MOBILESoft: G: Debugging of Mobile Apps in the Wild Guided by the Wisdom of the Crowd

María Gómez
Inria Lille - Nord Europe
University of Lille 1, France
Email: maria.gomez@inria.fr

I. Problem and Motivation

With the proliferation of mobile devices and app stores (*e.g.*, Google Play, Apple Store), the development of mobile applications (*apps*) is experiencing an unprecedented popularity. In 2013, the Google Play Store reached over 50 billion app downloads [3].

Despite the huge number of mobile apps available, the quality of these apps varies greatly. Unfortunately, end-users frequently experience crashes and errors for some apps installed on their devices, as highlighted by the apps' user feedback [22]. The majority of user complaints are related to app crashes [27].

Although mobile app developers can use a wide range of testing tools [18], [26], [11], [4] 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 identify and fix issues. Otherwise, due to abundant competition, they risk to lose customers to rival mobile apps and be forced out of the market. Any software developer knows that reproducing failures that users experience *in vivo* is a major challenge. This task is even more complicated in mobile environments, which suffer from device fragmentation and diverse operating conditions [8].

As an illustration, users recently experienced crashes with the Android Wikipedia app, 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.

The goal of this research is to drastically improve the quality of mobile apps by complementing existing testing solutions with novel mechanisms to monitor and debug apps after their deployment in the wild. In particular, we focus on a collaborative approach to assist developers in reproducing failures, based on the experience faced by a multitude of individuals. We chose the Android platform because, according to a recent study, the 70% of mobile app developers are targeting Android [32].

II. Background and Related Work

This section briefly summarizes the state of the art in the major disciplines that are related to this research: 1) mobile app testing, 2) failure reproduction, and 3) crowd monitoring.

Mobile app testing. Existing researches have proposed GUI-based testing approaches for Android apps [18], [26], [11], [4]. Hu et al. [19] propose AppDoctor, a tool for testing apps against many system and user actions, and helping developers to diagnose the resultant failure reports. Liang et al. [25] present Caiipa, a cloud service for testing apps over different mobile contexts. Caiipa provides information for developers to understand the root causes of errors and prioritize their correction. In addition, several commercial solutions (*e.g.*, Xamarin Test Cloud [7], testdroid [6]) exploit the cloud to test an app on hundreds of devices simultaneously. In spite of the prolific research in this area, testing approaches cannot guarantee the absence of unexpected behaviors once the apps are deployed in the wild.

Crash reproduction. Furthermore, there are techniques for detecting and reproducing crashes in desktop programs. Jin and Orso [20] introduce BugRedux to recreate field failures in the lab in desktop programs. STAR [12] provides a framework to automatically reproduce crashes from crash stack traces for object-oriented programs. Röβler et al. [31] 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 [10], a technique to generate unit tests that reproduce software failures. Nevertheless, the aforementioned techniques are not available for mobile platforms. 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).

Crowd monitoring. Another family of approaches monitor apps in the wild. Agarwal et al. [8] 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 the errors. AppInsight [30] is a system to monitor app performance in the wild for the Windows platform. In addition, current *crash reporting* systems (*e.g.*, SPLUNK [5], Google Analytics [2]) collect raw analytics on the execution of apps. Our approach goes beyond current crash reporting systems by providing developers with a test suite which defines the steps to reproduce the identified failures. The test suites are enhanced with the operating context conditions that favor the rise of failures, while preserving privacy.

The goal of this research is to improve the quality assurance of apps. The major expected contribution is a *crowd-sourced* framework to assist developers to detect, reproduce and diagnose bugs in mobile apps after their deployment in the wild.

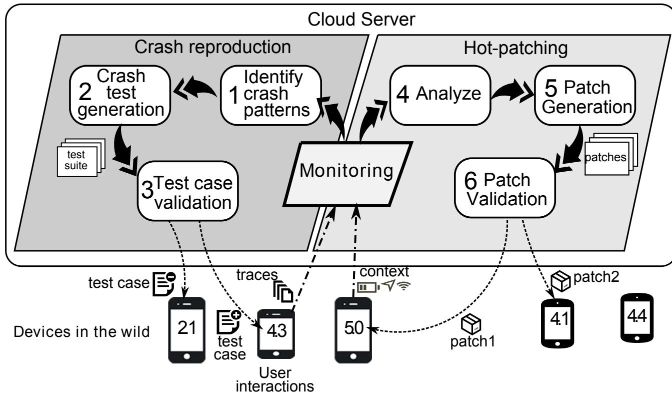

Fig. 1. Overview of the proposed approach.

