android apps, Context-sensitive crashes, Crowdsourcing

1. 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 [47], while projections of the number of app downloads across all app stores for 2016 ran up to 300 billion. Revenue-wise, app developers are expected to collect more than 70 billion dollar by 2017 [26].

Despite the huge number of mobile apps available, the quality of these apps varies greatly, as shown by the large variety in app ratings [31] and reviews [27, 30]. Apart from complaints about missing features, the majority of issues are related to app crashes, either upon fresh installation of an app or after an update [15]. Although mobile app developers can use a wide range of testing tools [6, 19, 29, 41] to detect such crashes prior to release, many bugs may still emerge once deployed to end-users.

When a crash is reported by users, developers must quickly fix their app to stay competitive in the ever-growing mobile computing landscape. This is especially important when considering that negative reviews for early releases make it almost impossible to recover later on [31]. Similar to desktop and web apps, the first task to fix a mobile app crash is to *reproduce* the problem [55]. Although any software developer knows that faithfully reproducing failures that users experience *in vivo* is a major challenge. Successful research has been conducted in crash reproduction in desktop programs [9, 10, 21, 39]. Nevertheless, the crash reproduction task poses additional challenges in mobile environments due to high device fragmentation, rapid platform evolution (SDK, OS), and diverse operating context (*e.g.*, sensors) [1].

As an illustration, users recently experienced crashes with the Android *Wikipedia* app [51], which crashed when the user pressed the *menu* button. However, this crash only emerged on LG devices running Android 4.1. Thus, app developers need to know the *user interactions* and the *execution context* (*i.e.*, software and hardware configuration) that led to crashes to faithfully reproduce such crashes.

To overcome this issue, we present MoTiF[1], which uses machine learning techniques atop of data crowdsourced from real devices and users, 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 crashes 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 aims to complement existing testing solutions with novel mechanisms to monitor and debug apps *in vivo*.

Beyond existing crash reporting systems for mobile apps

[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.

ACM ISBN .
DOI:

(such as *Google Analytics* [16] and *SPLUNK* [45]), which collect raw analytics on the execution of apps, MoTiF automatically identifies recurrent crash patterns among user actions and contexts to generate *in vivo crash test suites* to faithfully reproduce 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.

Finally, MoTiF uses the crowd of devices to assess whether or not the generated test suites truly reproduce the observed crashes and can generalize to other contexts. 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, and we therefore evaluate MoTiF on two crash data sets generated either manually or automatically on 5 real Android apps, and we demonstrate that crashes can be reproduced effectively, and with a low overhead.

The remainder of this paper is organized as follows. Section 2 provides an overview of the proposed approach. Section 3 describes the monitoring strategy followed by MoTiF. Section 4 introduces *Crowd Crash Graphs*, a technique to aggregate in a meaningful manner traces collected from a multitude of devices. Section 5 presents the algorithms to extract consolidated steps and contexts to reproduce crashes. Section 6 illustrates the test case generation technique from patterns extracted from the crowd. Section 7 provides implementation details. Section 8 evaluates the approach. Section 9 summarizes the related work. Finally, Section 10 concludes the paper and outlines future work.

2. OVERVIEW

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

1. *Collecting 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 3);

2. *Identifying 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 sequence of steps to reproduce crashes and characterize the operating conditions (*i.e.*, execution context) under which failures arise (cf. Sections 4 and 5);

3. *Synthesizing crash test suites.* Based on the crash patterns collected in the wild and the execution contexts identified in step 2, MoTiF synthesizes a *crash test suite* to faithfully reproduce a category of crashes experienced by users. This test suite will replay a sequence of user interactions that led to a crash of the application, while taking care not to disclose any sensitive information—*e.g.*, login, password (cf. Section 6);

4. *Assessing execution 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 whether or not they can generalize to other contexts. 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 6.2). Once a test suite is validated, MoTiF notifies the app developers.

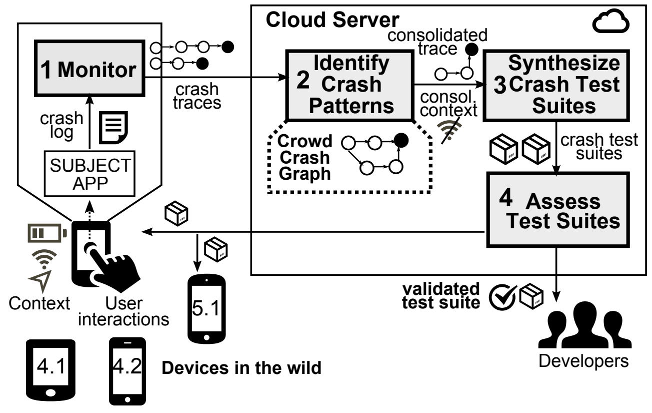


Figure 1: Overview of MoTiF.

The MoTiF architecture includes a cloud server component and an Android client library that runs on the mobile device. Our approach is transparent to users, who can keep on using their apps as usual. They only have to give their consent to automatically report debugging information when an app crashes in their devices, just like current error reporting systems do.

We envision two intended uses of MoTiF. First, when a developer cannot reproduce a crash, s/he can activate the monitoring in the wild to quickly reproduce and fix bugs, thus stopping negative reviews. Second, MoTiF can be used as a *beta-testing* platform to stress apps under real conditions and users before making available the final app release.

3. MONITORING THE CROWD

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

3.1 Causes of App Crashes

There are many possible causes for 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, a behaviour commonly referred to as a "crash". This paper focuses on bugs that manifest with crashes. Kechagia *et al.* [22] identify causes of Android app crashes within a dataset of stack traces collected from real devices. In addition, Liang *et al.* [24] identify *context-related bugs* in mobile apps, *i.e.*, *network conditions*, *device heterogeneity* and *sensor input.* Table 1 summarizes these categories of Android app crashes, together with a sample app exhibiting such a crash.

In particular, the crashes that depend on context are more challenging to isolate and to reproduce by developers in the lab [1]. Hence, we aim to complement existing in-house testing solutions with a collaborative approach that monitors apps after their deployment in the wild.

Table 1: Categories of Android app crashes.

