2016 IEEE/ACM 13th Working Conference on Mining Software Repositories

Mining Test Repositories for Automatic Detection of UI Performance Regressions in Android Apps

María Gómez, Romain Rouvoy
University of Lille & Inria, France
firstname.lastname@inria.fr

Bram Adams
MCIS, Polytechnique Montreal, Canada
bram.adams@polymtl.ca

Lionel Seinturier
University of Lille & Inria, IUF, France
lionel.seinturier@inria.fr

ABSTRACT

The reputation of a mobile app vendor is crucial to survive amongst the ever increasing competition. However this reputation largely depends on the quality of the apps, both functional and non-functional. One major non-functional requirement of mobile apps is to guarantee smooth UI interactions, since choppy scrolling or navigation caused by performance problems on a mobile device's limited hardware resources, is highly annoying for end-users. The main research challenge of automatically identifying UI performance problems on mobile devices is that the performance of an app highly varies depending on its context—*i.e.*, the hardware and software configurations on which it runs.

This paper presents DUNE, an approach to automatically detect UI performance degradations in Android apps while taking into account context differences. First, DUNE builds an ensemble model of the UI performance metrics of an app from a repository of historical test runs that are known to be acceptable, for different configurations of context. Then, DUNE uses this model to flag UI performance deviations (regressions and optimizations) in new test runs. We empirically evaluate DUNE on real UI performance defects reported in two Android apps, and one manually injected defect in a third app. We demonstrate that this toolset can be successfully used to spot UI performance regressions at a fine granularity.

1. INTRODUCTION

While the number of mobile applications (or apps) published by app stores keeps on increasing (Google Play currently has 1.6 million apps available, and the Apple App Store has 1.5 million apps [35]), the quality of these apps varies widely [30, 31, 28]. Previous studies [23] show that mobile app users are intolerant to such quality issues, as indicated by the 80% of users which abandon apps after facing less than three issues. In other words, app developers are challenged to rapidly identify and resolve issues. Otherwise, due to the abundant competition, they risk losing customers to rival mobile apps and be forced out of the market.

Apart from app crashes, performance issues heavily disrupt the user experience, as highlighted by apps' user feedback [26]. In particular, one important category of performance defects in Android apps is related to poor GUI responsiveness [27]. Ideally, a mobile app should run at a consistent frame rate of 60 *Frames Per Second* (FPS) [2], which is much higher than the traditional 24 FPS for movies. Frames that are dropped or delayed because the CPU is too busy making calculations are known in the Android developers community as *janks* [14]. If an app exhibits some janks, then the app will slow down, lag, freeze or, in the worst case, lead to ANR (*Application Not Responding*) errors. Thus, UI smoothness defects directly impact the user experience.

To avoid janks in a new release of their app, app developers should, in addition to functional testing, conduct *user interface (UI) performance testing* to ensure that the new release's UI smoothness is not worse than the one of previous versions. However, doing such testing automatically is not straightforward, since it requires to run UI tests on a multitude of different devices, then to identify performance deviations from the recorded performance data. While existing work has explored both challenges in the context of data centers [20], the mobile app ecosystem has a more extreme hardware fragmentation. For example, an app can perform well on a set of devices, but it may exhibit janks in a different environment consisting of (amongst others) a different SDK version, processor or amount of RAM. Unfortunately, the current state of the practice is to detect UI smoothness regressions manually, which is time-consuming, tedious, and error-prone.

To overcome these challenges, we present DUNE[1], which is a context-aware approach to help developers identify UI performance regressions in Android apps, at a fine granularity. In particular, DUNE builds an ensemble model from a repository of previous test runs and context characteristics. This model is then used to flag performance deviations (regressions and optimizations) in a new test run automatically. The fine-grained nature of DUNE refers to its ability to not only identify performance regressions, but also spot the specific UI events triggering the regression and the contextual conditions under which the regressions happen. This automated approach has the potential to save precious time for developers, which is a crucial factor for the success of apps in current app markets.

The contributions of this paper are:

1. We propose an approach (DUNE) to automatically detect UI performance regressions in Android apps, in heterogeneous contexts, at a UI event granularity;

2. We propose a batch model to aggregate metrics and contexts from a repository of historical test runs. This model enables to flag performance deviations in new test runs, taking into consideration the execution context of the tests;

3. We have implemented a proof-of-concept tool to support DUNE;

[1] DUNE stands for *Detecting Ui performaNce bottlEnecks*.

MSR'16, May 14-15, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2901739.2901747

4. We performed experiments with real UI performance defects with two Android apps: the *K-9* e-mail client and the *Space Blaster* game; and with one manually injected defect in the *ExoPlayer* media player (used by amongst others the popular YouTube). The experiments involve 45 test runs of each app.

The remainder of this paper is organized as follows. First, Section 2 provides the necessary background to support the understanding of the approach. We then present key challenges associated with the current practice of detecting UI performance regressions in mobile apps (cf. Section 3). After an overview of the proposal in Section 4, we describe the approach to detect UI performance regressions in-depth (cf. Section 5); followed by a discussion of the implementation in a proof-of-concept (cf. Section 6). Sections 7 and 8 report on our experiments on real apps to demonstrate the applicability of the approach. Finally, Section 9 summarizes related work, and Section 10 concludes the paper.

2. BACKGROUND

In this section, we provide a brief background about the type of performance defect and testing technique we aim to tackle.

2.1 UI "Jank" Measurement

Among the different types of performance bugs that affect mobile apps (such as energy leaks and memory bloat [27]), this paper focuses on the smoothness of an app's user interface (UI), since this type of bug is directly perceivable by a user and has a strong negative impact on the user's experience. This in fact can lead to bad ratings and reviews, and can turn users to the app's competition.

In Android, a lag in the UI is called a *jank* [14], particularly when dealing with scrolling, swiping, or other forms of animation [18]. To ensure smooth animations, apps must render at *60 frames per second* (FPS), which is the rate that allows animations to be perceived by the human eye [2]. Most Android devices refresh the screen every *16 ms* ($\frac{1\ sec}{60\ fps} = 16\ ms$ per frame). Consequently, any frame taking more than 16 ms will not be rendered by Android, and hence will be flagged as a jank. In this case, users will experience choppy animations and laggy GUI. The most common practice that leads to janks is to perform heavy computations in the UI thread of the app. Google therefore encourages developers to test the UI performance of apps to ensure that all frames are rendered within the $16\ ms$ boundary [2].

2.2 Performance Regression Testing

Performance regressions are defects caused by the degradation of the system performance compared to prior releases [19]. In current practices, developers often conduct performance tests on a system while collecting performance metrics during the test run. These metrics are stored in a *test repository* for further analysis.

When a new release of the system is available, developers repeat the performance tests and compare the newly collected metrics with the previous metrics available in the test repository. Thus, developers assess if the new release fulfills their expected performance goals—*i.e.*, preventing performance degradations, improving performance on low end devices, etc.

Although *performance regression testing* has been successfully studied for desktop and web applications [20], it has received less research interest for mobile apps. Nevertheless, the rapid evolution of the mobile ecosystem (OSs, APIs, devices, etc.) and the diversity of operating conditions (network, memory, etc.) make it difficult to guarantee the proper performance of mobile apps running on different environments. As an illustration, recently, Android developers complained because their apps experienced performance issues when running on the latest Android SDK version (Android Lollipop), while they performed well in previous versions [8]. Thus, detecting performance degradations among different app versions and execution contexts is crucial to ensure high quality apps. In particular, in this paper we focus on detecting *UI performance regressions* in mobile apps running on different execution contexts.

As a matter of clarification, we distinguish between two terms used along the paper: *test case* refers to the specification of a sequence of events (or actions) which comprise a usage scenario with an app; *test run* refers to the execution of a test case. During a test run, metrics are collected and stored in a repository—*i.e.,* the *test repository*. Hence, a *test repository* is a history of performance metrics recorded from multiple test runs along time across multiple app versions.

