Contextual Fuzzing: Automated Mobile App Testing Under Dynamic Device and Environment Conditions

MSR-TR-2013-100 – March 2013

Chieh-Jan Mike Liang[‡], Nicholas D. Lane[‡], Niels Brouwers[*], Li Zhang[⋆], Börje Karlsson[‡], Ranveer Chandra[‡], Feng Zhao[‡]

[‡]Microsoft Research [*]Delft University of Technology
[⋆]University of Science and Technology of China

Abstract

App experience drives healthy mobile ecosystems. However, mobile platforms present unique challenges to developers seeking to provide such experiences: device heterogeneity, wireless network diversity, and unpredictable sensor inputs. We propose *Context Virtualizer* (ConVirt), a cloud-based testing service that addresses two challenges. First, it provides a large set of realistic mobile contextual parameters to developers with emulators. Second, it enables scalable mobile context exploration with app similarity networks. To evaluate the system design, we profile 147 Windows Store mobile apps on our testbed. Results show that we can uncover up to 11 times more crashes than existing testing tools without mobile context. In addition, our app similarity network increases the number of abnormal performances found in a given time by up to 36%, as compared to the current practices.

1 Introduction

Mobile devices, such as smartphones and tablets, are rapidly becoming the primary computing platform of choice. This popularity and usage is fueling a thriving global mobile app eco-system. Hundreds of new apps are released daily, e.g. about 300 new apps appear on Apple's App Store each day [10]. In turn, 750 million Android and iOS apps are downloaded each week from 190 different countries [8]. However, success of this global market for mobile apps is presenting new demanding app testing scenarios that developers are struggling to satisfy.

Each newly released app must cope with an enormous diversity in device- and environment-based operating contexts. The app is expected to work across differently sized devices, with different form factors and screen sizes, in different countries, across a multitude of carriers and networking technologies. Developers must test their application across a full range of mobile operating contexts prior to an app release to ensure a high quality user experience (Section 2). This challenge is made worse by low consumer tolerance for buggy apps. In a recent study, only 16% of smartphone users reported they continue to use an app if it crashes twice soon after download [28]. As a result, downloaded app attrition is very high – one quarter of all downloaded apps are used just once and then eventually deleted [27]. Mobile users only provide a small opportunity for an app to demonstrate its worth, performance and crashes are not tolerated.

Today, developers only have a limited set of tools to test their apps under different mobile context. Tools for collecting and analyzing data logs from already deployed apps (e.g., [9, 5, 2]) require the app to be first released before problems can be corrected. Through limited-scale field tests (e.g., small beta releases and internal dogfooding) log analytics can be applied prior to public release but these tests lack broad coverage. Testers and their personal conditions are likely not representative of a public (particularly global) app release. One could use mobile platform simulators ([14, 20]) to test the app under a specific GPS location and network type, such as Wi-Fi, but in addition to being limited in the contexts they support, these simulators do not provide a way to explore all possible combinations of mobile contexts.

To address this challenge of testing the app under different contexts, we propose *contextual fuzzing* – a technique where mobile apps are monitored while being exercised within a host environment that can be programmatically perturbed to emulate key forms of device and environment context. By systematically perturbing the host environment an unmodified version of the mobile app can be tested, highlighting potential problems before the app is released to the public. To demonstrate the power of contextual fuzzing, we design and implement a first-of-its-kind cloud service *Context Virtualizer* (ConVirt) – a prototype cloud service that can automatically probe mobile apps in search of performance issues and hard crash scenarios caused by certain mobile contexts. Developers are able to test their apps by simply providing an app binary to the Context Virtualizer service. A summary report is generated by Context Virtualizer for the developer that details the potential problems observed and which conditions appear to act as a trigger. Context Virtualizer utilizes primarily cloud-based emulators but also integrates actual real devices to provide coverage of hardware-specific contexts that are difficult to otherwise emulate.

Key challenge when implementing Context Virtualizer is the scalability of such a service, across the range of scenarios and number of apps. Individual tests of context can easily run into the thousands when a comprehensive set of location inputs, hardware variations, network carriers and common memory and cpu availability levels. In our own prototype a library of 1,254 contexts is available and sourced largely from trace logs of conditions encountered by real mobile users (see §5.2.) Not only must conditions be tested in isolation, but how an app responds to a *combination* of conditions must also be considered (e.g., a low memory scenario occurring simultaneously during a network connection hand-off.)

We propose a new technique that avoids a brute force computation across all contexts. It incrementally learns which conditions (e.g., high-latency connections) are likely to cause problematic app behavior for detected app characteristics (e.g., streaming apps). Similarly, this approach identifies conditions likely to be redundant for certain apps (e.g., network related conditions for apps found to seldom use network conditions). Through this intelligent prioritization of context perturbation unexpected app problems and even crashes can be found much more quickly than is otherwise possible.

In this work we make the following contributions.

1. We show the need for additional testing framework for mobile apps, which extends beyond the traditionally available code testing tools. (§ 2)
2. We present a new concept, of *contextual fuzzing*, for systematically exploring a range of mobile contexts. We also describe techniques to make this approach scalable using a learning algorithm that effectively prioritizes contexts for a given run. (§ 3).
3. We propose a new, one of its kind cloud service, that consists of a hybrid testbed consisting of real devices and emulators, along with an energy testing framework, to which app developers can submit apps for contextual testing. (§ 5).

We evaluate our system using a workload of a representative 147 mobile Windows Store apps. Our experiments (1) validate the accuracy of all contexts simulated and measurements performed within our prototype; and, (2) examine the scalability through test time and resource efficiency relative to a collection of representative benchmark algorithms. We demonstrate the benefits to developers of contextual fuzzing by automatically identifying a range of context-related hard crashes and performance problems within our app workload. We describe both aggregate statistics of problems identified and in some cases supplement these with individual app example case studies. Finally, we discuss a series of generalizable observations that detail overlooked contexts that result in these issues and how they might be addressed.

2 The Mobile Context Test Space

We now present three different mobile contexts, and the variations therein, which we believe captures the majority of context-related bugs in mobile apps. To the best of our knowledge there doesn't exist ways to principally test all the variations of these contexts.

Wireless Network Conditions. Variation in network conditions leads to different *latency, jitter, loss, throughput and energy consumption*, which in turn impacts the performance of many network-facing apps (43% in the Google Play). These variations could be caused by the operator, signal strength, technology in use (e.g. Wi-Fi vs. LTE), mobile handoffs, vertical handoffs from cellular to Wi-Fi, and the country of operation. For example the RTTs to the same end-host can vary by 200% based the cellular operator used [17], even given identical locations and hardware, the bandwidth speeds between countries frequently can vary between 1 Mbps and 50 Mbps [33], and the signal strength variation changes the energy usage of the mobile device [22].

Device Heterogeneity. Variation in devices require an app to perform across different *chipsets, memory, cpu, screen size, resolution, and the availability of resources (e.g. NFC, powerful GPU, etc.).* This device heterogeneity is very severe. 3,997 different models of Android devices – with more than 250 screen resolution sizes – contributed data to OpenSignal database during a recent six month period [24]. We note that devices in the wild can experience low memory states or patterns of low CPU availability different from the expectation of developers, e.g. a camera temporarily needs more memory, and this interaction can affect user experience on a low-end device.

Sensor Input. Apps need to work across *availability of sensors, their inputs, and variations in sensor readings themselves.* For example, a GPS or compass might not work at a location, such as a shielded indoor building, thereby affecting end user experience. Furthermore, depending on the location or direction, the apps response might be different. Apps might sometimes cause these sensors to consume more energy, for example, by polling frequently for a GPS lock when the reception is poor. The sensors also sometimes have jitter in their readings, which an app needs to handle.

