Context Virtualizer: A Cloud Service for Automated Large-scale Mobile App Testing under Real-World Conditions

MSR-TR-2013-103

Chieh-Jan Mike Liang[‡], *Nicholas D. Lane*[‡], *Niels Brouwers*[*], *Li Zhang*[⋆], *Börje Karlsson*[‡], *Hao Liu*[†], *Yan Liu*[◁], *Jun Tang*[⋈], *Xiang Shan*[⋈], *Ranveer Chandra*[‡], *Feng Zhao*[‡]

[‡]*Microsoft Research* [*]*TU Delft* [⋆]*USTC* [†]*Tsinghua* [◁]*SJTU* [⋈]*HIT*

Abstract

Scalable and comprehensive testing of mobile apps is extremely challenging. Every test input needs to be run with a variety of *contexts*, such as: device heterogeneity, wireless network speeds, locations, and unpredictable sensor inputs. The range of values for each context, e.g. location, can be very large. In this paper we present a one of a kind cloud service, called ConVirt, to which app developers can submit their apps for testing. It leverages two key techniques to make app testing more tractable: (i) pruning the input space into representative contexts, and (ii) prioritizing the context test space to quickly discover failure scenarios for each app. We have implemented ConVirt as a cluster of VMs that can each emulate various combinations of contexts for tablet and phone apps. We evaluate ConVirt by testing 230 commercially available mobile apps based on a comprehensive library of real-world conditions. Our results show that ConVirt leads to an 11.4x improvement in the number of crashes discovered during testing compared to conventional UI automation (i.e., monkey-testing.)

1 Introduction

The popularity of mobile devices, such as smartphones and tablets, is fueling a thriving global mobile app ecosystem. Hundreds of new apps are released daily, e.g. about 300 new apps appear on Apple's App Store each day [9]. In turn, 750 million Android and iOS apps are downloaded each week from 190 different countries [7]. 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. This challenge is made worse by low consumer tolerance for buggy apps. In a recent study, only 16% of smartphone users continued to use an app if it crashed twice soon after download [31]. As a result, downloaded app attrition is very high – one quarter of all downloaded apps are used just once [30]. Mobile users only provide a brief opportunity for an app to show its worth, poor 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., [8, 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 local conditions are likely not representative of a public (particularly global) app release. One could use mobile platform simulators ([15, 23]) 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 systematically explore representative combinations of operating contexts under which their app might be used.

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 *ConVirt* (for Context Virtualizer) – 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 ConVirt service. A summary report is generated by ConVirt for the developer that details the problems observed, the conditions that trigger the bug, and how it can be reproduced.

A key challenge when implementing ConVirt 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 10,504 contexts is available and sourced largely from trace logs of conditions encountered by real mobile users (see §5.) 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 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 prioritization of context perturbation unexpected app problems and even crashes can be found much more quickly than is otherwise possible.

This paper, makes the following contributions:

- We present a new concept, of *contextual fuzzing*, which expands conventional mobile app testing to include complex real-world operating scenarios. This enables developers to identify context-related performance issues and sources of app crashes prior to releasing the app. (Section 2).
- We develop techniques for the scalable exploration of the mobile context space. First, we propose a method for synthesizing a representative, yet comprehensive, library of context stress tests from large repositories of available context sources. (Section 3). Second, we propose a learning algorithm that leverages similarities between apps to identify which conditions will impact previously unseen apps by leveraging observations from previously tested apps (Section 4).
- We design a new cloud service, called ConVirt, to which app developers and tests can submit their apps to identify context-related problems. This service consists of a pool of virtual machines that support either smartphone or tablet apps, while realistically emulating a variety of complex context combinations. (Section 5).

We evaluate ConVirt using a workload of 200 Windows 8 Store apps and 30 Windows Phone 8 apps based on a comprehensive library of real-world contexts. Our experiments show exploring the mobile context space leads to an 11.4x improvement in the number of crashes discovered during testing relative to UI automation based testing. Moreover, we find crashes we discover from automated pre-release testing take only around 10% of the time compared to waiting for the same bug to appear within a commercially available crash reporting system. Finally, ConVirt is able to identify 8.8x more performance issues compared to standard monkeying.

2 The ConVirt Approach

ConVirt is targeted towards two groups 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, e.g. another country, or different phones. **App distributors, or enterprises** who accept apps from developers and offer them to consumers, e.g. marketplaces of apps. The distributors must decide if an app is ready for public release.

For both classes of users, ConVirt has the following requirements.

- First, and most importantly, it needs to be *comprehensive*. Since testing all possible contexts, e.g. every possible lat-long location, will take a long time, ConVirt needs to come up with cases that are representative of the real-world, and where the app is most likely to fail or misbehave.
- Second, and usually in contradiction to the first requirement, it should be *responsive* and provide quick, timely feedback to users. We strive to provide feedback in the order of minutes, which is very challenging given the number of contexts, and their combinations for which the apps need to be tested. Consequently, ConVirt needs to scale with increasing number of simultaneously submitted apps.
- Third, it should be *black box*. We cannot always assume access to app source code. Although instrumentation of binaries has been shown to work for Windows Phone apps [28], we want our techniques to be general and work for Windows tablets, Android and iOS devices as well.

2.1 The Mobile Context Test Space

We believe the following three mobile contexts, and the variations therein, capture most context-related bugs in mobile apps. To the best of our knowledge there does not exist ways to principally test these context variations.

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 on the cellular operator [19], even given identical locations and hardware, the bandwidth speeds between countries frequently can vary between 1 Mbps and 50 Mbps [35], and the signal strength variation changes the energy usage of the mobile device [25].

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 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 [27]. 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 lower-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.

2.2 ConVirt Overview

ConVirt is implemented as a cloud service with the components shown in Figure 1. Users (app developers or distributors) submit binaries of their apps (packages for Windows). ConVirt then runs the App under an emulator environment (AppHost), which simulates various contexts (networks, carriers, devices, locations, etc.) using the Perturbation Layer. It continuously monitors the performance of the app under each context, and the PerfAnalyzer outputs a report with all cases where it found the app to have a bug, where a bug could be a crash, a performance anomaly or an unexpected energy drain.

