Towards Scalable Automated Mobile App Testing

Ranveer Chandra, Börje F. Karlsson, Nicholas D. Lane
Chieh-Jan Mike Liang, Suman Nath, Jitu Padhye, Lenin Ravindranath, Feng Zhao

Microsoft Research

MSR-TR-2014-44

The mobile app ecosystem continues to rapidly grow in importance as an increasing proportion of our daily computing needs shift away from desktop machines in favor of mobile devices. However, mobile apps must cope with extremely diverse operating conditions due to factors like device fragmentation, wireless network heterogeneity and varied user behavior. App developers and distributors (i.e., operators of app marketplaces) lack testing tools that can effectively account for such diversity and, as a result, app failures and performance bugs (e.g., excessive energy consumption) are commonly found in mobile apps today. Towards addressing this challenge to mobile app development, we have developed key techniques for scalable automated mobile app testing within two prototype app testing services – VanarSena and Caiipa. In this paper, we describe SMASH – our vision for a single unified cloud-based mobile app testing service that will combine the strengths of the VanarSena and Caiipa systems towards solving the complexities presently faced by testers of mobile apps.

I. Introduction

For the majority of users world-wide, computing has evolved to be now largely defined by their mobile devices and the apps that run on them. The apps are distributed through app stores that are brutally competitive [22, 21]. For an app to succeed, it must not only offer a useful service, but also provide a good user experience. An app that performs slowly or crashes frequently may easily be doomed to obscurity as unforgiving users quickly switch to alternate apps and/or leave poor reviews in app stores, discouraging new users to use it. To avoid such consequences, app developers therefore need to make sure that their apps run smoothly not just in their development environment, but also in the wild, in the hands of real users. Similarly, distributors of apps operating mobile app marketplaces must make sure apps in their catalogs meet certain standards of performance and quality.

Ensuring this, however, is extremely challenging [25, 1, 17]. Unlike traditional "enterprise" software, mobile apps are often used in more uncontrolled conditions, in a variety of different locations, over different wireless networks, with a wide range of input data from user interactions and sensors, and on a variety of hardware platforms. For example, wireless network speeds and latencies can fluctuate by a 100-fold at different locations across the world [12], mobile devices themselves differ by screen size, CPU speed, available memory and operating system versions [16]. An app running smoothly in the development environment with a good wireless network and a powerful device may run very slowly or crash when a user runs it with poor network and a weaker device. The space of real world conditions that an app may run on can be extremely large and coping with all the issues can be particularly acute for individual developers or small teams [24].

Existing mobile app testing tools available during the development process – for example those that perform static (e.g., [4, 10, 19]) or conventional-forms of dynamic analysis [13, 3, 20] – are not well-suited to this challenge. The principle limitation of existing techniques is their constrained ability to test app behavior under complex real-world conditions. At best, recent dynamic testing tools that exercise a mobile app by simulating user interaction are able to do so while simulating a small number of generic contexts, such as, 3G or WiFi network connections (e.g., [14, 11]). But such tools support only a small fraction of the conditions encountered in the real-world and do not attempt to represent complex environments in which multiple conditions (e.g., type of user, network, location, device state) can contribute towards unexpected consequences that negatively effect the app. As a result, app developers and distributors must heavily rely on the *post-facto* analysis of telemetry data (e.g., [2, 6, 23]) collected from a deployed mobile app, as a means catch many bugs related to users, devices and the environment. Although, this approach exposes apps to real-world conditions, developers get a chance to fix their bugs only *after* the apps are used by real users. It is often too late – users have already been exposed to buggy apps and posted poor ratings, discouraging new users to even give a chance to (potentially bug-fixed version of) the apps.

To address these challenges, we have recently developed two tools Caiipa [5] and VanarSena [18], which allow developers to automatically test their apps under a variety of conditions. Both tools have their strengths and their weaknesses, and the goal of this paper is to briefly describe how the tools can be combined into an unified system, that we plan to call *Scalable Mobile App Software Hardening* (SMASH).

