Caiipa: Automated Large-scale Mobile App Testing through Contextual Fuzzing

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

[‡]Microsoft Research [*]Delft University of Technology [⋆]University of Science and Technology of China
[†]Tsinghua University [◁]Shanghai Jiao Tong University [⋈]Harbin Institute of Technology

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 *Caiipa*, a cloud service for testing apps over an expanded mobile context space in a scalable way. It incorporates key techniques to make app testing more tractable, including a context test space prioritizer to quickly discover failure scenarios for each app. We have implemented Caiipa on a cluster of VMs and real devices that can each emulate various combinations of contexts for tablet and phone apps. We evaluate Caiipa by testing 265 commercially available mobile apps based on a comprehensive library of real-world conditions. Our results show that Caiipa leads to improvements of 11.1x and 8.4x in the number of crashes and performance bugs discovered compared to conventional UI-based automation (i.e., monkey-testing).

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Performance evaluation

Keywords

Mobile app testing; Contextual Fuzzing

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 [6]. In turn, 750 million Android and iOS apps are downloaded each week from 190 different countries [18]. 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 multiple form factors and screen sizes, in varied countries, across a multitude of carriers and networking technologies. Developers must test their applications across a full range of mobile operating contexts prior to an app release to ensure a high quality user experience. However, judging from the frequency of app failures and performance problems after release, it still seems a far off goal [31]. 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 [30]. As a result, 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, and poor performance and crashes are not tolerated.

Today, developers have only a limited set of tools to test their apps under different mobile contexts. Tools for collecting and analyzing data logs from already deployed apps (e.g., [5, 33, 7]) require them to be first released before problems can be corrected. Through limited-scale field tests (e.g., small beta releases or 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. Inducing issues via fault injection could be used to test consequences of specific situations. However, it requires specific tools to be available and expertise in choosing which issues to inject, while also being prone to the creation of inconsistencies [23] (i.e., artificial situations that may not be realistic).

One could use readily available platform simulators ([13, 20]) to test the app under a specific GPS location and network type, such as Wi-Fi. However, 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 mobile apps might be used. Knowledge of previous crashes (either from the developer's own experience or from data repositories) could guide the exploration, but those only cover previously observed scenarios and don't map well to performance analysis. Moreover, even if such data is available, one still has the problem of analysing it to identify the specific problematic contexts to use for testing; which has scalability issues if done manually and is noise prone if automatic (e.g., by association rule mining).

To address this challenge, we propose a new approach to mobile app testing named *contextual fuzzing* [10]. This technique incorporates two key advances over existing alternative methods such as simulators that simply round-robin over a limited sets of common conditions. First, apps are exposed

MobiCom'14, September 7-11, 2014, Maui, Hawaii, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2783-1/14/09 ...$15.00.
http://dx.doi.org/10.1145/2639108.2639131.

to a large-scale library of diverse contexts synthesized from the actual conditions observed in the wild. Thousands of contexts are considered; for example, the throughput and loss rates observed for Wi-Fi and 3G networks around the world – rather than only testing expected network behavior as quoted on technical specifications. Second, because of the wide-range of potential contexts to which an app can be exposed, a custom app-specific priority order of contexts is determined. Conditions expected to be highly relevant to the tested app are applied first in favor of less relevant ones; for example, an app might be sensitive to poor network connectivity or low memory conditions and so this category of contexts will be prioritized over CPU- or GPS-oriented tests. These two core techniques combine to enable contextual fuzzing to highlight potential problems before an app is even released to the public.

To demonstrate the power of contextual fuzzing, we design and implement *Caiipa*[1] – a prototype cloud service that can automatically probe mobile apps in search of performance issues and crash scenarios caused by certain mobile contexts. Developers are able to test their apps by simply providing an app binary to the Caiipa service. Apps are then 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 for context-related issues. A summary report is then generated detailing the problems observed, the trigger conditions, and how they can be reproduced. Detailing the scenarios in which issues happen is especially important for developers to be able to decide what actions to take.

This paper makes the following contributions:

- We extend *contextual fuzzing* – a new approach for mobile app testing – by introducing techniques for synthesizing a comprehensive library of context stress tests from readily available telemetry data sources. This enables developers to identify context-related performance issues and sources of app crashes prior to releasing their apps (§ 2).
- We develop techniques for the scalable exploration of the mobile context space. Specifically, we propose a learning algorithm that leverages similarities between apps to identify which conditions will impact previously unseen apps using observations from those apps already tested (§ 3).
- We design a cloud service, called Caiipa, to which app developers and testers can submit their apps to identify context-related problems. This service consists of a pool of virtual machines and real hardware devices that support either smartphone or tablet apps, while emulating a variety of complex context combinations (§ 4).

We evaluate Caiipa using a workload of 235 Windows 8 Store apps (for tablets) and 30 Windows Phone 8 apps based on a wide-ranging library of 10,504 real-world contexts. Our experiments show exploring the mobile context space leads to an 11.1x 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 at most around 2.6% of the time (for 94.7% of the crashes) compared to waiting for the same bug to appear within a commercially available crash reporting system. Finally, Caiipa is able to identify 8.4x more performance issues compared to standard monkeying.

[1] Caiipa means "island of monkeys" in Tupi-Guarani. We adopt this name because the contextual fuzzing approach provides a diverse environment for conventional UI-testing to be performed.

2. CONTEXTUAL FUZZING IN CAIIPA

Caiipa targets the needs of two types of users:

- **App Developers** who use Caiipa 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** 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 and cope with reviewing thousands of apps per week[2].

Consequently, Caiipa has the following design requirements:

- First, and most importantly, it needs to be *comprehensive.* Since testing all possible contexts, e.g. every possible lat-long location, would take a long time, Caiipa needs to come up with cases that are representative of the real-world, and where apps often 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, Caiipa needs to quickly determine the most relevant test cases to apply to submitted apps.
- Third, it needs to be able to detect unexpected problems, rather than perform spot tests for specific failures (e.g., common bugs that cause many failures but can easily be found by targeted approaches). In order to find these unforeseen issues, the conditions under which they might occur need to be tested for.
- Fourth, it should be *black box.* We cannot always assume access to app source code. Although instrumentation of binaries has been shown to work for some Windows Phone apps [28], even for a given platform there are many types of apps (e.g., managed, HTML, native). Also, we want methods that can generalize to Windows, Android, and iOS devices.