However, as mentioned earlier, running all possible combinations of contexts is not feasible. To address this challenge, we propose two techniques. First, ContextLib uses machine learning techniques to identify representative contents by (i) determining which combinations of contexts are likely to occur in the real world, and (ii) removing redundant combinations of contexts. This is a

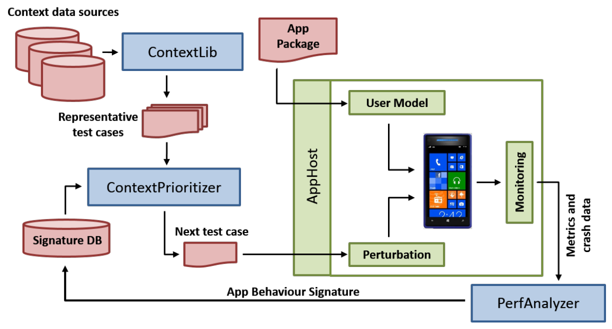

Figure 1: ConVirt System Diagram

preprocessing step, which we run periodically on crowd-sourced data as explained in Section 3. Second, ContextPrioritizer sorts different context combinations, and runs those with higher likelihood of failure before others. This helps ConVirt quickly discover the buggy scenarios. ContextPrioritizer sends one test case at a time, in prioritized order, to an AppHost. Both ContextLib and ContextPrioritizer are general techniques, which we believe have applications beyond ConVirt. (Section 7)

2.3 ConVirt System Components

As shown in Figure 1, ConVirt consists of four main components: (1) ContextLib, (2) ContextPrioritizer, (3) AppHost, and (4) PerfAnalyzer. A coordinator module is responsible for managing multiple AppHosts.

ContextLib. ContextLib stores definitions of various mobile context conditions (e.g. loss, delay, jitter at a location) and scenarios (e.g. handoff from Wi-Fi to 3G). 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. ContextLib is populated using datasets collected from real devices (e.g., OpenSignal [26]) in addition to challenging mobile contexts defined by domain experts. We describe this component in more detail in Section 3.

ContextPrioritizer. ContextPrioritizer determines the order in which the contexts from ContextLib should be performed, and then assigns each test to one of a pool of AppHosts. Aggregate app behavior (i.e., crashes and resource use) collected and processed by PerfAnalyzer from AppHosts, is used by ContextPrioritizer to build and maintain app similarity measurements that determines this prioritization. Both prioritization and similarity computation is an online process. As each new set of results is reported by an AppHost 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) contexts that negatively impacted other similar apps are prioritized (e.g., an app that behaves similarly to other streaming apps has network contexts prioritized).

AppHost. Apps run within a VM called AppHost. This helps isolate many apps that are simultaneously submitted to ConVirt, and can help parallelize various tests. The AppHost Coordinator picks a VM from the pool of AppHosts to perform the workload generated by ContextPrioritizer. The AppHost has three primary functions:

UI Automation (Monkeying): We use a User Interaction Model (see §5) that generates user events (e.g., touch events, key presses, data input) based on weights (i.e. probability of invocation) assigned to specific UI items. Our technique works on tablets and phones, and is able to execute most of the scenarios for an app. We also allow developers to customize this execution and personalize the test cases.

Simulating Contexts (Perturbation): This component simulates conditions, such as different CPU performance levels, amount of available memory, controlled sensor readings (e.g., GPS reporting a programmatically defined location), and different network parameters to simulate different network interfaces (e.g., Wi-Fi, GPRS, WCDMA), network quality levels, and network transitions (3G to Wi-Fi) or handoffs between cell towers. Each one of these is implemented using different various kernel hooks or drivers (5). This layer is extensible and new context dimensions can be added for other sensors or resource constraints.

Monitoring: During test execution AppHost uses different modules to closely record app behaviour in the form of a log of system-wide and per-app performance counters (e.g., network traffic, disk I/Os, CPU and memory usage) and crash data. To accommodate the variable number of logging sources (e.g., system built-in services and customized loggers), AppHost implements a plug-in architecture where each source is wrapped in 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, like specific rendering events).

PerfAnalyzer. This component identifies crashes, and possible bugs (e.g. battery drain, data usage spikes, long latency) in the large pool of monitoring data generated by AppHosts. To identify failures that do not result in a crash, it uses anomaly detection that assumes norms, based on previous behavior of (1) the app being tested and (2) a group of apps that are similar to the target app.

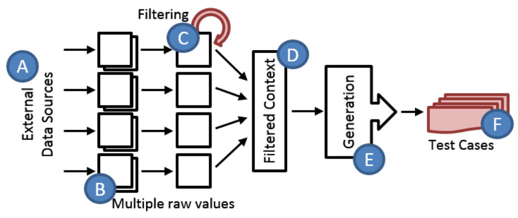

Figure 2: ContextLib Dataflow, showing the components and processes: (A) Context Sources; (B) Raw Context; (C) Redundancy Filtering; (D) Filtered Context; (E) Test Case Generation; and, (F) Test Cases

Crashes by themselves provide insufficient data to identify their root cause. Interpretation of them is key to providing actionable feedback to developers. PerfAnalyzer processes crash data across crashes to provide more focused feedback and helps narrow down the possible root cause. Also, if the app developer provides debug symbols, PerfAnalyzer can find the source code location where specific issues were triggered.

These bugs (crashes, performance, or energy anomalies) can be roughly categorized into three groups: implementation mistake, resource exhaustion, and unanticipated circumstances. While implementation mistakes are easier to reason about and debug, the other two categories are still hard to track with just crash episodes data.

PerfAnalyzer augments the bug data with relevant contextual information to help identify the source of problems. The generated report includes resource consumption changes prior to crash, the set of perturbed conditions, and the click trace of actions taken on the app (along with screenshots), thus documenting its internal computation state and how the app got there; which is missing in regular bug tracking systems or if only limited data on the crash moment is available. The generated aggregate report also allows developers to find commonalities and trends between different crash instances that might not be visible in coarser tests.

3 ContextLib: Generating Test Contexts

ContextLib automatically generates a library of realistic test cases from raw sources of context, such as OpenSignal. We note that ConVirt allows users to add specific additional test cases. The output of ContextLib seeds the list of contexts for which the app will be tested.