III. APPROACH AND UNIQUENESS

We propose MOTIF, a *crowdsourced approach* to assist developers to detect, reproduce and diagnose crashes in mobile apps after their deployment in the wild. MOTIF uses machine learning techniques atop of data crowdsourced from real devices and users. The key idea is that by exploiting the crashes faced by a multitude of users, it is possible to assist developers in isolating and reproducing such crashes in an automatic and effective manner. This automated approach has the potential to save precious time for developers, which is a crucial factor for the success of mobile apps in current app markets.

Figure 1 sketches a high-level overview of the proposal. The approach has three phases.

A. Phase 1: Monitoring the Crowd

Once an app is released and executed on mobile devices, MOTIF monitors app executions to detect the rise of unhandled exceptions. There are many possible causes for app failures [21]: memory exhaustion, network conditions, device incompatibility, etc. If apps manage failures inadequately in their source code, then the app throws an unhandled exception and the operating system terminates the app—*i.e.,* the app crashes).

Whenever a crash arises, MOTIF collects *context data* in order to confine bugs. In particular, we consider two types of relevant data:

- *App context*, information derived from the execution of the app—*e.g.*, exception traces, and app version;
- *Device context*, information related to the operating context—*e.g.*, device model, SDK version, memory, and state of sensors.

After detecting exceptions in a given app, MOTIF flags the app as *buggy-suspicious* and increases the monitoring depth to track additional *user interaction events* from different devices which run the same app in the crowd. Hence, each time the user interacts with a UI element (such as a button) in the app, MOTIF logs the event metadata (*i.e.,* timestamp, view id). 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. MOTIF reports the logs to a cloud server when the device is charging and connected to the Internet. Once uploaded, the synchronized traces are automatically removed from the local storage. MOTIF distributes the monitoring among devices and redistributes it periodically to avoid any accidental user's disturbance.

B. Phase 2: Identifying crash patterns across mobile app executions.

MOTIF uses a cloud environment to aggregate the crash traces collected from a multitude of devices in the wild. First, MOTIF transforms the collection of crash traces into a weighted directed graph that we denote as *Crowd Crash Graph*. MOTIF uses the *Crowd Crash Graph* to induce 1) the minimum sequence of steps to recreate a crash, and 2) characterize the execution context under which crashes arise.

1) Aggregating Crowd Data: The *Crowd Crash Graph* represents an aggregated view of all the UI events performed in a given app before a crash arises, with their frequencies. In such a graph, nodes represent UI events (*e.g.*, a click on a button), and edges represent sequential flows between events. Nodes and edges have attributes to describe event metadata and transition probabilities, respectively. In addition, each event records the context properties observed when the event was triggered. Figure 2 shows an example of a *Crowd Crash Graph*.

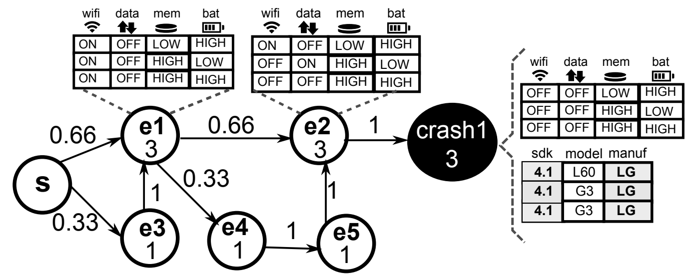

Fig. 2. Example of a Crowd Crash Graph.

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 UI events, and our edges represent sequential user interaction flows. 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.

2) Identifying Crash Patterns: MOTIF uses the *Crowd Crash Graph* to identify repeating patterns of UI events and contexts which appear frequently among crashes. MOTIF implements *Graph Traversal Algorithms*, *Sequential Patterns*, and *Set Operations* to effectively induce the minimal sequence of steps to reproduce a crash as well as the context under which this crash occurs.

MOTIF finds the shortest path from the starting node (S) to an exception node (e) which maximizes the Markov probability of the traversal. 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. 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.

Furthermore, MOTIF searches for recurrent *context patterns* within a *consolidated trace*, since not all devices suffer from the same bugs and, some crashes only arise under specific execution contexts—*e.g.*, network unavailable.

C. Synthesizing Crowd Crash Tests

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 [4]. Thus, MOTIF translates the sequence of steps in the *consolidated trace* into a test suite to 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.

2) Crowd-validation of Crash Tests: 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 context patterns 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 learned is not discriminative enough. If the test case only reproduces the failure on the consolidated context, this 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.