Cause	Sample app	App crashes
Missing or corrupted resource	PocketTool	*If the* Minecraft *game is not installed on the device*
Indexing problem	Ermete SMS	*Deleting a phone number taken from the address book*
Insufficient permission	ACV	*Long-pressing a folder*
Memory exhaustion	Le Chti	*After some navigation steps in the app*
Race condition or deadlock	Titanium	*Clicking the back button during the app launch*
Invalid format or syntax	PasswdSafe	*Opening a password that contains Norwegian characters*
Network conditions	Wikipedia	*Attempting to save a page without network connectivity*
Device heterogeneity	Wikipedia	*Pressing the Menu button on LG Devices*
Sensor input	MyTracks	*When GPS is unavailable*

3.2 What Context Information to Monitor?

To reproduce a crash, information regarding the actions that the user performed with the app, and the context under which the crash arose, are crucial. When enabled, MoTiF tracks input events (*e.g.*, user interaction events) and unhandled exceptions thrown during the execution of a subject app. To contextualize events, MoTiF records the following metadata and context information that is essential for being able to replicate a crash:

- *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 execution context, which we further classify as:
  - *Static context.* Properties that remain invariable during the whole execution—*e.g.*, device manufacturer, device model and SDK version,
  - *Dynamic context.* Properties that change along execution—*e.g.*, memory state, battery level, network state, and state of sensors.

3.3 Tracking 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 that declare public event handler methods. The Android framework calls these event handler methods when the respective event occurs [3]. For example, when a view (such as a button) is touched, the method `onTouchEvent` is invoked on that object. MoTiF intercepts the execution of these event handler methods. Hence, each time an event is executed, MoTiF logs both event metadata and context data. Table 2 reports on a subset of the handler methods intercepted by MoTiF.

Table 2: Examples of Android view types with their event listeners and handler methods.

Type	Event listener	Event handler
View	OnClickListener	onClick
ActionMenuView	OnMenuItemClickListener	onMenuItemClick
AdapterView	OnItemClickListener	onItemClick
Navigation Tab	TabListener	onTabSelected
	OnTabChangedListener	onTabChange
Orientation	OrientationEventListener	onOrientation-Changed

3.4 Logging Crash Traces

During the execution of an app, MoTiF keeps the observed events in memory. Only if the app crashes, MoTiF saves the trace of recorded 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—*i.e.*, $ct = \{e_1, e_2, ..., e_n\}$. Events can be of two types: *interaction* and *exception*. The last event of a logged 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 interaction events e_1, e_2, leading to an app crash $crash1$.

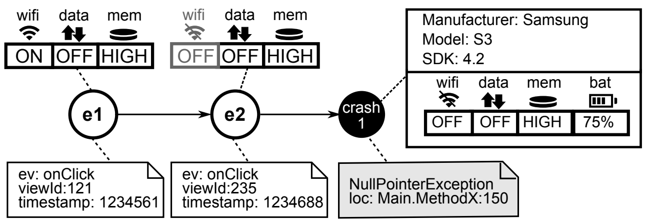


Figure 2: Example of a crowdsourced crash trace.

To minimize 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. Once uploaded, the synchronized traces are automatically removed from the local storage.

3.5 Adaptive Logging

To minimize the runtime overhead, MoTiF performs an *adaptive logging* strategy. In other words, MoTiF logs more information when the risk of a crash is higher, or when developers request to do it. By default, MoTiF only monitors and logs uncaught exceptions (*i.e.,* crashes) being thrown during the execution of apps. When the number of observed crashes for a given app reaches a predefined threshold of N crashes, MoTiF flags the app as *buggy-suspicious* and increases the monitoring depth to track additional user interaction events. N is a configuration parameter to be decided by app developers when using MoTiF. In addition, only one app is monitored in each device. The monitoring is distributed among devices and redistributed periodically to avoid any accidental user's disturbance.

4. AGGREGATING CROWD DATA

MoTiF uses a cloud environment to aggregate the crash traces collected from a multitude of devices in the wild. It transforms the collection of crash traces into a weighted directed graph that we denote as *Crowd Crash Graph.* The

Table 3: Example of *crash traces* and single steps split. In bracket, occurrences of each step.

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→e5(1)	e5→crash2(2)
e1→e6→e5→crash2	e1→e6(1)	e6→e5(1)

Crowd Crash Graph represents an aggregated view of all the events performed in a given app before a crash arises, with their frequencies, enabling MoTiF to induce 1. the minimum sequence of steps to recreate a crash, and 2. the context under which crashes arise.

4.1 Definition: Crowd Crash Graph

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 crash. Such a *crash graph* aggregates all crash traces that lead to the same exception. It is based on a Markov chain (1st order Markov model), which is a widely accepted formalism to capture sequential dependencies [36]. 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) measures how often, when e_i is fired, it is followed immediately by e_j. 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 Kim et. al. [23] to aggregate multiple crashes together in a graph. However, our crash graphs capture a different kind of information. Whereas the nodes of Kim et al. represent functions and edges represent call relationships between functions (extracted from crash reports); our nodes represent events, and our edges represent sequential user interaction flow. Our nodes and edges also store event and context metadata, and the graph forms a Markov model. In addition, we use crash graphs with a different purpose: to synthesize the most likely sequence of steps to reproduce a crash.

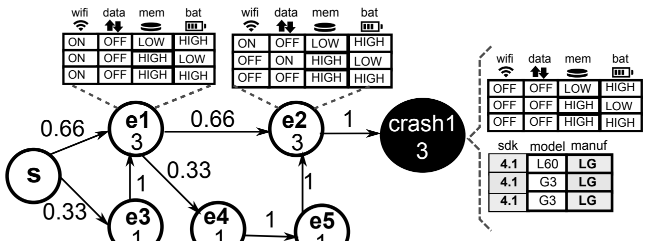
4.2 Building the Crowd Crash Graph

As illustration, we consider a version of the Wikipedia app (v2.0-alpha) that contained a crash-inducing bug—*i.e.*, the app crashes when the user tries to save a page and no network connectivity is available. Table 3 shows an example of five traces generated by the subject app.

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 comprises the following two steps.

