Reproducible Interference-aware Mobile Testing

Weilun Xiong*, Shihao Chen*, Yuning Zhang*, Mingyuan Xia† and Zhengwei Qi*
* Shanghai Jiao Tong University, Shanghai, China
† AppetizerIO
{azard5, julius_chen, yuning_zhang, qizhwei}@sjtu.edu.cn, ken@appetizer.io

***Abstract*—Mobile apps are born to work in an environment with ever-changing network connectivity, random hardware interruption, unanticipated task switches, etc. However, such interference cases are often oblivious in traditional mobile testing but happen frequently and sophisticatedly in the field, causing various robustness, responsiveness and consistency problems. In this paper, we propose JazzDroid to introduce interference to mobile testing. JazzDroid adopts a gray-box approach to instrument apps at binary level such that interference logic is inlined with app execution and can be triggered to effectively affect normal execution. Then, JazzDroid repeatedly orchestrates the instrumented app through app developers' existing tests and continuously randomizes interference on the fly to reveal possible faulty executions. Upon discovering problems, JazzDroid generates a test script with the user inputs from developers' tests and the interference injected for developers to reproduce the problems. At a high level, JazzDroid can be seamlessly integrated into app developers' testing procedures, detecting more problems from existing tests.**

We implement JazzDroid to function on unmodified apps directly from app markets and interface with *de facto* industrial testing toolchain. JazzDroid improves mobile testing by discovering 6x more problems, including crashes, functional bugs, UI consistency issues and common bug patterns that fail numerous apps.

***Keywords*-testing; mobile; Android;**

I. Introduction

The popularity of mobilized consumer electronics (smart phones, tablets, smart home appliances, smart watches, etc) is fueling a passionate global mobile application ecosystem. Due to the nature of mobility, mobile devices need to handle both user inputs and environmental changes. Often, app designation would impose implicit assumption about the environment such as stable network connectivity, dedicated hardware usage, and independent user inputs, etc. These assumptions could be nullified in real users' usage, causing apps to diverge from normal execution flow and leading to crashes, UI inconsistency, program state corruption, etc. Thus, understanding how environmental interference impacts app execution is a crucial task for mobile testing. However, environmental interference is hard to create in lab and becomes absent or insufficient in existing testing methods (manual testing, UI automation, fuzzing, etc). Consequently, current testing methods often fall short in face of such interference.

In this paper, we aim to improve the robustness, responsiveness and state consistency of apps against environmental interference. Fuzzing is a common technique used to stress an app with unanticipated user inputs [14], [15]. We develop JazzDroid, a tool that extends fuzzing to simulate environmental interference while the app is running existing test cases. The challenge for this design is the need to inject proper interference at the proper timing such that it has an impact on program execution. A white-box approach requires app developers to simulate interference by modifying app source code. Such approach results in fairly effective interference injection but requires significant efforts to identify, tune and maintain injection points. On the contrary, a black-box approach [15] automatically changes system states to simulate interference, which overcomes the shortage of white-box approaches. However, black-box approaches have no knowledge of the app execution state [10]. Thus, injection is less effective (e.g., emulating lost connectivity while the app is not performing network requests). Consequently, black-box approaches like monkey [20] and AimDroid [38] require significantly more time to detect more bugs. JazzDroid embodies a novel mechanism to automatically instrument an app at the bytecode level to achieve the injection granularity of white-box approaches and the automation of black-box counterparts. JazzDroid positions environmental interference at critical program points and code paths. Injected interference has direct effect on the program state (e.g., added network delay, failed network requests, etc) such that the injection is more effective for finding problems. Once a program bug is detected during interference-enabled execution, JazzDroid automatically generates a test script with user inputs and interference events to reproduce the faulty execution. This generation process ensures that detected problem is experimentally reproducible and also pinpoints the concrete interference (type(s), timing and ordering) that causes the program bug.

To demonstrate the usefulness of JazzDroid, we apply JazzDroid to 105 real-world apps to enhance their baseline testing methods, including monkey [20], hand-crafted test scripts and manual tests. Our results show that JazzDroid discovers 6x more bugs than baseline tests, each being reproducible with the detailed interference that triggers such incidents, which lead to easy fixes. The contribution of this paper is three-fold:

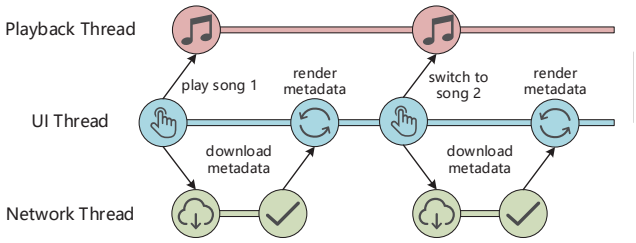
- We propose a methodology to effectively inject in-

terference for mobile testing. This methodology allows to trigger targeted interference at critical program points such that the app execution is affected to reveal potential robustness, responsiveness, and consistency problems.

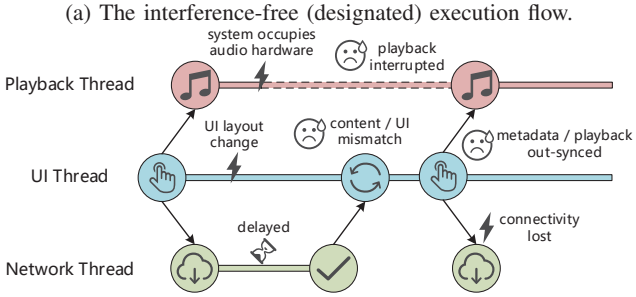
- We present a proof-of-concept implementation of interference injection, which can automatically retrofit Android apps released on app markets and easily be integrated with existing mobile testing workflows.
- We perform empirical studies of interference-aware mobile testing on well-tuned real-world and open source apps. Our study uncovers bugs that are hard be discovered by conventional testing techniques and common bug patterns that affect a wide range of apps.

The rest of the paper is organized as follows. Section II presents a simple example to motivate interference injection for mobile testing. Section III elaborates the design of JazzDroid. Section IV evaluates JazzDroid with real-world apps and presents our case studies and experiences. Section V presents related work and Section VI concludes.

II. Motivation



(a) The interference-free (designated) execution flow.



(b) Possible environmental interference and faulty execution consequences.

Figure 1: The impact of environmental interference on a music player app.