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 are no existing ways to systematically 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 (e.g., 84% of Android apps in a large-scale survey request networking permissions [37]). 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

[2] 8,500 app are reviewed weekly by the Apple App Store [6].

operator [17], 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 [22].

Device Heterogeneity. Variations in devices require an app to perform across different *chipset, memory, CPU, screen size, resolution, and 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 [26]. We note that devices in the wild can experience low memory states or patterns of low CPU availability different from developer expectations, e.g. a camera briefly needs more memory, and this can affect user experience on a low-end device.

Sensor Input. Sensors are commonly used by apps, for instance, surveys of Android marketplaces indicate between 22% and 42% of apps localize through GPS or Wi-Fi [34, 37]. These 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 app 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 Caiipa Overview

Caiipa is implemented as a cloud service with the components shown in Figure 1. App developers (or distributors) submit binaries of their apps (i.e. an app packages). Caiipa then runs the app under a controlled environment (AppHost) either in an emulator or on real hardware, while simulating various contexts (networks, carriers, locations, etc.) using the Perturbation Layer. It continuously monitors the performance of the app under each context; and PerfAnalyzer outputs a report with all cases where it found the app to have a bug, where a bug can be a crash, a performance anomaly (e.g., CPU), 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 contexts 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 preprocessing step, which we run periodically on crowdsourced data as explained in § 4. Second, ContextPrioritizer sorts different context combinations, and runs those with higher likelihood of detecting failures before others. This helps Caiipa to quickly discover the problematic scenarios. AppHost Dispatcher reads the prioritized test cases from ContextPrioritizer and sends each test to the appropriate AppHost.

2.3 Caiipa System Components

As shown in Figure 1, Caiipa consists of five components: (1) ContextLib, (2) ContextPrioritizer, (3) AppHost Dispatcher, (4) AppHost, and (5) PerfAnalyzer.

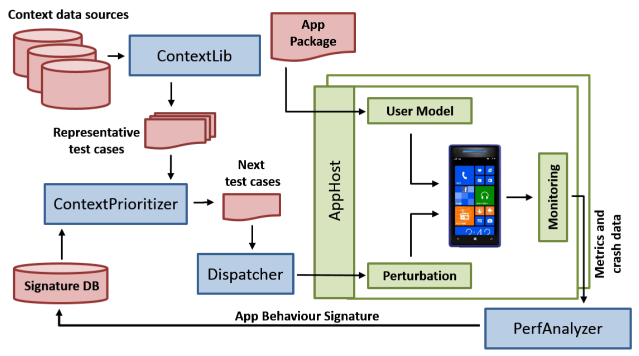

Figure 1: Caiipa system diagram.

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). ContextLib is populated using datasets collected from real devices (e.g., OpenSignal [25] and WER [7]) in addition to (1) challenging mobile contexts defined by domain experts; and, (2) a small number of test cases designed to target key device categories (see AppHost Dispatcher for more). Because context datasets are typically large-scale (OpenSignal, for instance, contains millions of network observations) and because not every raw context in a dataset will be an effective test condition, ContextLib performs a two-stage test case generation process that first filters redundant contexts that do not significantly impact app behavior (i.e., resource consumption); before then generating test cases not from just individual contexts (e.g., a network condition), but also based on context transitions (e.g., 3G to 4G) and the co-occurrence contexts of different types (e.g., fast network with low memory).

ContextPrioritizer. ContextPrioritizer determines the order in which the contexts from ContextLib should be performed, and communicates this ordering to AppHost Dispatchers. 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 determine this prioritization. Both prioritization and similarity computation are online processes. 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 impact similar apps are prioritized (e.g., an app that behaves similarly to streaming apps would have network contexts prioritized).

AppHost Dispatcher. Apps run within a controlled environment called AppHost. AppHosts can be instantiated either on a VM or on real hardware. The AppHost Dispatcher makes this decision based on the test case workload generated by ContextPrioritizer. In order to scale testing, most test cases are run on a VM, which AppHost Dispatcher selects from a AppHost pool. However, AppHosts running on real devices are used by Caiipa in two situations. First, in the event the app is native or mixed code and unable to be run on an emulator[3], or when an app uses specific hardware that is difficult to emulate, such as the camera or NFC interface. Second, when ContextPrioritizer selects any of the small number of device-related test cases included in ContextLib (such as the case corresponding to a low-end smartphone that causes testing to be done using a device representative of that category).

AppHost. The AppHost has three main functions:

UI Automation (Monkeying). We use a User Interaction Model (see §4) 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. As there is no dependency between our model and other Caiipa components, it can be replaced by a more elaborate one without negative impact. Moreover, we also allow developers to customize its weights.

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 various kernel hooks or drivers (4). This layer is extensible and new contexts can be added as required.

Monitoring. During test execution AppHost closely records 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 UI 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 target app and (2) an app group that are similar to the target app.

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, it can find the source code location where specific issues were triggered.

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 from 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 easily visible.

[3] This limitation applies only to ARM binaries.

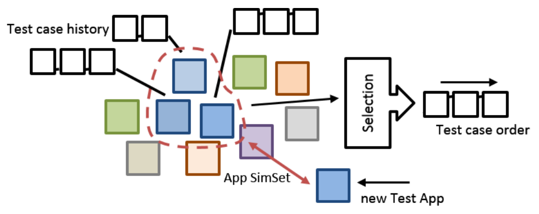

Figure 2: ContextPrioritizer dataflow.

3. PRIORITIZING TEST CASES

ContextLib maintains a test case collection that runs into the thousands. At this scale, the latency and computational overhead of performing the full test suite becomes prohibitive to users. To address this challenge, ContextPrioritizer finds 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 exercising only a fraction of the entire ContextLib can still discover important context-related crashes and performance bugs. We note that its use is optional, and users can also manually specify the order of test cases.

Overview. Figure 2 presents the dataflow of ContextPrioritizer. Within the overall architecture of Caiipa, the role of ContextPrioritizer is to determine the next batch of test cases (chosen from ContextLib) to be applied to the user-provided test app.

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 should perform similarly to the current test app (referred to as a `AppSimSet`). And then, it examines the test case history of each member of the `AppSimSet`, identifying potentially "problematic" test cases that may result in faults. These problematic test cases are then prioritized ahead of others in an effort to increase the efficiency by which problems in the current app are discovered.

