HOW TO THE NEXT BILLION MOBILE APP BUGS?

With users increasingly dependent on their phones, tablets, and wearables, the mobile app ecosystem is more important today than ever before. Creating and distributing apps has never been more accessible. Even single developers can now reach global audiences. But mobile apps must cope with extremely varied and dynamic operating conditions due to factors like diverse device characteristics, wireless network heterogeneity, and varied user behavior. App developers and operators of app marketplaces both lack testing tools that can effectively account for such diversity and, as a result, app failures and performance bugs (like excessive energy consumption) are commonly found today. To address this challenge to mobile app development, we have developed key techniques for scalable automated mobile app testing within two prototype services — VanarSena and Caiipa. In this paper, we describe our vision for SMASH, a unified cloud-based mobile app testing service that combines the strengths of both previous systems to tackle the complexities presently faced by testers of mobile apps.

Illustration, (this page) bigstockphoto.com

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

Editor: Geoffrey Challen

For the majority of users worldwide, computing has evolved to be largely defined by their mobile devices and the apps that run on them, which are distributed through app stores that are fiercely competitive [11]. An app that performs slowly or crashes frequently may easily fade into obscurity as unforgiving users are increasingly annoyed or switch to alternatives. Even brand-name apps for which there are no alternatives risk tarnishing their reputations through a poor user experience. To avoid such consequences, app developers need to make sure that their apps run smoothly not just in their development environment, but also in the hands of real users. Similarly, app marketplace operators must make sure apps they distribute meet certain standards of performance and quality.

However, ensuring this is extremely challenging [1]. Unlike traditional applications, mobile apps are often used in a variety of 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 speed and latency can fluctuate 100-fold at different locations across the world [7]; and mobile devices themselves differ by screen size, CPU speed, available memory and operating system version [8]. An app running smoothly during development with a good wireless network and on a powerful device may run very slowly or crash when a user runs it in other environments. Coping with all of these issues can be particularly challenging for individual developers or small teams [12].

Existing testing tools available during the development process — for example, those that perform static or conventional-forms of dynamic analysis [2,3,6] — are not well suited to this challenge. The principle failure of existing techniques is their restricted ability to test app behaviour under complex real-world conditions. Recent testing tools that exercise a mobile app by simulating user interaction do so while also simulating a small number of generic contexts, such as, 3G or Wi-Fi network connections. However, 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 (type of user, network, location, device state) can cause unexpected negative app side effects (excessive energy consumption, app stalls, crashes). As a result, app developers and distributors rely heavily on post-facto analysis of telemetry data collected from already deployed mobile apps[1]. Although this approach exposes apps to real-world conditions and developers get a chance to fix bugs, it is often too late—users have already used the buggy apps and posted poor reviews.

To address these challenges, we have recently developed two systems, Caiipa [4] and VanarSena [9], which allow developers to automatically test their apps under a variety of conditions. Both tools have their strengths and weaknesses, and the vision for this paper is to describe how they can be combined into a unified system to fulfill developer needs. We refer to this single system as SMASH: *Scalable Mobile App Software Hardening.*

SMASH consists of a configurable environment into which apps can be installed and within which their functionality explored. During exploration, the app can be systematically exposed to a variety of environments and system events. For example, a music streaming app might be repeatedly exercised through common user operations (playing music) while facing a broad variety of network conditions (a 3G to Wi-Fi network hand off) and common network faults (HTTP protocol errors). During tests, both the app and general system behavior can be closely monitored for problems ranging from the obvious, such as app crashes, to subtle 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 constraints. 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 where developers are able to submit an app binary to the system and, within a short amount of time, obtain a comprehensive report. This report includes performance problems, hangs, or crashes; with the app's execution point (interaction and stack traces) and external context (specific network characteristics, for example) that triggered them. Likewise, marketplace operators should be able to submit pre-release apps and determine if these apps meet distributor-defined policies for robustness and resource consumption. Report in hand, developers will be able to quickly fix problems by zooming in on the problematic code and having a better understanding of the environments that cause problems.