In this section, we present an example of how environmental interference impacts the ideal application execution flow, causing issues to emerge beyond developers' expectation. Detecting and resolving such issues motivate the need for interference-aware testing methods. Figure 1a showcases the designated work flow for a simplified music player app. This app has three threads. The playback thread controls the playback of a particular soundtrack. The UI thread processes user inputs and updates UI components. The network thread issues network requests and receives responses. The designated (interference-free) execution flow starts from a user click to play a particular song. The event handler running in the UI thread initiates the request for the metadata in the network thread and informs the playback thread to start playing the selected song. When the network request is completed, the network thread calls back UI thread to update the metadata to certain UI components. In this simple example, we have the asynchronous pattern, and we need manage the consistency of playback and metadata. Asynchrony and consistency management are core to app design and complicated apps can be decomposed to simple tasks as mentioned. The workflow is usually the validation target in the test phase, since mainstream testing tools and frameworks aim to walk through related procedures and check assertions for whether the metadata is properly rendered and whether playback starts/stops properly.

In Figure 1b, we demonstrate several possible environmental interference users could encounter in real world cases and how such interference impacts application behavior. First, during the playback of the first song, the mobile device could receive phone calls or alarm clocks, causing the system to take over the audio hardware to serve those events (similar to interrupts). The mobile system will inform the foreground music app of such interrupts, in the form of system broadcasts. If the music app does not handle such broadcasts, users could experience suspended and unrecoverable music playback. Second, most modern apps have images and other resources adapted for different UI layouts (optimized for different screen sizes and device orientations). If a UI layout change happens when the app still requests the resources for the old layout (e.g., because of a network delay for the request), users would witness a mismatch of the new layout with the old images. Third, mobile devices are prone to connectivity changes. Often if the application fails to consider the situation that network requests could fail due to connectivity lost, users would witness an inconsistent program state (e.g., metadata being inconsistent with the song being played). These three interference cases demonstrate that apps must properly handle interference in different stages of app execution. This is a non-trivial development task which involves the knowledge of what kind of events could interfere which part of the app. However, it is hard to enumerate all possible interference at all possible time points at the development phase and thus testing is essential to cover the execution flows that encounter interference. And often such interference cases unveil failure cases as mentioned in the simple example. In this paper, we aim to introduce interference to mobile testing to improve the robustness, responsiveness and state consistency of apps.

III. JazzDroid Design

JazzDroid is designed to introduce environmental interference to mobile testing. In this section, we begin with reviewing the design goals of JazzDroid and the considerations made to guarantee interference injection quality while maintaining the convenience for app developers to use JazzDroid. We then elaborate the choice for interference parameters and implementation details for a proof-of-concept implementation of JazzDroid.

A. Design Goals

We present the design goals for not only JazzDroid but also tools alike.

- Apps run in an ever-changing environment involving a diverse range of device hardware, random user inputs, system interrupts, etc. However, not all environmental changes are environmental interference that can impact app execution. Meantime, within a small time window, the app can only respond to an even smaller set of environmental changes. For instance, a music player will only be affected by audio hardware status changes when playing songs. Thus JazzDroid aims to emulate environmental changes at proper program points such that the interference *could affect dynamic app execution*. Conventional black-box approaches have no knowledge of the current app running status and thus lack the information to generate effective environmental interference. On the contrary, JazzDroid aims to trigger different interference according to app running status.
- The process of interference injection should be *transparent* to app developers and should be *non-intrusive* to existing testing toolchain. The ideal interference injection is one performed by app developers to manually merge interference logic with app logic, i.e., a white-box approach. However, this could be labour-intensive, error-prone and not maintainable. JazzDroid restrains from requiring modification on app source code. Also we are aware that real apps adopt numerous testing techniques (manual testing, script-based tests, automatic UI exerciser, etc), and altering tests for interference injection is also unrealistic and limits the capability of different testing methods. JazzDroid aims to automatically implant interference injection logic to app binary code, and a JazzDroid-enabled app should be compatible with existing tests.
- Since environmental interference is random in reality, only certain ordering could cause a problem for the app. Thus, JazzDroid should provide a procedure to *reproduce* the problem if a bug or an issue is detected after JazzDroid injects interference. The procedure should contain the user inputs and inject interference necessary to recreate a faulty execution. This design goal allows app developers to effectively debug and fix the problem caused by interfered execution.

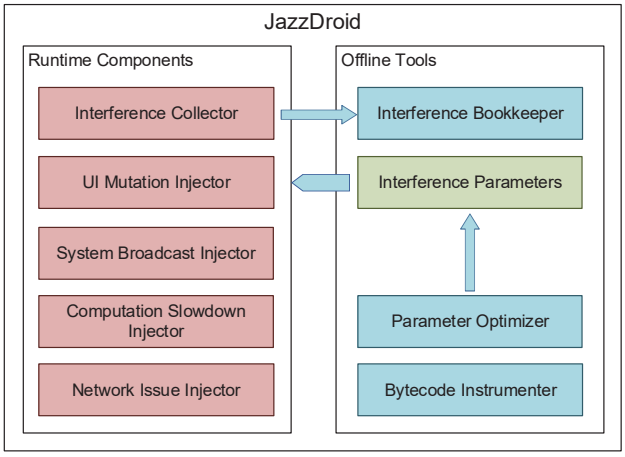


Figure 2: JazzDroid architecture.

B. Architecture

Figure 2 depicts the overview architecture of JazzDroid. JazzDroid leverages code instrumentation to add interference injection logic to a target app at bytecode level automatically. JazzDroid's instrumentation logic code that runs with the target app, i.e., JazzDroid runtime components, includes four *injectors* that inject different categories of interference on the fly, and the *interference collector* that logs the injected interference as well as other runtime diagnosis information for record. JazzDroid offline tools perform bytecode instrumentation, prepare interference parameters for injectors and collect injected runtime interference to generate test scripts that can reproduce captured problems.

As illustrated in Figure 3, the overall workflow for JazzDroid starts from the bytecode instrumenter, which extracts code files from an app, analyzes the bytecode to find proper instrumentation points, and inserts JazzDroid runtime components (see details about interference injection at Section III-C). Then the parameter optimizer performs dry runs on the JazzDroid-enabled app to obtain proper interference parameters such as injection frequency (see details at Section III-E). After these preparation procedures, the instrumented code and interference parameters are packed into a valid app format. App developers use the JazzDroid-enabled app instead of the original for testing. The interference collector will log every interference injected by JazzDroid runtime injectors. Finally, the interference bookkeeper synthesizes runtime interference log along with test operations from the developers into test scripts which can run to reproduce interference-enabled execution.