Figure 2 illustrates the main components of ContextLib. It pulls raw data from various online databases and services (A), such as those listed in Table 1, and populates the Raw tables (B). Redundant and duplicate contexts are then suppressed for each context (C), to lead to a filtered set of context instances (D). We then combine different contexts (network, memory, CPU, etc.) (E) that generates the list of test cases (F), which feeds to ContextPrioritizer. This process is repeated periodically, or as new data becomes available. We explain the two algorithmic components, (C) and (E), next in this section.

Redundancy Filtering. As a first step, we remove duplicate entries from the raw contexts (*rc*s). However, this step by itself is not sufficient. There are still several *rc*s that are not identical, such as two different networking scenarios, yet they may not cause a meaningful change in app behavior.

Simply clustering raw context parameters (such as, available memory or network latency) is not effective because fundamentally these values do not have linear correspondence to app behavior. For some combination of context parameters a small change will result in vastly different app behavior; while (as mentioned) other context parameters that are very different may cause nearly identical operation.

Instead, we use a different approach. ContextLib filters raw contexts if an app does not demonstrate a significant change in its *system resource usage* compared to other contexts. The underlying assumption is that significant changes in resources are an indicator that the app is being exercised in significant different ways by context[1].

Resource-based filtering is performed on a per context dimension basis; at each iteration the objective is to arrive at $\mathtt{f}dom_i$ – a set of key context parameters within the i^{th} context dimension that typically cause apps to behave (i.e., consume resources) differently. We begin this process by learning a dimension-specific matrix ($\mathtt{dom.i}_{prj}$) that can project each *rc* into a resource usage vector space. This resource usage space is parameterized by j resource usage metrics (see Table 2 for the 19 metrics used the ConVirt prototype;) each dimension is a time normalized statistic for the metric (e.g., average CPU utilization across one minute intervals.) We find $\mathtt{dom.i}_{prj}$ for dimension i by using matrix decomposition across a collection of training examples – that is, observations of pairs of *rc* and corresponding normalized resource usage for the default app workload collection. By performing matrix decomposition and learning $\mathtt{dom.i}_{prj}$ we are able to project any arbitrary point between the two spaces; this enables us to project *rc*s provided by context sources for which we do not yet have resource usage statistics, allowing them to still be included in the filtering process.

Training examples are collected in two ways (1) periodically testing a representative app workload based on popular mobile apps across multiple categories and (2) the output of app testing from standard ConVirt operation (see §6 for additional details.)

After all *rc*s are projected using $\mathtt{dom.i}_{prj}$ the resource vector space undergoes a dimensionality reduction step. This is done to remove the influence of irrelevant dimension for the particular context type being operated within. For example, disk related system metrics, that are part of the default j dimensions, might only be weakly relevant within a networking-related context dimension. This step is performed with Multi-dimensional Scaling [3] (MDS). Finally, to remove those *rc*s that have correlated system usage metrics (and thus are redundant) we apply a density-based clustering algorithm – DBSCAN [3]. [2]

Test Case Generation. To allow test cases to be a composite of different domains (e.g., network and memory), the follow steps are performed to generate candidate test cases (*ct*s.) This is necessary because single-domain fragments of context are readily available – but sources of complete multi-domain context are not.

First, additional *transition* context domains are generated. Specifically, for each filtered domain ($\mathtt{f}dom_i$) a companion transition domain is created. For example, the network domain is paired with a network-*transitions* domain. Transition domains contain the entries capturing all possible by transitions from one filtered context to another, within that domain (i.e., $\mathtt{f}dom_i \times \mathtt{f}dom_i$.) This step is necessary because context state changes (such as, switching from Wi-Fi to 3G network connectivity) are common situations for mobile apps to fail. Later, apps that are perturbed with a test case containing a transition are initially exposed to the first state (such as, a moderate amount of memory) before then exposed to the second state (such as, a lower level of memory.)

Next, a Cartesian product is performed across all filtered domains, including the new context transition domains ($\mathtt{f}dom_1 \times \mathtt{f}dom_2 \times \cdots \mathtt{f}dom_n$.) The side-effect of this process is that new redundant *ct*s can be introduced. To remove these we again perform one final round of redundancy filtering. Just as before, each *ct* is projected into a resource usage space based on projection matrix ($\mathtt{multi.dom}_{prj}$), trained exactly as previously described. MDS and DBSCAN are also applied again before filtered clusters are projected back into context parameters.

The final number of test cases generated are largely determined by clustering parameters[3] that influence both filtering phases of ContextLib. Later in §6.2.1, we evaluate this trade-off between the coverage of raw context and system overhead as the number of test cases is changed – primarily this is a decision by the user,

[1]Under the expectation the same User Interaction Model is applied

[2]Although we use MDS and DBSCAN many other dimensionality reduction and density-based clustering techniques are likely to perform equally well.

[3]In the case of DBSCAN these are ε and *MinPts* [3] but equivalent parameters are present in alternative clustering algorithms

depending on how they value testing fidelity relative to costs like latency in testing or computation used.

4 Prioritizing Test Cases

ContextLib maintains a test case collection that runs into thousands. At this scale, the latency and computational overhead of performing the full suite test cases becomes prohibitive to users. Instead, ConVirt must remain effective with a much smaller number of test cases. To meet this need, ContextPrioritizer is designed to find a *unique per-app sequence of test cases* that increases the marginal probability of identifying crashes (and performance issues) for each test case performed. The key benefit is that users are able to discover important context-related crashes and performance bugs while performing only a fraction of the entire ContextLib. However, just as in the case of ContextLib, its use is optional; we provide users a way to specify specific order of test cases.

Overview. Figure 3 presents the dataflow of ContextPrioritizer; it includes the following components and stages: (A) Test App; (B) AppSimSet; (C) Test Case History Categorization; (D) Test Case Sequence Selection; and, (E) Test Case Order. Within the overall architecture of ConVirt, the role of ContextPrioritizer is to determine the next batch of test cases to be applied to the user provided test app (chosen from ContextLib.)

The underlying approach of ContextPrioritizer is to learn from prior experience when prioritizing test cases for a fresh unseen test app. ContextPrioritizer first searches past apps to find a group that is similar to the current test app (referred to as a `SimSet`.) And then examines the test case history of each member of the `SimSet`, identifying any "problematic" test cases that resulted in the crashing of the test app. These problematic test cases are then prioritized ahead of others, for the current test app, in an effort to increase the efficiency by which test app problems are discovered.