3. CHALLENGES

Before jumping into the details of our approach, we first present three major challenges associated with the current practices to detect *UI performance regressions* in mobile apps:

Challenge #1: ***Automatic detection of UI performance degradations***. Android provides tools to profile the GPU rendering of apps in order to identify janky frames in an app [11]. The current UI performance testing consists of having a human tester which performs a set of user operations on the target app and either visually look for janks, or spend a large amount of time using the GPU profiler to identify janky frames. Hence, this *ad hoc* analysis is time consuming, tedious, and error prone. In addition, such a process relies on the human tester's ability to perceive frame rate changes, which could vary from person to person.

Challenge #2: ***Triaging UI performance root causes***. Once a jank is identified, developers rely on other tools, like Android Systrace [15], to profile concrete app executions in order to find the source of the jank. In particular, they need to filter the full data collected from a test execution in order to identify the specific sequence of user events that triggered the slow-down. This manual investigation again requires trial-and-error, which is even harder as the size of the app increases.

Challenge #3: ***Environment heterogeneity***. The app performance varies depending on the device on which it runs. For example, an app can perform well on one set of devices, or one combination of hardware and mobile app framework, but it can present performance bottlenecks when running in lower-end devices or in a different execution context (*e.g.*, using different network types). Thus, there can be performance deviations when running the app in different contexts, yet there is a lack of tools to identify UI performance deviations between different app versions and contexts.

4. OVERVIEW

Given the aforementioned challenges, we propose DUNE, which is a context-aware approach to help developers automatically identify UI performance regressions among different app releases and heterogeneous environments. DUNE does not only spot the UI events that potentially trigger a jank, but can also identify potential improvements in UI performance. The approach consists of 4 phases, as depicted in Figure 1.

To illustrate the intended use case of DUNE, let us introduce Ada, a mobile app developer who has published an app in an app store. After receiving several negative user reviews, she must quickly fix her app and update the new app version to avoid losing her reputation and customers. Before making the new version available to users, Ada wants to ensure that the users will have a high quality

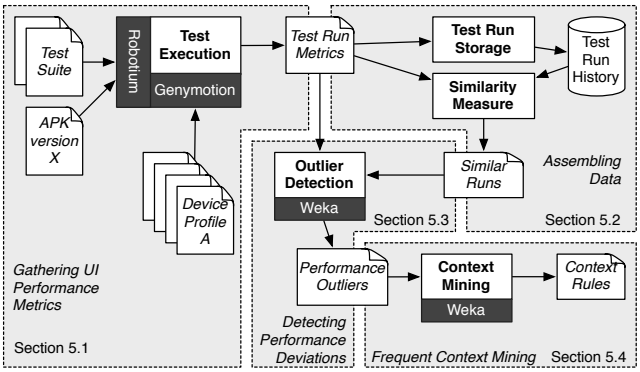

Figure 1: Overview of the DUNE approach for automatic detection of UI performance regressions.

experience with this new release. Nevertheless, she lacks the time to run tests on multiple devices, with different configurations in order to ensure that no UI performance degradation will affect her users. Instead, Ada could run DUNE to automatically and quickly identify performance degradations, and to start triaging potential problems.

All she needs to provide, is the following input: a) the *APK file of a new version* of the app under test; b) the *performance test suite* containing the scenarios to be covered as part of the UI performance evaluation; and c) the *list of device profiles* to be included as part of the UI performance evaluation.

After running the performance test suite with the provided APK on the selected device emulators, DUNE then automatically compares the performance metrics collected during the test executions to the history of metrics extracted from prior test runs in order to flag performance deviations.

5. PROPOSED APPROACH

In this section, we follow the workflow described in Figure 1 to present in detail the DUNE approach for automatic detection of *UI performance degradations* in mobile apps.

5.1 Phase 1: Gathering UI Performance Metrics

When a developer releases a new app version, this version is compared against the previous version to check whether it satisfies the app's specific performance goals—*i.e.*, to prevent regressions (or even to improve performance) on specific devices. The goal of the first phase of DUNE is to log key performance metrics during automated UI tests, in order to determine the current performance of the app and track future performance goals.

To determine the performance of an app, a variety of metrics must be collected during the execution of the apps, such as CPU utilization, memory usage or amount of input/output activity. Since DUNE targets UI performance testing, we specifically consider metrics related to UI performance that are available on mobile devices. In particular, DUNE mines two types of data during the execution of UI tests:

- *Frame rendering statistics*. The approach collects frame rendering metrics, particularly the *number of frames rendered*, the *average time to render a frame*, and the *smooth ratio*. We calculate the *smooth ratio* as the ratio of rendered frames that are not janky:

$$\text{smooth\_ratio} = 1 - \frac{\#janky\ frames}{\#total\ frames}$$

  We use the smooth ratio as a measure to determine the UI performance of an app. A higher ratio indicates better performance.

- *UI events*. Since mobile apps are UI-centric, the approach tracks the timestamps of UI events (*e.g.*, click button, select menu) during test executions. This will help developers to identify the root cause of the performance bottlenecks.

The metrics of each successful test are added to a historical test repository to help analyze future tests and performance goals.

5.2 Phase 2: Assembling Data

The goal of this phase is to build the models that can be used to detect deviations from the expected app performance.

5.2.1 Building the Batch Model for an App

App performance depends on the characteristics of the device on which it executes. Some apps can perform well in a set of devices, while failing on lower-end devices (*e.g.*, with slower CPUs or less memory available). Therefore, considering the execution context of tests is crucial to evaluate the app's actual performance.

We build a *batch model* that aggregates the collected metrics and the execution context across all the runs in the test repository of an app. In particular, the *batch model* contains the following data:

1. *Performance metrics:* We focus on UI rendering related metrics (*i.e.,* number of frames, smooth ratio, and rendering time);

2. *Execution context characteristics:* The context under which tests are executed. In particular, DUNE considers the following three major dimensions of execution context:

   - *Software profile:* Mobile SDK, API level, and version of the app under test;
   - *Hardware profile:* Device manufacturer, device model, CPU model, screen resolution;
   - *Runtime profile:* Network type, network quality, battery level.

Note that additional information can be included in the model (*e.g.,* memory and sensor state). Developers can select the relevant characteristics that they want to include in the model depending on the type of testing they target.

Figure 2 (left) depicts an example of a batch model created from a test repository with three test runs (T_1, T_2, T_3) in version $V1$ of an app. Each test run captures its *execution context* (*i.e.*, SDK, device manufacturer, device model, CPU chip, and network type); and the run's *UI performance metrics* (*e.g.*, #frames rendered, #janky frames, and smooth ratio). When the developer releases a new version of the app (*e.g.,* version $V1.2$), the batch model is used to flag performance deviations in future executions.

5.2.2 Aligning Metrics and UI Events

To flag performance deviations, DUNE implements two strategies. The *coarse-grained* mode considers the metrics collected during the whole test execution. Thus, the coarse-grained batch model stores each test run (T_i) as a vector: $T_i = [c_1, ..., c_n, m_1, ..., m_n]$. Each c_i represents a context property, and each m_i contains a global

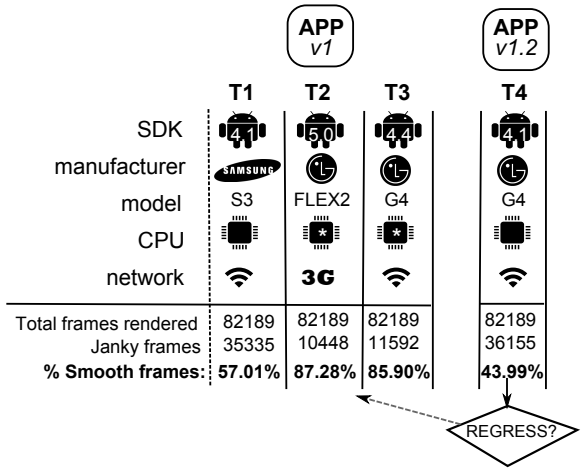

Figure 2: Example of a *Batch Model* from a test repository.