SMASH would consist of a configurable environment into which apps can be installed and their functionality explored. During exploration the app can be systematically

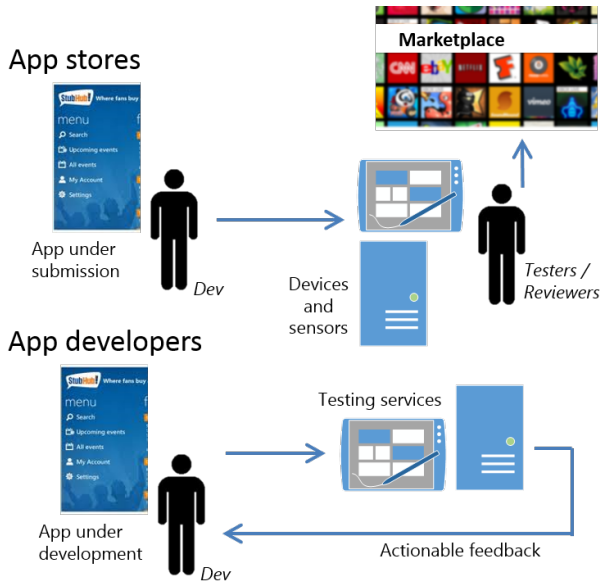

Figure 1: SMASH Usage Scenarios

exposed to a variety of environments and system events. For example, a music streaming app might be repeatedly exercised through common user operations (e.g., playing music, changing songs) while facing a variety of broad network conditions (e.g., a 3G to WiFi network hand-off) and common network faults (e.g., HTTP protocol error codes). During tests both the app and general system behavior can be closely monitored for problems ranging from the obvious, such as app crashes, to the subtle including memory leaks or excessive energy consumption. A key enabler for this approach is the use of cloud infrastructure allowing the number of hosting environments to be scaled up as required based on the number of relevant tests or completion time requirement. However, support for app execution on real hardware is also provided to cover specific scenarios and to provide performance baselines.

We envision deploying SMASH as a testing service. An app developer should be able to submit an app binary to the system, and then within a short amount of time obtain a report. This report should include performance problems or crashes, for each such problems, app's execution point (e.g., a stack trace for a crash) and external conditions (e.g., poor network condition) that triggered it. In a like manner, app distributors should be able to submit apps requested to be listed an app marketplace and determine if this app meets distributor-defined policies for app robustness and resource consumption.

The remainder of this paper is structured as follows. We begin in Section II by first outlining the design considerations of SMASH. Section III continues by presenting a high-level description of SMASH and outlines how the individual techniques found in VanarSena and Caiipa operate together within SMASH. Section IV presents the status of Caiipa and VanarSena towards building SMASH. Finally, we provide concluding remarks in Section VI.

II. SMASH Goals and Challenges

Our goal is to build a scalable, easy to use system that tests mobile apps for frequently occurring, externally-inducible faults as thoroughly as possible. As illustrated in Figure 1, we target two specific types of end-user scenarios:

- **App developers** who use SMASH to complement their existing testing procedures by stress-testing code under hard to predict combinations of contexts.
- **App distributors** who accept apps from developers and offer them to consumers (such as, entities operating marketplaces of apps) – distributors must decide if an app is ready for public release.

Since we anticipate the system being in daily use by both user categories – for example, in the case of the developer using it interactively while debugging; or, in the case of distributors whenever new batches of apps are submitted for release – we want to return the results of testing, in form of an actionable report, to users as quickly as possible. Finally, we want the system to be deployable as a cloud service in a scalable way.

II.A. Thoroughness

For SMASH to achieve its goal of thoroughness when testing for frequently occurring, externally-inducible faults our design targets the following characteristics.

High Execution Coverage. While testing an app, SMASH aims to execute as many of its execution paths as possible. Since mobile apps are UI-centric, one way to measure execution coverage is to use *page coverage* that represents the fraction of unique app pages visited by the system while testing an app. SMASH aims to maximize page coverage within a given time budget.

High Fault Coverage. While executing an app for testing, SMASH exposes it to many external environments or faults. Examples of faults include poor network connection, malfunctioning sensor, a hardware device with small screen, etc. Since the space of possible faults is potentially infinite, SMASH aims to cover the most common faults that appear in the wild.

Performance, Not Just Bugs. The resource usage of apps, such as energy, is just as important to the correctness and robustness of a given app. Users would be unwilling to use an app that quickly exhausts battery, no matter how "robust" it is. Thus, it is important for a testing solution to report test results that carefully consider app performance and efficiency – especially under changing conditions.