4.2.1 Clustering traces by type of failure

First, MoTiF clusters the traces leading to the same exception. To identify similar exceptions, different heuristics can be implemented. For example, Dang et al. [11] propose a method for clustering crash reports based on call stack similarity. In MoTiF, we use a heuristic that considers two exceptions to be the same if they have the same *type* (*e.g.*, `java.lang.NullPointerException`) and message, and they



wifi	data	mem	bat
ON	OFF	LOW	HIGH
ON	OFF	HIGH	LOW
ON	OFF	HIGH	HIGH

wifi	data	mem	bat
ON	OFF	LOW	HIGH
OFF	ON	HIGH	LOW
OFF	OFF	HIGH	HIGH

wifi	data	mem	bat
OFF	OFF	LOW	HIGH
OFF	OFF	HIGH	LOW
OFF	OFF	HIGH	HIGH

sdk	model	manuf
4.1	L60	LG
4.1	G3	LG
4.1	G3	LG

Figure 3: Crash Graph derived from Table 3.

are thrown from the same location—*i.e.*, same *class* and *line* number. For example, in Table 3 (first column), we identify two clusters of traces. The first cluster contains three traces leading to *crash*1, and the second cluster contains two traces leading to *crash*2.

4.2.2 Merging traces in a crash graph

Next, for each cluster of traces, we form a *crash graph* following the graph construction technique proposed by Kim et al. [23]. First, we decompose each trace into single steps—*i.e.*, pairs of events executed in sequence in a trace (cf. Table 3). 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 a step, we create a node in the graph. If the node already exists, we update its weight. For the same step, we then add a directed edge to connect the step's two events. 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 to the corresponding event nodes. Figure 3 shows the resulting *crash graph* from the cluster of traces leading to *crash*1 in Table 3.

For each step in the graph, we then 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 $P_{e1-e3} = 0.33$. We label each edge with the transition probabilities. In addition, each node contains a weight indicating the number of occurrences of the event. The event $e2$ was executed 3 times.

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 a consolidated view of the most frequent actions among users before a crash arises, together with the observed execution contexts.

5. IDENTIFYING CRASH PATTERNS

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 that appear frequently among crashes. While several data mining techniques can be used, MoTiF implements *Path Analysis*, *Sequential Patterns*, and *Set Operations* to effectively induce the minimal sequence of steps that reproduce a crash as well as the context under which this crash occurs.

5.1 Synthesizing 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, the trace e1→e4→e5→e2→crash1 in Table 3 includes four trace steps to crash the app. However, there is a two-step trace, e1→e2→crash1, that results in the same crash. By exploiting the *Crowd Crash Graph*, MoTiF reduces the size of traces and filters out the irrelevant steps.

The goal of this phase is therefore to find the shortest path from the starting node (S) to an exception node (e) that maximizes the markov probability of the traversal. For this purpose, MoTiF implements *Dijkstra*'s algorithm [12], which is a widely known algorithm to find the shortest path between nodes in a graph. Whereas *Dijkstra* aims to minimize the weights over the paths, our goal is to find the path that maximizes the transition probabilities over the S-e path. Given that *Dijkstra* does not work with negative weights, we reduce the problem to a standard shortest-path problem by replacing the transition probability (P_i) of each edge by $-\log P_i$. Since log is a monotonic function, maximizing P_i is equivalent to minimizing $\log P_i$. In addition, $P_i \in [0,1] \Rightarrow \log P_i \leq 0 \Rightarrow -\log P_i \geq 0$, hence all weights are positive. For example, in the crash graph of Figure 3, we convert the transition probability between the nodes $e1$ and $e2$ as: $P_{e1-e2} = 0.66 \Rightarrow P_{e1-e2}^{Dijkstra} = -\log 0.66 = 0.18$.

Therefore, the shortest S-e path is the maximum probability path, which we call the *consolidated trace* and is promoted as the candidate trace to reproduce the crash. In our example, Dijkstra's algorithm starts at node S and would select node $e1$, since has the minimum weight (0.18), which corresponds to the edge with the highest transition probability (0.66). After $e1$, it selects event $e2$ since it again has the minimum weight (0.18 or probability of 0.66). Therefore, the *consolidated trace* to reproduce *crash*1 is *e1→e2→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 one. Since the graph contains the traces of all crashes observed in practice, at least one of the traces is guaranteed to reproduce the crash.

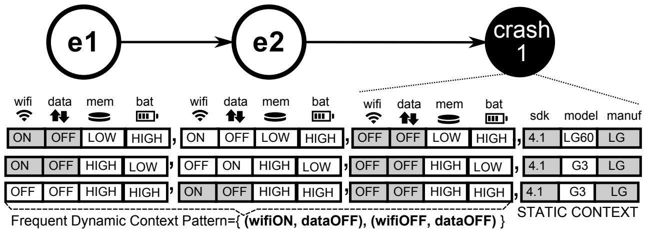
5.2 Learning Crash-prone Execution Contexts

As previously mentioned, not all the devices suffer from the same bugs and some crashes only arise under specific execution contexts—*e.g.*, network unavailable or high CPU load. Hence, MoTiF searches for recurrent context patterns within a consolidated trace, which 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. MoTiF learns both *dynamic* and *static* context patterns.

5.2.1 Dynamic Context

To learn frequent dynamic contexts from a trace, we use *Sequential Pattern Mining*, which is a data mining technique to discover frequent subsequences in a sequence database [28]. A sequence is a list of itemsets, where each itemset is an unordered set of items. We concatenate the context properties reported in each step of the consolidated trace. Fig. 4 shows 3 sequences of context properties observed for the consolidated trace (synthesized from Fig. 3). Each of these sequences contains 4 itemsets, one for each of the events in the trace, and each item maps to a context property.

In particular, we mine *frequent closed sequential patterns*—*i.e.*, the longest subsequence with a given support. The support of a sequential pattern is the percentage of sequences



e1
e2
crash 1
wifi data mem bat
wifi data mem bat
wifi data mem bat
sdk model manuf
ON OFF LOW HIGH , ON OFF LOW HIGH , OFF OFF LOW HIGH , 4.1 LG60 LG
ON OFF HIGH LOW , OFF ON HIGH LOW , OFF OFF HIGH LOW , 4.1 G3 LG
OFF OFF HIGH HIGH , ON OFF HIGH HIGH , OFF OFF HIGH HIGH , 4.1 G3 LG
Frequent Dynamic Context Pattern={ (wifiON, dataOFF), (wifiOFF, dataOFF) }
STATIC CONTEXT

Figure 4: Learning the crash-prone context from a candidate trace.

where the pattern occurs. To ensure that the context truly induces the crash, MoTiF searches for closed sequential patterns with support 100%—*i.e.*, patterns that appear in all the observed 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 [49]. In particular, we use the implementation of BIDE+ available in the SPMF tool [46].