While we informally refer to app(s) in this description (e.g., test app, or apps in a `SimSet`), more precisely this term corresponds to an *app package* that includes both (1) a mobile app and (2) an instance of a User Interaction Model (see §5.3). This pairing is necessary because the code path and resource usage of an app is highly sensitive to the user input. Conceptually, an app package represents a particular user scenario within an app.

App Similarity Set. In addition to the sheer number of potential test cases, context fuzzing is complicated by the fact a test app will likely be only sensitive to a fraction of all test cases in ContextLib. However, it is also non-trivial to predict which test cases are important for any particular test app – without first actually trying the combination. For example, an app that uses the network only

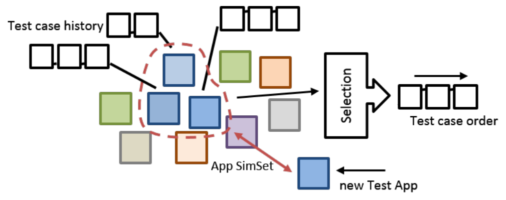

Figure 3: ContextPrioritizer Dataflow

sporadically may turn out to be fairly insensitive to many network related test cases; wasting resources spent on testing this part of the mobile context space. Reasonable heuristics for optimizing the assignment of test apps to test cases – such as, inspecting the API calls made by an app, and linking certain contexts (e.g., network-related test cases) to API calls (e.g., network-related APIs) – would have been confused by the prior example, and still have made the same mistake.

ContextPrioritizer counters this problem by identifying correlated system resource usage metrics as a deeper means to understand the relationship between two apps. The intuition underpinning this approach is that two apps that have correlated resource usage (such as, memory, CPU, network) are likely to have shared sensitivity to similar context-based test cases. For example, in deciding which test cases to first apply to the prior example app (with sporadic network usage) this approach would identify previous test apps that also had sporadic network usage – recognized by similarities in network resource consumption – and try test cases that also caused these previous apps to crash.

The building block operation within ContextPrioritizer is a *pairwise similarity comparison* between a new test app, and a previously tested app. This is done for just one system resource metric while both apps were exposed to the same test case (i.e., the same context). A standard statistical test is performed to understand if the distribution of the time-series data for this particular metric generated by each app is likely drawn from the same underlying population (i.e., the distributions are statistically the same.) To do this we use a standard approach and apply the Kolmogorov-Smirnov (K-S) test [3]; assuming an α of 0.05. The outcome of this test is binary, either the distributions are either found to be the same or they are not. We expect other statistical tests (i.e., a K-S alternative) would produce comparable results.

ContextPrioritizer uses the above described pairwise comparison multiple times to construct a unique `SimSet` for each new test app. During this process, ContextPrioritizer considers a collection of j resource metrics (the same group used by ContextLib and listed in Table 2.) For each resource metric the pairwise tests are performed between the current test app and all prior test apps. In many cases, ContextPrioritizer is able to compare the same pair of apps and same resource metrics more than once (for example, under different test cases.) For the prior test app to be included in `SimSet` it must pass the statistical comparison test a certain percentage of times (regulated by $\texttt{KS}_{thres}$ – set to 65% in our implementation.)

Importantly, no comparisons between the current and previous test apps can be performed until at least a few test cases are performed. ContextPrioritizer uses a bootstrapping procedure to do this, whereby a small set of test cases are automatically done before ContextPrioritizer is invoked. We select these test cases experimentally by identifying *k* test cases with a high initial rate of causing crashes[4]. Bootstrapping is important only briefly as test app data quickly accumulates once ConVirt begins.

Test Case History Categorization. As many crashes are caused by abnormal resource usage (e.g., excessive memory usage), they can be tied to resource metrics. The motivation for such categorization is that each `SimSet` should consider only crashes matching its resource metric, in the Test Case Sequence Selection stage.

The intuition is that a rapid fall or rise in the resource metric consumption immediately prior to a crash is a signal of a likely cause. For example, memory may increase at a rapid rate immediately before a crash. Importantly, this process enables non-context related crashes (e.g., division by zero) to be ignored during prioritization. Crashes are assigned to one of the *j* monitored resource metrics (described earlier); otherwise, the crash is categorized as being non-context related. Resource consumption leading up to the crash (our current implementation uses a 120 second window) are tracked. Typically, the crash is then tied to the resource metric with the largest gradient (either positive or negative) compared to all other metrics.

Test Case Sequence Selection. ContextPrioritizer uses a two-tiered voting process to arrive at the sequence of test cases to be applied to the test app. At each tier the voting setup is the same. Each time a test case results in a crash is treated as a vote, any crash that is categorized as being non-context related is ignored. The order of test cases is determined by the popularity of crash causing test cases tied to prior test apps contained within `SimSet`.

The first tier of voting operates at intra-resource metric level – in other words, multiple voting process are performed – one for each resource metric. Within each resource metric vote, only those prior test apps that passed the pairwise comparison test for the resource metric in question are included. By performing intra-resource metric voting problems related to a diverse set resources and contexts can be identified. By their nature some resources-related issue occur much more frequently than others – this approach helps this situation.

The second tier of voting operates between the top *n* popular test cases selected from each resource metric. A single test case is chosen, but this process is repeated multiple times to determine a test case sequence. This is typically done because ConVirt performs batches of test cases rather than one test case at a time. At all stages, ContextPrioritizer breaks ties by random selection.

[4]We set *k* to 3 and find the best test cases are: `GPRS`, `802.11b`, `4G`

Context Dimension	Raw Entries	Description	Source
Network	9,500+	Cellular in Locations	Open Signal
		Wi-Fi Hotspot	
		Cellular Operators	
Platform	220/23	CPU Utilization Ranges	Watson
	(W8/WP8)	Memory Ranges	
Various	45	...	Hand-coded

Table 1: Context Library

5 ConVirt Implementation

This section presents our current implementation of the testing framework components (see Figure 1). The entire ConVirt implementation consist of approximately 31k lines of code, broken into: ContextLib 1.9k; ContextPrioritizer 2.2K; AppHost 20K; and PerfAnalyzer 6.8K.

ContextLib. Our current ContextLib contains 10,504 raw mobile contexts, as summarized in Table 1. The algorithms that rely on this data are described in §3.