While we informally refer to app(s) in this description (e.g., test app, or apps in a `AppSimSet`), more precisely this term points to an *app package* that includes both (1) a mobile app and (2) an instance of a User Interaction Model (see §4). This pairing is necessary because the code path and resource usage of an app are highly sensitive to the user input. Conceptually, an app package represents a particular usage scenario within an app.

Algorithm 1 details precisely ContextPrioritizer operation. In what follows we explain the key algorithmic phases.

App Similarity Set. In addition to the sheer number of potential test cases, context fuzzing is complicated by the fact that a given app will likely only be 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 sporadically uses the network for checking program updates may be fairly insensitive to many network related test cases; wasting resources 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 cer-

Algorithm 1: ContextPrioritizer

```
Input  : Prioritization Request for $\texttt{App}_i$
         $\texttt{Len}_i$: Length of required $\texttt{TestSeq}_i$
Output: $\texttt{TestSeq}_i$ for $\texttt{App}_i$
AppSimSet ⟵ {}                    /* compute AppSimSet */
For ∀$\texttt{App}_j$ ∈ AppHist
    ResSimSet ⟵ {}
    For ∀$\texttt{Res}_k$ ∈ ResSet
        If KS.ResTestIsTrue($\texttt{App}_i$, $\texttt{App}_j$,$\texttt{Res}_k$)
            ResSimSet ⟵ $\texttt{Res}_k$
        EndIf
    End
    If |ResSimSet| / |ResSet| ≥ $\texttt{KS}_{thres}$
        AppSimSet ⟵ $\texttt{App}_j$
    EndIf
End
ResCrash ⟵ {}                     /* compute ResCrash */
For ∀$\texttt{App}_k$ ∈ AppSimSet
    For ∀$\texttt{Crash}_l$ ∈ AppHist[$\texttt{App}_k$]
        ResCatSet ⟵ ResCat($\texttt{Crash}_l$, $\texttt{CAT}_{thres}$)
        For ∀$\texttt{Res}_j$ ∈ ResSet
            If $\texttt{Res}_j$ ∈ ResCatSet
                ResCrash[$\texttt{Res}_j$] ⟵ $\texttt{Crash}_l$
            EndIf
        End
    End
End
$\texttt{TestSeq}_i$ ⟵ {}    /* select Test Case Sequence */
While |$\texttt{TestSeq}_i$| ≤ $\texttt{Len}_i$
    CrashVote ⟵ {}            /* voting - Tier One */
    For ∀ $\texttt{Res}_j$ ∈ ResCrash
        Temp ⟵ TopN(ResCrash[$\texttt{Res}_j$],$\texttt{RES}_{thres}$)
        CrashVote ⟵ Temp
        ResCrash[$\texttt{Res}_j$] − Temp
    End
    $\texttt{TestSeq}_i$ ⟵ TopN(CrashVote,1)  /* Tier Two */
End
```

tain contexts (e.g., network-related test cases) to API calls (e.g., network-related APIs) – would have been confused by the prior example.

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. Then, it prioritizes other test cases over network-related ones, with the insight of the potential network insensitivity.

The building block operation within ContextPrioritizer is a *pairwise similarity comparison* between a new test app, and a previously tested app (called `KS.ResTestIsTrue()` in Algorithm 1). This is done for each system resource metric while both apps were exposed to the same test case (i.e., 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 apply the Kolmogorov-Smirnov (K-S) test [4], with an α of 0.05. The outcome of this test is binary, either the distributions are found to be the same or 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 `AppSimSet` for each new test app. During this process, ContextPrioritizer considers a collection of j resource metrics – called `ResSet` (listed in Table 2). For each resource metric, the pairwise tests are performed between the current test app and all prior test apps (i.e., `AppHist`.) 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 `AppSimSet`, it must pass the statistical comparison test a certain percentage of times ($\texttt{KS}_{thres}$). We empirically set $\texttt{KS}_{thres} = 65\%$, as it is not too loose to differentiate cases such as image-centric and video-centric news apps, and not too restrictive to result in similar clusters with virtually interchangeable members.

Importantly, no comparison between the current and past 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 executed before prioritization. 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 Caiipa starts to run.

Test Case History Categorization. As many crashes are caused by abnormal resource usage (e.g., excessive memory), they can be tied to resource metrics. The motivation for categorization of test case history is two-fold. First, it allows non-context related crashes (e.g., division by zero) to be ignored during prioritization. Second, the frequency of crashes fluctuates significantly between resource categories (e.g., network-related crashes are much more common that hardware-related ones). As a result, fair comparisons for later Test Case Sequence selection are only possible *intra*-category instead of inter-category.

The intuition supporting the use of abnormal resource usage to categorize test case history is that a fall or rise in the resource metric consumption immediately prior to a crash is a signal of a likely cause. As we find that most real-world contexts have an immediate impact on app behavior, we compare the resource consumption of the last three user clicks to that of the rest. Typically, the crash is then tied to the resource metrics with the largest gradient (either positive or negative). This procedure is called `ResCat()` in Algorithm 1. Due to potential inaccuracies, during ContextPrioritizer execution the top $\texttt{CAT}_{thres}$ most likely resource metrics are tied to each crash (by default $\texttt{CAT}_{thres}$ is set to 3).

Test Case Sequence Selection. ContextPrioritizer uses a two-tiered voting process to arrive at a sequence of test cases ($\texttt{TestSeq}_i$) to be applied to the test app ($\texttt{App}_i$). The tiered process ensures no single resource category dominates test case selection.

[4] We set k to 3 and find the best test cases. As test case creation is data-driven, we manually name them: `GPRS*`, `802.11b*`, and `4G*`, for their similarity to conditions for those network types.

Context Dimension	Raw Entries	Description	Source
Network	9,500+	Cellular in Locations Wi-Fi Hotspot Cellular Operators	Open Signal
Platform	220/23 (W8/WP8)	CPU Utilization Ranges Memory Ranges	Watson
Sensor	300	Locations	FourSquare
Others	49	Transitions, Extreme Cases	Hand-picked

Table 1: Context Library.

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 in terms of weighted crashes observed in past test apps contained within `AppSimSet` – in Algorithm 1 this is computed by `TopN()`.

The differences between the two tiers are in the pools of context-related crashes considered. For each past test app, the first tier looks at which test case crashes happened for each resource metric (`ResCrash`). Then, the top $\texttt{RES}_{thres}$ popular test cases is picked for each app, and aggregated at the second tier. A single test case is chosen, but this process is repeated for a test case sequence. We set $\texttt{RES}_{thres}$ to be 10 in our current Caiipa implementation.