In Fig. 4, the algorithm identifies the following frequent context pattern: {(`wifiON`, `dataOFF`), (`wifiOFF`, `dataOFF`)}. This means that the properties (`wifiON`, `dataOFF`) are observed at the same time, and eventually are followed by the properties (`wifiOFF`, `dataOFF`) appearing together. Note that the itemsets appear in sequence, *i.e.*, the second itemset is always observed after the first itemset, but it does not need to be occurring at the same time in each trace. In the example, the pattern reveals that the crash arises when the network is being disconnected (`wifiOFF`).

5.2.2 Static Context

The example above shows the power of sequential pattern mining compared to simple intersection operations, since the former enables to capture relevant context changes, for example running out of memory (*e.g.*, the sequence {`memHIGH`, `memLOW`}), a network disconnection, or an empty battery.

However, to identify relevant static contexts, which do not evolve over time, we can just use *set operations*. For example, in Fig. 4, the union set across all traces for the *sdk* property is {`4.1`}, for the *manufacturer* property is {`LG`}, and for the *model* property is {`LG60, G3`}. In other words, the crash affects LG devices that run Android 4.1, and has been observed in LG60 and G3 models. The resulting sets are the relevant static contexts and will be used to select the devices to reproduce the crashes.

6. SYNTHESIZING CRASH TEST SUITES

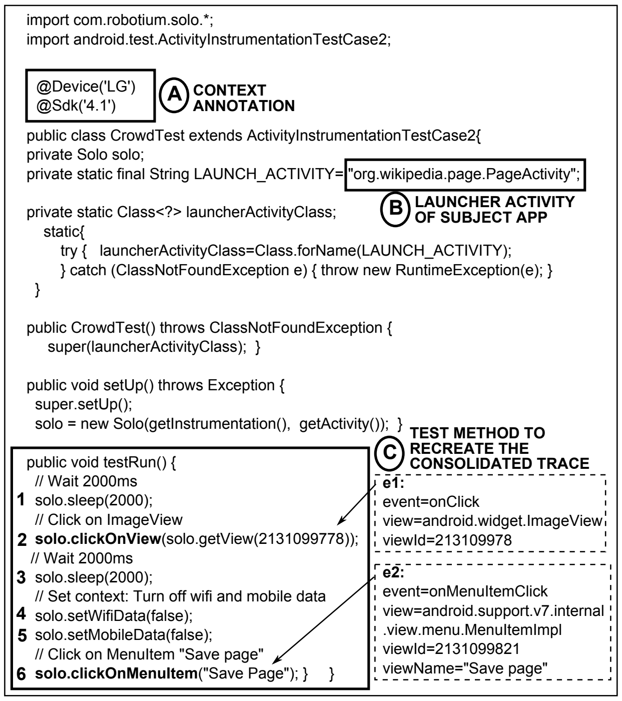
Based on the *consolidated trace* (cf. Section 5.1) and its *crash-prone execution context* (cf. Section 5.2), MoTiF generates a test suite to faithfully reproduce the crash. These tests recreate a sequence of user interactions that lead to the crash of the app, while taking care of not disclosing any sensitive information (*e.g.*, password). 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.

6.1 Generating Crowd Tests

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 for automatic black-box UI tests of Android applications [41]. We chose *Robotium* because it has full support for native and hybrid applica-

Table 4: Examples of mappings between Android event handler methods and Robotium methods.

Element	Android method	Robotium method
View	onLongClick	clickLongOnView
Button	onClick	clickOnButton
TextField	setText	typeText
ActionMenu	onMenuItemClick	clickOnActionBarItem
Orientation	onOrientationChange	setActivityOrientation



```
import com.robotium.solo.*;
import android.test.ActivityInstrumentationTestCase2;

@Device('LG')
@Sdk('4.1')

public class CrowdTest extends ActivityInstrumentationTestCase2{
private Solo solo;
private static final String LAUNCH_ACTIVITY="org.wikipedia.page.PageActivity";

private static Class<?> launcherActivityClass;
   static{
      try {   launcherActivityClass=Class.forName(LAUNCH_ACTIVITY);
      } catch (ClassNotFoundException e) { throw new RuntimeException(e); }
 }

public CrowdTest() throws ClassNotFoundException {
    super(launcherActivityClass);  }

public void setUp() throws Exception {
  super.setUp();
  solo = new Solo(getInstrumentation(),  getActivity());  }

 public void testRun() {
   // Wait 2000ms
1  solo.sleep(2000);
   // Click on ImageView
2  solo.clickOnView(solo.getView(2131099778));
  // Wait 2000ms
3  solo.sleep(2000);
   // Set context: Turn off wifi and mobile data
4  solo.setWifiData(false);
5  solo.setMobileData(false);
   // Click on MenuItem "Save page"
6  solo.clickOnMenuItem("Save Page"); }    }
```

```
e1:
event=onClick
view=android.widget.ImageView
viewId=213109978
```

```
e2:
event=onMenuItemClick
view=android.support.v7.internal
.view.menu.MenuItemImpl
viewId=2131099821
viewName="Save page"
```

A CONTEXT ANNOTATION
B LAUNCHER ACTIVITY OF SUBJECT APP
C TEST METHOD TO RECREATE THE CONSOLIDATED TRACE

Figure 5: Generated MoTiF test case for the Wikipedia app.

tions, does not require the source code of the application under test, and provides fast test case execution.

We propose mapping rules between the Android event handler methods (Section 3.3) and the methods provided by the Robotium API [42]. For example, the Android event `onClick` in a view of type `Button` is mapped to the *Robotium* method `clickOnButton`. Table 4 shows a subset of the mapping rules identified. These rules guide the automatic generation of test cases.

MoTiF defines a base template for a Robotium test case (Figure 5). First, MoTiF adds the crash-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 5 shows the crash 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 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 other 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.

6.2 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 assess whether or not 1. they truly reproduce the observed crashes, and 2. they can generalize to other contexts/devices.

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

Later, MoTiF selects a random sample of devices that do not match the context profile, and tests whether they reproduce the crash. If the test case indeed reproduces the crash in a different context, MoTiF concludes that the context it learnt is not discriminative enough. In this case, MoTiF adds the context in the test case as a note to developers, mainly informing him or her about the devices most frequently running their apps. If on the contrary, the test case only reproduces the failure on the consolidated context, that context will be included as a *critical* annotation in the test.