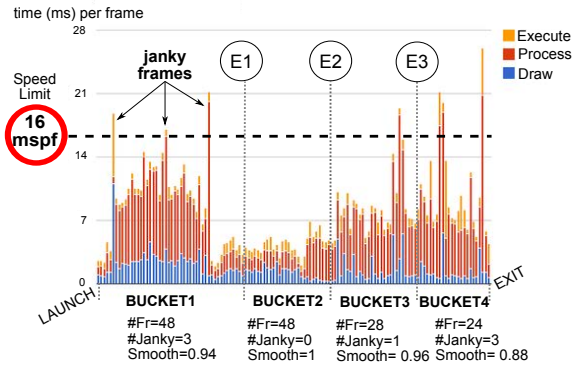

Figure 3: Frame rendering metrics for a test run divided in buckets per UI event.

performance metric (#janky frames, smooth ratio, rendering time) across the whole test execution. For example, $T1$ is stored as:

$$T1_{coarse} = [v1, 4.1, samsung, S3, x86, wifi, 148, 0.95, 45]$$

Figure 3 depicts an example of the UI frame performance metrics collected during the execution of test T_1 in Figure 2. It shows the frames rendered during the test run from app launch to finish. Each vertical bar represents one frame, and its height represents the time (in ms) spent to render the frame. The horizontal dashed line represents the speed limit (16 ms), in the sense that frames taking more time than this limit—*i.e.*, janky frames—will be skipped, leading to the app's lag. The different colors in the bars represent the time spent in each stage of the frame rendering process (execute, process, and draw). During the test run, 148 frames were rendered and 7 frames were flagged as janky.

In addition, the *fine-grained* approach, links (aligns) the metrics to specific user input events that happened during the test run and hence could be the trigger of janky frames. This *fine-grained* strategy aims to help developers to isolate the root cause of performance problems. With this goal, we distribute and aggregate the collected metrics into N different buckets, where N is the number of UI events plus 1. The first bucket contains all collected metrics before the first user event, the second bucket those between the first and second user event, etc. In the example of Figure 3, there are 3 UI events ($E1$, $E2$, $E3$) in the test run, which means that the frame metrics from app launch to finish are aggregated into 4 buckets. In particular, each bucket summarizes the number of frames, the number of janky frames, and the smooth ratio observed in that bucket. Thus, the batch model in the *fine-grained* mode stores each test run as a vector that includes one dimension per bucket to store the metrics observed in those buckets. For example, $T1$ is:

$T1_{fine}$ =[v1, 4.1, samsung, S3, x86, wifi, [48,0.94,3], [48, 1,0], [28, 0.96,1], [24, 0.88,3]]

5.3 Phase 3: Detecting Performance Deviations

When there is a new version of an app or a new execution context (such as a new SDK released by Android), the app developer needs to guarantee that the new release (or the app on a new device) meets the expected performance requirements to fulfill the user expectations. Then the developer executes the performance tests in the new context and compares the metrics with previous runs. DUNE automatically compares the new test run (T_4) with the historical test repository (T_1,T_2,T_3) and flags potential performance degradations at specific locations in the app. Note that comparisons are only done between test runs replicating the same scenario in the application, executing exactly the same functionalities and features. The process to identify performance degradations consists of 3 steps.

5.3.1 Calculating Context Similarity

Since the app performance varies depending on the device on which it runs, we rank previous test runs according to the context similarity with the new test run [20]. Thus, a previous test run on a more similar environment receives more weight and will be taken more seriously during the comparison than a test run on a completely different environment. To calculate *context similarity* between two test runs T_1 and T_4 (cf. Figure 2), first we extract from each test run the subvector that contains the context dimensions. For example, the context of the test runs in Figure 2 are represented by the subvectors:

$$T_1 = [v1,\ \underline{4.1}, samsung, S3, \underline{x86}, \underline{wifi}\,]$$
$$T_4 = [v1.2,\ \underline{4.1}, lg, G4, \underline{x86}, \underline{wifi}\,]$$

Then, we generate a binary *similarity vector* to capture the context properties that are common in the two runs. A value of 1 indicates that the two runs share the context property. Conversely, the value 0 indicates that the context property differs between the two runs. The similarity vector between tests T_1 and T_4 is:
$Sim(T_1, T_4)$=[0, 1, 0, 0, 1, 1], which means that both T_1 and T_4 share SDK version 4.1, CPU chip $x86$ and $wifi$.

Finally, we measure the *similarity degree* ($simD$) between the new and the past test run as the Cartesian length of the similarity vector. The Cartesian length is calculated as $\sqrt{N}$, with N the number of 1s in the similarity vector. For example, the similarity degree between T_1 and T_4 is $\sqrt{3} = 1.73$. The higher the degree, the higher the similarity between the context of the tests.

5.3.2 Ranking Previous Tests

Using the similarity degree, we can assign a *weight* (w) to each previous test to describe the importance of that test's batch model to analyze the new test run. Since we expect that an app behaves similar in similar contexts, tests having similar or identical contexts as the new test, should indeed have the largest weights.

Hence, to rank tests, we follow the approach proposed by Foo et al. [20], which calculates the weight of a previous test as $w(T_i) = \frac{simD(T_i)}{\sum_{j=0}^{n} simD(T_j)}$. Thus, the sum of all weights is 1. Table 1 reports

on the similarities and weights between the new test run T_4 and the historical test runs. For example, the weight of T_1 is $w(T_1) = \frac{1.73}{1.73+1+1.73} = 0.39$.

Table 1: Context similarities between the new test run (T_4) and the historical test runs (T_1, T_2 and T_3).

$test_N$	$test_H$	**simD**($Test_N$,$Test_H$)	**w**($Test_H$)
T_4	T_1	1.73	0.39
	T_2	1	0.22
	T_3	1.73	0.39

We use the similarity weight to rank the previous test runs, and cluster them according to the similarity degree with the new tests. Each cluster groups the tests which have the same degree (and hence weight). In the example, the tests result in two clusters: the first cluster contains the tests T_1 and T_3, while the second cluster only test T_2. Since the first cluster contains the set of test runs most similar to the new test, it will be used to identify performance deviations.

5.3.3 *Identifying Performance Deviations*

In the next step, DUNE aims to identify performance outliers by comparing the metrics collected in the new test run with the set of historical test runs obtained in the previous step. If the set of metrics of the new run are flagged as an outlier, DUNE notifies the developer, providing information about the violated metric and the location of the offending UI event.

To detect such outliers, we apply the *Interquartile Range* (IQR) statistical technique. This technique filters outliers and extreme values based on interquartile ranges—*i.e.*, a metric (x) is considered as an outlier if:

$$(1)\ x < Q1 - OF \times (Q3 - Q1)$$
$$(2)\ x > Q3 + OF \times (Q3 - Q1)$$

where $Q1$ is the 25% quartile, $Q3$ is the 75% quartile, and OF is the outlier factor, which is an input parameter of the technique. In this paper, we use $OF = 1.5$ because it is the default value provided by the Weka library (used in our implementation) to identify outliers with this technique. However, OF is a configuration parameter that can be overridden by the DUNE user. With a lower OF factor, metrics are flagged faster (*i.e.*, with slighter difference) as outliers.

The outliers flagged by *equation (1)* are values that deviate below the normal performance (with negative offset), whereas outliers flagged by *equation (2)* are values above the normal performance (with positive offset). Thus, DUNE can detect performance deviations that can be categorized as either performance regressions or optimizations with respect to previous test runs. Remark that the interpretation of outliers as regression or optimization depends on the type of the metric x. For example, for the metric *number of frames rendered*, an outlier with negative offset is a performance regression. If the new test renders less frames means that some frames were skipped during the execution, thus resulting in laggy animations perceivable by users.