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 user scenarios:

- **App developers:** who use SMASH to complement their existing testing procedures by stress-testing code under hard-to-predict combinations of contexts.

[1] Usually from services like http://applause.com/ or http://www.flurry.com/.

- **App marketplace operators:** who accept apps from developers and offer them to consumers, and so must decide if an app is ready for public release.

We anticipate the system being in daily use by both user categories. For example, a developer using it interactively while debugging, or, in the case of marketplaces, whenever new batches of apps are submitted for release. As such, the results of testing must reach users as quickly as possible and in an actionable form. Moreover, the system also needs to balance thoroughness of tests with speed of results.

Thoroughness

For SMASH to achieve its goal of thoroughness when testing, 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, an important measure of execution coverage is the fraction of unique app pages visited and UI elements manipulated by the system while testing an app.

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, unavailable functionalities, etc. Since the space of possible faults is potentially infinite, SMASH aims to cover the most probable 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.

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 distributors can more easily incorporate it into their workflow.

A number of obstacles typically limit the scalability of app testing. First, faithfully simulating 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 every day and often only have 20 or 30 minutes to examine an app before deciding if it should be released [5].

Actionable Reports

Test results must be customized for the type of end-user. For example, an app distributor may require analysis of test results with respect to certain store policies, such as app startup time. Alternatively, a developer will require much more detailed outputs (stack traces, 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.

SMASH ARCHITECTURE

At a high level, SMASH consists of three components: an app interaction engine that executes and manipulates the app; an execution environment that exposes the executing app to various external conditions and injects faults; and 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 collect additional information [10]. A test scheduler module controls the workflow of the system. Figure 1 illustrates the SMASH architecture.