Note that, to avoid any user disturbance, MoTiF executes the tests for validation only during periods of phone inactivity, *e.g.,* during the night, and when the device is charging.

6.3 Privacy Issues

All approaches that record user inputs put privacy at risk [55]. Since our approach provides test suites to replay a sequence of user interactions that lead to a crash of the application, we took care not to disclose any sensitive information (*e.g.*, password, login, address). Specifically, MoTiF incorporates two privacy mechanisms: *anonymization* [2] and *input minimization* [56] 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 (since that could expose the user identity).

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 [56] to simplify the input to only the relevant parts. For example, let us consider the Android app *PasswdSafe*, which allows users to store all passwords in a single database. The app had a bug [7] and crashed when opening a password that contained the Spanish character *ñ*. Since it is undesirable that MoTiF provides all users' passwords to developers, MoTiF applies the minimization technique to all crowd users' inputs and extracts the minimum relevant part that produces the crash. For example, consider the following 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 includes this input in the tests instead of the original input, which would reveal sensitive information.

7. IMPLEMENTATION DETAILS

This section provides details about the infrastructure that supports our approach. MoTiF can monitor any *debuggable* app[2] running on a mobile device, without requiring access to its source code. Our prototype implementation is composed of two parts: a mobile client library that runs on the mobile device and a cloud service.

7.1 Android Client Library

The Android virtual machine (named *Dalvik*) implements two debugging interfaces: the *Java Debug Interface (JDI)* and the *Java Debug Wire Protocol (JDWP)*, which are part of the *Java Platform Debugger Architecture (JPDA)* [20]. This technology allows tools such as the `adb` tool (Android Debug Bridge) to communicate with a virtual machine. MoTiF's client app runs `adb` on the device and communicates with Dalvik via `adb` and the standard debugging interfaces JDWP and JDI over a socket. For this, our tool extends and reuses part of the implementation provided by *GROPG* [35], an on-phone debugger. The *GROPG* implementation ensures low memory overhead and fast execution, and it enables to monitor apps and to intercept user interaction and exception events.

7.2 Cloud Service

MoTiF sends the data collected in devices to a cloud service for aggregation and analysis using APISENSE [4]. APISENSE provides a distributed crowd-sensing platform to design and execute data collection experiments in mobile devices [17].

To store and aggregate the crash traces collected from the crowd and the crowd crash graphs, MoTiF creates a *graph database* with *Neo4J* [33]. Graph databases provide a powerful and scalable data modelling and querying technique capable of representing any kind of data in a highly accessible way [40]. We can then query the graph database using the Cypher graph query language, which is a widely used pattern matching language. To extract the *consolidated traces* from the Neo4J graph database, we have implemented the *Dijkstra* algorithm as a Cypher query.

8. EVALUATION

In this section, we report on empirical experiments that we performed to evaluate the applicability and performance of our approach. In particular, we address the following research questions:

- **RQ1:** *What is the overhead of* MoTiF*?*
- **RQ2:** *Can* MoTiF *identify crash patterns effectively?*
- **RQ3:** *Can* MoTiF *synthesize test suites that reproduce crashes effectively?*

8.1 Data Sets

To perform the experiment, we use two data sets—*i.e.*, the Chimp and CrowdSourced data sets. Both are based on 5 Android apps that experience crashes. These apps were selected based on their popularity, app category, different size and complexity of functionality, mixture of open-source and proprietary, and the types of crashes that occur. Table 5 lists the apps used in the study, with their version, category, size, and type.

The Chimp data set is obtained from the 5 mobile apps using the *Monkey* testing tool (provided by Android). This tool generates pseudo-random user events such as clicks, touches, gestures as well as system-level events in apps running on a device or in emulators [32]. If the app crashes or receives an unhandled exception, *Monkey* stops and reports the error. To build the Chimp data set, we first let *Monkey* send 1,000 events to each of our apps. We then repeated this, but this time using 50,000 events. Finally, for the Bites app, we repeated the 50,000 events 49 more times (such that this app in fact had 50 such executions). These three different types of executions will be used across different research questions.

[2]These are apps that have the `android:debuggable` attribute in their manifest.

The CrowdSource data set is obtained through crowdsourcing with students of 1 computer science lab in Lille and 2 in Montreal. Since engaging users to participate in crowdsourced experiments is a challenge [54], we designed the experiment as a contest with a prize as incentive for users. The goal of the contest was to try crashing the 5 candidate apps as many times as possible in as many different ways as possible, during a maximum time of 60 minutes. The participants were unaware of the number and types of crashes in the apps. Eventually, 10 participants engaged in the contest, each of whom was an experienced mobile user.

To run the contest, we provided 5 Android devices with different characteristics to simulate a diverse crowd (Table 6). We pre-installed MoTiF and the set of apps under test and borrowed the devices to the participants for the duration of the contest.

8.2 Exploratory Data Analysis

Before jumping to the research question results, we first want to compare the two obtained data sets in more detail, since the Chimp data set is obtained automatically, compared to the manually gathered CrowdSource data set.

In the Chimp data set (for the 50,000 events sent to each app), the apps *Bites* and *PocketTool* crashed after executing 9,480 events and 33 events, respectively. However, no crashes could be found for the other three apps.

In the CrowdSource data set, on the other hand, the participants were able to generate 52 crashes (each yielding a trace for analysis) across the five apps, distributed across the different devices and Android versions. Table 5 shows, for each subject app, the distribution of crash traces (*i.e.*, crashes) per app and the number of unique crashes amongst them. The observed crashes belong to 4 of the 7 categories of crashes identified in the literature (cf. Table 1).

Out of the 52 crashes, we could identify 6 unique crashes, as shown in the table. For the Google I/O app, the crowd discovered two crashes that we were unaware of beforehand. The first crash is a context-related crash occurring only on the devices with Android 4.0.3 and Android 4.0.4. The second crash happens when searching for a word that contains the quote character: *"*. Since the Google I/O app was developed on purpose by Google for their annual Android developer conference as an example of best practices (its source code was made publicly available), these two discovered bugs underline the quality of the CrowdSource data set. Indeed, it demonstrates that even apps that are well-designed and tested, crash in certain contexts. Furthermore, crowd-based crash monitoring seems a valid basis for gathering a wide variety of crashes.

In the remainder of this section, we discuss the findings for the 3 research questions, using the two data sets. For each question, we provide a motivation, approach and findings.