By default, the outlier detection considers the set of metrics of a test together (coarse-grained, cf. Section 5.2.1), then determines if the test is an outlier in comparison with the test repository. As illustration, consider a test repository of an app with five test runs

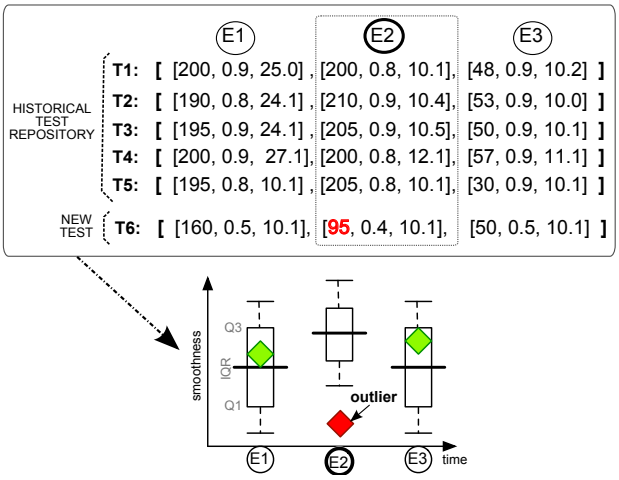

Figure 4: Automated detection of performance deviations through Interquartile Ranges with fine-grained (UI) metrics.

that recorded the following performance metrics (#Fr, Rat, T):

$$\begin{array}{rl} & (\#Fr)\ \ (Rat)\ \ (T) \\ T_1 = & [\ 448,\ \ 0.95,\ \ 45.3\ ] \\ T_2 = & [\ 453,\ \ 0.94,\ \ 44.5\ ] \\ T_3 = & [\ 450,\ \ 0.93,\ \ 44.9\ ] \\ T_4 = & [\ 457,\ \ 0.80,\ \ 50.3\ ] \\ T_5 = & [\ 430,\ \ 0.85,\ \ 45.7\ ] \end{array}$$

Recently, Android has released a new SDK version (Android Marshmallow 6.0.0) and the app developer wants to ensure that his app has a good performance in this new context. Thus, he runs the same performance test in a new device with Android 6.0.0, which records the following metrics:

$$T_6 = [\ 270,\ \ 0.40,\ \ 40.3\ ]$$

To automatically detect any performance deviation, DUNE applies the IQR filtering to the data available in the test repository (T_1, T_2, T_3, T_4, and T_5) and the new test run (T_6). If we assume all 5 test runs in the repository to be clustered together based on similarity degree, a new test run is flagged as outlier if at least one of its metrics is flagged as outlier compared to these 5 runs.

For example, the first metric of T_6 ($x_1 = 270$) is an outlier in comparison with previous observations of such metric: {448, 453, 450, 457, 430, 270}, according to *equation (1)*:

$$Q1 = 390,\ \ Q3 = 454$$
$$270\ < 390\ - 1.5(64)$$

As a result, T_6 is flagged as an outlier.

Finally, DUNE adds a class label ($Outlier_+$/$Outlier\_$/N) to the analyzed test to indicate if it is a performance optimization, regression or normal execution.

Furthermore, if the *fine-grained* mode is activated, DUNE can spot specific UI events associated with the outliers. In this case, the test runs contain a list of sets of metrics, in particular as many sets of metrics as UI events in the test (cf. Figure 4 top). The outlier detection process follows the same IQR filtering strategy, but in this case, DUNE flags individual UI events as outlier.

Figure 4 illustrates the automated outlier detection process in the *fine-grained* mode. The test repository contains five historical tests

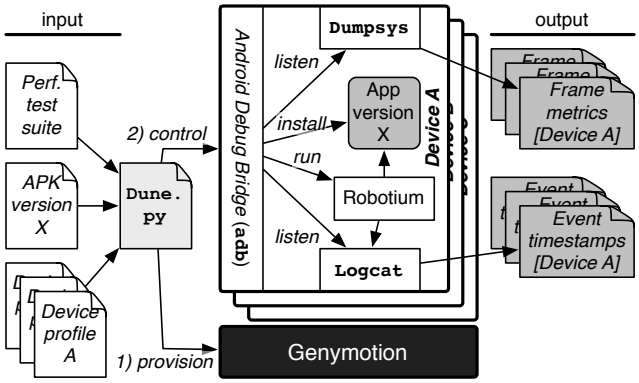

Figure 5: DUNE metrics acquisition component.

(T_1 to T_5) and a new test T_6 to analyze. DUNE applies the IQR technique to identify outliers per attribute. As a result, DUNE flags T_6 as an outlier and reports the event $E2$ as the location of the outlier. Then DUNE adds the class label $E2Outlier\_$ to the test T_6, to indicate that the event $E2$ exhibits a performance degradation. If several UI events are flagged as outliers, then DUNE adds several class labels to the test.

5.4 Phase 4: Frequent Context Mining

Finally, DUNE characterizes the context under which the performance outliers arise. To identify common context patterns that induce outliers, DUNE uses *Association Rule Mining*. An association rule is an implication of the form $X \implies Y$ that states "when X occurs, Y should occur" with a certain probability. The support of such a rule is the frequency of the rule in the dataset, while the confidence of a rule indicates the percentage of instances of X for which Y occurs. In our case, X is a context property, and Y is the class label (*e.g.,* $Outlier\_$) learnt in the previous step. To build a context rule, we require a confidence value of 90%, which is the default parameter value provided by Weka. However, this configuration parameter can be overridden by the users of DUNE.

For example, DUNE learns rules of the form:
$\{sdk = 4.1\} \implies Outlier\_$
$\{v = 1.2, dev = LG\} \implies E2Outlier\_$

These context rules help the developer to narrow down the different contexts under which performance bottlenecks appear.

6. IMPLEMENTATION DETAILS

This section provides details about our proof-of-concept implementation of DUNE, which consists of two parts. The first component is in charge of executing tests on devices and collecting metrics during the execution, while the second component is in charge of detecting performance deviations.

Figure 5 shows an overview of our tool implementation in charge of collecting performance metrics during execution. This component is implemented as a Python script that installs and runs tests on mobile devices and collects metrics during the test executions. To communicate with devices, our implementation relies on the *Android Debug Bridge* (`adb`) [1]. `adb` is a command line tool included in the Android SDK that acts as a middleman between a host and an Android device. We use `adb` to install/uninstall apps and tests, log UI events, and profile GPU rendering.

To track UI events, the tests write the timestamps of the events in the *Android Logging system* (`logcat`) [4]. In Android, the system collects debug information from apps and from the system into logs, which can be viewed and filtered by `logcat`. We have implemented a listener that monitors `logcat` and subscribes to the UI event messages.

To profile GPU rendering, we use the Android `dumpsys` tool, which runs on a device and dumps relevant information about the status of system services [3]. Since `dumpsys` only provides information about the last 120 frames rendered, our tool read the frames rendered every second, to ensure that no frame information is missed. To ensure a low overhead, this component is built on top of low-overhead tools provided by Android. Instead of requiring tests to be run on physical devices, we leverage the fast third-party Android emulator Genymotion [7], which allows to generate emulators with specific context configurations, on-the-fly. Following the concepts of Infrastructure-as-a-Service [25], Genymotion just requires a textual specification, after which generation, deployment and execution of the emulators is fully automated. In particular, one can configure the emulated device model, SDK, network, etc.

The second component of our proof-of-concept implementation is a Java application that interfaces with Weka [22]. Weka is an open-source Java library that implements several data mining algorithms. In particular, we use the *InterquartileRange* filter implemented in Weka to identify outliers, and the *Apriori* algorithm to mine association rules to point out offending contexts.