D. Phase 3: Hot-patching

Once a crash is isolated, the app developer can use the generated test cases to manually fix the bugs, or use our approach to automatically generate temporary patches. We propose to incorporate automatic software repair techniques (*e.g.*, GENPROG [24]) to generate patches against the failing test suite. We consider all the alternative valid patches generated by these techniques and we send them to the crowd of devices for validation.

Finally, we run all tests against the patches and check, which patches pass the tests. Some tests can only pass in specific group of devices with specific characteristics (*e.g.*, same SDK). Thus, our approach learns which patches are acknowledged for different types of devices and contexts.

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.

The main novelty of this research is to propose a collaborative approach to debug mobile apps when running in the wild. Thus, the bugs are reproduced and fixed in the field, instead of in a simulated environment. This approach exploits crowd feedback to: a) characterize the context under which crashes arise; b) generate in vivo crash test suites; c) select devices for assessing field tests; and d) select devices for assessing app fixes.

IV. RESULTS AND CONTRIBUTIONS

As a starting point, we performed a static analysis to automatically identify potential error-prone apps [16]. We began by mining $1,400,000$+ online reviews posted by users in the Google Play Store to identify error-related reviews which point out error-suspicious apps. Specifically, we identify $10,658$ buggy-suspicious apps in a dataset of $46,644$ Android apps.

To evaluate MOTIF, we randomly selected 5 Android apps for which users had reported crashes. We pre-installed MOTIF and the set of *apps under test* in 5 different Android devices with different characteristics to simulate a diverse crowd[1]. Since engaging users to participate in crowdsourced experiments is a challenge [33], we designed the experiment as a contest with a prize as incentive for users. The goal of the contest was trying to crash the 5 candidate apps as many times as possible in as many different ways as possible, during 60 minutes. Eventually, 10 participants engaged in the contest. Each time an unhandled exception raises, the exception trace and state information is reported to the MOTIF server, together with the user interaction events performed before the crash. The participants were able to generate 52 crashes (each yielding a trace for analysis) across the five apps, distributed across the different devices. Furthermore, to assess the impact of noise, we use the *Monkey* testing tool [28] to increase the number of traces. *Monkey* generates pseudo-random user events (*i.e.*, clicks, touches) in apps running on a device. We let *Monkey* send $50,000$ events to one of the apps (*Bites* app) in our dataset, and we repeated the process 50 times. The Monkey-generated traces were added to the dataset of crowdsourced traces. Table I (left) shows, for each subject app, the distribution of crash traces per app, the number of unique crashes amongst them, and the type of crash.

From the collected traces, MOTIF generates a *Crowd Crash Graph*. By using the graph, it extracts for each app the consolidated trace candidate to reproduce each crash. Table I (right) illustrates for each app, the average number of events in the traces, the number of events in the extracted consolidated trace, and the compression factor (ratio between Avg. # Events and #Consol. Events). MOTIF obtains compression factors of 7.5 up to 22. In the presence of noise (Bites-Monkey), MOTIF achieved a compression factor of 389.40.

[1]The devices used are: LG-E617G, Samsung Galaxy Nexus, GT-I9100 (2X) and Nexus S with Android SDKs from 4.03 to 4.1.2.

TABLE I. STATISTICS OF THE ANDROID APPS USED IN THE EXPERIMENT

Android App	#Traces (#unique)	Crash type	Avg. # Events	#Consol. Events	Compr. # Factor
Google I/O 2014	11 (2)	Invalid format	22	1	22
Wikipedia (2.0α)	4 (1)	Network conditions	29.5	2	14.75
OpenSudoku (1.1.5)	5 (1)	NullPointerException	60	8	7.5
PocketTool (1.6.12)	16 (1)	Missing resource	9.4	1	9.4
Bites (1.3)	16 (1)	Invalid format	56	6	9.33
Bites (Monkey)	50 (1)	Invalid format	778.79	2	389.40

The next phase of the approach is the generation of a test suite to reproduce crashes. To check if the promoted traces from crowdsourcing can reproduce the crashes, MOTIF generates the corresponding *Robotium* tests. Then, the test cases are executed on the devices to check whether the app crashes again and, if so, whether the same exception types occur. The test cases correctly reproduce the crashes in 4 (out of 5) apps. Only in the *OpenSudoku* app, the first consolidated trace failed when trying to reproduce. One benefit of the Crowd Crash Graph is indeed that it will always contain a path that reproduces the crash. Further details of the experiment are available [15].