4. CAIIPA IMPLEMENTATION

This section presents the implemented testing framework components (see Figure 1). The entire Caiipa prototype consists of $\approx$ 31k lines of code, broken into: ContextLib 1.9kloc; ContextPrioritizer 2.2kloc; AppHost 20kloc; and PerfAnalyzer 6.8kloc.

ContextLib. Our current library contains 10,504 raw contexts, as summarized in Table 1. The majority of the ContextLib is populated by Open Signal [25] (a global public dataset of crowd-sourced cellular measurements) as an external data source. We limit the number of cities to 400, which include 50 mobile carriers. Additional tests (e.g., memory and CPU) can be sourced from telemetry databases. Finally, a number of hard-coded raw ContextLib records are included for: (1) challenging scenarios that are not available from current context sources – examples include sudden drops in available memory; in addition to (2) a small number of device-specific test cases (e.g., low-end smartphone, ARM-architecture tablet) that correspond to actual devices in our testbed.

Figure 3 illustrates the main components of ContextLib. It pulls raw data from various 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. The two algorithmic components – (C) Redundancy Filtering and (E) Test Case Generation – shown Figure 3 operate as follows.

Redundancy Filtering. As a first step, we remove duplicate entries from the raw contexts (rcs). However, this step by itself is not sufficient. ContextLib also filters rcs if apps do not demonstrate large changes in *system resource usage* relative to other contexts. The underlying assumption is that large changes in resources are an indicator that the app is being exercised in significantly different ways by a con-

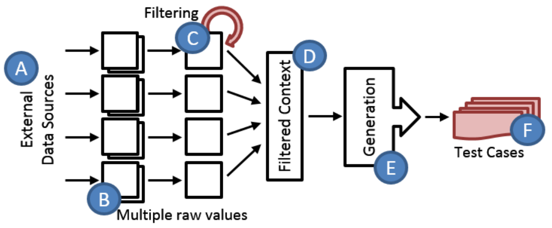

Figure 3: ContextLib dataflow, showing the components and processes: (A) Context Sources; (B) Raw Context; (C) Redundancy Filtering; (D) Filtered Context; (E) Test Case Generation; and, (F) Test Cases.

text[5]. Resource-based filtering is performed separately for each category of context (e.g., network, memory, CPU). The goal is to arrive at $\texttt{f}dom_i$ for each category – a collection of key context conditions (e.g., a problematic or common memory level) within the i^{th} context type (e.g., memory-related context) that often cause apps to behave (i.e., consume resources) differently.

To find $\texttt{f}dom_i$ we begin by using training examples – that is, pairs of rc and corresponding normalized resource usage. Training examples are collected in two ways: (1) by periodically testing a representative workload of popular mobile apps; and (2) via the output of app testing from standard Caiipa operation. We perform matrix decomposition on this training data to learn a projection matrix $\texttt{dom.i}_{prj}$ for each context category. This matrix maps contexts (e.g., a change in device memory availability) to a n-dimension resource vector space of app usage parameterized by the same 19 system metrics used by ContextPrioritizer (see Table 2).

Initially each $\texttt{f}dom_i$ contains all rcs (i.e., the 10,504 context-only records available from context sources) projected using $\texttt{dom.i}_{prj}$ and so represented as 19-dimension resource usage vectors. Next, dimensionality reduction is performed on each $\texttt{f}dom_i$ using Multi-dimensional Scaling [4] (MDS). This removes irrelevant dimensions for a particular context category (e.g., I/O-related system metrics might only be weakly relevant within a networking-related context category). Finally, to remove rcs with correlated system metrics (and thus redundant) we apply a density-based clustering algorithm – DBSCAN [4][6].

Test Case Generation. Filtered contexts are used to form three types of test cases. First, test cases based solely on individual filtered contexts themselves. Second, contexts fused together to form *transition* context test cases within the same domain (e.g., a transition from one memory condition to another). Third, *multi-domain* context test cases made from the combination of contexts that test the co-occurrence of multiple types of context conditions (e.g., simultaneously-occurring specific network and CPU contexts).

Generation of test case types is as follows. First, all filtered contexts are automatically used to generate test cases – a trivial step. Next, candidate *transition* and *multi-domain* test cases are generated. For each filtered domain ($\texttt{f}dom_i$) a companion *transition* domain is created. For example, the network domain is paired with a network-*transitions* domain. Transition domains contain entries capturing all pos-

[5] Under the expectation that the same UI steps are being repeated.

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

sible transitions from one context to another, within that domain (i.e., $\mathtt{f}dom_i \times \mathtt{f}dom_i$). This step is needed because context transitions (such as, switching from Wi-Fi to 3G network connectivity) are common situations for mobile apps to fail. A Cartesian product is then performed across all filtered domains, including the new transition ones ($\mathtt{f}dom_1 \times \cdots \mathtt{f}dom_n$). This creates candidate test cases that span context domains. However, a side-effect of this process is that new redundant *transition* and *multi-domain* test cases can be introduced. To remove these we perform one final round of redundancy filtering over all test cases; the output of this filtering is the final set of test cases. In the end, the number of test cases (i.e., ContextLib size) is determined by clustering parameters that influence both phases of ContextLib. For DBSCAN and MDS these principally are dim and $MinPts$ [4] but equivalent parameters exist in alternative clustering algorithms. In our implementation these values are empirically set and default to $\{dim : 5, MinPts : 15\}$.

AppHost Dispatcher. AppHosts run either on VMs or on real devices. VMs are used for the majority of tests for testbed scalability and are maintained as a pool of AppHosts on Microsoft Azure. To support the testing of device-sensitive context, we use four categories of devices in our testbed – two smartphones (Lumia 520 and 1020) and two tablets (Microsoft Surface RT, Samsung 700T1A Slate) – each corresponding to specific device test cases in ContextLib.

AppHost Dispatcher allocates test cases picked by ContextPrioritizer for an app to either the VM pool or specific hardware. Tests are assigned to a device if they require a certain hardware architecture (e.g., requiring ARM to run), or when the app uses certain components flagged as requiring real-hardware. In such cases all tests picked for the app are performed on a real device with the AppHost perturbing, for example, the network or GPS, to examine different contexts. Alternatively, for apps that are otherwise able to be run on an emulator, certain test cases may still be performed on hardware, with the remainder being executed on emulators. This occurs when ContextPrioritizer includes any of the device-specific test cases (e.g., low-end smartphone) in the collection of tests picked for the app.