C. Interference Injection

Interference injection is the key part of JazzDroid. The questions to be considered include: What kinds of interference should be emulated? What program points should have

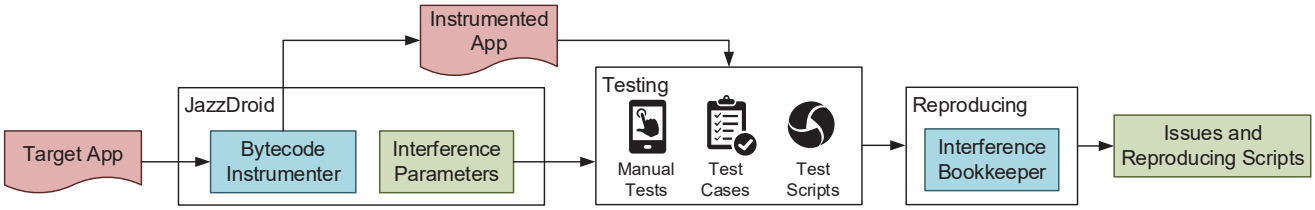


Figure 3: JazzDroid workflow.

interference injection? How to instrument app bytecode to achieve transparent interference injection?

1) Interference: **System Broadcast Injector.** System broadcasting is a common mechanism used by the mobile OS to inform running apps of major events in the system, e.g., incoming text message, connectivity changes, etc. System broadcasts would interrupt normal app execution to invoke the broadcast handlers registered by the app. Different broadcasts would have different impacts on the app depending on the current program state. Thus, system broadcasts serve as useful interference. JazzDroid' system broadcast injector would emulate three families of system broadcasts, indicating the occupancy of certain hardware, network connectivity changes and callbacks from hardware, respectively. Some apps would only respond a subset of these broadcasts and thus the injector would identify those to focus interference.

Network Issues Injector. Modern apps use network for extensively purposes, e.g., fetching content and images, advertising, etc. Many functionalities rely heavily on the availability of network [19]. Thus interfering network traffic is useful for mobile testing. The network issues injector interposes on HTTP requests from the app and manipulates their responses to simulate network delay and connectivity instability.

Computation Slowdown Injector. There could be other apps running in the background, which affects foreground tasks of the app in question. The computation slowdown injector performs heavy computation periodically in a dedicated thread to simulate the case when app methods could run slower than expectation due to high CPU utilization.

UI Mutation Injector. Mobile apps often spawn worker threads to perform blocking or lengthy operations (such as network requests, image decoding, etc) asynchronously to keep the UI thread responsive to user inputs. To process a user input, the event handler running in the UI thread starts by creating one or multiple asynchronous tasks, each fulfilling a piece of the work. When the asynchronous tasks complete, results are transmitted back to the UI thread to update the UI. Asynchronous tasks are not instantaneous. If the UI state changes when an asynchronous task is not yet completed, the running asynchronous task might need to be cancelled and restarted. For instance, modern apps load large images from the network asynchronously such that

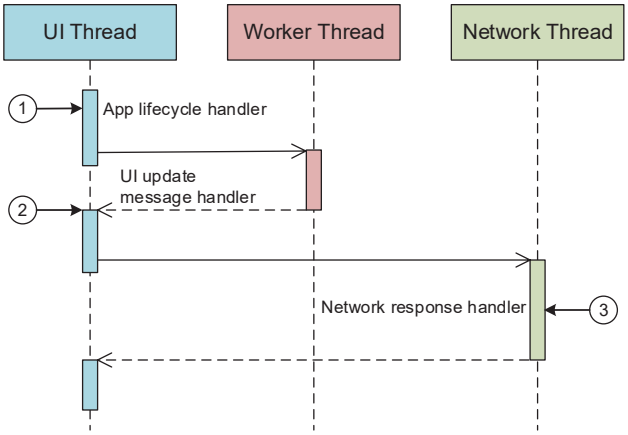


Figure 4: Important program points for injectors to trigger interference. "App lifecycle handler" typically rearranges the UI elements for the entire app. "UI update message handler" handles updates for one or multiple UI elements. "Network response handler" gives network data back to the app and is very likely to trigger UI update.

image loading does not block user inputs. The image widget holding the image could change its state while the image is still loading. A user could navigate to a new app window (the widget disappears) or have a device orientation change (the widget is resized). In such cases the asynchronous task loading the image should be cancelled and/or restarted to fetch the image suitable for the new UI widget state. Yet, the possibility of UI state changes during asynchronous tasks is often overlooked. Thus mutating UI state while asynchronous tasks are running is a useful interference for improve mobile testing. JazzDroid's UI mutation injector triggers valid UI state change while asynchronous tasks are still running. There are many possible UI state changes. In practice, JazzDroid only emulates screen orientation change which causes to recreate all the UI widgets on the screen. This interference is a valid random event for the app and could effectively stress all the running asynchronous tasks to handle UI changes properly.

2) Instrumentation Points: Certain interference only affects app execution when the app is performing related tasks. For instance, the network changes have very limited

impact on the app if the app is not performing any network requests. To improve the impact from its injectors, JazzDroid identifies several critical program points that app is performing tasks related to one or more interference injectors. Figure 4 illustrates these instrumentation points. First, the mobile system defines several callback functions for apps to handle major app state change (e.g., navigate to a new app window, task switched out, etc). Such callbacks are called app component life cycle methods (shown as point 1 in Figure 4). After the life cycle methods are invoked, the app typically involves the creation, destruction and state changes for a large number of UI elements. Thus UI-related interference is most effective when such life cycle methods happen. In addition, while the app is running, the UI thread is the only thread that can update UI state. All other worker/network/background threads need to pass messages to the UI thread to indirectly update UI. Thus the UI mutation injector also interferes the app when the UI thread receives any message (shown as point 2 in Figure 4). Aside from UI states, the network issue injector is most interested in network activities. Thus, JazzDroid intercepts all network request and response handlers to potentially emulate a network issue (network delay or failed request), shown as point 3 in Figure 4. The UI mutation injector would also trigger when a network response is processed, as modern apps are very likely to update UI upon receiving data from the network.