3 Context Virtualizer Design

In the following section we describe the design considerations and the overall architecture of Context Virtualizer.

3.1 Design Challenges

The goal of Context Virtualizer (ConVirt) is to address the mobile app testing needs of two specific types of end-users:

- **App developers** who use ConVirt 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.

Before we can build an automated app testing service able to examine a comprehensive range of mobile contexts we must solve two fundamental system challenges:

Challenge 1. High-fidelity Mobile Context Emulation. Cloud-based emulation of realistic mobile contexts enables more complete *pre-release* testing of apps. Developers can then incorporate such testing into their development cycle. Similarly, distributors can be more confident of the user experience consumers will receive. A viable solution to the emulation problem will have two characteristics. First, it must be comprehensive and be capable of emulating various key context dimensions (viz. network, sensor, hardware) in addition to the key tests within each dimension (e.g., wireless handoffs under networking). Second, it must be an accurate enough emulation of the real-world phenomena such that app problematic behavior still manifest.

Challenge 2. Scalable Mobile Context Exploration. As detailed in §2, the number of mobile contexts that may impact a mobile app is vast – for example, consider just the thousands of device types and hundreds of mobile carriers in daily use today. Because of the complexity of the real-world developers and distributors are unable to correctly identify which conditions are especially hazardous, or even relevant, to an app. Instead automated exploration is required. Ideally, not only are contexts tested in isolation, but in *combination*, resulting in a combinatorial explosion that makes brute-force testing infeasible.

To solve these challenges we propose the following solutions that are embedded within the architecture and implementation of Context Virtualizer.

Contextual Fuzzing. To address Challenge 1, we propose a hybrid cloud service architecture that blends conventional servers and real mobile devices. Servers running virtual machines images of mobile devices provide our solution with scalability. A *perturbation layer* built into both server VMs and mobile devices allows a set of context types to be emulated (for example, networking conditions). Under this design, certain tests cases that are logic-based or that rely on context that be effectively emulated are executed on server VMs. Similarly, test cases that require hardware specific conditionsare performed directly on a mobile device.

App Similarity Network. Our solution to Challenge 2 relies on constructing a similarity network between a population of apps. Under this network, apps are represented as network nodes that are connected by weighted edges. Edge weights capture correlated behavior (e.g., common patterns of resource usage under the same context). By projecting a new previously unseen app into the network we can identify app cliques that are likely to respond to mobile contexts in similar ways. Using the network, mobile contexts previously observed to cause problems in apps similar to the app under test can be selected. Similarly, those contexts that turned out to be redundant can be avoided. One strength of this approach is that it allows information previously discovered about apps to benefit subsequently tested apps. The more apps to which the network is exposed, the better it is able to select context test cases for new apps to be tested.

Not only do these solutions enable Context Virtualizer to meet its design goals, but we believe these are generalizable techniques able to be applied to other mobile system challenges that we leave to be explored in future work (see §7 for further discussion.)

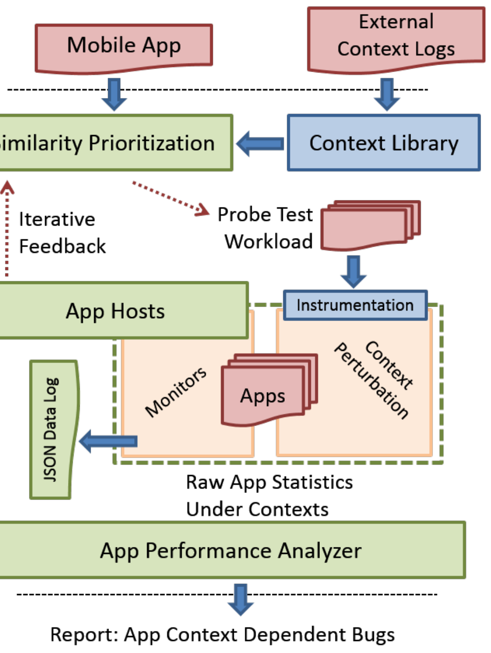


Figure 1. Context Virtualizer Architecture

3.2 Architectural Overview

As shown in Figure 1, Context Virtualizer consists of four principle components: (1) Context Library (CL); (2) App Similarity Prioritization (ASP); (3) App Environment Host (AEH); and, (4) App Performance Analyzer (APA). We now describe in turn each component along with component-specific design decisions.

Context Library. Stored in CL are definitions of various mobile context scenarios and conditions. One example scenario is a network handoff occurring from Wi-Fi to 3G. An example of a stored condition are the network characteristics for a specific combination of location and cellular provider. Every context relates to only a single system dimension (e.g., network, cpu); this enables contexts to be composable – for instance, a networking condition can be run in combination with a test limiting memory. CL is populated using datasets collected from real devices (e.g., OpenSignal [23]) in addition to challenging mobile contexts defined by domain experts.

Through our design of CL that packages together context parameters we (1) reduce the overall test space to search while also (2) restricting tests that are executed to particular combinations of parameters that actually occur in the wild (e.g., a cellular provider and a specific location.) If instead an unstructured parameter search within each mobile context dimension was performed it is unclear how valid parameter combinations could be enforced while the search scalability challenges would be even worse. Similarly, our decision to integrate external data sources directly into CL enables it to better reflect reality and to even adapt to changes in the mobile market (e.g., new networking parameters.) Without this integration CL would be limited to only the mobile context that users anticipate.

App Similarity Prioritization. Tested apps are exposed to mobile contexts selected from CL. ASP determines which order these contexts are to be performed and then assigns each test to one of a pool of AEHs. Aggregate app behavior (i.e., crashes, and resource use), collected by AEHs, is used by ASP to build and maintain an app similarity network that determines this prioritization. Both prioritization and similarity network computation is an online process. As each new set of results is reported by an AEH more information is gained about the app being tested, resulting potentially in re-prioritization based on new behavior similarities between apps being discovered. Through prioritization two outcomes occur: (1) redundant or irrelevant contexts are ignored (e.g., an app is discovered to never use the network, so network contexts are not used); and, (2) those contexts that negatively impacted apps similar to the one being tested are performed first (e.g., an app that behaves similarly to other streaming apps has a series of network contexts prioritized).

Our design decision to automate testing with a systematic search of mobile context enables any inexperienced user successfully use Context Virtualizer. An alternative design would require users to specify a list of mobile contexts to test their app against – however, the ability of the user to determine such a list would largely determine the quality of results.

Instead of adopting general purpose state space reduction algorithms we instead develop a novel domain-specific approach. Mobile apps have higher levels of similarity between each other (compared to desktop apps) because of factors including (1) strong SDK-based templates, and (2) a high degree of functional repetition (e.g., networking functions).

App Environment Host. An AEH provides an environment for a tested app to run while providing the required context requested by the PSP. App logic and functionality are exercised assuming a particular usage scenario and encoded with a Weighted User Interaction Model (WUIM, see §5.1) provided by users (i.e., distributors or developers) when testing is initiated. In the absence of a user provided WUIM a generic default model is adopted. Specific contexts are produced within the AEH using using perturbation modules (see Figure 1, central component) that realistically simulate a specific context (e.g., GPS reporting a programmatically defined location). However, certain contexts can not be adequately simulated – such as, particular hardware – and in these cases the app is tested on being performed on the same, or equivalent, hardware. During testing AEH uses monitor modules to closely record app behavior. Monitors primarily record app resource usage (e.g., memory) but they also record app state, such as an app crashes.