The Dispatcher manages each AppHost node (both VM- and device-based nodes) via an RPC-like framework.

AppHost. Each AppHost is controlled by the AppHost Dispatcher. AppHosts support both Windows 8 (W8) and Windows Phone 8 (WP8) 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 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 W8 app, the controller and all other modules run on the target OS. This allows W8 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 WP8 apps, an inner host running the WP8 OS is used. WP8 apps run inside this inner host (a WP8 VM) and are exercised and monitored with phone-specific modules. The W8 Controller and Data Manager still run unchanged. But AppHost sub-modules (e.g., Daemon, the

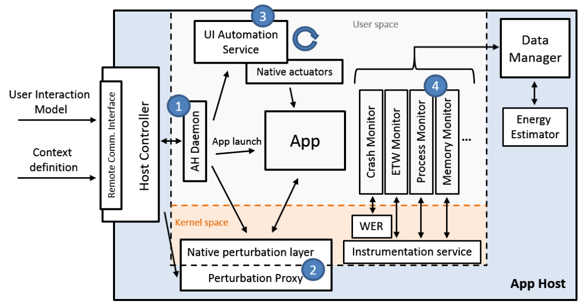

Figure 4: Implementation components of AppHost.

native layer of the Perturbation service) run on the Windows Phone OS. Some perturbation modules (Perturbation Proxy) are reused without change, whereby the outer host OS has its context manipulated which propagates into the inner phone VM.

Monitoring Modules. DataManager instantiates the appropriate monitors to collect data depending on app type and platform. Under W8, 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 WP8, monitors also log crash data and system and app counters, but use OS-specific infrastructure. Finally, under both platforms we enable an energy consumption estimation monitor based on WattsOn [22].

Perturbation Modules. Network perturbation is implemented on top of Network Emulator for Windows Toolkit (NEWT)[7], a kernel-space network driver. NEWT exposes four 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 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 AppHost to change available system memory. Finally, we implemented a virtual GPS driver to feed apps with spoofed coordinates and other GPS responses. Our UDP-controlled virtual GPS driver raises a state-updated event and a data-updated event to trigger the OS location services to refresh the geolocation data.

As noted, the WP8 AppHost can re-use parts of the W8 perturbation layer (like CPU or network throttling), rather than implementing its own.

User Interaction Model. As W8 Store apps and WP8 apps are "page"-based, the app UI is represented as a tree, where nodes represent the pages (or app states), and edges represent the invoke-able UI elements. We provide a stand alone authoring tool allowing a user to assign a weight to each UI element. Higher weights indicate a higher probability of a particular UI element being invoked. However, by default – if the authoring tool is not used – each UI element has

[7] Part of the Microsoft Visual Studio package.

an equal weight. Due to platform differences in UI 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 consumption that is higher than expected under certain contexts (e.g., energy bugs); and, (2) Crashes linked to metadata assisting in root cause analysis.

Performance Outliers. To determine if the target app being tested exhibits abnormal behavior, we first find the set of similar apps over measurements from previous tests (see §3). Then, we perform MeanDIST-based outlier detection [16], 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 by the user. 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.

Crash Analysis. To collect and interpret crash data, Caiipa connects to 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 (see [7] for details). The resulting data is then correlated to changes in resource consumption prior to the crash for reporting.

5. EVALUATION

This section is organized by the following major results: (1) ContextLib is able to effectively balance the trade-off between increasing the rate of discovering app problems (e.g., crashes and freezes) 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) Caiipa 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 future mobile app development.

5.1 Methodology

In the proceeding experiments, we use two datasets that are tested using our Caiipa prototype hosted in Azure (see §4), specifically: (1) 235 mobile Windows 8 Store apps that target tablet devices (hereafter W8 apps); (2) 30 Windows Phone 8 apps that target smartphones (hereafter WP8 apps). All apps are free to download from the Microsoft Windows Store and Windows Phone Store, respectively; selected from the list of top-free non-game downloads.

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 Table 1. In addition, we add five hand-coded network profiles not present in the OpenSignal database, namely: `802.11b`, `WCDMA`, and `4G` – as familiar network scenarios developers can use as reference points; and `GPRS out of range` and `GPRS handoff`, as scenarios involving cell tower changes. We limit the potential memory configurations generated by ContextLib to just two to focus more closely on the networking parameter space.

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 the Caiipa prototype.

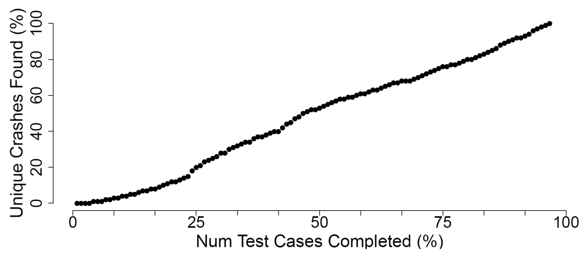

Figure 5: CDF of unique crashes found when incrementally adding test cases from ContextLib.

Finally, we include four device-specific test cases tied to real hardware: two phones (Lumia 520 and 1020) and two tablets (Microsoft Surface RT, Samsung 700T1A), selected as representatives of device classes.

We configure Caiipa 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. Finally, we use the term *crash* to represent both app freezes and unexpected app terminations.

5.2 Mobile Context Test Space Exploration

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

ContextLib. We begin by examining key questions regarding test cases in our ContextLib design: Is a large-scale ContextLib really necessary, or are many of our tests in ContextLib redundant? And, how effective is ContextLib in capturing the same detected crashes if exhaustively testing a much larger volume of raw context conditions?

To highlight any potentially redundant test cases in ContextLib, we perform an experiment where we test each app (both W8 and WP8) while withholding differing fractions of the 350-tests in the library. If unnecessary test cases are present in ContextLib we should see certain test cases being responsible for a disproportionate amount of the crashes.