Table 5: Statistics of the Android apps used in our experiments. The number of users and average rating were determined at the time of writing this paper. For Google I/O, only the 2015 ratings were available.

Android App	Category	Size in Kb	Type	#Users	Avg. rating	#Traces (#unique)	Crash type
Google I/O 2014	Books	15,060	open	500k–1M	4.3	11 (2)	Device heterogeneity
							Invalid format
Wikipedia (2.0α)	Books	5650	open	10M–50M	4.4	4 (1)	Network conditions
OpenSudoku (1.1.5)	Games	536	open	1M–5M	4.6	5 (1)	NullPointerException
Bites (1.3)	Lifestyle	208	open	10k–50k	3.2	16 (1)	Invalid format
PocketTool (1.6.12)	Tools	1410	closed	N/A	N/A	16 (1)	Missing resource

Table 6: Crowd of devices used in the experiment.

Device model	Android SDK
LG-E617G	4.0.3
Samsung Galaxy Nexus	4.0.4
Samsung GT-I9100	4.1.1
Samsung GT-I9100	4.1.2
Samsung Nexus S	4.1.2

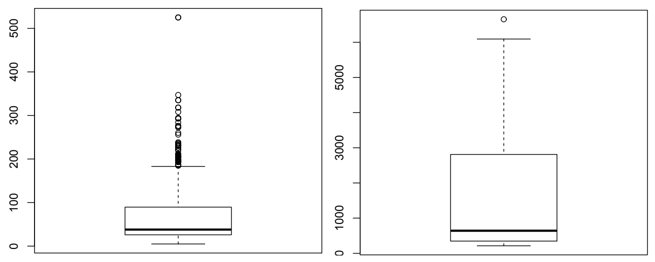


Figure 6: Runtime overhead (ms) of monitoring user interactions (left) and send traces to server (right).

RQ1. What is the overhead of MoTiF?

Motivation: The first phase of MoTiF is the monitoring of an app's execution on a client device. The more overhead monitoring causes, the more users will be aware that MoTiF will be running, and the more it can influence their user experience. Furthermore, users without crashes should not be punished with slow performance due to monitoring. Hence, this question aims to quantify the overhead of MoTiF.

Approach: We study the runtime overhead introduced by MoTiF by monitoring the app executions of the Chimp data set on the Samsung Galaxy Nexus with Android 4.1.2 and 2 processors. In particular, we used the executions of $1,000$ events of the Chimp data set, then measured the average execution time across the recorded events as well as the average time required to send traces to the server.

Findings: **The mean overhead to log a user event is** 39 *ms***.** Due to MoTiF's adaptive logging strategy (Section 3.5), MoTiF initially only listens for uncaught exception events. Therefore, the corresponding runtime overhead to store exception events is 0, given that MoTiF logs the exception events only after the app has crashed. Only when an app is suspected to be crash-prone, MoTiF augments the monitoring strategy to log user interactions. As such, the obtained average overhead of 39 *ms* of the current proof-of-concept implementation is imperceptible to users when interacting with the apps. A response delay < 500 *ms* is acceptable for users according to the Intel industry experience values [53]. Nevertheless, additional engineering efforts should be invested to minimize overhead. Different approaches, *e.g.,* code instrumentation, could be explored. Finally, it is important to note that MoTiF distributes its experiments among the different devices available in its crowd, and redistributes them periodically, hence any accidental overhead will even out across different devices.

The median execution time to send crash traces to the server is 666 *ms***.** The crash traces are temporarily stored in JSON files in the device memory, until the data is automatically flushed to the remote server for processing. For example, the 50 random traces in the Chimp data set generated for the Bites app contain $36,603$ events and consume 31 MB. Since MoTiF sends the traces to the server only when the device is charging, and the majority of users charge their devices on a daily basis, MoTiF liberates the temporary storage in a short time. Hence, in a typical use scenario, only a limited number of traces will be stored in a device. Since modern devices have several GBs of memory available and incorporate external storage cards with additional memory, the temporal storage of traces in devices is feasible. Furthermore, to minimize the impact on a user's device, only one app is monitored at a given time on each device. Figure 6 shows the statistical distribution of the overhead measures.

RQ2. Can MoTiF identify crash patterns effectively?

Motivation: The second phase in MoTiF is the identification of crash patterns, which is crucial to filter noise from user interaction events and context. The more irrelevant steps MoTiF can eliminate, the more succinct the resulting crash pattern, and hence the less effort is required from developers to interpret the crash patterns. Here, we study the filtering performance of MoTiF for the crash traces, as well as its resilience to noise introduced in the data.

Approach: For each app, we extract the crowd-consolidated traces using MoTiF and measure their compression factor as $\frac{Avg.\ \#events\ in\ crash\ traces}{\#Eventsincrowdconsolidatedtrace}$. The higher this factor, the better MoTiF was able to filter the trace.

Furthermore, in order to assess the impact of noise, we used the 50 executions of the Bites app in the Chimp data set (each of which crashed the app). In particular, we added these 50 traces to the crash graph of the Bites app obtained from CrowdSource. Since the amount of random crash data from Chimp outweighs the amount of manually generated crash data, this experiment allows to measure how effective MoTiF can deal with noise in the crash data.

Findings: **MoTiF obtains compression factors of 7.5 up to 22.** As shown in Table 5, the number of events per trace can significantly differ among apps, since it will depend on the design of the app and the location of the bug

Table 7: Number of events and compression factor of the crash traces for the CrowdSource (first five rows) and Chimp (last row) data sets.

Android App	Avg. # Events	#Consol. Events	Compr. Factor
Google I/O 2014	22	1	22
Wikipedia	29.5	2	14.75
OpenSudoku	60	8	7.5
Bites (CrowdSource)	56	6	9.33
PocketTool	9.4	1	9.4
Bites (Chimp)	778.79	2	389.40

causing the crash. However, for all apps the total number of events in the crowd-consolidated trace (generated by MoTiF) is smaller than the average size of the original traces. For example, in the *Wikipedia* app, the average size of traces is 29.5, while MoTiF synthesizes a 2-event trace from the crowd data, together with a relevant context: *network disconnection*. Table 7 shows the resulting compression factors, which are the highest for the Google I/O app. However, even for the apps with the longest traces (60 for OpenSudoku and 56 for Bites), MoTiF is able to reduce the size of the event trace substantially to 8 and 6, respectively.