The majority of the ContextLib currently uses Open Signal [26] as an external data source. This public dataset is comprised of crowd-sourced cellular measurements collected around the world. Based on data from Open Signal, we limit the number of cities to 400, which include 50 mobile carriers. Additional examples of ContextLib tests are memory and CPU events that can be sourced from telemetry databases (we use the Microsoft Watson Database). Finally, we hard-code a number of raw ContextLib records for challenging mobile scenarios that are not supported by current context sources.

AppHost Coordinator. AppHosts are designed to run either on real devices or in virtual machines (VM) for testbed scalability. Currently we maintain a pool of AppHosts hosted on Microsoft Azure. A central server manages each AppHost node via an RPC-like framework. On Windows 8, Windows Management Instrumentation (WMI) is used. But on Windows Phone 8, WMI is replaced by a test harness (TH) that communicates over IP, providing the same functionality. Both the WMI and TH interfaces are tightly integrated with PowerShell (a shell environment for Windows), so each test executed is de-

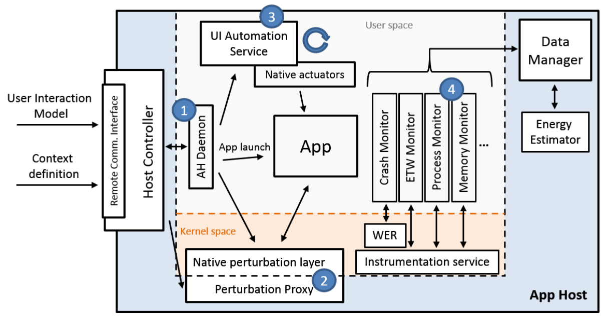

Figure 4: Implementation Components of AppHost

scribed in a PowerShell script that specifies the mobile contexts to emulate. Finally, at the end of a test, the central server imports the data to a MS SQL Server database.

AppHost. Each AppHost runs in a host virtual machine, controlled by the AppHost Coordinator. AppHosts support both Windows 8 and Windows Phone 8 apps. As shown in Figure 4, an AppHost is comprised of four main components: 1) Controller, 2) UI Automation Service, 3) Perturbation Service, and 4) Data Manager.

The Controller is responsible for the communication with the rest of the ConVirt system and for orchestrating the other components inside the AppHost. A sub-module of the Controller, called AppHost Daemon, encapsulates the OS-specific coordination of UI automation, context emulation, and app monitoring. When the app under test is a Windows 8 app, the controller and all other modules run on the target OS. This allows Windows 8 apps within AppHost to be exercised (with user input), observed (via monitors and data manager), and for the context to be carefully controlled (via perturbation).

To support Windows Phone 8 apps, an inner VM running the Windows Phone 8 OS is used. WP8 apps run in this VM and are exercised and monitored with phone-specific modules. The Controller and Data Manager still run on Windows 8 unchanged. But the AppHost Daemon, UI Automation Service, monitors, and a native layer of the Perturbation Service (drivers, for example) run on the Windows Phone OS. Some perturbation modules (Perturbation Proxy) are reused without change, whereby the host OS has its context manipulated which propagates into the phone VM (i.e., by altering the background context changes the context for the whole emulator, for example, to throttle network traffic).

Monitoring Modules. Under Windows 8, two monitors log system-wide and per-app performance counters through WMI. A third monitor collects crash data from the system event log and the local WER system. Finally, a fourth monitor hooks to the Event Tracing for Windows (ETW) service to capture the output of `msWriteProfilerMark` JavaScript method in Internet Explorer; which allows writing debug data from HTML5/JS Windows Store apps. Under Windows Phone 8, monitors also log crash data and system and app counters, but using OS-specific telemetry infra-structure.

Finally, under both platforms we enable an energy consumption estimation monitor based on WattsOn [25].

Perturbation Modules. Network perturbation is implemented on top of 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. To which we introduced real-time network property updates to emulate network transitions and cell-tower handoffs.

Modern VM managers expose settings for CPU resource allocation on a per-instance basis. By manipulating these processor settings, we can make use of three distinct CPU availability states: 20%, 50%, and 100%. To control the amount of available memory to apps, we use an internal tool that allows us to change available system commit memory. AppHost then uses this tool to create the required levels specified by the current test context. Finally, we implemented a virtual GPS driver to feed apps with spoofed coordinates and other GPS responses. Upon receiving a valid UDP command, our virtual GPS driver signs a state-updated event and a data-updated event to trigger the OS location services to refresh the geolocation data.

As noted, the Windows Phone 8 AppHost can leverage part of the perturbation layer (like CPU or network throttling), rather than implementing its own.

User Interaction Model. As Windows 8 Store apps and Windows Phone 8 apps are “page”-based, the app UI is represented in our model as a tree, where nodes represent the pages (or app states), and tree edges represent the UI element being invoked. 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. Due to platform differences in user interface automation APIs (i.e., W8 vs. WP8) we implement two native UI actuators, each using a OS-specific automation framework.

PerfAnalyzer. To report actionable feedback to developers PerfAnalyzer focuses on two areas: (1) System resource consumptions that are higher than expected under certain contexts (e.g., energy bugs); and, (2) Crashes linked to additional data towards identifying root causes.

Performance Outliers. To determine if the target app under a test case exhibits abnormal behaviour, we first find the set of similar apps over measurements from all other test cases (see §5). Then, we perform MeanDIST-based outlier detection [18], and see whether the test app is in the outlier group. Our current implementation assumes 5% of the population are outliers, which can be adjusted as we collect developers' feedback. Users can filter and rank outliers within a table view based on: (1) the frequency at which they occur; (2) the magnitude in difference to the comparison 'norm'; and (3) a particular metric type (e.g., network-related). Limited support for simple visualizations (e.g., bar charts) are also provided.

Crash Analysis. To collect and interpret crash data ConVirt connects with the Microsoft WER (Windows Error Reporting), a service on every Windows system to gather information about crashes for reporting and debugging. WER aggregates error reports likely originating from the same bug by a process of labeling and classifying crash data. More details on WER can be found in [10]. The resulting data is then correlated to changes in resource consumption prior to the crash for reporting.