II.B. Scalability and Speed

We want SMASH to scale to testing a large number of apps. SMASH needs to thoroughly test each app and generate test results within a matter of hours, so developers and app

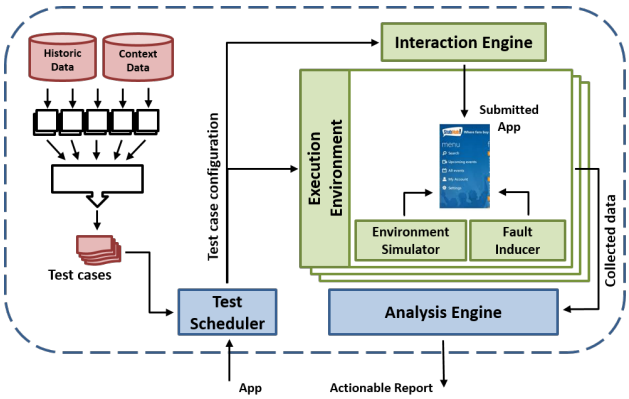

Figure 2: SMASH Architecture

distributors can more easily incorporate SMASH into their work flow.

A number of obstacles typically limit the scalability of app testing. First, the amount of time necessary to simulate faithfully UI interactions with apps can be time consuming. For example, waiting for app network traffic to complete before progressing to the next UI interaction. Second, app distributors release hundreds of new apps to the public everyday [8, 7]. Each of these must be verified before this can be done. Distributors often only have 20 or 30 minutes to examine an app and decide if it should be released – further motivating SMASH to operate efficiently.

II.C. Actionable Reports

Test results must be customized for each end-user. For example, an app distributor may require analysis of test results with respect to certain store policies (such as, max CPU load on the device, or a time limit for app startup). Alternatively, a developer will require much more detailed outputs (e.g., stack traces, root cause analysis, causal relationship between various asynchronous calls) that direct them towards which part of the app should be debugged and how to decide what problems to address first.

III. SMASH Architecture

At a high level, SMASH will consist of three components: (1) an app interaction engine that executes and manipulates the app; (2) an execution environment that exposes the executing app to various external conditions and injects various faults; and (3) an analysis engine that processes and reasons over the collected data to produce a tailored report for the system user. For certain apps, SMASH can also use an instrumenter, to instrument the app binary. A test scheduler module controls the workflow of the system. Figure 2 illustrates the SMASH architecture.

III.A. App Interaction Engine

The App Interaction Engine will spawn a number of *monkeys* to test the app. A monkey is a UI automation tool to explore various parts of an app. It can launch the app on a real mobile device or an emulator and interact with it by mimicking user interactions (e.g., clicking a button or swiping a page) to recursively visit various pages of the app. The monkey can thus explore the UI-state transition graph of the app, where each page corresponds to a UI-state and each interaction results in a state transition. The key optimization goal of a monkey is to maximize the number of explored states (i.e., coverage) within a given time budget (i.e., speed). If the app is instrumented, this exploration can be aided by the information collected by the instrumentation. In addition, an initial instrumented run can generate data to inform subsequent tests.

SMASH will incorporate various optimizations described in Caiipa and VanarSena to improve coverage and speed of the monkey.

First, SMASH will try to prune the state space without sacrificing testing coverage. It can identify all state transitions that invoke the same event handler or lead to a similar state and will explore only one of such transitions. Second, it can dynamically track when a state transition has completed so that it can immediately initiate the next transition. Third, in addition to exhaustively exploring the UI-state graph, it can prioritize its exploration paths so that more important states (e.g., states that are visited by real users more often, given by the developer or identified by telemetry data from real users) are explored before others. SMASH can exploit a User Interaction Model, built based on telemetry data of real app usage, that generates user events (e.g., touch events, key presses, data input) based on weights (i.e. probability of invocation) assigned to specific UI items. Such prioritization is useful when the monkey does not have enough time to explore all UI-states or when the developer wishes to test for problems that are more likely to affect real users. Finally, it can utilize semantically meaningful inputs such as login/passwords, unique gestures, etc. provided by the developer, which are needed for transitioning to various UI-states but a fully automated system cannot generate.