The design choice of a mixed architecture is motivated by the competing goals of scalability and realism. An alternative approach using only real hardware will struggle to scale. Furthermore it is unnecessary as many tests that are related, for instance, to app logic do not strictly require real hardware. Similarly, an approach purely using VM emulators will have limited realism. Certain context dimensions can never be performed with such as design. In contrast, our design is extensible to include improved context support (e.g., additional sensors) within our proposed architecture.

App Performance Analyzer. Collectively, all AEHs produce a large pool of app behavior (i.e., statistics and recorded crashes) collected under various contexts. APA is designed to extract from these observations unusual app behavior that warrants further investigation by developers or distributors. It relies on anomaly detection that assumed norms based on previous behavior of (1) the app being tested and (2) a group of apps that are similar to the target. As shown in Fig. 1, APA collates its findings into a report made available to Context Virtualizer users.

Our design of APA demonstrates the larger potential for our App Similarity Networks approach. In this component we leverage the similarity network used for prioritization as a means to identify accurate expectations of performance for a tested app.

4 App Similarity Prioritization

In the following section, we describe (1) the similarity network used to guide context test case prioritization, and (2) the prioritization algorithm itself.

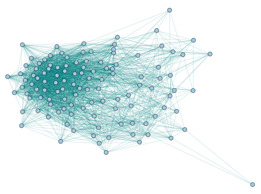
This discussion makes use of two terms. First, the term "target app" refers to a representative app for which context test cases are being prioritized. Second, the term "resource measurement" is one of the hundreds of individual measurements of system resources – related to networking, CPU, memory, etc. (see §5.1) – made while Context Virtualizer tests an app.

4.1 App Similarity Network

A weighted fully connected graph is used by ASP to capture inter-app similarity. We now describe the network and its construction.

Nodes in the network represent not only apps, but a pairing of an app and a specific WUIM (Weighted User Interaction Model). A WUIM describes a particular app usage scenario in terms of a series of user interaction events. An example scenario is where a user listens to music on a radio and periodically changes radio station. Since app behavior (e.g., resource usage) can differ significantly from scenario to scenario, apps must be paired with a specific WUIM when modeled in the similarity network. Throughout this section, while we refer to an "app" being represented in the network, in all cases this more precisely refers to a app and WUIM pair.

Edges in the network between two nodes are weighted according to the similarity between two apps with respect to a specific resource measurement (e.g., memory usage). We calculate similarity by computing the correlation coefficient (i.e., Pearson's Correlation Coefficient [3]) between pairs of identical resource measurements observed under identical context tests for two apps. This estimates the strength of the


(a) TCP packets received

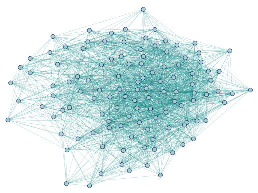

(b) Number of disk reads

Figure 2. Similarity graphs depict the degree of correlation in consuming two resources when exercising 147 app under eight networking contexts.

linear dependence between the two sets of variables. While the correlation coefficient reports a signed value depending on if the relationship is negative or positive, we use the absolute value for the edge weight.

Bootstrap Scenario. Initially, there is no existing resource measurement regarding the target app, with which similarity to other apps can be computed. From our experiments, we find the most effective (i.e., leading to higher prioritization performance) initial set of three context tests are three wireless networking profiles (c.f. §5.2): `GPRS`, `802.11b`, and `cable`. As more and more resource measurements are made for the target app, the similarity networks improve (and largely stabilize).

Per-Measurement Networks. We utilize multiple similarity networks, one for each type of resource measurement (e.g., allocated memory and CPU utilization). Figure 2 presents similarity graphs for two different resource metrics, TCP packets received and disk reads for a population of 147 Microsoft Windows Store apps (detailed in §6). Each edge weight shown in both figures (equal to correlation coefficient) is larger than 0.7. We note that the graph, even with this restriction applied, is well connected and has high average node degree. Furthermore, both graphs are clearly different and support our decision to use per-measurement networks instead of a single one.

4.2 App Similarity-based Prioritization

ASP prioritization is an iterative four-step process that repeatedly occurs while the target app is tested by Context Virtualizer. These steps are: (1) cluster, (2) predict, (3) rank, followed by (4) update.

Cluster. At the start of each iteration, an app cluster is selected for the target app within each app similarity network (one for each resource measurement type). The app cluster is based on network edge weights. Experimentally, we find an edge weight with a threshold of 0.70 is effective during this process. All nodes with an edge weight greater or equal to this threshold are assigned to be a member of the cluster.

Predict. A prediction for the target app is made for each resource measurement within each pending (i.e., yet to be performed) context test. Predictions are made using a simple linear model trained for each member of the app cluster. These models predict the unseen resource measurement for a particular context test based on (1) the value for this measurement for the app cluster member, and (2) the relationship between the target app's measurements and the app cluster members under all previously computed context tests.

For those tests when a member of the app cluster crashed a prediction can not be made. Instead their prediction is replaced with a crash flag. Crash flags indicate a similar app to the target app crashed under the context test attempting to be predicted. When this context test is ranked against other tests, the number of crash flags boost its ranking.

Rank. Based on predicted resource estimates and crash flags, a ranking is given for all context test cases yet to to be executed for the target app. The priority of test cases is determined on this ranking, which is computed as follows. First, the variance is calculated for each resource measurement within each test case (excluding those with crash flags). Variance provides a notion of prediction agreement between each estimate. The intuition behind this decision is that tightly clustered (i.e., low variance) estimates indicate higher uncertainty regarding the estimate. We want to rank test cases high when they have large uncertainty across their resource measurements. Second, the variance for each resource measurement is compared with all other resource measurements of the same type. A percentile is assigned based on this comparison. The average percentile for each resource measurement within every context test case is then computed. Third, crash flags for all resource measurements are counted within each test case. We apply a simple scaling function to the number of crashes that balances the importance of many crashes against a high percentile. Finally, scaled crash flag count is added to the percentile to compute a final rank.

Update. The final phase in prioritization is to revise the app similarity network based on new resource measurement collected about the target app. These measurements may have altered the edge weights between the target app and the other apps in the network.

The new rankings determined at this iteration are utilized as soon as an AEH completes its previously assigned context test. Once the head of context test ranking has been assigned to an AEH then it is no longer considered when ranking occurs at the next iteration.

5 Implementation

This section details our current implementation of the framework components (see Figure 1).

5.1 App Environment Host

AEH can run in virtual machines (VM) for testbed scalability, or on real Intel x86 mobile devices (e.g., Samsung 7 Slate [26]) for hardware fidelity. AEH runs Windows 8 OS with our own monitoring and perturbation modules.

Monitoring Modules. AEH logs a set of system statistics of some running apps, as specified by the *process matching rules* in a configuration file. In its simplest form, a process matching rule would be a process name (i.e., 'iexplore.exe'). If AEH detects a new process during the periodic polling of running process list, it monitors the process if there is a rule match.

We note the caveat that HTML5/JS-flavored Windows Store Apps [1] run inside a host application called `WWAEHost.exe`. Therefore, AEH takes the hint from the working directory argument of the command line that starts the `WWAEHost.exe` process.

To accommodate the variable number of logging sources (e.g., system built-in services and customized loggers), AEH implements a plug-in architecture where each source is wrapped by a *monitor*. Monitors can either be time-driven (i.e., logging at fixed intervals), or event-driven (i.e., logging as an event of interest occurs). Logging can take place on both system-wide or per-app fashion. The final log file is in JSON format.