7. EMPIRICAL STUDY SETUP

In this section, we perform experiments to evaluate how well the proposed approach can detect *UI performance regressions* in tests executed on different devices with different hardware and software configurations. We first describe the *apps under test* (AUTs), the devices used, the tests developed, the data collection procedure, and the experiments performed. The next section will present the results for these experiments.

7.1 Experiments

To perform the study, we first mine real UI performance defects that are documented in issue tracker discussions and commit messages in software repositories of Android apps. In particular, we study DUNE on real UI performance issues reported in two Android apps: the *K-9 Mail* client, and the *Space Blaster* game. To further evaluate the approach, we use an additional app, the *ExoPlayer* Android media player. In this app, we manually inject janks in the context of our evaluation process. We provide a description of each AUT, and the UI performance issues that were reported.

K-9 Mail. *K-9* is an open-source e-mail client for Android [10]. The K-9 app (version 5.006) lags when opening a message from the list of messages. This issue happens after upgrading devices to Android Lollipop (V 5.∗) [9].

Space Blaster. *Space Blaster* is a space shooter game that originated on the Nintendo and Sega consoles, and later has been ported to Android [13]. Users experience performance degradations when playing on Android 5.0 devices.

ExoPlayer. *ExoPlayer* is an open source media player for Android developed by Google [5]. *ExoPlayer* provides functionality to play audio and video both locally and over the Internet. It is currently used inside Google's popular *YouTube* and *Play Movies* applications [6].

We chose these apps because UI animations are prime (*e.g.,* scroll mails, game animations and play video). Thus, the UI performance is crucial and any performance degradation quickly impacts users.

7.2 Building the Device Lab

To automatically run different versions of the AUTs in heterogeneous contexts in our experiments, we leverage the *Genymotion*

Table 2: Sample of the Device Lab used in the experiment.

Device	Brand	Model	SDK	API	CPU	Screen
D1	LG	Optimus L3	4.1.1	16	x86	240x320
D2	Google	Galaxy Nexus	4.2.2	17	x86	720x1280
D3	Motorola	Moto X	4.2.2	17	x86	1080x1920
D4	Samsung	Galaxy S4	4.3	18	x86	1080x1920
D5	Samsung	Galaxy Note 3	4.3	18	x86	1080x1920
D6	HTC	Evo	4.3	18	x86	720x1280
D7	Sony	Xperia Z	4.3	18	x86	1080x1920
D8	Google	Nexus 5	4.4.4	19	x86	1080x1920
D9	Google	Nexus 9	5.0.0	21	x86	1536x2040
D10	Google	Nexus 5X	6.0.0	23	x86	1080x1920

component of our proof-of-concept implementation. As mentioned in Section 6, *Genymotion* provides a wide range of images of real devices (with distinct hardware and software profiles), and it allows to simulate different execution environments—*i.e.*, network types.

We select the images of 20 different Android devices with different hardware and software profiles to emulate a heterogeneous context. These context profiles correspond to common configurations of popular devices at the time of performing our experiments. The selected devices run the Android SDKs: 4.1.1, 4.2.2, 4.3 and 4.4.4. We call this lab the *Device lab*.

We then build an additional lab that contains 5 additional images of Android devices running the Android SDKs: 5.0, 5.1, and 6.0.0. The latter forms the *Upgraded device lab*. These two device labs will be used across different experiments. Table 2 shows a subset of the virtual devices we use in the study, with their hardware and software characteristics.

7.3 Setting Up Tests to Collect Data

We then define, for each AUT, user interaction scenarios to mimic the real usage scenarios that were reported in the issue trackers to reproduce the performance bugs. We specify 3 test scenarios, one for each AUT.

To automatically run and repeat these scenarios, we use *Robotium*, which is a test automation framework for automatic black-box UI tests for Android apps [12]. *Robotium* executes a sequence of user interactions —*i.e.*, clicks, scrolls, navigations, typing text, etc.—on an app without requiring the source code, and it is able to do this fast. During such a test execution, we log the timestamp of each executed *UI event*, and we profile the *GPU rendering* of the app.

7.4 Benchmark Suite

To evaluate the performance of DUNE, we run each test scenario on the 20 emulated devices in the *Device lab*. To attenuate noise and obtain more accurate metric values, we repeat the same test 10 times and accumulate the results as a mean value. We repeat this process twice on each device with 2 different types of networks—*i.e.,* WiFi and 3G. This results in $20 \times 2 = 40$ different test runs for each app, which constitutes the *historical test repository* of an app. We consider all test runs in the historical test repositories as having acceptable performance.

Finally, we run the same tests in the 5 devices in the *Upgraded device lab*. This results in 5 new test runs per app.

Using these repositories, we then perform a series of 4 experiments to evaluate:

- **A:** *The effectiveness of the approach to identify new test runs with performance degradation*.
- **B:** *The effectiveness of the approach to isolate contextual conditions under which performance degradations arise*.
- **C:** *The effectiveness of the approach to identify performance optimizations*.
- **D:** *The effectiveness of the approach to identify performance degradations at a fine granularity*—i.e.*, UI event level*.

8. EMPIRICAL STUDY RESULTS

For each experiment, we provide a motivation, present the approach followed and our results.

8.1 Experiment A. Performance Degradation

Motivation: The first goal of DUNE is detecting performance deviations in a new test run in comparison with previous test runs. In this experiment, we study the coarse-grained filtering performance of DUNE to flag tests that contain performance outliers.

Approach: To perform this experiment, we select two real context-related performance bugs reported in the *K-9* and *Space Blaster* apps. Both apps reported UI performance issues after upgrading devices to Android Lollipop ($v5.*$). The app developers confirmed these issues. While the most recent version of the *K-9* app fixes the problem, the bug remains in the *Space Blaster* app.

In order to simulate the device upgrades reported in the issues for the AUTs, we select, for each app, the 5 test runs which were executed in the *Upgraded device lab* (cf. Section 7.4). These test runs exhibit performance degradation. We then compare each new test with the repository of historical tests, thus we perform 10 of such comparisons.

Results: **Dune correctly flags all performance regressions in our dataset.** Out of 10 new tests (5 per app), DUNE flags the 10 tests as containing performance degradations.

Figure 6 compares the frames rendered during the execution of the test scenario in the *Space Blaster* app in the historical repository (left) and in an updated device (right). In particular, we illustrate the test runs in devices Google Nexus 4 with Android 4.3 and 5.0.0, respectively. In the historical test, the average time to render each frame is $8\ ms$ which is under the $16\ ms$ threshold. On the contrary, the average rendering time per frame is $144\ ms$. In addition, while the historical test execution rendered $2,145$ frames and only 6 frames were janky (the smooth ratio is 99%), the new test only rendered 176 frames, thus showing that more than the 90% of frames were skipped and resulting in laggy animations perceivable by users. Similarly, Figure 7 shows the frames rendered during the execution of the test scenario in the *K-9* app on a Google Nexus 7 device with Android 4.3 in the historical test repository (left) and in the same type of device with Android 5.1.0 (right).

The task to manually determine if a new test induces a performance degradation along its execution is challenging and time-consuming. This task becomes harder as the size of the historical test repository augments and the number of considered metrics increases. Nevertheless, DUNE is able to correctly identify each performance outlier in $\sim 1\ ms$ in our datasets. Figure 8 shows the outliers flagged in the new test runs in comparison with the historical test runs in the *K-9* app for the metrics: *number of frames*, *smooth ratio*, and *average time to render each frame*.

8.2 Experiment B: Context Isolation

Motivation: The second goal of DUNE is to pinpoint potential contexts that induce janks. This experiment aims to assess this capability.

Approach: To perform this evaluation, we use the test repositories generated in *Experiment A* for the *K-9* app. We then evaluate the effectiveness of DUNE to identify the conflictive contexts that were reported as causing the performance issues in the *K-9* app.