6 Evaluation

This section is organized by the following major results: (1) ContextLib is able to effectively trade-off the rate of discovering app problems (e.g., crashes) while reducing the number of test cases required; (2) ContextPrioritizer can find up to 47% more crashes than the conventional baselines, with the same amount of time and computing resources; (3) ConVirt increases the number of crashes and performance outliers found over the current practice by a factor of 11× and 8×, respectively; and (4) we share lessons learned to help developers improve their apps.

6.1 Methodology

In the proceeding experiments, we use two datasets that are tested using our ConVirt prototype hosted in Azure (see §5), specifically: (1) 200 mobile Windows 8 Modern apps that target the Microsoft Surface Tablet PC (referred to subsequently as W8 apps); (2) 30 Windows Phone 8 Modern apps that target smartphones (referred to subsequently as WP8 apps.) All apps are free to download from the Microsoft Windows Store and Windows Phone Store, respectively.

To define our test case workload we first pick three representative cities from different continents with a large number of mobile device users: Seattle, London, and Beijing. Next, we utilize ContextLib to generate a total of 350 test cases. We use the standard context sources part of our current ContextLib implementation listed in

Resource Type	Description
Network	{datagrams/segments} {recieved/sent} per sec
	total TCP {connection/failure/active}
	total TCP {established/reset}
Memory	current amount of % {virtual/physical} memory used
	max amount of % {virtual/physical} memory used
	{current,max} amount of % {paged} memory used
CPU	% {processor/user} time
Disk	bytes {written/read} per sec

Table 2: System Resources used in ConVirt prototype

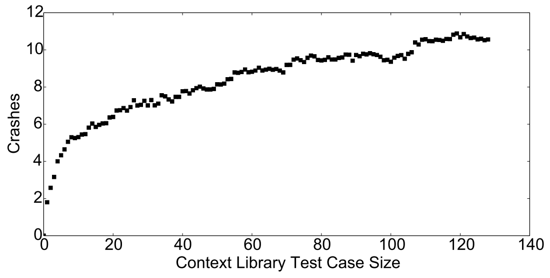

Figure 5: ContextLib Deep App Experiment.

Table 1. In addition, we add five hand-coded network profiles not present in the Open Signal database, namely: 802.11b, WCDMA, GPRS (out of range), GPRS (handoff), 4G – these are familiar network scenarios to developer. We limit the potential memory configurations generated by ContextLib to just two to focus more closely on the networking parameter space.

We configure ConVirt to test individual apps three times under each test case, with an individual test session under one test case being five minutes in duration. Table 2 lists the 19 system resource metrics and performance counters logged during our experiments.

6.2 Mobile Context Test Space Exploration

Our first experiments examines two key components of ConVirt, namely: ContextLib and ContextPrioritizer.

6.2.1 ContextLib

To investigate the effectiveness of the ContextLib, we perform the following two experiments. In each experiment we compare the detection rate of crashes assuming ConVirt is executed with different parameterizations of ContextLib. The objective is to observe how many crashes go undetected as the number of test cases is lowered. If ContextLib is effective, it will be able to maintain reasonably high crash detection rates even when the number of test cases drops dramatically.

Figure 5 shows an experiment using four representative Windows 8 Modern Apps – one from four distinct

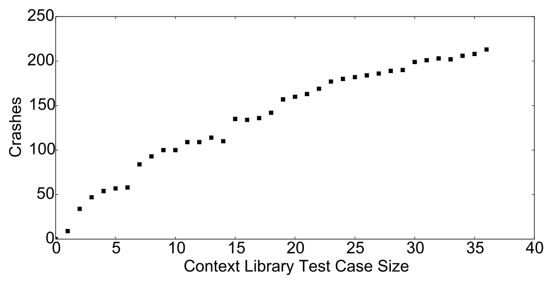

Figure 6: ContextLib Broad App Experiment.

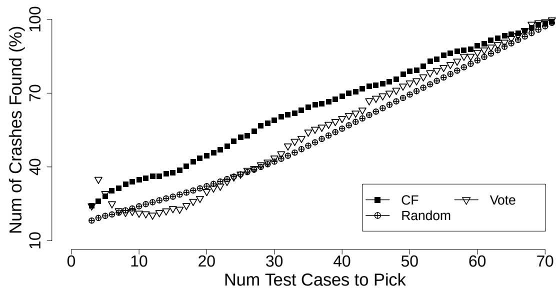

Figure 7: The number of crashes found by different techniques under different time budget.

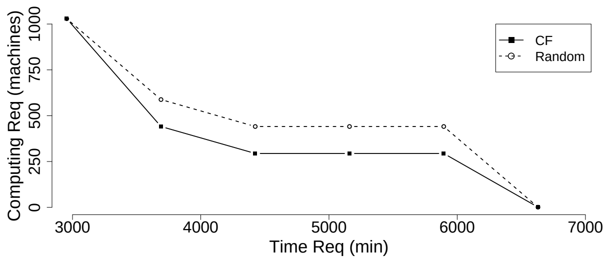

Figure 8: Feasible combinations of time and computing budget to find at most 10 crashes in each of the 200 Windows 8 apps.

app categories. By using fewer apps we are able in this experiment to use a large number of raw context, specifically: 500. Using ConVirt we test each app under each raw context. On average, each app crashes 16 times. We then repeat this experiment but use ContextLib to produce a library of representative test cases, and vary the size of the library. The figure reports the per-app average crash number. It shows by using ContextLib we are able to find a relatively high fraction of the crashes even as the number of test cases is lowered. For example, we find on average by using only 60 test cases (only 12% of the original total) we are able to find 50% of all crashes.

Figure 6 presents an experiment with the same methodology except for a much larger number of apps. We use 50 of the Windows 8 Modern Apps detailed in §6.1, but consequently must lower the number of raw contexts used to only 200. In this figure we report the total number of crashes in this app population. When testing all the raw contexts we find in total there are 312 crashes. Again we find ContextLib to be effective in discovering most of these app crashes with significantly fewer than the total raw contexts. Figure 6 shows ContextLib is able to find around 60% of all of these crashes using only $\approx$ 35 test cases.