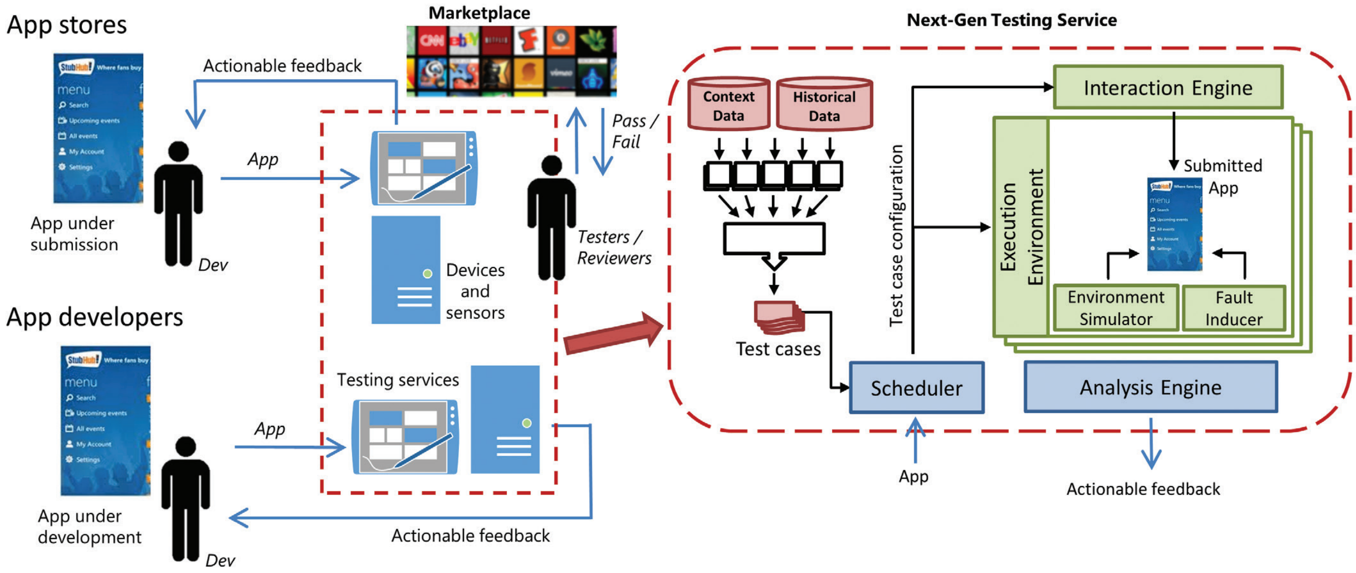


FIGURE 1. SMASH Usage Scenarios and Architectural Overview

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 (clicking a button, swiping a page) to recursively visit various pages of the app. The key optimization goal of a monkey is to maximize the number of explored states within a given time budget. SMASH will incorporate various optimizations developed in VanarSena and Caiipa to improve coverage and speed of the monkey.

First, SMASH tries to 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 are explored before others. Such states could be specified by developers or identified via telemetry data from real users. These forms of prioritization are useful when SMASH does not have enough time to explore all UI-states or when developers wish to test for specific problems that are more likely to affect real users.

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 both occur in the real world and are unusual enough to be missed or hard for developers to reproduce while testing their own apps. For thoroughness, SMASH considers a diverse set of faults, due to environment (network connectivity, locations), device (low memory, busy CPU), user behaviour (impatient interactions), inputs (incorrect text entry, sensor readings), etc.

The key challenge is to identify which external conditions to emulate. SMASH can leverage two types of data sources to address this: databases of historical crash or telemetry data; and collected data about the mobile environment, such as network conditions and CPU and memory availability. By mining such datasets, SMASH can identify and rank problematic situations. Our initial experience from an analysis of 25 million real-world crashes [9] shows that most of them are caused by a small number of root causes, making it feasible for SMASH to systematically induce them.

To go beyond testing for previously detected common scenarios, SMASH makes use of a comprehensive library of stress tests from repositories of mobile context sources. It uses machine learning techniques to identify representative contexts by determining which combinations of contexts are likely to occur in the real world, and removing redundant combinations. This library generation process is fed datasets collected from real devices[2]. With such a context library, SMASH can simulate conditions, such as different CPU performance levels, amount of available memory, controlled sensor readings (GPS locations), and different network parameters (Wi-Fi, WCDMA), network quality levels, and network transitions (3G to Wi-Fi).

SMASH can also prioritize test cases for a given app. To this end, it utilizes a learning algorithm that leverages similarities between apps to identify which conditions are most likely to impact previously untested apps via observations from previously exercised apps.

Analysis Engine

After testing an app, SMASH will generate a report that the developer can use to help reproduce any problem found and pinpoint the likely causes behind it. The information includes replay logs, detailed user transaction traces, and the performance problems/crashes.

In addition to reporting crashes, SMASH also reports anomalous app performance. However, understanding performance data (energy usage, latency) is challenging. It is difficult to identify truly abnormal app behavior compared to changes in performance that are unavoidable given the tested conditions. SMASH can build upon the techniques used in Caiipa. For example, to estimate normal behavior in a given setting, SMASH considers the performance of previous runs of the same app, as well as that of other apps that are similar to the target app. Because the number of issues that appear to require investigation can be quite large, SMASH will also provide rankings of severity to help developers prioritize their time.

Although SMASH cannot detect all forms of problematic app behavior, new analysis modules can be added to the system as challenges in detecting additional problems are overcome.

PROGRESS TOWARDS SMASH

Assembling SMASH is facilitated by our two already working mobile app testing prototypes — Caiipa and VanarSena. Each prototype has been used to 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. It 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 like excessive energy consumption or the app process hanging. In contrast, VanarSena focuses on app crashes and seeks to efficiently test common-case failure conditions of mobile apps — attempting to identify and focus on root cause conditions that are responsible for the majority of failures observed in the wild.

Caiipa

As illustrated in Figure 2, Caiipa [4] has been deployed as an internal service at Microsoft. Caiipa can test 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.

By focusing on the impact of mobile contexts in app behaviour, Caiipa's default interaction engine is relatively simple and applies a light weighted exploratory user model, customizable as needed. One limitation of UI interaction comes from adopting a black-box approach during app testing. However, this allows the system to test any app regardless of the languages used during development.