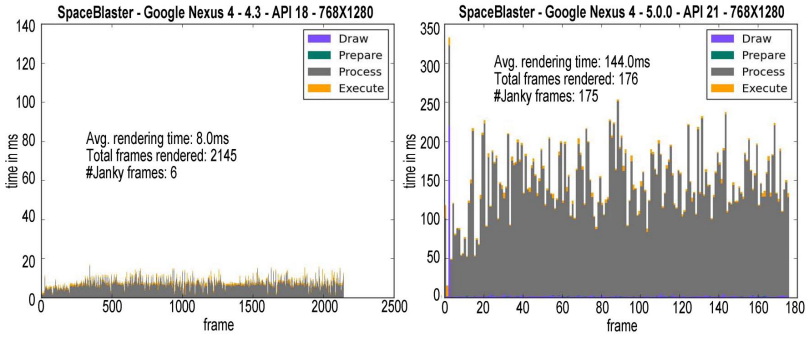

Figure 6: Frames rendered during the test scenario execution in the *Space Blaster* app in the historical dataset (left) and in the new context (right).

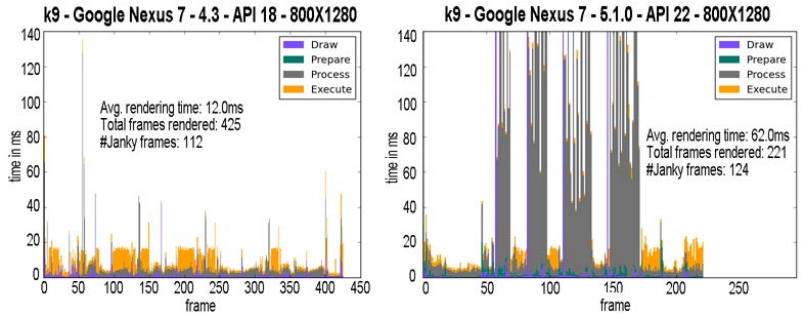

Figure 7: Frames rendered during the test scenario execution in the *K-9* app in the historical dataset (left) and in the new context (right).

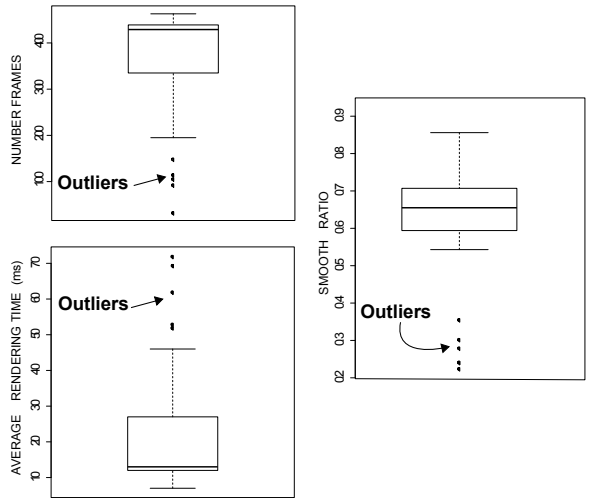

Figure 8: Outliers flagged in the new test runs in the *K-9* app in comparison with the historical test runs.

We let DUNE learn context rules with minimum support 0.01 and minimum confidence 0.9. We set minimum support 1%, since the proportion of new test runs is small in comparison with the number of test runs in the historical dataset. For example, when a new test run includes a new context property that was not available in the historical tests, *e.g.,* a new SDK version is released, these properties have a low frequency. By setting a low minimum support value we ensure that the rules capture these new infrequent properties as they emerge. On the contrary, we set confidence to 90%, since we want to ensure that the context properties are highly correlated with outliers. This is the default value provided in the Weka library used in the implementation.

Results: **DUNE effectively learns context rules where janks arise.** As a result, DUNE identifies 7 context rules (after pruning redundant rules). All these rules have a confidence of 100%. Figure 9 shows a graph-based visualization of the context rules identified in the *K-9* app. In the graph, nodes represent items (or itemsets) and edges indicate relationship in rules. The size of the nodes represents the support of the rule. In particular the contexts are the SDKs: 5.0.0, 5.1.0, and 6.0.0, and their associated API levels: 21, 22, and 23, respectively. Thus, DUNE effectively identifies the contexts that were reported as problematic in the issue tracker system. Furthermore, even though the SDK 6.0.0 was not reported in the apps' issue reports[2], we manually repeated the scenario on a device with 6.0.0 and we could perceive UI lags as described. This finding underlines the potential of the technique to identify performance regressions across contexts. In addition, the *Google Nexus 5X* device appears in a rule, since it is the unique device in the repository that runs the version 6.0.0.

To analyze the sensitivity of the parameters in this experiment, we increase the minimum support threshold with deltas of 0.01 and explore the rules obtained. When the minimum support reaches 0.03, the number of context rules found is 5. In particular, the contexts `SDK=6.0.0` and `device=Google Nexus 5X` are pruned. Whereas the remaining 5 rules are the same as those previously identified, which effectively reproduce the performance regressions. Moreover, when we set the minimum support to 0.07, no rules are found. In this experiment, all the rules exhibit a confidence of 1,

[2]At the time of writing, Android 6.0.0 (Marshmallow) is the latest Android version released, however it is only available on Nexus devices manufactured by Google, thus currently few widespread among users.

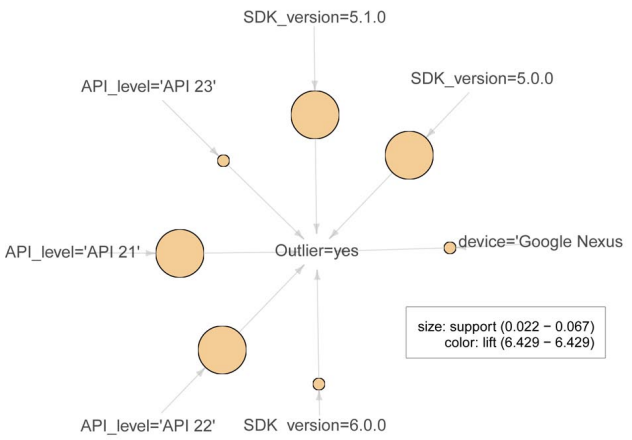

Figure 9: Graph visualization of the context rules identified in the *K-9* app.

which is the highest possible value. As mentioned in section 5, developers can calibrate the parameters of DUNE (minimum support and confidence) according to their needs. If they are too restrictive, they risk to miss some defects, and harm the experience of their users. On the contrary, if they are too permissive, they can obtain context rules that are not discriminative enough.

8.3 Experiment C: Performance Optimization

Motivation: In addition to detecting performance degradations, improving app performance (for example in low-end devices) is another significant performance goal which app developers persecute. Hence, developers can use DUNE to assess performance optimizations. The goal of the third experiment is to assess this capability.

Approach: We select the 10 new tests that were generated in *Experiment A* for both apps *K-9* and *Space Blaster*. Then, we randomly select one of the tests in each app from the *historical* test repository, *i.e.*, we swap the new tests and one of the elements of the test repository. Finally, we check if DUNE correctly flags the historical test as an outlier (optimization) in comparison with the new tests. In this case, the outlier should have a positive offset, hence indicating a performance optimization.

Results: **The approach effectively reports the historical test as an outlier with positive offset**, hence indicating that this run is better than previous tests. This result confirms the utility of DUNE, which can spot both regressions and optimizations effectively.

8.4 Experiment D: UI Event Filtering

Motivation: The last experiment aims to evaluate the capability of DUNE to filter out individual events that trigger janks. Hence, in this experiment, we evaluate the fine-grained filtering performance to identify performance degradations at UI event level. We check if DUNE correctly points out the events in a test that trigger a jank. The more irrelevant UI events DUNE can discard, the less effort developers require to find the root cause of problems.

Approach: In this evaluation, we use the *Exoplayer* app with manually injected janks. Since we know the location where the janks were injected, we can assess the effectiveness of the approach.