In the presence of noise, MoTiF achieved a compression factor of 389.40. Indeed, the graph from the Chimp data set for Bites contains 629 different event nodes and 3, 596 relationships among them (extracting the consolidated trace took 267ms). Whereas the average number of events in the randomly generated crash traces is 778.79, the consolidated crash trace contains only two events:
$keyUp(keyCode = 22) - > onMenuItemClick(id = 6)$.

Although this trace reproduces the original crash, we observe that it slightly differs from the consolidated trace synthesized from CrowdSource. This is because the input data in this case contains more randomly generated traces (50) than manually generated ones (16), hence the random Chimp events dominate. However, both 2-event traces are correct and their main difference is that the Chimp traces contain event subsequences that never could be crowdsourced because of the physical limitations of a mobile device.

RQ3. Can MoTiF synthesize test suites that reproduce crashes effectively?

Motivation: The third and final phase of the approach is the generation of a test suite to reproduce crashes. The main challenge here is to generate the same exception types as the original crashes. This is what is evaluated in this question.

Approach: To check if the promoted traces from CrowdSource can reproduce the crashes, we generate the corresponding Robotium tests with MoTiF. We then execute the test cases on the devices and check whether the app crashes again and, if so, whether the same exception types occur.

Findings: **The test cases correctly reproduce the bugs in 4 (out of 5) apps.** In other words, the execution of the test cases generates the same exception type in the same stack trace location as the original crashes of CrowdSource. Only in the OpenSudoku app, the first consolidated trace failed when trying to reproduce. Closer analysis of the source code learnt that this failure is due to the appearance of the same dialog box, from two different locations in the app. Hence, in the graph, the event was merged into a single node. We plan to further explore such situations to improve the effectiveness of MoTiF. In any case, when we extracted the next most weighted consolidated-trace from the graph, the crash could be reproduced. One benefit of the Crowd Crash Graph is indeed that it will always contain a path that reproduces the crash.

Table 8: Candidate traces to reproduce crashes.

App	Events
Google I/O	1) Click on ImageView Search 2) Type "
	1) Click on a Session [Android 4.0.3/4.0.4]
Wikipedia	1) Click on ImageView Menu 2) Disconnect WiFi and mobile data 3) Click on MenuItem "Save Page"
OpenSudoku	1) Click on ListItem position 1 2) LongClick in ListItem position 1 3) Click on MenuItem "Edit note" 4) Click on Button "Save" 5) LongClick in ListItem position 1 6) Click on MenuItem "Delete puzzle" 7) Click on Button "Ok" 8) Change orientation
Bites	1) Click on Tab "Method" 2) Click on context menu 3) Click on MenuItem "insert" 4) Touch text field "Method" 5) Click Button "ok"
PocketTool	1) Click on Button "Level Editor"

Table 8 summarizes the steps extracted from the crowd-consolidated traces to reproduce the context-sensitive crashes found by the participants of the experiment. Each of them is not only compact, but also easy to interpret by developers. Note that the apps used in the experiment, the generated crowd crash graphs and tests are available in the online appendix [37].

8.3 Threats to Validity

One important threat to construct validity is the choice of CrowdSource participants which could be biased to young, experienced mobile app users who knew that they had to find crashes. This might not be representative of the typical users of some apps, and hence impact (either positively or negatively) the ability of crowdsourcing to generate certain crash traces. Nonetheless, our experiment is performed under realistic conditions with 5 real apps with different types of crashes.

Although performing a crowdsourcing-based experiment is challenging, especially taking into account the added difficulty of factoring in different contexts, 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 larger the number of users (with higher diversity), the more accurate the results that MoTiF produces and the more crash types can cover.

Our approach also has limitations induced by the implementation of the client library reported in this paper. First, for convenience, the apps must have their debug flag enabled to be monitored by MoTiF. Second, the implementation of MoTiF requires root access and only runs in devices with Android SDKs under 4.2 due to a limitation introduced by the GROPG implementation. We plan to make MoTiF compatible with the latest Android versions. To alleviate any potential security risk inherent to the current proof-of-concept implementation, alternative techniques (*e.g.,* bytecode instrumentation, embedding a library in the apps source code) could be used to collect the user traces. Further threats are associated to the way in which we measured the overhead of MoTiF.

Furthermore, the choice of apps and devices are a threat to external validity. Further analyses are necessary to evaluate the efficiency of this approach on different types of apps and crashes. For example, different app categories and sizes of apps should be considered. Finally, other ecosystems than the Android one should be explored.

9. RELATED WORK

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

Mobile App Testing. Currently, a wide range of testing tools for Android apps is available: *Monkey* [32], *Calabash* [8], *Robotium* [41], *Selendroid* [44]. In addition, previous research has investigated GUI-based testing approaches for Android apps [6,19,29,34]. However, the aforementioned approaches do not include execution contexts in the tests, therefore they cannot detect device-specific bugs. Furthermore, Liang et al. [24] present Caiipa, a cloud service for testing Windows phone apps over different execution contexts. Several commercial solutions (*e.g.*, Xamarin Test Cloud [52], testdroid [48]) 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 crowdsourced monitoring after deployment to help developers to quickly detect and fix crashes. The evaluation of this paper provides an example of such complementarity, by combining MoTiF with Monkey.

Crash Reporting Systems. Current crash reporting systems on mobile apps (*e.g.*, SPLUNK [45], Google Analytics [16]) collect raw analytics on the execution of apps. MoTiF goes beyond current crash reporting systems by exploiting crowd feedback in a smarter way. MoTiF provides developers *in vivo test suites*, which define the steps to reproduce crashes and the context that induce the failures. The test suites are crowd-validated before delivery to developers.

Mobile Monitoring in the Wild. Agarwal et al. [1] 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. On the contrary, our approach synthesizes test suites to enable developers to automatically reproduce crashes. Additionaly, we exploit the crowd, not only to learn failure contexts, but also to assess the consolidated traces and context. AppInsight [38] 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. However, they do not synthesize test cases to reproduce crashes.

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 [18,43]. *MonkeyLab* [25] is an approach to mine GUI-based models based on recorded executions of Android apps. The extracted models can be used to generate actionable scenarios for both natural and unnatural sequences of events. Our approach also characterizes the execution contexts under which crashes arise. Although our approach is related to record and replay approaches, we do not use existing tools (*e.g.*, *RERAN* [14], *Android getevent tool* [13]) because in these approaches the recorded actions are specific to the device on which they were recorded. Instead, we are interested in reproducing context-related and device-specific crashes, thus we need generic scripts that can be reproduced on different devices in order to assess the validity of the consolidated contexts.