[2] Microsoft's Windows Error Reporting (WER) or OpenSignal (http://opensignal.com).

To cover a wide breadth of mobile conditions, Caiipa currently includes a large context library of 10,504 test cases. 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.

VanarSena

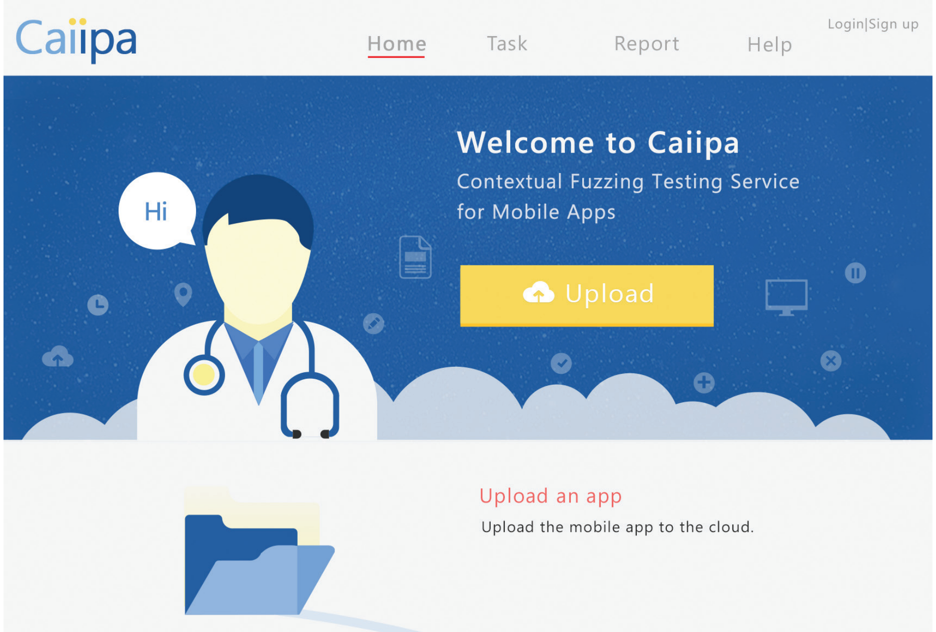
VanarSena's architecture is described in detail in [9]. VanarSena is designed to test apps for a small set of externally inducible common faults using a greybox approach; the app binary is first instrumented and then run within the Windows Phone Emulator using a UI monkey. Several fault inducing modules (FIMs) induce faults such as network errors and simulating an impatient user.

VanarSena uses two main techniques for improving the speed of testing. The first one is called hit testing. When presented with the app UI, VanarSena classifies various UI controls into equivalent classes that exercise the same code path in the app. The monkey then invokes only one control from each class, which considerably speeds up testing. Second, the added instrumentation generates *ProcessingCompleted* events, which allows the monkey to precisely decide when to next interact with the app.

VanarSena supports testing apps written in C# for the Windows Phone platform and was used to test 3,000 Windows Phone apps, uncovering 2,969 distinct faults, of which 1,227 were previously unreported.

CONCLUSION

In this paper, we have described how, by integrating features from Caiipa and VanarSena, a next generation test service SMASH can be built. Currently both systems address different requirements including Microsoft's internal quality assurance processes and cross-platform developer tools. Needless to say, new additional challenges will arise as we move forward in the combined system. However, we expect the foundational vision laid out in this paper to guide the development of SMASH and help improve mobile app testing in general. ■



Welcome to Caiipa

Contextual Fuzzing Testing Service for Mobile Apps

Upload the mobile app to the cloud.

FIGURE 2. Caiipa Prototype

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.

[2] 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.

[3] 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.

[4] Chieh-Jan Mike Liang, Nicholas D. Lane, Niels Brouwers, Li Zhang, Börje F. 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.

[5] 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/.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] Wall Street Journal. The Surprising Numbers behind Apps. http://blogs.wsj.com/digits/2013/03/11/the-surprising-numbers-behind-apps/.