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 unsympathetic 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. 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, Caiap [4] and VanaSena [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 is 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.
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 the 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.

**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 can be time consuming. 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.

**App Interaction Engine**

The App Interaction Engine will spawns 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 exploration 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 by our analysis modules. 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 states that developers never anticipated occurring. SMASH can also simulate conditions that developers never anticipated occurring. SMASH can also simulate conditions that developers never anticipated occurring. SMASH can also simulate conditions that developers never anticipated occurring.

**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 other apps that are similar to the target app. When SMASH finds conditions that are likely to require investigation can be quite large, SMASH will also provide rankings of likely to help developers prioritize their time.

Although SMASH cannot detect all forms of problematic app behavior, new analytical models 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 (low battery, high CPU usage), etc. 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.

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 environments, 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 behavior, Caiipa’s default interaction engine is rich and simple and applies a light-weighted exploratory user model, customizable as needed. One limitation of UI interaction engines is that they do not adapt to a developer’s user model, and as such, Caiipa adopts a black-box approach during app testing. However, this allows the system to test any app regardless of the languages used during development.
To cover a wide breadth of mobile conditions, Caipa currently includes a large context library of 10,504 test cases. Caipa 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, Caipa detects 11 times more crashes and 8 times more performance problems.

VanarSena
VanarSena 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 execute 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 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 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.

REFERENCES

CALL FOR CONTRIBUTIONS
GetMobile aims to foster the development and growth of the mobile research community. We encourage researchers and practitioners to submit proposals for articles for future issues. Proposed articles should address topics of broad interest and should be accessible to the wider mobile research community.

WE SEEK CONTRIBUTIONS that fall within the scope of the departments listed in the table below.
GetMobile articles are limited to 3,000 words or less.
To submit a proposal, please email the appropriate contact.
To propose an article that does not fit well within the scope of any of these departments, please send your proposal to getmobile_editor@acm.org.

<table>
<thead>
<tr>
<th>DEPARTMENT NAME</th>
<th>SCOPE</th>
<th>CONTACT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mobile Platforms</td>
<td>Developments in mobile platforms and their use in research.</td>
<td><a href="mailto:getmobile_platforms@acm.org">getmobile_platforms@acm.org</a></td>
</tr>
<tr>
<td>(Almost) Unpublishable Results</td>
<td>Work not usually taken by conferences or journals but valuable to the mobile research community (e.g., experience from deployment and development and effort to reproduce important results).</td>
<td><a href="mailto:getmobile_unpublishable@acm.org">getmobile_unpublishable@acm.org</a></td>
</tr>
<tr>
<td>Education</td>
<td>Developments in mobile and wireless education.</td>
<td><a href="mailto:getmobile_education@acm.org">getmobile_education@acm.org</a></td>
</tr>
<tr>
<td>Experimental Methods</td>
<td>Topics that every mobile researcher should know about before conducting user-centered experiments.</td>
<td><a href="mailto:getmobile_experimental@acm.org">getmobile_experimental@acm.org</a></td>
</tr>
<tr>
<td>Makers</td>
<td>Developments on the growing maker community.</td>
<td><a href="mailto:getmobile_makers@acm.org">getmobile_makers@acm.org</a></td>
</tr>
<tr>
<td>Arm’s Length</td>
<td>Developments in fields that fall outside of the traditional SIGMOBILE core, but which have applications in mobile systems and networks.</td>
<td><a href="mailto:getmobile_armslength@acm.org">getmobile_armslength@acm.org</a></td>
</tr>
<tr>
<td>Standards</td>
<td>Developments at standards bodies, such as the IETF and W3C.</td>
<td><a href="mailto:getmobile_standards@acm.org">getmobile_standards@acm.org</a></td>
</tr>
<tr>
<td>_past* = pull*</td>
<td>Highlights cutting edge pre-publication research, as well as articles that reach back five to ten years and reexamine previous work in mobile systems, to help us better understand where we are now and where we might end up next.</td>
<td><a href="mailto:getmobile_pastfuture@acm.org">getmobile_pastfuture@acm.org</a></td>
</tr>
</tbody>
</table>