For scalability, we assume that multiple parallel runs of the apps do not affect each other. Thus, the execution engine can spawn a large number of monkeys, each testing the app for a given external condition.

III.B. Execution Environment

The goal of each instance of the Execution Environment is to systematically emulate various operating conditions while the App Interaction Engine exercises an app. SMASH selects conditions that (1) occur in the real world and (2) are "unusual" enough to be missed or hard to produce by most developers while testing their own apps. For thoroughness, SMASH will consider a diverse set of faults, due to environment (e.g., network connectivity, locations), device (e.g., low memory), user behavior (e.g., impatient interactions), sensors (e.g., sensor timeout), inputs (e.g., incorrect text inputs), etc.

The key challenge is to identify what external conditions to emulate. SMASH will leverage two types of data sources to address this: (i) databases of historical crash or telemetry data; and (ii) collected data about mobile environment dimensions (e.g. network conditions, CPU and memory

availability, sensor output). SMASH will mine databases of historical crash or telemetry data to identify and rank (for example, by frequency) common faults (i.e., problematic situations) and emulates the most common faults to induce problems in the app. Our initial experience from an analysis of 25 million real-world crashes shows that most of them are caused by a small number of root causes, making it feasible for SMASH to systematically induce them to app to test if the app would crash in the wild due to those common situations. A few examples of common faults are impatient user interaction while the app is still loading a page, malformed XML data from the network, slow network, etc. SMASH can induce such faults by an impatient monkey, a network proxy, and a network emulator, respectively.

To go beyond testing for previously detected common scenarios, SMASH will also use a representative, yet comprehensive, library of context stress tests from large repositories of available context sources. It will use machine learning techniques to identify representative contexts by (i) determining which combinations of contexts are likely to occur in the real world, and (ii) removing redundant combinations of contexts. This library generation process will be fed using datasets collected from real devices (e.g., WER[1] [9], OpenSignal [15]) in addition to challenging mobile contexts defined by domain experts. This process will happen as a pre-processing step that is re-run periodically with updated source data. With such a context library, SMASH can simulate conditions, such as different CPU performance levels, amount of available memory, controlled sensor readings (e.g., GPS drivers reporting programmatically defined locations), and different network parameters to simulate different network interfaces (e.g., WiFi, GPRS, WCDMA), network quality levels, and network transitions (e.g., 3G to WiFi) or handoffs between cell towers.

SMASH can also prioritize test cases for a given app. To this end, it will use a learning algorithm that leverages similarities between apps to identify which conditions are most likely to impact previously unseen apps via observations from previously tested apps. Based on such similarities, SMASH can determine the order in which test cases are applied to specific apps. As each new set of results is reported, tests can potentially be re-prioritized.

III.C. Analysis Engine

At the end of testing an app, SMASH will generate a report that the developer can use to reproduce the problem and to pinpoint the code point and likely causes behind it. The information will include replay logs and detailed user transaction traces that elucidate the causal relationship between user actions, various thread invocations, and the performance problems/crashes.

In addition to reporting crashes, SMASH also detects and reports anomalous app performance. Understanding app performance data (e.g., energy, latency) is challenging, because it is difficult to identify truly abnormal app behavior compared to changes in performance that are unavoidable given the conditions to which the app is exposed. SMASH will use several techniques, that build upon the techniques used in Caiipa. For example, to determine "normal" behavior in a given setting, SMASH will consider the performance of previous runs of the same app, as well as that of other apps that are similar to the target app. Looking at how different metrics are affected by changes in context helps identify the real anomalous cases.

Finally, since the number of issues that appear to require investigation can be quite large; SMASH will also provide rankings of severity issues to help developers prioritize their time.