We have implemented several monitors. Two separate monitors track system-wide and per-app performance counters (e.g., network traffic, disk I/Os, CPU and memory usage), respectively. Specifically, the per-app monitor records kernel events to back-trace a resource usage to the process. The third monitor hooks to the Event Tracing for Windows (ETW) service to capture the output of `msWriteProfilerMark` JavaScript method in Internet Explorer. This method allows us to write debug data from HTML5/JS Windows Store apps. Finally, we have a per-app energy consumption estimation monitor based on JouleMeter [22].

Perturbation Modules. We built the network perturbation module ontop of the Network Emulator for Windows Toolkit (NEWT), a kernel-space network driver. NEWT exposes four main network properties: download/upload bandwidth, latency, loss rate (and model), and jitter. We introduced necessary enhancements such as the real-time network property update to emulate network transitions and cellular tower handoffs. Another crucial enhancement is the per-app filtering mechanism for perturbation without impact on on other running processes.

Next, we describe two separate approaches to adjust the CPU clock speed. First, modern VM managers (e.g. VMWare's vSphere) expose settings for CPU resource allocation on a per-instance basis. Second, most modern Intel processors support EIST that allows setting different performance states. By manipulating the processor settings, we can make use of three distinct CPU states: 20%, 50%, and 100%.

To control the amount of available memory to apps, we use a tool called Testlimit from Microsoft, which includes command-line options that allows AEH to change the amount of system commit memory. AEH then uses this tool to create three levels of available memory for the apps to execute.

[1] formerly referred to as Metro apps

Finally, we implemented a virtual GPS driver to feed apps with spoofed coordinates, as Windows 8 assigns the highest priority to the GPS driver. AEH instructs the virtual driver through the UDP socket to avoid the overhead associated with polling. Upon receiving a valid UDP command, our virtual GPS driver signs a state-updated event and a data-updated event to trigger the Windows Location Service to refresh the geolocation data.

These CPU and memory perturbations can be used not only for changing a given test context, but also as a way to emulate more limited hardware, if necessary.

Weighted User Interaction Model. User inputs (e.g., navigation and data) can significantly impact the app behavior and resource consumption. For example, a news app containing videos will consume different resources depending on how many (and which) videos are viewed.

Our goal is to exercise apps under common usage scenarios with little guidance from developers. We represent app usage as a tree. Tree nodes represent the pages (or app states), and tree edges represent the UI element being invoked. Then, invoking UI elements (e.g., clicking buttons) essentially traverses the UI tree.

Each usage scenario is a tree with weighted edges based on the likelihood of a user to select a particular UI element on a page. We generate a usage tree for each scenario with a stand alone authoring tool, which allows the user to assign a weight to each UI element. Higher weights indicate a higher probability of a particular UI element being invoked.

5.2 Context Library

Our complete Context Library includes 1,254 mobile context tests, the bulk of these being network and location related. CL currently uses Open Signal [23] as an external data source. This public dataset is comprised of user contributions of wireless networking measurements collected around the world. It provides CL with data from more than 800 cellular networks and one billion WiFi points readings.

From datasets as large as Open Signal each data point could be extracted into one (or more) context tests. However, this would be intractable during testing but more importantly data is often redundant. Instead we extract aggregates – bucketization – that summarize classes of data points. In the case of Open Signal we aggregate data representing carriers at specific cities. Aggregation is performed by computing the centroid of all data points when grouped by carrier and city. We limit cities to 400 selected based on the volume of data collected at Open Signal. 50 carriers are also included. Although we do not maintain the same amount of data for each carrier.

Additional examples of CL tests are networking, memory and cpu events. These are a variety of hand coded scenarios replicating specific events typically challenging to apps. For example, sudden drops in memory and fluctuations in cpu availability. Device profiles are primarily combinations of memory and CPU to represent classes of device with certain levels of resources. Device profiles also include specific real devices available within the ConVirt testbed.

5.3 Interfacing with App Environment Hosts

AEHs are controlled via PowerShell scripts, a Windows shell environment. Each test script specifies the system conditions to emulate and the system metrics to log. To scale up the testbed to hundreds of machines, we set up a client-server infrastructure where a central server (that implements the ASP algorithm) configures each AEH node via Windows Management Instrumentation (WMI) technology, an RPC-like framework. The tight integration between WMI and PowerShell simplifies the process of pushing test scripts from the central server. In turn, a service running on the AEH node accepts the script and initiates its execution. Finally, after a test script is executed, the output log is copied to a designated network share. Then, the central server cleans up the data and writes to a MSSQL database.

5.4 App Performance Analyzer

APA is responsible for (1) highlighting the system metric measurements that are unexpected under certain mobile contexts; and (2) providing actionable reports that help users interpret the raw measurements.

To judge whether a system metric measurement is within expectation, APA applies the common anomaly detection algorithm: if the data is more than some standard deviations away from the group mean, then it is flagged. APA defines two notions of group. First, APA compares multiple iterations of the same app being tested under the same context. Second, APA compares an app with a comparison group of apps, which can include a broad app population or simply a set of apps considered to be similar in resource consumption. APA reuses the same similarity equation described in §4.1.

APA copes with false positives by a 'novelty' weight, which is proportional to the number of times the same combination of emulated context and statistic is confirmed as norm by developers of similar apps.

Finally, the reporting interface of APA can present data analysis in several ways. Figure 3 shows a radar plot that highlights the system metrics that developers should focus on. These are the system metrics that have a high probability of being outliers. In addition, identified outliers can be ranked as to how severe they are. Ranking considers three factors: (1) the frequency at which the outlier occurs; (2) the difference to the comparison 'norm'; (3) the relative importance of each metric. A variety of report templates that provide actionable items for developers can be generated on demand based on crash and outlier data.

6 Evaluation

This section is organized by the following major results: (1) our system can emulate device and network conditions with high fidelity; (2) bucketization can reduce the continuous geolocation parameter space of a city to one single point, at the expense of measurement error rate between 7% and 30%; (3) GCF can find up to 36% more outliers than the current industry practices, with the same amount of time and computing resources; (4) contextual fuzzing increases the number of crashes and performance outliers found over the current practice by a factor of 11 and 8, respectively; and (5) we share lessons learned to help developers improve their apps.

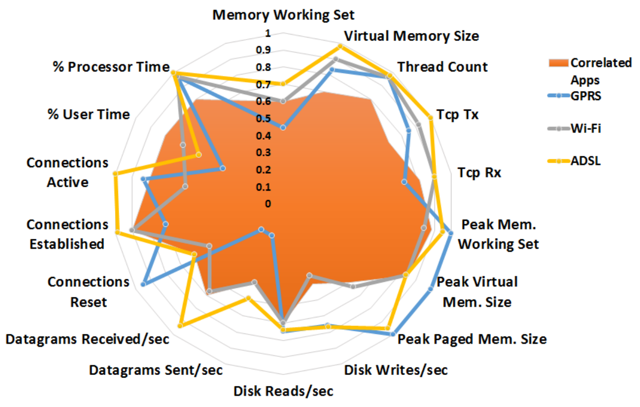


Figure 3. Radar plot of statistics for an app under different network conditions. The filled area represents the set of similar apps.

	Latency		Bandwidth	
	(ms)	Error (%)	(MBps)	Error (%)
ADSL	373.38	12.03	544.98	0.96
Cable	338.18	12.34	641.18	2.65
802.11g #1	118.60	14.05	5000.37	4.84
802.11g #2	119.90	1.42	4000.06	0.99
802.11b #1	459.40	14.89	260.04	2.56
802.11b #2	367.49	12.11	567.77	3.86
KPN 2G	508.02	0.27	25.48	4.87
KPN 3G	166.05	1.07	99.05	1.81
China Mobile 3G	674.87	9.25	71.93	0.72

Table 1. Emulated network performance numbers and error for various network types.