Google states that most of the performance issues in apps come from performing heavy operations in the main UI thread. Thus, we inject a synthetic jank, extracted from Sillars et al. [18], in the *Exoplayer* app to cause UI performance degradation. The jank consists of inserting a heavy computation in the UI thread. We inject the jank in two locations of the app—*i.e.*, when the user selects the option *Logging* during video playback and when the user touches the screen during video playback. We then define a realistic user interaction scenario for the AUT. At a high level, the scenario consists of selecting and playing two videos from the Internet, and change different configuration settings while the video is playing. In total, the scenario is composed of 20 user interactions and takes 102 seconds to execute on average. Across the execution of the test scenario the jank manifests 4 times after events: *#3, #7, #16, #17*.

We then run the test scenario with the mutated version of the *Exoplayer* app on 15 devices randomly selected in our *Device lab*. This results in a *mutated test repository* with 15 test runs that exhibit performance degradation. Remark that all the test executions traverse the injected faults.

We measure the effectiveness of DUNE in terms of *Precision* and *Recall* which are computed as follows [20]:

$$\text{Precision} = 1 - \frac{\#false\ positives}{total\#\ of\ events\ flagged}$$

$$\text{Recall} = \frac{\#janky\ events\ detected}{\#expected\ janky\ events}$$

False positives are events flagged as janky, while we did not include a jank in such a position. Note that the precision metric is only an approximation, since we cannot ensure that the false positives really are false alarms. The app could experience performance degradations at some points due to external factors such as network variations, device state (*e.g.*, device running out of battery), etc. The less false positives, the less locations the developer will need to explore to fix the janks. On the contrary, high recall means that the approach detects most performance degradations at specific events. The recall metric is the most important in this setting, since a high recall means that DUNE finds most of the janks.

Results: **The approach obtains an average precision and recall of 53% and 83%, respectively.** Table 3 reports the results of the experiment. For each mutated test, it shows the events that were flagged (bold events were flagged correctly), together with the approximate precision and the recall. Recall is the most important metric for DUNE, since it aims to find all occurrences of janks. The low precision means that several events were flagged without any jank being injected. After manual inspection of these incorrectly flagged events, we observe that many performance deviations are induced by the effect of the previous jank. For example, we injected a jank in event #7. We can observe that when event #7 is flagged in a test, the next event #8 is also flagged in most of the tests, since the app has not recovered yet. Although event #8 is considered as a false positive in the evaluation, it is observed as janky. In addition, we observe that in some test runs (*e.g.*, $T7$, $T8$, $T9$) the number of falsely flagged events is higher. The rational of this is that such test runs correspond with lower-end devices which make the app to perform worse in general along the whole execution. Finally, some external factors such as network variations, can be responsible to introduce lags at some points of the execution. Nonetheless, DUNE is a powerful tool which can filter out the most relevant events, thus saving precious time to developers.

8.5 Threats to Validity

Our results show that the approach can be successfully used to spot UI performance deviations at a fine granularity. Nevertheless, further analyses are necessary to evaluate the efficiency of this approach on different types of apps. For example, we should consider different app categories and sizes. The setting to evaluate the approach (number of scenarios and test runs) is similar to the settings used in previous works [20]) that successfully tackled a similar problem.

Table 3: Summary of the performance of DUNE to spot UI events.

	Janky events:**#3, #7, #16, #17**		
Test	**Flagged Events**	**Precision**	**Recall**
$T1$	**3**,5,**7**,**16**	0.75	0.75
$T2$	2,**3**,5,**7**,**16**,**17**	0.67	1
$T3$	**3**,5,**7**,**16**	0.75	0.75
$T4$	**3**,6,11,**16**,**17**	0.60	0.75
$T5$	**3**,11,**16**,**17**	0.75	0.75
$T6$	**3**, 6,11,**16**,**17**	0.60	0.75
$T7$	1,6,**7**,8,9,12,14,**16**,**17**,18	0.30	0.75
$T8$	1,2,**3**,**7**,9,11,12,**17**,18	0.33	0.75
$T9$	1,2,**3**,6,**7**,8,9,11,12,**16**,**17**,18	0.33	1
$T10$	**3**,4,6,**7**,8,**16**,18	0.43	0.75
$T11$	**3**,4,6,**7**,8,**16**,18	0.43	0.75
$T12$	**3**,6,**7**,8,**16**,**17**	0.67	1
$T13$	2,**3**,4,6,**7**,8,**16**,**17**,18	0.44	1
$T14$	1,2,**3**,4,6,**7**,8,**16**,**17**,18	0.40	1
$T15$	2,**3**,4,6,**7**,8,**16**,**17**,18	0.44	1
	average:	**0.53**	**0.83**

In order to better capture the user experience, the performance test scenarios that are used by DUNE are expected to provide a good coverage of the app functionalities and be representative of app usages. With regard to this threat, the introduction of lightweight usage monitoring techniques can help to leverage the generation of user-realistic scenarios [21].

In addition, one important threat to construct validity is the choice of devices used in the experiment. Similar to user scenarios, the selected device profiles should be representative of the devices owned by users. One alternative technique to user monitoring is to select random profiles during the test execution and to rely on the similarity measure to better approximate the performance bottlenecks.

For the purpose of our experimentations, we use Android emulators, although the performance of an app could vary when running on a emulator compared to on a real device. Nevertheless, in practice, app developers use emulators to test their apps due to budget and time constraints. Given the large mobile fragmentation, developers use both emulators and real devices. To address this threat, we plan to extend our experiments in a crowd of real devices.

The proposed approach is based on predefined thresholds specified by various parameters. The first parameter is the *outlier factor* used to detect performance outliers (cf. subsection 5.4). The second parameter is the *confidence factor* used to learn context rules (cf. subsubsection 5.3.3). The calibration of these parameters can impact the results and constitute a potential threat to validity.

Finally, other ecosystems than the Android one should be explored. The conceptual foundations of our approach are independent of Android and the type of performance defect. We only need an oracle of performance and some observation features (in our case, context characteristics). To gain confidence in the external validity of our approach, more evaluations are needed on other platforms, using different performance metrics.

9. RELATED WORK

This section summarizes the state of the art in the major disciplines related to this research.

Mobile App Testing. A large body of research exists in the general area of functional testing in mobile apps [29, 24, 17, 16]. These works have proposed GUI-based testing approaches for Android apps. These approaches focus on functional correctness and aim to reveal faults in the apps under test. In contrast, we are interested in detecting UI performance defects—*i.e.*, janks.

Automatic Detection of Performance Bugs. Liu *et al.* [27] performed an empirical study on 8 real-world Android applications and classify performance bugs in mobile apps as: GUI lagging, memory bloat, and energy leak. Additionally, they present *PerfChecker*, a static code analyzer to identify such performance bugs. Later, Liu *et al.* [27] present an approach for diagnosing energy and performance issues in mobile internetware applications.

Another group of works includes approaches to detect performance bugs in mobile apps after deployment in the wild. *AppInsight* [34] is a system to monitor app performance in the wild for the Windows Phone platform. *CARAT* [32] is a collaborative approach to detect and diagnose energy anomalies in mobile apps when running in a crowd of mobile devices. On the contrary, our approach focuses on the detection of a specific type of performance defect—*i.e.*, UI janks.

Other works have focused on the detection of UI performance defects. Yang *et al.* [36] propose a test amplification approach to identify poor responsiveness defects in android apps. Ongkosit *et al.* [33] present a static analysis tool for Android apps to help developers to identify potential causes of unresponsiveness, focusing on long running operations. The aforementioned approaches tackle the jank defect, as our approach does. However, these approaches focus on stressing an app to expose UI responsiveness lags. In contrast, our approach aims to identify UI performance regressions between different versions of an app running in different execution contexts.

Despite the prolific research in this area, there is a lack of automated regression testing approaches in the mobile world. Our approach aims to complement existing testing solutions with a performance regression approach that takes into consideration the highly diverse execution context.