3) Bytecode Instrumentation: The bytecode instrumenter of JazzDroid uses two operators to transform app bytecode for interference injection. This process is automatic and transparent to app developers. JazzDroid runtime components have the instrumentation rules for the two operators to instruct the instrumenter to position injector at desired program points.

```
android.app.Activity.foo(x, y); // original API call
```

HookAPICall operator

```
JazzDroidHooks.hookFoo(x, y);    // modified call

class JazzDroidHooks {
  static void hookFoo(int x, int y) { ... }
}
```

Figure 5: The transformation effect of `HookAPICall` operator at source code level (for illustration).

HookAPICall Operator searches for function calls to an Android API of interest and redirects the calls to a hook function of JazzDroid. This transformation is shown in Figure 5 where the instrumenter modifies function call instructions of interest to achieve this. This transformation can not only intercept static method calls, but also virtual calls. This operator allows JazzDroid to interpose on arbitrary system calls from app code and manipulate the return

```
class MyActivity extends android.app.Activity {
  @Override
  void onCreate(Bundle b) {  // original callback
    ...
  }
}
```

WrapCallback operator

```
class MyActivity extends android.app.Activity {
  @Override
  void onCreate(Bundle b) {  // modified callback
    JazzDroidHooks.hookEntry(this, b);
    ...
    JazzDroidHooks.hookExit(this, b);
  }
}

class JazzDroidHooks {
  static hookEntry(Activity _this, Bundle b) {...}
  static hookExit(Activity _this, Bundle b) {...}
}
```

Figure 6: The transformation effect of `WrapCallback` operator at source code level (for illustration).

value of these calls. JazzDroid uses this operator to intercept and manipulate network requests where JazzDroid's network issue injector would simulate network exceptions or network delays.

WrapCallback Operator modifies the function body of a target callback functions to insert calls to hook functions in JazzDroid at the callback entry and exit, as illustrated in Figure 6. Callback methods are usually invoked by Android programming framework to inform the app of certain state change or user inputs. This operator helps to implant interference logic at various callback methods so as to trigger interference more effectively. For instance, UI mutation injector uses this operator to emulate device orientation change in app component lifecycle methods and UI callback methods.

JazzDroid's instrumenter fulfills tasks similar to aspect-oriented programming (AOP) tools [7], while functions on compiled form of app code (Dalvik bytecode).

D. Reproducing Faulty Execution

The interference bookkeeper logs all user input events and all the interference injected by the four injectors. When a problem (denoted as X) is detected during execution, the bookkeeper outputs a trace of input events and injected interference events happened before the problem, sorted chronologically. This trace is denoted as *the full trace*. Then the bookkeeper generates a test script based on the full trace for the app developer to reproduce the faulty execution for problem X.

The generation process contains three steps: 1) the bookkeeper picks *all user input events* but *part of the interference*

events and generates an *experimental trace*; 2) the app is restarted and the experimental trace is replayed R times (R being an odd integer). When replaying, user input events are replayed faithfully[1] and interference events are emulated as system-wide automation (e.g., network cut-off, orientation change, incoming call, etc). All events are emitted strictly according to the timing logged previously; 3) if the app encounters the same problem X in the majority ($> 50\%$) of these test runs, the experimental trace is marked as a *stable* recreation of problem X. If the occurrence is minor ($< 50\%$ *but* $> 0\%$), the experimental is treated as an *unstable* recreation of problem X. If problem X is not recreated after replaying the experimental trace, then the trace is a *negative* recreation.

The generation process is repeated for all possible combinatorics cases of interference events. Quantitatively, suppose there are N user inputs events and M interference events in the full trace, there are 2^M possible experimental traces. If all experimental traces for X are negative, then problem X is not reproducible, which is marked as a false positive case for JazzDroid. If any experimental trace generates a stable recreation, then JazzDroid reports the problem X along with the stable experimental trace for the app developer to reproduce the problem. If none experimental trace is stable, then the unstable experimental trace with the least number of interference events and highest recreation success rate is selected and reported.

The interference bookkeeper provides the following vital guarantees: 1) all reported problems with stable recreation are reproducible experimentally; 2) the test script generated for recreating the problem has equivalent human interaction steps; 3) false positive cases are pruned; 4) problems with unstable recreation can be reproduced at least once with no more than R trials. Note that such unstability is inevitable as some program problems could be results of heisenbugs [40], [41].

E. Balancing Interference Parameters

Runtime injectors would trigger interference at designated program points while the app is running. Yet, injectors only trigger according to a certain probability. This is because interference exists in the reality but happens with low occurrence probability. There is an inherent trade-off for interference-aware mobile testing where a higher triggering probability is less likely in the field but could potentially increase the number of detected problems. JazzDroid adopts a procedure to balance its interference parameters and avoid over-aggressiveness. JazzDroid quantifies the aggressiveness for the four injectors with the following four heuristic equations:

[1]For Android, UI Automator [46] can faithfully mimic arbitrary touch-screen inputs from a human user.

$$A_u = p_{app} + p_{ui} + p_{network} + (1 - \frac{\tau_u}{T_u}) \tag{1}$$

$$A_b = \frac{\sum^i b_i}{\ln(1+\tau_b)} \tag{2}$$

$$A_c = r \tag{3}$$

$$A_n = p_{delay} t_{delay} + p_{loss} T_{loss} \tag{4}$$

The UI mutation injector triggers device orientation change at three instrumentation points with a probability of $p_{app}, p_{ui}, p_{network}$, respectively as well as periodically with an interval of τ_u, where $\tau < T_u = 60 seconds$. JazzDroid uses A_u to characterize the aggressiveness of this injector.

The system broadcast injector sends one broadcast every τ_b seconds. b_i is a boolean variable indicating if the i-th system broadcast would be injected. A_b represents the aggressiveness of the system broadcast injector.

The computation slowdown injector operates periodically with an interval of T_c seconds (in our case, 10 seconds). Each time this injector would perform computation lasting for rT_c seconds, where $0 < r < 1$. Thus, $A_c = r$ captures the aggressiveness of this injector.

The network issue injector operates with three parameters, $p_{delay}, t_{delay}, p_{loss}$. p_{delay} is the probability to add a delay to a network request. t_{delay} is the network delay introduced. p_{loss} is the probability to trigger a failed request. Upon triggering a failed request, the injector would first inject a delay of T_{loss} and then throw a network exception. T_{loss} is a constant value of 10 seconds which is the default timeout value for network client library. Note that $p_{delay} + p_{loss} \leq 1$ and $t_{delay} < T_{loss}$. A_n in equation 4 characterizes the aggressiveness of the network issue injector.