6.1 Micro-benchmarks

This sub-section presents experiments to verify the accuracy of our simulated network conditions and quantify the system overhead when probing apps; to ensure that measurements are not adversely affected by the system components.

Network Emulation Validation. To assess network emulation fidelity, we first configure NEWT parameters to match bandwidth, latency, jitter and packet loss from network traces collected under different real world settings. Then, network performance is measured under NEWT.

Real world network traces were collected in Amsterdam and Beijing, as listed in Table 1. Our benchmark client is a laptop with a fast Ethernet connection, 802.11b/g radio, and a USB-based cellular dongle. The backend server is a VM on Amazon's EC2. We measured the network bandwidth from the laptop to the backend with iperf and latency by sending 100 ICMP pings.

The emulated testbed consists of two laptops connected via a 100-Mbps switch. As our LAN outperforms all the

	CPU (%) *App*	CPU (%) *FW*	RAM (MB)
Spotify	0.37		107.14
Spotify (+FW)	0.51	1.04	105.46
Youtube	5.73		178.91
Youtube (+FW)	5.72	1.09	176.12
MetroTwit	0.52		150.73
MetroTwit (+FW)	0.52	0.83	160.03
Hydro Thunder Hurricane	40.71		170.20
HTH (+FW)	42.42	1.04	169.82

Table 2. App performance statistics with the framework enabled and disabled.

emulated network types, it does not introduce any artifact to the emulation results. One laptop acts as the server, and the other repeats the same iperf and ping tests while cycling through all nine networks.

Table 1 shows the emulated network bandwidth and latency as observed. NEWT can throttle the bandwidth well, with an error rate between 0.72% and 4.87%. On the other hand, latency has an error rate of 18.05%. Fortunately, since the error rate is relatively constant between iterations, we can compensate by adjusting the latency setting accordingly.

System Overhead Analysis. To ensure that our measurement techniques do not skew the results, we measured the overhead introduced by our system. We selected four apps, as listed in Table 2, for benchmark. This app selection covers static web content access, streaming media (audio and video), social networks, and CPU/GPU intensive game. Each app ran two one-hour sessions; only one of which had the framework enabled. Each session of the same app followed the same user interaction model. In addition, for sessions with the framework enabled, we instructed NEWT to limit bandwidth to 1 Gbps. In this way, we ensure that NEWT is actively monitoring TCP connections, without disturbing the available bandwidth.

We used Perfmon, a standard Windows tool, to track global and per-process performance counters. Table 2 shows that the framework has a very low resource overhead, and the framework does not significantly impact the app behavior. Specifically, the framework uses an average of 1% CPU time, and the difference in memory usage is less than 2% in most cases.

6.2 Technique Validations

This section evaluates the scalability of our system in terms of parameter space searching. We start by describing the four datasets used for evaluation.

6.2.1 Evaluation Datasets

We collected four datasets from testing a total of 147 Windows 8 Store apps. Individual test cases lasted five minutes, and they were repeated on four machines: two desktops, one laptop, and one slate. All apps followed a random user interaction model, and all datasets are categorized according to their target scenarios.

Dataset #1 aims for 8 common network conditions that mobile devices may encounter: WCDMA, 802.11b, GPRS, GPRS (out of range), GPRS (handoff), ADSL, and Cable internet. *Dataset #2* tests how apps behave under common network transitions: <802.11b, WCDMA>, <802.11b, GPRS>, <GPRS, GPRS (out of range)>, and <GPRS, GPRS (tower handoff)>. *Dataset #3* emulates a WCDMA network at 10 locations with large mobile device populations: Beijing, Berlin, Jakarta, Moscow, New Delhi, Seattle, Seoul, Teheran, Tokyo, Washington D.C. *Dataset #4* emulates a WCDMA network at 12 locations uniformly spread out in Seattle.

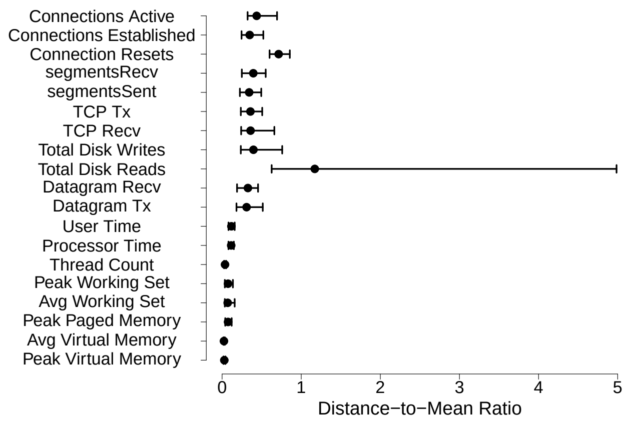


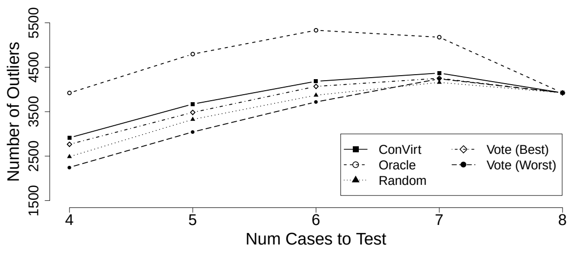
Figure 4. Per-metric variation of WCDMA (T-Mobile) emulation with traces from 12 locations in Seattle.

6.2.2 Test Case Bucketization

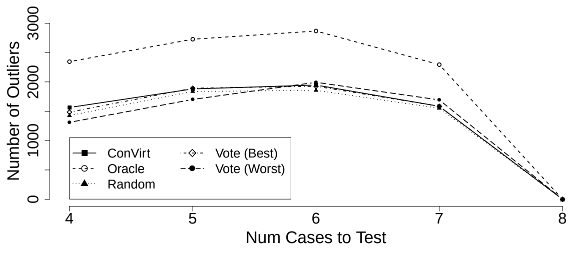
Dataset #4 emulated real life WCDMA performance of 12 uniformly random locations in the Seattle area. The OpenSignal data suggests that the variations in bandwidth and loss rate are relatively low, as compared to latency. The standard deviation for upstream and downstream bandwidth, loss rate, and latency are 0.20, 0.48, 0.01, and 39.75, respectively. We next quantify the impact of these variations on bucketization.

Figure 4 shows the trade off of bucketization. Suppose we pick one point to represent the 12 test points in Seattle, we effectively reduce the number of profiles by a factor of 12. However, doing so introduces error into measurements. Figure 4 shows the distance-to-mean ratio for each metric across all apps. If we exclude the total disk reads (with the highest variance), 85% of the metrics have an average distance-to-mean ratio less than 30%. In other words, if we use a single point to represent the WCDMA condition in Seattle, most measurement errors are between 7% and 30%. However, the speed up (by a factor of 12) might outweigh the error in this case.

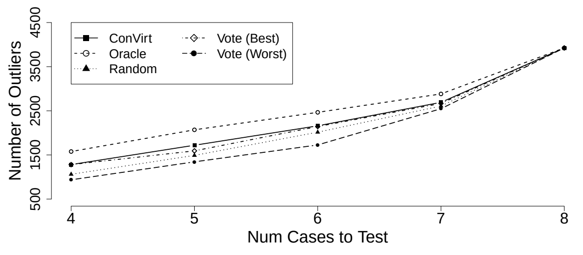
Figure 4 also shows that the network and disk metrics observe a relative high variance, especially for the app categories of sports and travel. This result suggests that the granularity of the buckets should adapt to each app.