6.2.2 ContextPrioritizer

The evaluation metric is the number of crashes found as **(1)** the time budget varies, and **(2)** the amount of available computing resource varies. We used three comparison baselines. First, *Oracle* has the complete knowledge of the measurements for all apps (including untested ones), and it represents the upper-bound. *Random* is a common approach that randomly picks an untested case to run at each step. Finally, *Vote* does not rely on finding apps with similar behavior, and uses test cases that yield the most crashes in all previously tested apps.

Time Budget. We start with the question: given sufficient time to exercise the target app under x test cases, which x cases would reveal the most number of crashes. We note that a one test case runs for a fixed duration (e.g., five minutes). Figure 7 shows the results with the Windows 8 dataset, and we highlight two observations.

First, ConVirt reports a higher number of crashes than Random and Vote. On average, ConVirt can find 30.90% and 28.88% more crashes than Random and Vote, respectively. We note the ConVirt exhibits the most gain when the time budget is relatively limited, or selecting less than 60% of all test cases. In fact, ConVirt can find up to 47.63% and 77.61% more crashes than Vote and Random, respectively. These results demonstrate the gain from two mobile app testing principles: learning from app test history, and considering only apps with similar behavior in resource consumption.

Second, as the time budget increases, the testing tool has more time to explore the entire testing space. Therefore, the probability of picking the set of test cases with crashes also increases. This suggests that the gain from using different techniques will eventually experience a diminishing return. In our dataset, the boundary is at around 60 test cases, or around 85.71% of all test cases.

Resource Tradeoff. An important observation is that, since app testing is highly parallelizable, multiple apps can be exercised at the same time on different machines. At the extreme with an 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 assumption is not practical in the real world, we calculate the speed up that ConVirt offers under various amounts of available computing resources.

Figure 8 illustrates combinations of computing resources and time required to find at most 10 crashes in each of the 200 Windows 8 Store apps. First, the figure shows 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 Fuzzer 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 6,630 minutes, or $> 90\%$ of the total possible time for testing all combinations of apps and test cases.

6.3 Aggregate App Context Testing

In our next set of experiments, we investigate crashes and performance outliers identified by ConVirt within a set of popular publicly available mobile apps.

Comparison Baseline. We compare ConVirt to a conventional UI automation based approach as a comparison baseline. This baseline represents current common practice for testing mobile apps. To implement a UI automation approach we use the default UIM already part of ConVirt (see §5.3). However, during app testing under the UI automation approach context is never perturbed.

To perform these experiments everything is repeated twice. Once using ConVirt and then repeated under the UI automation approach. Since the setup is identical for each run of the same app, differences in crashes and performance outliers detected are due to the inclusion of contextual fuzzing by ConVirt.

Summary of Findings. Overall, ConVirt is able to discover significantly more crashes (11.4×) and performance outliers (8.8×) than the baseline solution for W8 apps. And approximately 5.5× more crashes and 9× more outliers for WP8 apps. Furthermore, with ConVirt, 75 out of the 200 W8 apps tested observe at least one crash – in aggregate, ConVirt discovers a total of 1,170 crash incidents and 4,589 performance outliers. Similarly, 17 of the 30 WP8 apps crash 192 times and register 635 performance anomalies. This result is surprising, as these apps are in production and have passed testing.

Findings by Categories. Figure 9 shows the number of crashes and performance outliers categorized by app source code type: HTML-based vs. compiled managed code. The observation is that ConVirt 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×. It is important to note that all tested WP8 apps are managed code apps.

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 supports the use of contextual fuzzing in mobile app testing.

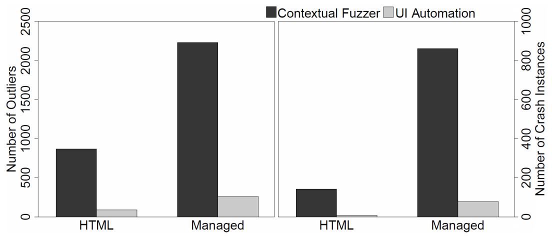

Figure 9: App performance outliers and crashes categorized by app source code type.

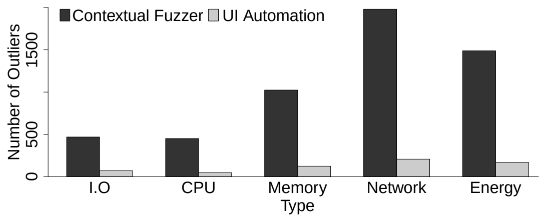

Figure 10: App performance outliers, by resource usage.

Figure 10 shows performance outliers broken down by resource type. The figure suggests that most outliers are network or energy related. Both disk activity (i.e., I/O) and CPU appear to have approximately the same number of performance outlier cases.

Time to Discovery: Comparison with Crash Report Analysis. In an effort to understand how the time taken for ConVirt to identify crash conditions compares to the analysis of user submitted crash reports we are given limited access to the WER backend database (see §5). We are provided with the internal WER reports for all the crashes ConVirt is able to find during our experiments. Our preliminary analysis of this data compares how long it took for these same crashes to appear in WER database after the app was released.

Overall, we find for W8 apps ConVirt takes, on average, only 11.8% (std dev 0.588) of the time needed by WER. Similarly, for WP8 apps this percentage is even lower at 1.3% (std dev 0.025). We speculate that the better performance of WP8 is related to their lower complexity and current lower adoption rates than W8.

6.4 Experiences and Case Studies

Finally, we highlight some identified problem scenarios that mobile app developers might be unfamiliar with, thus illustrating how ConVirt can help prevent ever more common context-related bugs and performance issues.

Location Bugs. Seemingly every mobile app today is location-aware. Unfortunately, location also introduces the opportunity for developer errors – *location bugs*. We now detail one representative location bug, concerning an app released by a US-based magazine publisher. Test results from ConVirt emulating location (i.e., GPS input and network conditions) discovered while the app was robust in the U.S it was very brittle in other countries. For example, we find the app is 50% more likely to crash under Chinese conditions compared to U.S conditions.

Network Transitions. Mobile devices experience network transitions frequently throughout the day. For example, handoffs from Wi-Fi to 3G when the user leaves their home, and in the reverse direction once they arrive at work. It is critical apps be robust to such transitions.