JazzDroid bootstraps the balancing process with all-zero parameters, where JazzDroid would add no interference to normal execution. Then JazzDroid counts the problems found with current parameter settings and computes the *Efficiency Index* as

$$EI = \frac{\text{issues}}{\text{time}} \cdot \frac{1}{1 + W_u A_u + W_b A_b + W_c A_c + W_n A_n} \tag{5}$$

The first term represents the number of issues detected per second. The second term weakens the first term with the inverse aggressiveness of all injectors. Four weighting factors W_u, W_b, W_c, W_n are introduced to adjust the value range of four aggressiveness indices, such that each does not overshadow others. JazzDroid iteratively increases one of the variables (adding aggressiveness to one of the injectors), reruns the tests and obtains a new efficiency index. The per-iteration increment is 5% for any of the probabilities, one second for $t_{delay}, \tau_b, \tau_u$, and 0.05 for r. This iterative process stops (converges) when the EI stops increasing.

Table I: Normalized number of detected problems with baseline tests and test enhanced by JazzDroid.

	Crash	ANR	Functional	UI	Total
Manual	0.27	0.28	0.43	0.59	1.58
Monkey	0.15	0.02	0.15	0.28	0.60
Test Scripts	0	0	0	0.43	0.43
Average	0.19	0.13	0.26	0.42	**1**
Manual†	1.07	0.89	0.43	3.78	6.17
Monkey†	0.28	0.28	1.90	2.76	5.22
Test Scripts†	0.43	0	1.30	3.04	4.77
Average†	0.63	0.53	1.24	3.21	**5.61**

† tests enhanced by JazzDroid

F. Implementation

JazzDroid's offline tools run once to instrument the app to be JazzDroid ready and obtain interference parameters suitable for the target app. The offline tools include a custom bytecode instrumenter based on smali [21] and PATDroid [27], a strategy optimizer based on heuristic algorithms and a test driver based on Appium [25]. JazzDroid runtime components have 4.5k Java SLOC, compiling down to Dalvik bytecode and are merged into the target app code during the development phase, adding a negligible footprint of 52 kB. JazzDroid is only a development dependency for the target app and its footprint will not appear in the release build of the app.

IV. Evaluation

This section evaluates JazzDroid in terms of its capability of detecting new bugs, the characteristics of detected bugs, experiences we learned from case studies of real apps and practical considerations when adopting JazzDroid.

A. Methodology

We present the real-world apps we choose for JazzDroid evaluation and the test environments. Since JazzDroid is a tool that is compatible with most existing testing techniques, we adopt multiple baseline testing methods that are common in industrial practice and compare them with tests enhanced by JazzDroid.

Datasets: The 105 apps used for evaluation are collected from three sources, falling into representative app categories, including music, news, social media, shopping, productivity, tools, etc. (1) Top-hit apps from Google Play and Tencent market (the largest app market in China). All Apps have a user rating over 4.3 and the total downloads sum up to over ten billion. These apps are mature and well-tuned apps that serve users across the global over many years. (2) Apps with public bug reports from Testerhome [26], the largest Android testing community in China. We cross reference the problems detected by JazzDroid with public information to bug reports to testify that JazzDroid can detect known problems. (3) Popular open-source app projects on GitHub (2,000 stars+) and F-Droid [44].

Environments: For test devices, we use three real Android devices and three emulators, covering Android version 4.4, 5.0 and 6.0 (representing 60% end users [30]). Real devices have different hardware capability, with release years from 2013 to 2015. Our Android devices and emulators have low-latency Wi-Fi connectivity and real devices have additional 4G cellular connectivity. In our experiments, JazzDroid could generate switches between Wi-Fi and 4G networks on real devices.

Baseline Testing Methods: We apply three testing methods commonly adopted by the industry. 1) Recorded manual testing: we first manually test an app for its main functionality and record the touchscreen inputs with a tool [45]. The recorded input trace is then replayed to recreate same execution during subsequential experiments. During these tests, we ensure that over 90% of app windows (Activities) are covered. 2) Automated UI testing: the official Android SDK ships with a stress testing utility called `monkey` [20] to randomly exercise an app. This tool has fairly high adoption for automatic app exercising. 3) Hand-crafted Appium test scripts: UI automation is also a common technique used for automated test cases, which encloses a sequence of simulated user interaction (clicks, text inputs, etc) with assertions for correctness. Our hand-crafted Appium test scripts cover over 90% of the app windows, with assertions on app correctness, UI element placements, etc.

Comparison Study and Bug Identification: We first perform the three baseline tests on every app in the app dataset. Then we instrument the app with JazzDroid and rerun baseline tests with the same settings (same input trace, monkey seed and test scripts). During these experiments, we widely collect screenshots (on a per-action basis), system log (logcat), and JazzDroid log for analysis. All (unique) bug/issue screenshots and logs are manually identified and made public [29]. We repeat each baseline test three times and report the all detected problems.

In our experiments, we choose manual bug identification because no prior knowledge of these bugs is available and no test checkpoints and assertions are available due to the absence of app source code. Should app developers adopt JazzDroid in development, this tedious manual bug identification process can be largely replaced by designated test checkpoints and assertions.

B. Effectiveness

We first present the overall effectiveness of JazzDroid in detecting bugs and issues. We run the original apps and the JazzDroid-enabled apps through three kinds of baseline tests and report detected problems. Due to test time difference (across apps and testing methods), we report the number

of unique bugs/issues detected per minute. Also all results are normalized to total bugs-per-minute of the average case of the original app for comparison. We classify detected problems into four categories: 1. *Crash bugs*: the app in test encounters unhandled exceptions and the system forcefully terminates it with the notorious "App has stopped" popup dialog; 2. *ANR (Application Not Responding) bugs*: the app runs lengthy operations in the UI thread causing the app to be unresponsive to any user inputs. The system forcefully terminates the app with an "App is not responding" popup dialog; 3. *Functional bugs*: The app fails to fulfill its designated functions. For example, the window shows incorrect content during navigation, and certain functional widget does not respond to user inputs, etc; 4. *UI issues*: The app has oversized, empty-content, overlapping or misplaced UI widgets.

Overall, 5.6x more problems detected. Table I outlines the improved bug discovery capability (5.6x on average) of JazzDroid for three baseline testing methods. Further examination on these problems reveals that over 70% of the newly detected problems can only be reproduced with the presence of environmental interference. We will further present the breakdown of problems detected by individual interference injector in Section IV-C and case studies in Section IV-D to understand how interference uncovers these subtle problems.