(a) All outliers found



(b) False positives



(c) True outliers

Figure 5. The number of outliers found for all 147 app packages in dataset #1.

6.2.3 *App Similarity Prioritization*

The evaluation uses two metrics: number and relevance of outliers found as **(1)** the time budget varies, and **(2)** the amount of available computing resource varies.

We picked three comparison baselines to represent the absolute upper-bound and common approaches. First, Oracle has the complete knowledge of the measurements for all apps (including untested ones), and it represents the upper-bound. Vote picks the most popular test case during each step from all previously tested apps. In addition, as a single ordering is not optimal for all apps, we coin the term ”vote-best” and ”vote-worst” for its upper and lower bound. Finally, Random randomly picks an untested case to run at each step.

Assessment of Outliers Found. We start by studying the question: given sufficient time to exercise $AppPkg_{test}$ under x test cases, what should those x cases be to maximize the number of reported problems. The assumption is that a single test case runs for a fixed amount of time (e.g., five minutes). Figure 5 and Figure 6 illustrate the results from dataset #1 and #2, respectively. Next, we use the former to highlight three observations.

First, the total number of outliers reported by GCF is better than the non- oracle baselines, and only the best cases of Vote can match this number. Second, we estimate the false positives by assuming the complete dataset (of all eight cases) as the ground truth. False positives are inevitable for all approaches because only estimations are possible without complete measurements. In addition, while GCF reports slightly more false positives initially, this count drops faster as the number of cases increases. Finally, considering both results, Figure 5(c) suggests that GCF can report more true outliers than the non-oracle baselines. Even with only eight profiles, it can find up to 21%, 8%, and 36% more true outliers than Random, Vote-Best and Vote-Worst, respectively. The difference between GCF and Oracle is due to the non-zero error rate of the information gain prediction, as we quantify next.

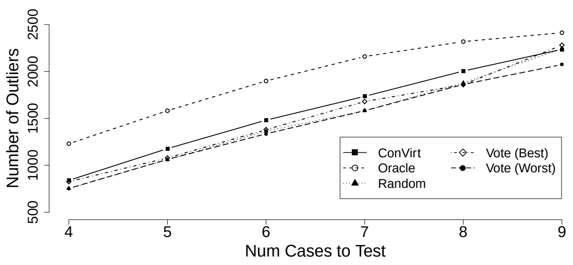


Figure 6. The number of true outliers found for all 147 app packages in dataset #2.

We quantify two sources of the prediction error by calculating the fit score relative to the oracle, or the percentage of matching profiles in both sets. First, all apps start with a fixed bootstrapping set, which has a fit score of 48%. We note that this fit is better than the non-oracle approaches (e.g., 32% for Vote). Second, GCF's set selection accuracy for 4, 5, 6, 7, 8 test cases are 53.57%, 66.94%, 75.85%, 87.95% and 100%, respectively. The accuracy increases with the number of measurements, only the bast case of Vote comes near this result.

Finally, we note that the degree of gain from prioritization is proportional to the degree of parameter variations between test cases. For example, compared to dataset #1 (network profiles of different physical mediums), dataset #2 (WCDMA cellular profiles from different countries) has a less variation. In addition, Figure 6 suggests that Context Virtualizer finds only up to 12%, 7%, and 11% more true outliers than Random, Vote-Best and Vote-Worst, respectively.

Resource Requirements. An important observation is that, since app testing is highly parallelizable, multiple apps can be exercised at the same time on different machines. On the extreme of infinite amount of computing resources, all prioritization techniques would perform equally well, and the entire dataset can finish in one test-case time. Given this is not practical in the real world, we measure the speed up that GCF offers under various amounts of available computing resources.

Figure 7 illustrates combinations of computing resources and time required to find at least 10 potential problems in each of the 147 apps in dataset #2. First, the figure shows

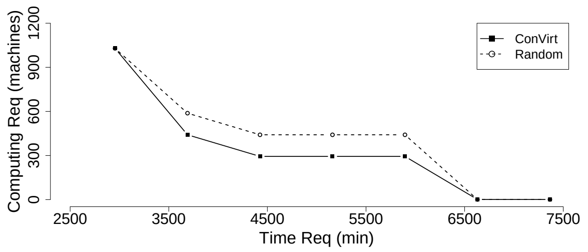


Figure 7. Feasible combinations of time and computing budget to find at least 10 potential problems in each of the 147 apps in dataset #2.

increasing the resource in one dimension can lower the requirement on the other. Second, by estimating the information gain of each pending test case, Context Virtualizer can reach the goal faster and with fewer machines. For example, given 4425 minutes of time budget, ConVirt needs 294 machines – 33% less. Finally, the break-even point for Random is at the time budget of 6630 minutes, or $>$ 90% of the total possible time for testing all combinations of apps and test cases.

6.3 Aggregate App Context Testing Analysis

In this section, we (1) investigate crashes and performance outliers identified via contextual fuzzing, and (2) examine the issues we have found from a set of publicly available apps that presumably had already been tested.

For these experiments, we exercised the same 147 Windows Store apps on test cases of dataset #1, as described in §6.2. Individual apps were tested with four 5-min rounds under both ConVirt and the context- free baseline solution described below. Since the setup is identical for each run of the same app, differences in crashes and performance outliers detected are due to our inclusion of contextual fuzzing.

Comparison Baseline. We use a conventional UI automation based approach as the comparison baseline. The baseline represents current common practice of testing mobile apps. We use the default WUIM that randomly explores the user interface, which is functionally equivalent to the Android Exerciser Monkey [13].

Summary of Findings. Overall, Context Virtualizer is able to discover significantly more crashes (11×) and performance outliers (11×) than the baseline solution. Furthermore, with Context Virtualizer, 75 out of the 147 apps tested observed a total of 1,170 crash incidents and 4,589 performance outliers. This result is surprising, as these production apps should have been thoroughly tested by developers.

Findings by Categories. Figure 8 and Figure 9 show the number of crashes and performance outliers categorized by app source code type and targeted hardware architecture: (1) HTML-based vs. compiled managed code, and (2) x64 vs. architecture-neutral. The observation is that Context Virtualizer is able to identify significantly more potential app problems across both categories. For example, in both categories, this difference in the number of outliers found is a factor of approximately 8 times.

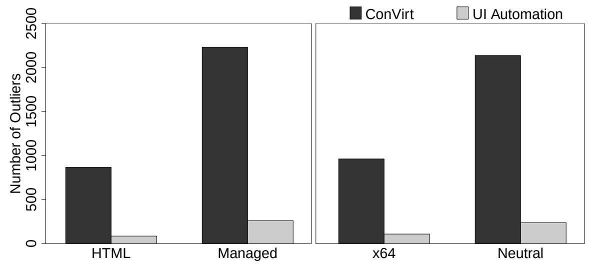


Figure 8. App performance outliers categorized by app source code type and targeted hardware architecture.

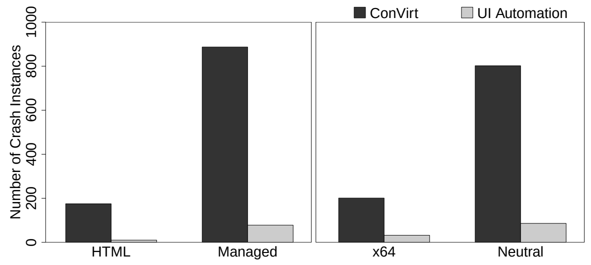


Figure 9. Crashes categorized by app source code type and targeted hardware architecture.