[1]Windows Error Reporting

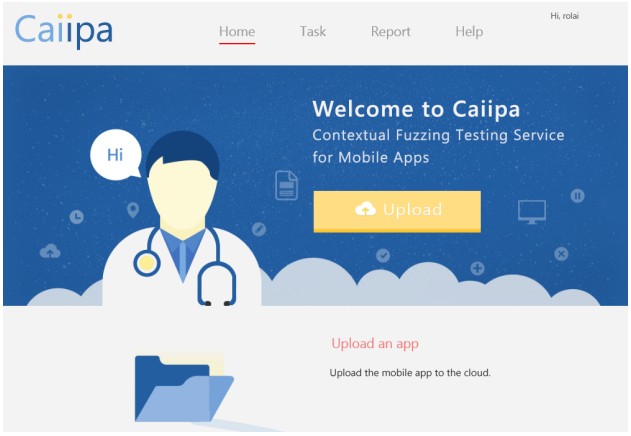

Figure 3: Caiipa beta service

IV. Progress Towards SMASH

Assembling SMASH will be simplified because of the two already working mobile app testing prototypes – Caiipa and VanarSena. Each prototype has been used to evaluate and explore separate techniques and scenarios required by SMASH.

Caiipa aims to stress test mobile apps to cover a wide range of potential real-world conditions apps may encounter. Caiipa is designed to test an app under conditions a developer never anticipated occurring. Tests consider both app failures as well as identifying resource consumption outliers (e.g., excessive energy consumption).

In contrast, VanarSena seeks to efficiently test common case failure conditions of mobile apps – attempting to catch root cause conditions that are responsible for the majority of failures observed in the wild. VanarSena focuses on app failures (i.e., crashes) only but can generalize to anomalous resource consumptions using techniques similar to those developed by Caiipa.

IV.A. Caiipa

As illustrated in Figure 3, Caiipa [5] has been deployed as an internal service available to Microsoft employees. The Caiipa service tests mobile apps under a variety of mobile environment conditions, including network bandwidth and

quality, varied device types, memory levels, locations, and running key tests on real hardware.

At a high-level, its workflow is similar to the one for SMASH, shown in Figure 2. The responsibility of selecting which environments to be run by the Environment Emulation Engine belong to two components under Caiipa: (i) a generator of context library tests and (ii) a prioritizer that selects the most important test to run next for a given individual app or group of apps. These two modules are key to scalably search the space of real-world conditions.

By focusing on the impact of mobile contexts in app behaviour, Caiipa's default interaction engine is relatively simple and is most effective when primed with specific sequences of user interaction (e.g., by the developer) – otherwise a simple weighted exploratory user model is applied. However, the system is built such that the UI manipulation module can easily be replaced.

One limitation of interaction interface comes from adopting a black-box approach during app testing. However, this allows the system to test any app regardless of the languages and tools (e.g., JavaScript, C#, Silverlight) used during development, which are common in the Windows app ecosystem.

To cover a wide breadth of mobile conditions, Caiipa currently includes a large context library (10,504 test cases). Test cases are synthesized largely from databases containing global conditions of cellular and WiFi networks, but also memory and CPU test events sourced from Microsoft WER [9].

Caiipa tested 265 commercially available Windows Store and Windows Phone 8 apps in depth. Our results show that test prioritization can find up to 47% more crashes than the conventional baselines, with the same amount of computing resources. Additionally, by considering the different real-world contexts, Caiipa detects $11\times$ more crashes and $8\times$ more performance problems.

IV.B. VanarSena

VanarSena architecture is described in detail in [18]. VanarSena is designed to thoroughly test mobile apps for a small set of externally inducible common faults using a greybox approach. It neither treats the app as a blackbox, nor does it try to understand the app semantics (i.e. whitebox testing). Instead, it instruments the app binary and runs it within the Windows Phone Emulator using an UI automator monkey. Several fault inducing modules (FIMs) induce faults such as network errors and simulating an impatient user. Both the monkey and the FIMs rely on information collected by the added instrumentation to speed up app testing without sacrificing coverage.

VanarSena uses two main techniques for improving speed of testing. The first one is called *hit testing*. When presented with the app UI, VanarSena uses hit testing to quickly classify various UI controls (buttons, lists, etc.) into equivalent classes, such that controls within a class exercise the same code path in the app. The UI automator then invokes only one control from each class, which considerably speeds up testing. Second, the added instrumentation generates *ProcessingCompleted* event, which allows the UI automator to precisely decide when to interact with the app next.

The current implementation of VanarSena can test apps written in C# (Sliverlight) for Windows Phone platform. We tested 3,000 apps from the Windows phone marketplace using VanarSena, and uncovered 2,969 distinct faults, of which 1,227 were previously unreported.