Interfering vs. Input Generation. The Dynodroid [36] is an advanced input generator for Android apps, which detects bugs in 12 apps from F-Droid [44]. Unfortunately, Dynodroid targets Android version < 3.0 and can not work on modern apps. We apply JazzDroid to the same app dataset evaluating Dynodroid to compare effectiveness. Only four out of 12 apps are runnable on modern Android (5.0 and 6.0) because most of them are not forward compatible (many due to the dynamic permission request from 5.0). For the four apps, JazzDroid not only detects the bugs identified by Dynodroid, but also uncovers some more issues caused by screen orientation change and network packet loss, both being absent from Dynodroid's design.

C. Bug Characteristics

Table II: Bugs detected by baseline testing methods vs. individual JazzDroid components. "Jointly" stands for bugs detected by the joint effect of more than one JazzDroid components.

	Crash	ANR	Functional	UI	Subtotal
UI Mutation	7	0	8	42	57
Network	1	4	6	11	22
Broadcast	1	0	2	3	6
Computing	0	4	0	0	4
Jointly	2	0	6	2	10
Baseline	4	3	5	9	21

Each bug reported by JazzDroid is associated with a test script generated by the interference bookkeeper to reproduce the faulty execution. We inspect the generated test script to characterize these bugs. Table II presents the breakdown of which injector contributes to the discovery of the problem. The "jointly" row indicates that detected problems are a joint effort from more than one injector (e.g., orientation change after a network failure). The "baseline" row represents the problems detected with baseline testing methods without JazzDroid.

Bug Validity and Recreation Stability. There are in total 99 bugs detected by JazzDroid, as shown in Table II. All these bugs come with a test script that can reproduce the problem and no false positive is observed. Among these, 95 problems ($\approx 96\%$) can be stably reproduced, i.e., at least two out of three runs of the generated test script can reproduce the problem in question. Four problems can be instable reproduced, i.e., one out of three runs is successful. These problems involve one interference event from the network injector and one from the broadcast receiver, which only happen when such interference meets certain timing requirements.

Single Interference Being Dominant Root Cause. We also notice that for the 99 problems detected by JazzDroid, 85 problems ($\approx 85\%$) can be reproduced with an experimental trace of only one interference event (recall Section III-D). That indicates that only one particular interference event is needed to trigger the problem. The remaining 14 problems require the combination of two interference events (10 jointly cases, 2 network injector cases and 2 broadcast injector cases). The common pattern of such issues is to impose orientation change and network delay simultaneously, which jointly changes the execution order of events, causing state inconsistency or UI mismatch. A typical scenario for such overlapping is to rotate the screen quickly twice, when the first orientation change triggers several requests, and the second collides with network issues generated from former requests. Such rare (but possible) scenarios can be only created by a system with fine-grained control over every network requests.

D. Case Studies and Experiences

We present two case studies that outline two common patterns that fail more than one apps in our dataset. Also we share the experiences of using JazzDroid with real apps.

Dangling Reference Problem. We notice that many apps experience problems of various severity when a device orientation change is issued at particular moments of app execution. A device orientation change would cause the app to recreate an app window object (`Activity`) but inherit some old UI widget objects from the previous window. In some cases, the inherited UI widgets keep a reference to the old window object, which will be destroyed immediately after the orientation change process is finished. Then, when the new window is shown to the user and user interacts with

Figure 7: Device orientation change at the dialog crashes BBC news app with `NullPointerException`.

one of the inherited widgets, certain errors happen because the inherited widgets are still bound to the destroyed old window object. Such dangling reference problem affects four apps in our dataset. We elaborate with the BBC news case, where JazzDroid detects a crash bug due to this matter.

BBC News app is the official Android client for BBC, the famous news media. When the app starts for the first time, it shows a popup dialog for push notification authorization. However, if a device orientation change happens at this moment and the user clicks on either the "Yes" or "No" button of this dialog, then this app will crash due to a `NullPointerException`. We decompile its code and locate the problematic `Activity`:

```
// bbc.mobile.news.v3.app.TopLevelActivity.onCreate() initialization
if (appSettings.firstLaunch) { // if it's the first time to launch the app
  // create a dialog, binding it to the current window object
  new PushDialog(this).show(this);
} else {
  // following launches or orientation change
  // will not create the dialog again
  appSettings.firstLaunch = false;
}
```

At the first launch of this app, the popup dialog `PushDialog` is created, bound to the window object (`this`) and displayed. The `else` branch ensures that this dialog is not shown to the user in the following launches. However, if an orientation change happens before the user dismisses this dialog, a new window object will be created and the dialog object will be inherited. Then, in the invocation of `onCreate` on the new window object, the app believes that it is not the first launch and no PushDialog is created. The old dialog object is then displayed to the user with a reference to the old (destroyed) window object. When the user clicks on the dialog buttons, the app crashes with a null pointer. JazzDroid discovers this bug with its UI mutation injector triggering orientation change after this life cycle method (`onCreate`). In other similar problems detected by JazzDroid, we notice crashes and mismatched UI contents due to the similar reason. An easy fix is to store an additional boolean flag whether the dialog has been answered and then create/recreate the dialog whenever the dialog is not answered. This bug has been reported to BBC news developers.

(a) Premium plans and payment. (b) Missing content.

Figure 8: WPS office functional issue and ANR.

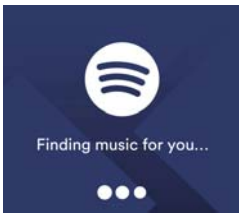


Figure 9: Spotify splash window stuck when packet loss.

Unhandled Exception Paths. A considerable number of Android apps build their key functionality upon online resources and fetch them through HTTP protocol. However, network connectivity is often uncertain and an app should handle possible network exceptions and network delays. With JazzDroid, we discover common bug patterns regarding network usage, resulting in crashes, ANRs and functional issues.

WPS Office is a popular document creation and editing app, with over a billion downloads in the market. JazzDroid discovers bugs related to network when an app is requesting for premium service rates over network. In usual cases, the app would display the possible premium plans in a UI widget and a user could choose to see the price, shown in Figure 8a. However, if the particular network request experiences a failure (`IOException`) or a network delay, the app would have an empty UI widget displayed and sometimes the app would hang and eventually throw an ANR crash, shown in Figure 8b. After examining the decompiled code, we notice the app would wait for the premium rates in the UI thread with infinite retries, which is an anti-pattern for using data from unreliable networks.