Table 3 categorizes apps in the same way as the Windows Store. It shows that media-heavy apps (e.g., music, video, entertainment, etc.) tend to exhibit problems in multiple contexts. This observation motivates the use of contextual fuzzing in mobile app testing. Furthermore, the use of media often increases the memory usage, which results in crashes.

Figure 10 shows performance outliers broken down by resource type. The figure suggests that most outliers are network or energy related. While tools for testing different networking scenarios are starting to emerge [20], the same has not yet happened for energy-related testing (which also heavily depends on the type of radios and communication protocols in use). Both disk activity (i.e., I/O) and CPU appear to have approximately the same number of performance outlier cases.

6.4 Experience and Case Studies

In this section we highlight some identified problem scenarios that mobile app developers might be unfamiliar with, thus illustrating how a tool like Context Virtualizer can be used to prevent ever more common issues.

Geolocation. As an increasing number of apps on mobile platforms become location-aware, services start providing location-tailored content. For example, a weather app can provide additional weather information for certain cities, such as Seattle. Another example is content restrictions in some streaming and social apps. Unfortunately, many developers are unaware of the implications of device geolocation

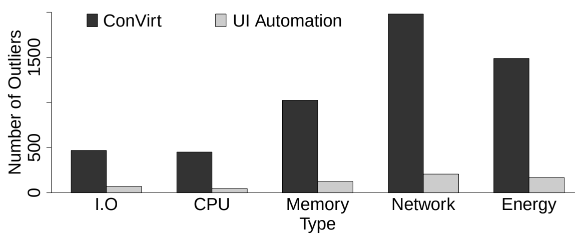


Figure 10. App performance outliers categorized by resource usage.

	Context Virtualizer	Auto UI	Context Virtualizer	Auto UI
	(Outliers)		(Instance of Crash)	
News	1437	147	284	24
Entertainment	667	62	101	6
Photo	90	10	17	1
Sports	304	41	194	15
Video	688	123	142	12
Travel	238	12	25	1
Finance	193	13	0	0
Weather	31	2	1	0
Music	737	85	289	38
Reference	12	0	0	0
Education	125	13	0	0
E-reader	24	5	0	0
Social	20	1	112	5
Lifestyle	23	4	5	0

Table 3. App crashes and performance outliers categorized the same as the Windows Store.

on app behavior and energy consumption. We use dataset #3 to illustrate these impacts.

Our first case study focuses on an app released by a US-based magazine publisher. Test results from ten world-wide locations show that the app crashed frequently outside of North America; sometimes even not proceeding beyond the loading screen. This problem was confirmed by users on the app marketplace and verified by our manual testing and the publisher later released an update (after our first round of testing). Re-exercising the app with our tool suggests that the likelihood of crashes in China was reduced from 80% to 50% with the new version, but the problem was not completely resolved.

A second case study on location exemplifies how resource consumption variance can be significant with geolocation. Figure 11 depicts a snapshot of dataset #3, where we see that network-related metrics typically exhibit the largest variance. We note that network-related metrics can impact the system-related metrics such as CPU. The implications are two fold. First, apps can present a higher energy consumption in certain locations. For example, a particular weather app uses 8% more energy in Seattle than Beijing. Second, the excessive memory usage in some locations can translate to crashes.

Network Transitions. In contrast to PCs, mobile devices are rich in mobility and physical radio options. In fact, network transitions can happen frequently throughout the day, e.g., handoffs between cellular towers, switches from 3G to 2G, and transition between Wi-Fi and cellular network. While an increasing number of developers start to test apps under discrete network profiles [20], testing for network transitions is not yet a common practice.

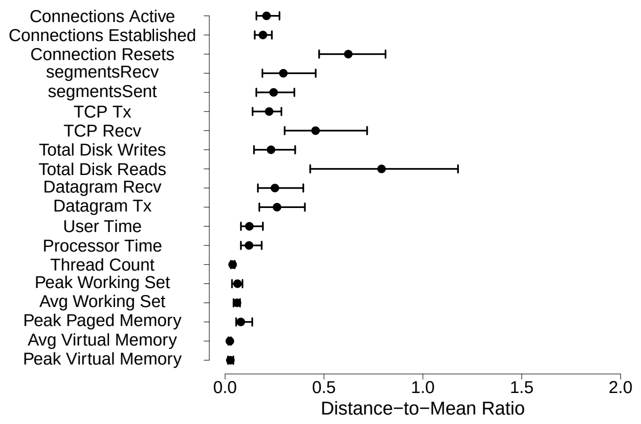


Figure 11. Per-metric variation of WCDMA emulation with traces from top 10 worldwide locations with high mobile device usage.

An example case that demonstrates these issues is a popular Twitter client app. Our system logs and user traces suggested that the app crashed if users tried to *tweet* after a transition to a slower network. Any attempt, by a user, to post two separate messages, one over a Wi-Fi network and a second one after switching to a slower GPRS network, was enough to repeatedly cause a crash. Interestingly, if the opposite network transition occurs, it does not seem to affect the app execution. Without peeking into the source code, it is difficult to point out the exact root cause of the issue. However, all results from our logs (and posterior manual exploration) suggest that the app does not consider network dynamics and assumes the network is always available, after an initial successful connection.

Exception Handlers. Results of dataset #1 also indicate that some music streaming apps tend to crash with higher frequency on slow and lossy networks. Decompiled code analysis [29] reveals that the less crash-prone apps apply a more comprehensive set of exception handlers around network-related system calls. The effort in handling such exceptional cases is extremely important in mobile platforms as they can experience a much wider range of network conditions than traditional PCs. Although, as previously highlighted[7], the lack of a priori design of the exceptional behavior can lead to multiple issues, it is not feasible to expect that developers correctly create such code without tools that support them in checking for the necessary environment conditions.

Device Service Misuse. On this case ConVirt highlighted a possible energy bug in a location tracking app. The app registers for location updates (by setting a minimum distance threshold for notifications) to avoid periodic polling the platform. Events are then signaled if the location service detects the device has moved beyond the threshold. However, the app set the threshold to a value lower than the typical accuracy of the location providers (e.g. 5m accuracy at best on typical smartphone GPS[31]). This resulted in the app constantly receiving location updates and keeping the process up, which then consumed more energy than expected.

7 Discussion

We now examine some of the overarching issues related to Context Virtualizer design.

Generality of the System. While the paper focuses on app testing, our core ideas can be generalized to other scenarios. In privacy, applying app similarity networks to packet inspections can discovery apps that transmit an abnormal amount of personal data. In energy optimization, measurement can help determine whether an app would experience significant performance degradation on a slower but more energy-efficient radio.

Real-World Mobile Context Collection. Our system relies on real-world traces to emulate mobile context. We recognize that some traces are more difficult to collect than others. For example, while the data on cellular carrier performance at various locations is publicly available [23], an extensive database on how users interact with apps is not easily accessible. We leave the problem of collecting user interaction traces at large scale as a future work.

Hardware Emulation Limitation. ConVirt currently only exposes coarse-grained hardware parameters: CPU clock speed and available memory. While this suggests that certain artifacts of the hardware architecture cannot be emulated, our system can accommodate real devices in the test client pool to achieve hardware coverage.

User Interaction Model Limitation. Our current implementation interacts with apps by invoking their user interface elements (e.g, buttons, links, etc.) through a UI automation library. One limitation is that user gestures can not be emulated, so many games cannot be properly tested. Gesture is left as mid-term future work.