Reproducing Field Failures. The last group includes techniques to reproduce crashes. Jin and Orso [21] introduce BugRedux, an approach that applies symbolic execution from different types of failure data to recreate field failures for desktop programs. Rö*β*ler et al. [39] 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 [5], a technique to generate unit tests that reproduce program failures. ReCrash stores partial copies of method arguments in memory to create unit tests to reproduce failures. STAR [10] provides a framework to automatically reproduce crashes from crash stack traces of object-oriented programs. Despite the prolific research in this area, none of the aforementioned approaches are available for mobile apps. Mobile apps pose additional challenges for the reproduction task, *i.e.,* context-induced crashes that are invisible from an app's code. Thus, some crashes only exhibit under specific device models or SDK versions. We propose an approach to reproduce crashes in mobiles apps that takes context into consideration.

For the mobile platform, *Crashdroid* [50] automatically generates steps to reproduce bugs in Android apps, by translating the call stack from crash reports. 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. Hence, our approach complements existing approaches by providing a solution that can characterize contexts and isolate device-specific crashes.

10. CONCLUSION

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

This paper presents MoTiF, a crowdsourced monitoring approach to help developers to detect and reproduce context-related crashes in mobile apps after their deployment in the wild. MoTiF leverages, in a smart way, crash and device feedback to quickly detect crash patterns across a crowd of 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 if the tests can expose the crashes and no other contexts can reproduce the same crash. We empirically evaluated the approach in an experiment with crowdsourced and automatically generated crashes of 5 existing mobile apps under realistic conditions.

As future work, we plan to analyze trade-offs between the amount of data collected and the reproducibility of the approach. Furthermore, we will evaluate the approach with different types of crashes and apps, in particular on larger crowds. Finally, we plan to study further mechanisms to encourage users to collaborate in debugging experiments.

11. REFERENCES

[1] 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.

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

[3] Android developers guide. Input events. http://developer.android.com/guide/topics/ui/ui-events.html. [Online; accessed Jan-2016].

[4] APISENSE. http://apisense.io. [Online; accessed Jan-2016].

[5] 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. Springer, 2008.

[6] 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 and Applications*, OOPSLA, pages 641–660. ACM, 2013.

[7] Bug report PasswdSafe. http://sourceforge.net/p/passwdsafe/bugs/3. [Online; accessed Jan-2016].

[8] Calabash. http://calaba.sh/. [Online; accessed Jan-2016].

[9] Y. Cao, H. Zhang, and S. Ding. Symcrash: Selective recording for reproducing crashes. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 791–802, New York, NY, USA, 2014. ACM.

[10] N. Chen and S. Kim. STAR: Stack Trace based Automatic Crash Reproduction via Symbolic Execution. *IEEE Transactions on Software Engineering*, 41:1–1, 2014.

[11] 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.

[12] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[13] Getevent tool. https://source.android.com/devices/input/getevent.html. [Online; accessed Jan-2016].

[14] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. Reran: Timing-and touch-sensitive record and replay for android. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 72–81. IEEE, 2013.

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

[16] Google Analytics. http://www.google.com/analytics. [Online; accessed Jan-2016].

[17] 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.

[18] S. Herbold, J. Grabowski, S. Waack, and U. Bünting. 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.

[19] 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.

[20] Java. Javatm platform debugger architecture. http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/. [Online; accessed Jan-2016].

[21] 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.

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

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

[24] 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.

[25] 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.

[26] Louis Columbus. Roundup of mobile apps and app store forecasts, 2013. http://www.forbes.com/sites/louiscolumbus/2013/06/09/roundup-of-mobile-apps-app-store-forecasts-2013/, June 2013. [Online; accessed Jan-2016].

[27] W. Maalej and H. Nabil. Bug report, feature request, or simply praise? on automatically classifying app reviews. In *Proceedings of the 23rd IEEE International Requirements Engineering Conference*, 2015.

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

[29] 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.

[30] S. McIlroy, N. Ali, H. Khalid, and A. E. Hassan. Analyzing and automatically labelling the types of

user issues that are raised in mobile app reviews. *Empirical Software Engineering*, pages 1–40, 2015.

[31] I. J. Mojica, M. Nagappan, B. Adams, T. Berger, S. Dienst, and A. E. Hassan. An examination of the current rating system used in mobile app stores. *IEEE Software*, 2015.

[32] UI/Application Exerciser Monkey. http://developer.android.com/tools/help/monkey.html. [Online; accessed Jan-2016].

[33] Neo4J. Neo4j. http://www.neo4j.org. [Online; accessed Jan-2016].

[34] 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.

[35] 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.

[36] J. R. Norris. *Markov chains*. Cambridge university press, 1998.

[37] Online Appendix. https://sites.google.com/site/motifandroid.

[38] 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.

[39] J. Rö*β*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.

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

[41] Robotium. https://code.google.com/p/robotium. [Online; accessed Jan-2016].

[42] Robotium API. http://robotium.googlecode.com/svn/doc.

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

[44] Selendroid. http://selendroid.io/. [Online; accessed Jan-2016].

[45] SPLUNK. https://mint.splunk.com. [Online; accessed Jan-2016].

[46] SPMF: An open-source data mining library. http://www.philippe-fournier-viger.com/spmf/index.php. [Online; accessed Jan-2016].

[47] 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 Jan-2016].

[48] Testdroid. http://testdroid.com. [Online; accessed Jan-2016].

[49] 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.

[50] 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.

[51] Wikipedia Android app. https://play.google.com/store/apps/details?id=org.wikipedia&hl=en. [Online; accessed Jan-2016].

[52] Xamarin Test Cloud. http://xamarin.com/test-cloud. [Online; accessed Jan-2016].

[53] Xiao-Feng Li. Quantify and Optimize the User Interactions with Android* Devices. https://software.intel.com/en-us/android/articles/quantify-and-optimize-the-user-interactions-with-android-devices, Apr. 2012. [Online; accessed Jan-2016].

[54] D. Yang, G. Xue, G. Fang, and J. Tang. Incentive mechanisms for crowdsensing: Crowdsourcing with smartphones. *Networking, IEEE/ACM Transactions on*, PP(99):1–13, 2015.

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

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