The third and final phase of MOTIF is to hot-patch the apps to avoid crashes. We randomly selected one of the buggy apps used in the experiment—*i.e.*, the *PocketTool* app. To automatically avoid the observed failures, we have implemented a basic patching strategy [13], which consists in inserting `try/catch` blocks in different suspicious locations in the code of the app. These locations are the different methods that appear in the exception traces. The patching strategy is performed by rewriting the bytecode of apps and updating the apps in the devices. After deploying the patched apps, we observe if the apps stop sending exceptions and we learn which patches avoid the bugs in which contexts. In this case study, we generated two patches and only one of them actually avoids the crash.

A. Implementation Details

We have implemented a proof-of-concept prototype to demonstrate the feasibility of the approach. 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.

The Android client library runs the `adb` tool (Android Debug Bridge) in the device to communicate with the Android virtual machine (named *Dalvik*) using the *Java Debug Interface (JDI)*. This library enables to intercept *user interaction* and *exception* events.

MOTIF sends the data collected in devices to a cloud service for aggregation and analysis using `APISENSE` [1]. `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 [29].

All approaches that record user inputs put privacy at risk [34]. 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* [9] and *input minimization* [35] techniques.

To sum up, the main contribution derived from this research is a crowdsourced monitoring approach, MOTIF, to support developers to detect, reproduce, and fix crashes faced by end-users in the wild. 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. 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 demonstrate the power of the crowd and the benefits of considering a multitude of devices. As future work, we plan to analyze trade-offs between the amount of data collected and the reproducibility of the approach.

Although this approach focuses on bugs that manifest with crashes, the conceptual foundations of our approach could be extended in order to tackle other types of problems (*e.g.*, performance or energy bugs). For example, we have also exploited the power of the crowd to help developers to identify another critical issue for the quality of mobile apps: *UI performance regressions* [14].

REFERENCES

[1] APISENSE. http://apisense.io.

[2] Google Analytics. http://www.google.com/analytics.

[3] Google: Android app downloads have crossed 50 billion. http://www.theverge.com/2013/7/24/4553010/google-50-billion-android-app-downloads-1m-apps-available.

[4] Robotium. https://code.google.com/p/robotium.

[5] SPLUNK. https://mint.splunk.com.

[6] Testdroid. http://testdroid.com.

[7] Xamarin Test Cloud. http://xamarin.com/test-cloud.

[8] 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-IX, pages 22:1–22:6, New York, NY, USA, 2010. ACM.

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

[10] S. Artzi, S. Kim, and M. Ernst. ReCrash: Making software failures reproducible by preserving object states. volume 5142 of *Lecture Notes in Computer Science*, pages 542–565. Springer Berlin Heidelberg, 2008.

[11] 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.

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

[13] 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.

[14] M. Gomez, R. Rouvoy, B. Adams, and L. Seinturier. Mining Test Repositories for Automatic Detection of UI Performance Regressions in Android Apps. In R. Robbes and C. Bird, editors, *13th International Conference on Mining Software Repositories (MSR'16)*, Proceedings of the 13th International Conference on Mining Software Repositories, Austin, Texas, United States, May 2016. IEEE.

[15] M. Gómez, R. Rouvoy, B. Adams, and L. Seinturier. Reproducing Context-sensitive Crashes of Mobile Apps using Crowdsourced Monitoring. In L. Flynn and P. Inverardi, editors, *IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft'16)*, Proceedings of the 3rd IEEE/ACM International Conference on Mobile Software Engineering and Systems, Austin, Texas, United States, May 2016. IEEE.

[16] 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.

[17] N. Haderer, R. Rouvoy, and L. Seinturier. Dynamic deployment of sensing experiments in the wild using smartphones. In *DAIS*, pages 43–56, 2013.

[18] 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.

[19] G. Hu, X. Yuan, Y. Tang, and J. Yang. Efficiently, effectively detecting mobile app bugs with AppDoctor. In *Proceedings of the 9th European Conference on Computer Systems*, page 18. ACM, 2014.

[20] W. Jin and A. Orso. BugRedux: Reproducing field failures for inhouse debugging. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 474–484, Piscataway, NJ, USA, 2012. IEEE Press.

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

[22] H. Khalid, E. Shihab, M. Nagappan, and A. Hassan. What do mobile app users complain about? *Software, IEEE*, 32(3):70–77, May 2015.

[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. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1):54–72, 2012.

[25] 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.

[26] 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.

[27] 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.

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

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

[30] 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.

[31] J. Röβler, G. Fraser, A. Zeller, and A. Orso. Isolating failure causes through test case generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 309–319. ACM, 2012.

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

[33] 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.

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

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