During tests, ConVirt uncovered a prototypical example demonstrating these issues in a popular Twitter app. We noticed frequent crashes under certain network conditions. By performing a larger number of test iterations we find that whenever a (simulated) user attempts to tweet during a network handoff from a "fast" (e.g., Wi-Fi) to "slow" (e.g., 2G) network, the app crashes nearly every time. Without the source code, it is hard to know the root cause of the issue. However, it is a clear example of the type of feedback that are possible using ConVirt.

Exception Handlers. During our experiments, we notice a group of music streaming apps some of which tend to crash with higher frequency on slow and lossy networks. By performing decompiled code analysis [32], we find that the less crash-prone music stream apps apply a significantly more comprehensive set of exception handlers around network-related system calls. Although not surprising, this highlights how ConVirt is a promising way to compare mobile apps at scale and develop new best practices for the community.

Unintended Outcomes from Sensor Use. ConVirt highlighted an interesting *energy bug* within a location tracking app. The app registers for location updates to be triggered whenever a minimum location displacement occurs. However, we find the app set the threshold to be very tight (we estimate $\approx$ 5m accuracy). During ConVirt testing we perturb the reported accuracy of the location estimate provided to the app (see §5.3). We find at typical location accuracy values ($\approx$ 25m) the app requests location estimates at a much higher frequency. As a result, the app consumes energy at much higher rates than expected. This unexpected outcome would be otherwise hard to recognize during non-context based testing.

	ConVirt	Monkey	ConVirt	Monkey
	(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	161	18	0	0
Social	43	5	116	5

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

7 Discussion

We discuss some overarching issues related to ConVirt.

Generality of the System. Besides app testing, our tools and algorithms can be applied to other scenarios. In privacy, outlier detection (with similarity tests) can identify apps that access or transmit personal data differently from the norm. In energy optimization, contextual fuzzing can help determine whether an app would experience significant performance degradation on slower but more energy-efficient radios.

Real-World Mobile Context Collection. While our system utilizes real-world data to emulate mobile contexts, we recognize that some datasets are difficult to collect. For example, an extensive, easily generalized, database regarding users app interaction is not yet available. We leave the problem of developing an expanded set of context sources as future work.

Context Emulation Limitations. While ConVirt currently exposes only coarse-grained hardware parameters (i.e., CPU clock speed and available memory), it can accommodate real devices in the test client pool to achieve hardware coverage. However, our system lacks support for sophisticated user gestures and low-level hardware, such as Wi-Fi energy states. We leave the support for an expanded perturbation layer as future work.

Applicability To Other Platforms. While our prototype runs on Windows 8 and Windows Phone 8, the core system ideas also work on platforms (e.g., Android). Because our design only makes a black-box assumption regarding access to app source code ConVirt is much easier to port to alternative mobile platforms.

8 Related Work

In the area of software testing, our work proposes to expand the testing space for mobile apps to include real-world context. Our results complement existing techniques, including static analysis and fault injection.

Mobile App Testing. There are industrial telemetry solutions on mobile devices such as Microsoft Windows Error Reporting (WER) [10]. AppInsight [28] adds significant visibility into the critical paths of asynchronous and multi-threaded mobile apps. Carat [5] focus on energy diagnosis by periodically uploading coarse-grained battery measurements and system status. Finally, a number of services analyze telemetry logs [8, 6, 4, 33]. Unlike ConVirt, these are post-release solutions, so users are exposed to app faults.

A number of proactive testing tools are available. However, most tools cover only a small subset of mobile contexts. First, Windows and Android offer specialized tools [15, 14, 22] and libraries [24] for custom UI automation solutions. Also, significant effort has been invested into generating UI input for testing with specific goals in mind (e.g., code coverage) [36, 16].

Second, some testing tools can emulate a limited number of predefined network conditions. This emulation can be controlled either manually [15, 23] or via scripts [20]. In contrast to ConVirt, these tools do not allow automatic fine-grained control over network parameters, nor accurate emulation of mobile network contexts such as network hand-offs (e.g., 3G to Wi-Fi).

State Space Exploration. The software testing community has proposed techniques to efficiently explore program state space, which mostly rely on hints extracted from the source code or test history of the target program.

Whitebox fuzzing is one technique that requires source code. For example, SAGE [12] tests for security bugs by generating test cases from code analysis. [1] explores code paths within mobile apps and reduces path explosion by merging redundant paths. Model checking is another popular technique, where the basic idea is to build models of the test target via knowledge of specification or code [17]. Model checking has also been used in other communities such as distributed systems [37]. Finally, [13] propose a test case prioritization scheme that exploits inter-app similarity between code statement execution patterns. However, because ConVirt does not require source code it is more broadly app compatible. Directed testing, e.g. DART [11], is a category of techniques that implement a feedback loop for the same app. ConVirt uses similar techniques, but across apps.

Simulating Real-world Conditions. ConVirt targets context emulation. Other domains, notably sensor networks, have also developed testing frameworks [21], incorporated energy simulation [29], and support execution emulation [34]. While ConVirt conceptually shares similarities with this work, its foundations are in mobile context not present in these domains, such as network transitions, mobility patterns, and hardware diversity.

9 Conclusion

This paper presents ConVirt, a testing framework that applies the *contextual fuzzing* approach to test mobile apps over an expanded mobile context space. Results from our cloud-based prototype suggest that ConVirt can find many more app performance problems than existing tools that consider none or a subset of the mobile context.

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

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

[9] 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/`.

[10] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *SOSP*. ACM, 2009.

[11] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[12] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, Mar. 2012.

[13] 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.

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

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

[16] 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.

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

[18] V. Hautamaki, I. Karkkainen, and P. Franti. Outlier detection using k-nearest neighbour graph. In *ICPR*, 2004.

[19] 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.

[20] 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.

[21] 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.

[22] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: an input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 224–234, New York, NY, USA, 2013. ACM.

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

[24] Microsoft. UI Automation Verify. `http://msdn.microsoft.com/en-us/library/windows/desktop/hh920986(v=vs.85).aspx`.

[25] 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.

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

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

[28] 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.

[29] V. Shnayder, M. Hempstead, B.-r. Chen, G. W. Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, pages 188–200, New York, NY, USA, 2004. ACM.

[30] 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`.

[31] 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`.

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

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

[34] 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.

[35] 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.

[36] 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.

[37] J. Yang, C. Sar, and D. Engler. Explode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 10–10, Berkeley, CA, USA, 2006. USENIX Association.