Figure 5 compares the fraction of *unique* crashes found by Caiipa against the fraction of ContextLib made available during testing. This figure suggests that each test case observes, in general, new additional unique crashes. This observation justifies that the entire ContextLib we use is necessary for testing. In fact, Figure 5 has a reasonably constant slope, which also implies that all test cases are equally important in testing for app crashes. We find there is no small subset of tests in ContextLib that are significantly more responsible for detecting more crashes than others. Therefore, it is not possible to find a significant fraction of the different types of crashes we find with Caiipa by using only a few common failure-causing contexts.

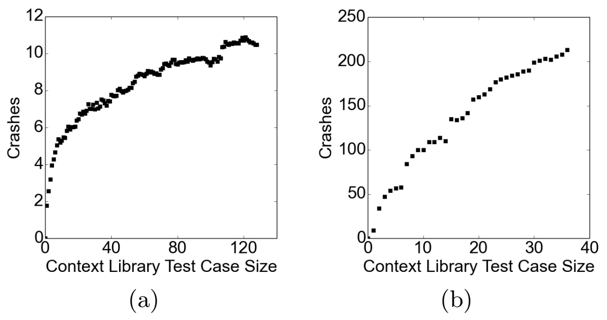

Figure 6: Context Library filtering performance: a) 500 raw contexts for 4 apps; b) 200 raw contexts for 50 apps.

Next, to further investigate the effectiveness of the ContextLib techniques, we perform the following two experiments. In each experiment, a selection of apps is initially tested under a set of raw contexts taken directly from our context sources (detailed in §4). We then repeat the experiment using a experiment-specific ContextLib of varying size synthesized from the same set of raw contexts (not the full ContextLib used by Caiipa). The objective is to observe how many crashes identified with the full-set of raw contexts later go undetected as the number of test cases in the ContextLib is lowered. If ContextLib techniques are effective, it will be able to maintain reasonably high crash detection rates even when the number of test cases drops dramatically.

Figure 6(a) shows an experiment using four representative W8 apps – one from four distinct app categories. By using fewer apps we are able in this experiment to use a large number of raw context, specifically: 500. Using Caiipa, we test each app under each raw context and on average, each app crashes 16 times. The figure reports the per-app average crash number. It shows that by using ContextLib we are able to find a relatively high fraction of the crashes with the raw contexts 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 of 500) we are able to find 50% of all crashes.

Figure 6(b) presents an experiment with the same methodology except for a much larger number of apps. We use 50 of the W8 apps detailed in §5.3, 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 a total of 312 crashes. Again we find ContextLib to be effective in discovering most of these app crashes with significantly fewer than the total raw contexts. As one example, Figure 6(b) shows it is able to find around 60% of all of these crashes using only ≈ 35 test cases.

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.

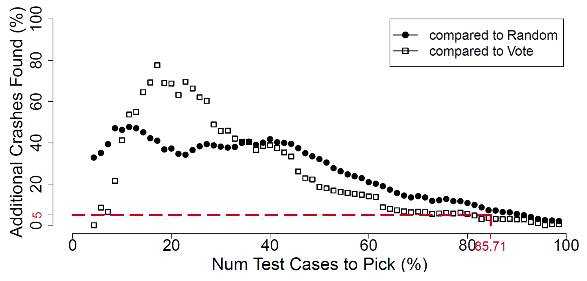

Figure 7: % more crashes that ContextPrioritizer finds given a time budget. 41.1% and 77.6% more crashes than Random and Vote, respectively, within 1-hour. Its gain over baselines falls below 5% only after exploring 85.71% of the entire test space.

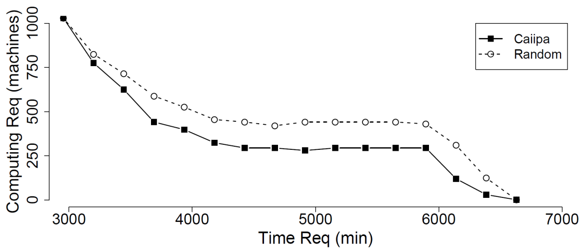

Figure 8: Feasible combinations of time and computing budget to try to find at least 10 crashes in each of the 235 tablet apps. Caiipa uses much less time and computing resources.

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

Second, as the time budget increases, testing tools can run through more test cases. This implies that the probability of picking the set of test cases that cause crashes also increases, regardless of the technique used. Therefore, the gain from using different techniques will eventually experience a diminishing return. For example, the dotted lines in Figure 7 show that the gain from ContextPrioritizer falls below 5% when there is enough time budget to explore at least 85.71% of the test space. Assuming 85.71% of our context library of 350 test cases, running five minutes for each test case would take a total of 24.99 hours to complete.

Resource Tradeoff. We note 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 Caiipa offers under various amount of available resources.

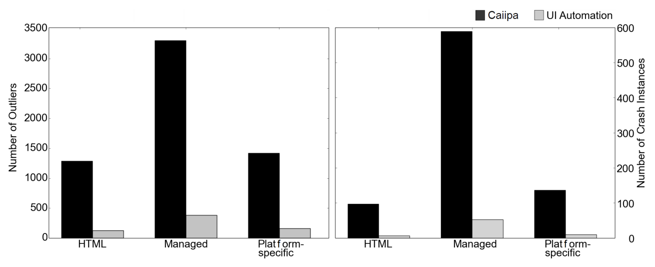

Figure 9: Performance outliers and **unique** crashes by type.

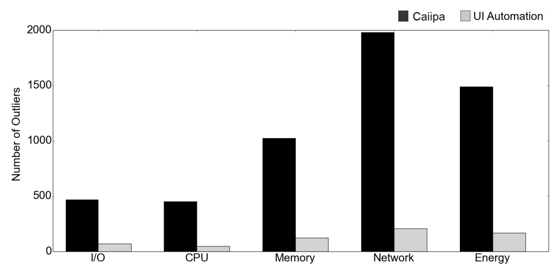

Figure 10: App performance outliers, by resource usage.

Figure 8 illustrates combinations of computing resources and time required to find at least 10 crashes in each of the 235 Windows 8 Store apps. For apps that have less than 10 crashes, we run through all test cases. 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, Caiipa can reach the goal faster and with fewer machines. For example, to find **all** crashes as set in our target within a time limit of 4425 minutes, Caiipa needs 294 machines – 33% less than the nearest benchmark. Finally, in this dataset, 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.

5.3 Aggregate App Context Testing

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

Comparison Baseline. We compare Caiipa to a conventional UI automation 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 Caiipa (see §4). However, during app testing under the UI automation approach context is not perturbed.