JazzDroid also detects a similar bug in Spotify, a popular music and radio app. As shown in Figure 9, a guide page appears at first time to assist new user to join and find music. However, if the network request for querying music for the new user fails, the application will stall in this UI forever.

In this specific case, even the back button cannot finish the stalled activity. The app becomes literally unusable, similar or even worse than an explicit crash. JazzDroid would stress every network requests with probabilities of being delayed or failed, such that an app would better understand the consequence of network instability.

E. Discussion

JazzDroid is rather a test enhancement tool than a standalone tool for mobile testing. In practical use, JazzDroid needs to team up with other tools to achieve better performance [33], [35], [39].

First, JazzDroid adds interference to baseline tests. Thus, the effectiveness of interference is largely coupled with the quality of baseline tests, i.e., the code coverage of baseline tests. If baseline tests can not cover certain app functionality, JazzDroid can merely discover problems in it.

Second, JazzDroid adopts a heuristic algorithm to obtain interference parameters to avoid detecting bugs that are caused by purely excessive interference. A high interference frequency indicates better discovery but means less likely to happen in reality. Nevertheless, this balancing mechanism could cause JazzDroid to miss bugs that happen with extremely low probability in reality. We argue that such cases could be complemented by a production-stage error reporting mechanism.

Third, the goal of JazzDroid is to add interference and lead app to a faulty execution state. However, identifying faulty states is not within the scope of JazzDroid. This requires better test assertions, automated UI layout checkers [3], [28], exception tracking tools, etc.

V. Related Work

JazzDroid is not the first to discover that environmental interference could impact app execution in various aspects. Previous papers discover that app could have high energy consumption due to background network tasks [11] or background activities [12], [13]. Wu et al. further explore to crash the Android system by trapping uncaughtException handlers. These known bug/issue patterns highlight the need to consider environmental interference when testing mobile apps.

Context-aware testing. Previous work outlines that certain contexts, such as device models, device hardware states, system settings, slow network connection, could cause an app to malfunction in different ways [31]. Thus some approaches try to emulate black-box [16], [36] or grey-box [34] troublesome contexts by software on Android, on Windows Phone [14], [32]. Some approaches emulate troubles in hardware [15] while testing an app. Other approaches aim to emulate as many different devices as possible to detect UI performance regressions [3] and device-specific UI issues [9]. These approaches are similar to JazzDroid in linking the environmental factors with app problems, yet different because JazzDroid injects dynamically while the app is running. In essence, JazzDroid instruments an app to gain knowledge of its running status and triggers interference that could cause program state change to uncover faulty code paths.

Automated testing. Automated testing techniques (fuzzers) are a popular direction to explore fully automatic or semi-automatic input generation for testing apps. The Android SDK ships with a tool called monkey [20] that randomly clicks on screen to test an app. Zheng et al. [5] address the limitation of monkey [20] in testing industrial apps by avoiding repetitious exploration patterns and reducing execution runtime of the utility. Appdoctor [1] utilizes approximate execution and false-positive pruning to achieve efficient UI exercising. Arnatovich et al. [8] explore and analyze dynamically UI hierarchy dump of an app to perform tailored input generation. Anbunathan et al. [4] propose a method to simulate user interface events by parsing UML Sequence diagram. Automated testing techniques are complementary to JazzDroid, where fuzzers focus on generating user inputs and JazzDroid focuses on generating interference. When used with a powerful fuzzer that could generate user inputs with high code coverage, JazzDroid can discover more problems. Chronicler [42] and SymCrash [43] introduce methods about reproducing failures.

VI. Conclusion

Modern applications call for robustness, responsiveness and state consistency during execution. Environmental interference is the main trigger of bugs that compromise such standards; therefore, testing with interference is a vital process to guarantee application quality. This paper presents JazzDroid, an automated Android gray-box fuzzer that injects into applications various environmental interference on the fly to detect issues. JazzDroid is designed to be lightweight, non-intrusive and highly compatible, requiring no change to application source code, testing techniques or testing environments. Our results show that JazzDroid can help application developers to detect significantly more bugs and to uncover subtle code paths that potentially fail an application. We believe that JazzDroid is a promising testing technique that is practical, easily-deployable, complementary to current testing framework and conductive to prospering the existing Android automated testing ecosystem.

Acknowledgment

This work was supported in part by National Key Research & Development Program of China (No. 2016YFB1000502), National NSF of China (NO. 61672344, 61525204, 61732010), and Shanghai Key Laboratory of Scalable Computing and Systems.

REFERENCES

[1] Hu, G., Yuan, X., Tang, Y., Yang, J., "Efficiently, effectively detecting mobile app bugs with appdoctor," Proceedings of the Ninth European Conference on Computer Systems. ACM, 2014, p. 18.

[2] Wu, J., Liu, S., Ji, S., Yang, M., Luo, T., Wu, Y., Wang, Y., "Exception beyond Exception: Crashing Android System by Trapping in'Uncaught Exception'," Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), 2017 IEEE/ACM 39th International Conference on. IEEE, 2017, pp. 283-292.

[3] Gómez, M., Rouvoy, R., Adams, B., Seinturier, L., "Mining test repositories for automatic detection of UI performance regressions in Android apps," Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on. IEEE, 2016, pp. 13-24.

[4] Anbunathan, R., Basu, A., "Automatic Test Generation from UML Sequence Diagrams for Android Mobiles," International Journal of Applied Engineering Research 11.7, 2016, pp. 4961-4979.

[5] Zheng, H., Li, D., Liang, B., Zeng, X., Zheng, W., Deng, Y., Lam, W., Yang, W., Xie, T., "Automated test input generation for android: Towards getting there in an industrial case," Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track. IEEE Press, 2017, pp. 253-262.

[6] Amalfitano, D., Amatucci, N., Memon, A. M., Tramontana, P., Fasolino, A. R., "A general framework for comparing automatic testing techniques of Android mobile apps," Journal of Systems and Software 125, 2017, pp. 322-343.

[7] Liu, J., Wu, T., Deng, X., Yan, J., Zhang, J., "Insdal: A safe and extensible instrumentation tool on dalvik byte-code for Android applications," Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on. IEEE, 2017, pp. 502-506.

[8] Arnatovich, Y. L., Ngo, M. N., Kuan, T. H. B., Soh, C., "Achieving High Code Coverage in Android UI Testing via Automated Widget Exercising," Software Engineering Conference (APSEC), 2016 23rd Asia-Pacific. IEEE, 2016, pp. 193-200.