Performance Regression Testing. Performance regression testing has been vastly studied in desktop and web application domains. Foo *et al.* [20] introduce an approach to automatically detect performance regressions in heterogeneous environments in the context of data centers. We adapted this approach to the area of mobile apps. To the best of our knowledge, we propose the first automated approach to automatically identify UI performance degradations in mobile apps taking into consideration different execution contexts.

10. CONCLUSION

Due to the abundant competition in the mobile ecosystem, developers are challenged to provide high quality apps to stay competitive in the app market. In particular, providing smooth UI interactions is a key requirement to ensure a high quality user experience.

This paper introduces DUNE, a context-aware approach to help developers identify UI performance regressions among different Android app releases and heterogeneous contexts. First, DUNE builds an ensemble model of the UI performance metrics from a repository of test runs collected for different configuration contexts. Second, this model is used to flag UI performance deviations (regressions and optimizations) in new test runs. DUNE spots the specific UI events that potentially trigger a jank, and isolates the context properties under which janks arise (*e.g.*, a specific SDK version).

We provide a prototype implementation to support DUNE. Then, we empirically evaluate DUNE on real UI performance defects reported in two Android apps and one injected defect in a third Android app. Our results show that DUNE can successfully spot UI performance regressions, even at the UI event level (modulo false positives).

11. REFERENCES

[1] Android adb. http://developer.android.com/tools/help/adb.html. [Online; accessed Mar-2016].

[2] Android Developers. Android Performance Patterns: Why 60 FPS? https://www.youtube.com/watch?v=CaMTIgxCSqU. [Online; accessed Mar-2016].

[3] Android dumpsys. https://source.android.com/devices/tech/debug/dumpsys.html. [Online; accessed Mar-2016].

[4] Android logcat. http://developer.android.com/tools/help/logcat.html. [Online; accessed Mar-2016].

[5] ExoPlayer. http://developer.android.com/guide/topics/media/exoplayer.html. [Online; accessed Mar-2016].

[6] ExoPlayer Adaptive video streaming on Android (YouTube) . https://www.youtube.com/watch?v=6VjF638VObA. [Online; accessed Mar-2016].

[7] Genymotion. https://www.genymotion.com. [Online; accessed Mar-2016].

[8] How To Fix Freezing, Lag, Slow Performance Problems on Samsung Galaxy Note 3 After Updating to Android Lollipop. http://thedroidguy.com/2015/06/how-to-fix-freezing-lag-slow-performance-problems-on-samsung-galaxy-s5-after-updating-to-android-lollipop-part-1-107475. [Online; accessed Mar-2016].

[9] K-9 issue report. https://github.com/k9mail/k-9/issues/643. [Online; accessed Mar-2016].

[10] K9-Mail Android app. https://play.google.com/store/apps/details?id=com.fsck.k9&hl=en. [Online; accessed Mar-2016].

[11] Profiling GPU Rendering Walkthrough. http://developer.android.com/tools/performance/profile-gpu-rendering/index.html. [Online; accessed Mar-2016].

[12] Robotium. https://code.google.com/p/robotium. [Online; accessed Mar-2016].

[13] Space Blaster Android app. https://play.google.com/store/apps/details?id=com.iraqdev.spece&hl=en. [Online; accessed Mar-2016].

[14] Testing Display Performance. http://developer.android.com/training/testing/performance.html. [Online; accessed Mar-2016].

[15] Testing Display Performance. http://developer.android.com/tools/help/systrace.html. [Online; accessed Mar-2016].

[16] AMALFITANO, D., FASOLINO, A. R., TRAMONTANA, P., TA, B. D., AND MEMON, A. M. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *Software, IEEE 32*, 5 (2015), 53–59.

[17] AZIM, T., AND NEAMTIU, I. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (2013), OOPSLA, ACM, pp. 641–660.

[18] DOUG SILLARS. *High Performance Android Apps: Improve ratings with speed, optimizations, and testing*, first ed. O'Reilly Media, Sept. 2015.

[19] FOO, K. C., JIANG, Z. M., ADAMS, B., HASSAN, A. E., ZOU, Y., AND FLORA, P. Mining performance regression testing repositories for automated performance analysis. In *Proceedings of the 10th International Conference on Quality Software, QSIC 2010, Zhangjiajie, China, 14-15 July 2010* (2010), pp. 32–41.

[20] FOO, K. C., JIANG, Z. M., ADAMS, B., HASSAN, A. E., ZOU, Y., AND FLORA, P. An industrial case study on the automated detection of performance regressions in heterogeneous environments. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2* (2015), pp. 159–168.

[21] GÓMEZ, M., ROUVOY, R., ADAMS, B., AND SEINTURIER, L. Reproducing Context-sensitive Crashes of Mobile Apps using Crowdsourced Monitoring. In *Proceedings of the 3rd IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft'16)* (Austin, Texas, United States, May 2016), L. Flynn and P. Inverardi, Eds., IEEE. To appear.

[22] HALL, M., FRANK, E., HOLMES, G., PFAHRINGER, B., REUTEMANN, P., AND WITTEN, I. H. The WEKA data mining software: An update. *SIGKDD Explor. Newsl. 11*, 1 (Nov. 2009), 10–18.

[23] HEWLETT PACKARD. Failing to Meet mobile App User Expectations: A Mobile User Survey. Tech. rep., 2015.

[24] HU, C., AND NEAMTIU, I. Automating GUI Testing for Android Applications. In *Proceedings of the 6th International Workshop on Automation of Software Test* (2011), AST, ACM, pp. 77–83.

[25] HUMBLE, J., AND FARLEY, D. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, 1st ed. Addison-Wesley Professional, 2010.

[26] KHALID, H., SHIHAB, E., NAGAPPAN, M., AND HASSAN, A. What do mobile app users complain about? *Software, IEEE 32*, 3 (May 2015), 70–77.

[27] LIU, Y., XU, C., AND CHEUNG, S.-C. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), ICSE 2014, ACM, pp. 1013–1024.

[28] MAALEJ, W., AND NABIL, H. Bug report, feature request, or simply praise? on automatically classifying app reviews. In *Proceedings of the 23rd IEEE International Requirements Engineering Conference* (2015).

[29] MACHIRY, A., TAHILIANI, R., AND NAIK, M. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2013), ESEC/FSE, ACM, pp. 224–234.

[30] MCILROY, S., ALI, N., KHALID, H., AND E. HASSAN, A. Analyzing and automatically labelling the types of user issues that are raised in mobile app reviews. *Empirical Software Engineering* (2015), 1–40.

[31] MOJICA, I. J., NAGAPPAN, M., ADAMS, B., BERGER, T., DIENST, S., AND HASSAN, A. E. An examination of the current rating system used in mobile app stores. *IEEE Software* (2015).

[32] OLINER, A. J., IYER, A. P., STOICA, I., LAGERSPETZ, E., AND TARKOMA, S. Carat: Collaborative energy diagnosis for mobile devices. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems* (2013), ACM, p. 10.

[33] ONGKOSIT, T., AND TAKADA, S. Responsiveness analysis tool for android application. In *Proceedings of the 2Nd International Workshop on Software Development Lifecycle for Mobile* (2014), DeMobile 2014, pp. 1–4.

[34] RAVINDRANATH, L., PADHYE, J., AGARWAL, S., MAHAJAN, R., OBERMILLER, I., AND SHAYANDEH, S. AppInsight: Mobile App Performance Monitoring in the Wild. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation* (2012), OSDI, pp. 107–120.

[35] STATISTA. Number of apps available in leading app stores as of July 2015. http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores, 2015. [Online; accessed Mar-2016].

[36] YANG, S., YAN, D., AND ROUNTEV, A. Testing for poor responsiveness in android applications. In *Engineering of Mobile-Enabled Systems (MOBS), 2013 1st International Workshop on the* (2013), IEEE, pp. 1–6.