To perform these experiments all steps are repeated twice. Once using Caiipa and then repeated under UI automation. 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 Caiipa.

Summary of Findings. Overall, Caiipa 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 Caiipa, 107 out of the 235 W8 apps tested observe at least one crash – in aggregate, Caiipa discovers 661 **unique** crash incidents (from a total of 9,103 crashes) and 4,589 performance outliers. Similarly, 17 of the 30 WP8 apps crash 197 times (26 **unique** crashes) and register 635 performance anomalies. Crash uniqueness is defined by error code and stack trace. This result is somewhat surprising, as these apps are in production and already presumedly went through testing.

Findings by Categories. Figure 9 shows the number of crashes and performance outliers categorized by app source code type: HTML-based, managed code, and apps containing platform-specific code (native or mixed code apps). The observation is that Caiipa is able to identify significantly more potential app problems across categories. For example, in all categories, this difference in the number of outliers found is a factor of approximately 8×.

Figure 10 shows performance outliers broken down by resource type. The figure suggests that both disk activity (i.e., I/O) and CPU appear to have approximately the same number of performance outlier cases. Interestingly, most outliers are network or energy related. From the top energy outliers, we observe that the root cause of high energy consumption correlates with the underlying network condition. Under lossy network conditions, the cause is typically connection resets. Under reasonably good network conditions, the cause is typically the amount of incoming network traffic.

Time to Discovery. In an effort to understand how the time taken for Caiipa to identify crash conditions compares to the analysis of user submitted crash reports we looked into the WER backend database (see 4). We analyzed the internal WER reports for all the crashes Caiipa 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, for W8 apps, Caiipa takes, on average, only 11.8% (std dev 0.588) of the time needed by WER (2.6% of the time if considering 94.7% of the crash population). Similarly, for WP8 apps this percentage is even lower at 1.3% (std dev 0.025). By closely looking at the data, we see that a few W8 apps quickly crashed in the wild, with crash reports being submitted less than one hour after release, thus reducing the time difference to our proactive approach. We speculate that part of the much better performance in WP8 apps is related to their lower complexity and current lower adoption rates when compared to W8 apps.

5.4 Experiences and Case Studies

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

Location Bugs. Seemingly every mobile app today uses location information. Unfortunately, this 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 Caiipa emulating location (i.e., GPS input and network conditions) detected that while the app was robust in US conditions, it was very brittle in other countries. For example, we find the app is 50% more likely to crash under China conditions then US ones.

Network Transitions. Mobile devices experience network transitions frequently throughout the day. For example, handoffs from Wi-Fi to 3G when users leave their home. It is critical apps be robust to such transitions.

During tests, Caiipa 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 is possible using Caiipa.

Exception Handlers. During our experiments, we notice a group of music streaming apps 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 apps apply a significantly more comprehensive set of exception handlers around network-related system calls. Although not surprising, this highlights how Caiipa is a promising way to compare mobile apps at scale and develop new best practices for the community.

Unintended Outcomes from Sensor Use. Caiipa highlighted an interesting *energy bug* in 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 ($\approx$ 5m accuracy). During testing we perturb the reported accuracy of the location estimate provided to the app (see §4). 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.

6. DISCUSSION

We discuss overarching issues related to Caiipa.

Caiipa Software Release. Currently available as a web service in beta trial at Microsoft (Figure 11), the system is already in use by app development teams responsible for 11 apps. The service also allows us to test more effective ways of providing feedback to developers on the detected issues. We are preparing for a wider release in the coming months and targeting additional scenarios, such as developers who must test for compatibility issues against multiple platforms.

Hardware vs Emulators. Currently Caiipa makes limited use of the support for physical devices (i.e. to run platform-specific code and as individual test cases). A better analysis of the tradeoffs between emulation and real devices is ongoing as well as tracking of hardware-specific metrics (e.g. GPU usage, device temperature) and usage of chipset features (e.g. co- and multi-processor).

Context Emulation. While emulating mobile context can be seen as a form of bug inducement, it presents a number of advantages over other such approaches. For example, by exploring context Caiipa can detect unforeseen situations. Other intrusive approaches like fault injection lose these benefits. They must rely on historical data on past issues, which limits their scope to previously detected problems. Furthermore, this data needs to be processed either manually (which does not scale) or automatically (which is

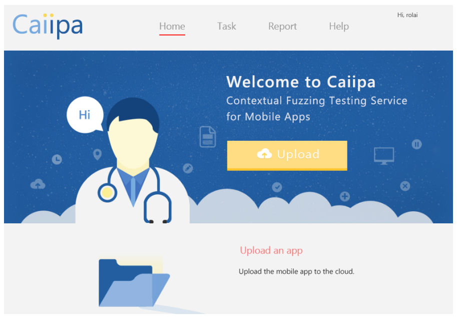
Caiipa
Home
Task
Report
Help
Hi, rolai
Hi
Welcome to Caiipa
Contextual Fuzzing Testing Service for Mobile Apps
Upload
Upload an app
Upload the mobile app to the cloud.

Figure 11: Caiipa beta web service.

error prone) to identify the root cause issues to be injected in the target system. Special care needs to be paid to this analysis as they can cause faults not possible during actual usage, or detect bugs not representative of real workloads.

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

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. Of special interest for connected apps, carrier-specific characteristics (e.g. blocking of ports or protocols, DNS solving, routing) can be a source of problems. Collecting and representing this information as part of context present many challenges. We will explore additional context sources in future work.

Testing Limitations. Many apps suffer from issues that are not directly related to crashes nor performance, such as layout issues, language errors, or usability problems. While Caiipa can detect app hangs or excessive battery drain issues, it does not address the category of content-related visual problems. Games also present challenges for automated testing, from precision of UI interactions (e.g. gestures) to gameplay constraints (e.g. only pressing jump when approaching an enemy), and they are out of scope for Caiipa.

Applicability to Other Platforms. While our prototype runs on Windows-based OSs, the core design also works on other platforms (e.g., iOS). Recently, we completed a preliminary Android prototype that is functionally equivalent to the original Window-centric implementation detailed in this paper. This prototype has yet to be evaluated, but we expect similar findings for Android-based apps as we report for Windows apps. Because of its design, such as our black-box app assumption and replaceable UI automation, Caiipa is much easier to port to alternative platforms.

7. RELATED WORK

We propose new methods for using context during mobile app testing. Our results complement existing techniques, including static analysis and fault injection.