Applicability To Other Platforms. While our current implementation works for Windows 8, the core system ideas can also work on platforms that fulfill three requirements: (1) the network stack should allow a method to manipulate incoming and outgoing packets; (2) provide a way to emulate inputs to apps, such as user touch, sensors and GPS; and (3) good performance logging provided by the OS. Additionally, the backend network should have higher performance than the emulated network profiles. These requirements are not onerous, and Android is another readily suitable platform.

8 Related Work

Clearly, a wide variety of testing methodologies already exist for discovering software, system, and protocol problems. Context Virtualizer contributes to this general area by proposing to expand the testing space for mobile devices to include a variety of real-world contextual factors. In particular, we advance prior investigations of fuzz testing [21, 11] with techniques that enable the systematic search of this new context test space; this new approach can then complement existing techniques including static analysis and fault injection.

Mobile App Testing. In response to the strong need for improved mobile app testing tools, academics and practitioners have developed a variety of solutions. A popular approach is log analytics, with a number of companies [9, 6, 4, 30] offering such services. Although this data is often insightful, it is only collectable post-release, thus exposing users to a negative user experience.

Similarly, although AppInsight [25] enables significant new visibility into app behaviour, it is also a post deployment solution that requires app binary instrumentation. Context Virtualizer, in contrast, enables *pre-release* testing of unmodified app binaries.

Emulators are also commonly used and include the ability to test coarse network conditions and sensor (GPS, accelerometer) input. [14, 20], for instance, offer such controls and allow a developer to test their app by selecting different speeds for each network type. More advanced emulator usage, such as [18], enables developers to have scripts sent to either real hardware or emulators and to select from a limited set of network conditions to apply during the test. Unlike Context Virtualizer, neither emulator systems nor manual testing offer the ability to control a wide range of context dimensions – simultaneously, if required; neither do they allow the high-fidelity emulation of contextual conditions, such as handoff between networks (e.g. 3G to WiFi) that can be problematic for apps. Finally, the conditions tested are typically defined by the developers – instead, ConVirt generates the parameters for test cases automatically.

Testing methods based on UI automation, when applied to mobile app testing adopt emulators to host applications. Android offers specialized tools [14, 13] for custom UI automation solutions. Also, significant effort has been invested into generating UI input for testing (e.g., [34, 15]) with specific goals in mind (e.g., code coverage). ConVirt allows users to define usage scenarios or falls back to random exploration. However, any automation technique can be added into our system.

Efficient State Space Exploration for Testing. A wide variety of state exploration strategies have been developed to solve key problems in domains such as distributed systems verification [35] and model checking [16]. State exploration is also a fundamental problem encountered in many testing systems. Many existing solutions assume a level of internal software access. For example, [1] explores code paths within mobile apps and reduces path explosion by merging redundant paths. Context Virtualizer test apps as a blackbox and so such techniques do not apply. In [12], a prioritization scheme is proposed that exploits inter-app similarity between code statement execution patterns. However, ConVirt computes similarity completely differently (based on resource usage) – allowing use with blackbox apps.

Simulating Real-world Conditions. ConVirt relies on high-fidelity context emulation along with real hardware. Other domains, notably sensor networks have also developed testing frameworks (e.g., [19]) that incorporate accurate network emulation. In particular, Avrora [32] offers a cycle-accurate emulation of sensor nodes in addition to network emulation. Conceptually ConVirt has similarity with these frameworks, but in practice the context space (low-power radio) and target devices (small-scale sensor nodes) are completely different.

9 Conclusion

This paper presents Context Virtualizer (ConVirt) – an automated service for testing mobile apps using an expanded mobile context test space. By expanding the range of test conditions we find ConVirt is able to discover more crashes and performance outliers in mobile apps than existing tools, such as emulator-based UI automation.

10 References

[1] S. Anand, M. Naik, H. Yang, and M. Harrold. Automated concolic testing of smartphone apps. In *Proceedings of the ACM Conference on Foundations of Software Engineering (FSE)*. ACM, 2012.

[2] Apigee. `http://apigee.com`.

[3] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, August 2006.

[4] BugSense. BugSense — Crash Reports. `http://www.bugsense.com/`.

[5] Carat. `http://carat.cs.berkeley.edu/`.

[6] Crashlytics. Powerful and lightweight crash reporting solutions. `http://www.crashlytics.com`.

[7] R. Di Bernardo, R. Sales Jr, F. Castor, R. Coelho, N. Cacho, and S. Soares. Agile testing of exceptional behavior. In *Proceedings of SBES 2011*. SBC, 2011.

[8] Flurry. Electric Technology, Apps and The New Global Village. `http://blog.flurry.com/default.aspx?Tag=market%20size`.

[9] Flurry. Flurry Analytics. `http://www.flurry.com/flurry-crash-analytics.html`.

[10] Fortune. 40 staffers. 2 reviews. 8,500 iphone apps per week. `http://tech.fortune.cnn.com/2009/08/21/40-staffers-2-reviews-8500-iphone-apps-per-week/`.

[11] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Proceedings of NDAA 2008*, 2008.

[12] A. Gonzalez-Sanchez, R. Abreu, H.-G. Gross, and A. J. van Gemund. Prioritizing tests for fault localization through ambiguity group reduction. In *Proceedings of ASE 2011*, 2011.

[13] Google. monkeyrunner API. `http://developer.android.com/tools/help/monkeyrunner_concepts.html`.

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

[15] F. Gross, G. Fraser, and A. Zeller. Search-based system testing: high coverage, no false alarms. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2012.

[16] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of SOSP2011*. ACM, 2011.

[17] J. Huang, C. Chen, Y. Pei, Z. Wang, Z. Qian, F. Qian, B. Tiwana, Q. Xu, Z. Mao, M. Zhang, et al. Mobiperf: Mobile network measurement system. Technical report, Technical report, Technical report). University of Michigan and Microsoft Research, 2011.

[18] B. Jiang, X. Long, and X. Gao. Mobiletest: A tool supporting automatic black box test for software on smart mobile devices. In H. Zhu, W. E. Wong, and A. M. Paradkar, editors, *AST*, pages 37–43. IEEE, 2007.

[19] P. Levis, N. Lee, M. Welsh, and D. E. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In I. F. Akyildiz, D. Estrin, D. E. Culler, and M. B. Srivastava, editors, *SenSys*, pages 126–137. ACM, 2003.

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

[21] B. P. Miller, L. Fredrikson, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32, December 1990.

[22] R. Mittal, A. Kansal, and R. Chandra. Empowering developers to estimate app energy consumption. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, Mobicom '12, pages 317–328, New York, NY, USA, 2012. ACM.

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

[24] Open Signal. The many faces of a little green robot. `http://opensignal.com/reports/fragmentation.php`.

[25] 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.

[26] Samsung. Series 7 11.6" Slate. `http://www.samsung.com/us/computer/tablet-pcs/XE700T1A-A01US`.

[27] 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`.

[28] 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`.

[29] Telerik. JustDecompile. `http://www.telerik.com/products/decompiler.aspx`.

[30] TestFlight. Beta testing on the fly. `https://testflightapp.com/`.

[31] A. Thiagarajan, L. Ravindranath, K. LaCurts, S. Madden, H. Balakrishnan, S. Toledo, and J. Eriksson. Vtrack: accurate, energy-aware road traffic delay estimation using mobile phones. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 85–98, New York, NY, USA, 2009. ACM.

[32] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press.

[33] 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.

[34] L. Yan and H. Yin. DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *Proceedings of the 21st USENIX Security Symposium*. USENIX, 2012.

[35] J. Yang, C. Sar, and D. R. Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In B. N. Bershad and J. C. Mogul, editors, *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 131–146. USENIX Association, 2006.