V. Conclusion

In this paper, we have described how we plan to integrate features from Caiipa and VanarSena to build SMASH. Currently both systems address different requirements. VanarSena is currently being incorporated in Microsoft's internal quality assurance toolchain. While Caiipa is in use by partner product teams during app development. Needless to say, new additional challenges will arise as we actually build the combined system. However, we expect that the foundational vision laid out in this report will be adequate to guide the development of SMASH and help improve mobile app testing in general.

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

[2] Applause. `http://applause.com/`.

[3] 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 '13, pages 641–660, New York, NY, USA, 2013. ACM.

[4] L. Batyuk, M. Herpich, S. A. Camtepe, K. Raddatz, A.-D. Schmidt, and S. Albayrak. Using Static Analysis for Automatic Assessment and Mitigation of Unwanted and Malicious Activities within Android Applications. In *Proceedings of the 2011 6th International Conference on Malicious and Unwanted Software*, MALWARE '11, pages 66–72, Washington, DC, USA, 2011. IEEE Computer Society.

[5] Chieh-Jan Mike Liang, Nicholas D. Lane, Niels Brouwers, Li Zhang, Börje Karlsson, Hao Liu, Yan Liu, Jun Tang, Xiang Shan, Ranveer Chandra and Feng Zhao. Caiipa: Automated Large-scale Mobile App Testing through Contextual Fuzzing. In *Proceeding of the 20th Annual International Conference on Mobile Computing and Networking*, MobiCom '14, New York, NY, USA, 2014. ACM.

[6] Flurry. `http://www.flurry.com/`.

[7] Flurry. Electric Technology, Apps and The New Global Village. `http://www.flurry.com/bid/91911/Electric-Technology-Apps-and-The-New-Global-Village`.

[8] Fortune. 40 Staffers. 2 Reviews. 8,500 iPhone Apps per week. `http://fortune.com/2009/08/22/40-staffers-2-reviews-8500-iphone-apps-per-week/`.

[9] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 103–116, New York, NY, USA, 2009. ACM.

[10] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS '08. The Internet Society, 2008.

[11] Google. UI/Application Exerciser Monkey. `http://developer.android.com/tools/help/monkey.html`.

[12] J. Huang, F. Qian, Q. Xu, Z. Qian, Z. M. Mao, and A. Rayes. Uncovering Cellular Network Characteristics: Performance, Infrastructure, and Policies. Technical Report MSU-CSE-00-2, 2013.

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

[14] Microsoft. Simulation Dashboard for Windows Phone. `http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj206953(v=vs.105).aspx`.

[15] Open Signal. `http://opensignal.com`.

[16] Open Signal. The Many Faces of a Little Green Robot. `http://opensignal.com/reports/fragmentation.php`.

[17] A. Pathak, Y. C. Hu, and M. Zhang. Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, pages 5:1–5:6, New York, NY, USA, 2011. ACM.

[18] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and Scalable Fault Detection for Mobile Applications. In *Proceeding of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, New York, NY, USA, 2014. ACM.

[19] 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 Conference on Operating Systems Design and Implementation*, OSDI'12, pages 107–120, Berkeley, CA, USA, 2012. USENIX Association.

[20] A. Reina, A. Fattori, and L. Cavallaro. A System Call-centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware Behaviors. In *Proceedings of the 6^{th} European Workshop on System Security (EUROSEC)*, Prague, Czech Republic, April 2013.

[21] Techcrunch. Mobile App Users Are Both Fickle And Loyal: Study. `http://techcrunch.com/2011/03/15/mobile-app-users-are-both-fickle-and-loyal-study`.

[22] Techcrunch. Users Have Low Tolerance For Buggy Apps: Only 16% Will Try A Failing App More Than Twice. `http://techcrunch.com/2013/03/12/users-have-low-tolerance-for-buggy-apps-only-16-will-try-a-failing-app-more-than-twice`.

[23] TestFlight. `http://testflightapp.com/`.

[24] Wall Street Journal. The Surprising Numbers Behind Apps. `http://blogs.wsj.com/digits/2013/03/11/the-surprising-numbers-behind-apps/?mod=dist_smartbrief`.

[25] A. I. Wasserman. Software Engineering Issues for Mobile Application Development. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 397–400, New York, NY, USA, 2010. ACM.