Mobile App Testing. There are many commercial telemetry solutions for mobile devices [5, 33, 7]. Carat [24] focuses on energy bugs by periodically uploading coarse-grained battery and system status. AppInsight [28] adds visibility into the critical paths of asynchronous mobile apps. Unlike Caiipa, these are post-release solutions.

Many proactive testing tools are also available. However, most tools cover only a small subset of mobile contexts. For example, specialized tools [13] and libraries [21] exist for UI automation testing. Also, there is considerable work into generating UI input for testing of a specific goal (e.g., code coverage) [36, 14, 3, 1, 27]. Additional proactive testing tools emulate a limited set of predefined network conditions. This emulation is controlled either manually [13, 20] or via scripts [19]. In contrast to Caiipa, these tools do not allow fine-grained control over network parameters, nor accurate emulation of contexts such as network handoffs.

VanarSena [29] is a related effort that employs a greybox testing approach, leveraging instrumentation, to find bugs. It focuses on increased monkey UI testing efficiency (using a proposed *hit testing* approach along with instrumentation) and conventional fault injection of common crash inducing conditions (e.g., unhandled HTTP error codes). Unlike Caiipa, [29] can not test app performance (such as energy bugs) and is unable to test unmanaged code due to its dependence on instrumentation. We envision combining Caiipa and VanarSena into a seamless testing service that leverages the strengths of both systems [9].

State Space Exploration. The software testing community has proposed techniques to efficiently explore program state space, which mostly rely on hints extracted from source code or app test history. Whitebox fuzzing is one technique that requires source code. For example, SAGE [11] tests for security bugs by generating test cases from code analysis. [2] explores code paths within mobile apps and reduces path explosion by merging redundant paths. Model checking is another popular technique, where the idea is to build models of the app based on specs or code [15].

Finally, [12] proposes a test prioritization scheme using inter-app similarity between code statement execution patterns. However, as Caiipa does not require source code it is more broadly applicable. Directed testing (e.g. [8]), is a type of technique using a feedback-loop but for the same app. Caiipa uses similar methods, but across apps.

8. CONCLUSION

This paper presents Caiipa, a testing framework that applies the *contextual fuzzing* approach to test mobile apps over an expanded mobile context space in a scalable way. Our results show Caiipa can find many more bugs and performance issues than existing tools that consider none or a subset of the mobile context. Additionally, by linking context parameters to the detected issues, Caiipa provides additional information for developers to understand the causes of bugs and better prioritize their correction.

9. REFERENCES

[1] D. Amalfitano, et al. Using GUI Ripping for Automated Testing of Android Applications. In ASE 2012.

[2] S. Anand, M. Naik, H. Yang, and M. Harrold. Automated Concolic Testing of Smartphone Apps. In FSE 2012.

[3] T. Azim et al. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In OOPSLA 2013.

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

[5] Crashlytics. `http://www.crashlytics.com`.

[6] Fortune. `http://fortune.com/2009/08/22/40-staffers-2-reviews-8500-iphone-apps-per-week/`.

[7] K. Glerum et al. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In SOSP 2009.

[8] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed Automated Random Testing. In PLDI 2005.

[9] R. Chandra et al. Towards Scalable Automated Mobile App Testing. Technical Report MSR-TR-2014-44, 2014.

[10] C. Liang et al. Contextual Fuzzing: Automated Mobile App Testing Under Dynamic Device and Environment Conditions. Technical Report MSR-TR-2013-100, 2013.

[11] P. Godefroid et al. Sage: Whitebox Fuzzing for Security Testing. Communications of the ACM, 55(3):40–44, 2012.

[12] A. Gonzalez-Sanchez, et al. Prioritizing Tests for Fault Localization through Ambiguity Group Reduction. In Proceedings of the IEEE/ACM ASE 2011.

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

[14] F. Gross, G. Fraser, and A. Zeller. Search-based System Testing: High Coverage, No False Alarms. In ISSTA 2012.

[15] H. Guo, et al. Practical Software Model Checking Via Dynamic Interface Reduction. In SOSP 2011.

[16] V. Hautamaki, I. Karkkainen, and P. Franti. Outlier Detection Using k-nearest Neighbour Graph. In ICPR 2004.

[17] J. Huang, et al. Uncovering Cellular Network Characteristics: Performance, Infrastructure, and Policies. Technical Report MSU-CSE-00-2, 2013.

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

[19] B. Jiang, et al. Mobiletest: A Tool Supporting Automatic Black Box Test for Software on Smart Mobile Devices. In Proceedings of AST 2007.

[20] Simulation Dashboard for Windows Phone. `http://msdn.microsoft.com/library/windowsphone/develop/jj206953`.

[21] UI Automation Verify. `http://msdn.microsoft.com/en-us/library/windows/desktop/hh920986`.

[22] R. Mittal, et al. Empowering Developers to Estimate App Energy Consumption. In Mobicom 2012.

[23] R. Natella et al. On Fault Representativeness of Software Fault Injection. IEEE Trans. on Software Eng., 39(1), 2013.

[24] A. J. Oliner, et al. Carat: Collaborative Energy Diagnosis for Mobile Devices. In SenSys 2013.

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

[26] Open Signal. Signal Reports. `http://opensignal.com/reports/fragmentation.php`.

[27] V. Rastogi et al. Appsplayground: Automatic Security Analysis of Smartphone Applications. In CODASPY 2013.

[28] L. Ravindranath, et al. AppInsight: Mobile App Performance Monitoring in the Wild. In OSDI 2012.

[29] L. Ravindranath, et. al. Automatic and Scalable Fault Detection for Mobile Applications. In MobiSys 2014.

[30] Techcrunch. `http://techcrunch.com/users-have-low-tolerance-for-buggy-apps-only-16-will-try-a-failing-app-more-than-twice`.

[31] Crittercism. `http://pages.crittercism.com/rs/crittercism/images/crittercism-mobile-benchmarks.pdf`.

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

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

[34] Bit9. `https://www.bit9.com/download/reports/Pausing-Google-Play-October2012.pdf`.

[35] A. I. Wasserman. Software Engineering Issues for Mobile Application Development. In FSE - FoSER workshop, 2010.

[36] L. Yan and H. Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In USENIX Security Symposium, 2012.

[37] T. Boksasp, et al. Android Apps and Permissions: Security and Privacy Risks. NTNU Trondheim TR, June 2012.