[9] Kaasila, J., Ferreira, D., Kostakos, V., Ojala, T., "Testdroid: automated remote UI testing on Android," Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia. ACM, 2012, p. 28.

[10] Zhauniarovich, Y., Philippov, A., Gadyatskaya, O., Crispo, B., Massacci, F., "Towards black box testing of android apps," Availability, Reliability and Security (ARES), 2015 10th International Conference on. IEEE, 2015, pp. 501-510.

[11] Rosen, S., Nikravesh, A., Guo, Y., Mao, Z. M., Qian, F., Sen, S., "Revisiting network energy efficiency of mobile apps: Performance in the wild," Proceedings of the 2015 Internet Measurement Conference. ACM, 2015, pp. 339-345.

[12] Chen, X., Jindal, A., Ding, N., Hu, Y. C., Gupta, M., Vannithamby, R., "Smartphone background activities in the wild: Origin, energy drain, and optimization," Proceedings of the 21st Annual International Conference on Mobile Computing and Networking. ACM, 2015, pp. 40-52.

[13] Lee, J., Lee, K., Jeong, E., Jo, J., Shroff, N. B., "Context-aware application scheduling in mobile systems: What will users do and not do next" Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing. ACM, 2016, pp. 1235-1246.

[14] Liang, C. J. M., Lane, N., Brouwers, N., Zhang, L., Karlsson, B., Chandra, R., Zhao, F., "Contextual fuzzing: automated mobile app testing under dynamic device and environment conditions," Microsoft. Microsoft, nd Web , 2013.

[15] Liang, C. J. M., Lane, N. D., Brouwers, N., Zhang, L., Karlsson, B. F., Liu, H., Liu, Y., Tang, J., Shan, X., Chandra, R., Zhao, F., "Caiipa: Automated large-scale mobile app testing through contextual fuzzing," Proceedings of the 20th annual international conference on Mobile computing and networking. ACM, 2014, pp. 519-530.

[16] Zhang, L. L., Liang, C. J. M., Zhang, W., Chen, E., "Towards a contextual and scalable automated-testing service for mobile apps," Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications. ACM, 2017, pp. 97-102.

[17] Li, Y., Yang, Z., Guo, Y., Chen, X., "DroidBot: a lightweight UI-Guided test input generator for android," Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on. IEEE, 2017, pp. 23-26.

[18] Linares-Vásquez, M., Bavota, G., Tufano, M., Moran, K., Di Penta, M., Vendome, C., Bernal-Crdenas, C., Poshyvanyk, D., "Enabling mutation testing for android apps," arXiv preprint arXiv:1707.09038, 2017.

[19] Nikravesh, A., Yao, H., Xu, S., Choffnes, D., Mao, Z. M., "Mobilyzer: An open platform for controllable mobile network measurements," Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services. ACM, 2015, pp. 389-404.

[20] "UI/Application Exerciser Monkey," https://developer.android.com/studio /test/monkey.html.

[21] "JesusFreke/smali," https://github.com/JesusFreke/smali.

[22] "openstf/minitouch," https://github.com/openstf/minitouch.

[23] "Android network libraries - AppBrain," https://www.appbrain.com/stats/libraries/tag/network/android-network-libraries.

[24] "openstf/adbkit-apkreader," https://github.com/openstf/adbkit-apkreader.

[25] "Appium," http://appium.io/.

[26] "TesterHome: Bug Lighthouse," https://testerhome.com/bugs.

[27] "PATDroid," https://github.com/mingyuan-xia/PATDroid.

[28] Méndez-Porras, A., Alfaro-Velásco, J., Jenkins, M., Martínez Porras, A., "A User Interaction Bug Analyzer Based on Image," CLEI Electronic Journal 19.2, 2016, 4-4.

[29] "JazzDroid evaluation screenshots," https://www.dropbox.com/sh/0xhb37 btbxrnbb-v/AAB5znAvGAzqZLURItzb9bHca.

[30] "Dashboards," https://developer.android.com/about/dash-boards/index.html.

[31] Joorabchi, Mona Erfani, Ali Mesbah, and Philippe Kruchten, "Real challenges in mobile app development," Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on. IEEE, 2013, pp. 15-24.

[32] Ravindranath, L., Nath, S., Padhye, J., Balakrishnan, H., "Automatic and scalable fault detection for mobile applications." Proceedings of the 12th annual international conference on Mobile systems, applications, and services. ACM, 2014, pp. 190-203.

[33] Linares-Vásquez, M., Vendome, C., Luo, Q., Poshyvanyk, D., "How developers detect and fix performance bottlenecks in android apps," Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on. IEEE, 2015, pp. 352-361.

[34] Hao, S., Liu, B., Nath, S., Halfond, W. G., Govindan, R., "PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps," Proceedings of the 12th annual international conference on Mobile systems, applications, and services. ACM, 2014, pp. 204-217.

[35] Tian, Y., Nagappan, M., Lo, D., Hassan, A. E., "What are the characteristics of high-rated apps? a case study on free android applications," Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on. IEEE, 2015, pp. 301-310.

[36] Machiry, A., Tahiliani, R., Naik, M., "Dynodroid: An input generation system for android apps," Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, 2013, pp. 224-234.

[37] "LeakCanary," https://github.com/square/leakcanary.

[38] Gu, T., Cao, C., Liu, T., Sun, C., Deng, J., Ma, X., Lü, J., "AimDroid: Activity-Insulated Multi-level Automated Testing for Android Applications," Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on. IEEE, 2017, pp. 103-114.

[39] Das, Teerath, Massimiliano Di Penta, and Ivano Malavolta, "A Quantitative and Qualitative Investigation of Performance-Related Commits in Android Apps," Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on. IEEE, 2016, pp. 443-447.

[40] Maiya, P., Kanade, A., Majumdar, R., "Race detection for Android applications," ACM SIGPLAN Notices Vol. 49, No. 6, ACM, 2014, pp. 316-325.

[41] Hsiao, C. H., Yu, J., Narayanasamy, S., Kong, Z., Pereira, C. L., Pokam, G. A., Peter M. Chen, Flinn, J., "Race detection for event-driven mobile applications," ACM SIGPLAN Notices, Vol. 49, No. 6, ACM, 2014, pp. 326-336.

[42] Bell J, Sarda N, Kaiser G. "Chronicler: Lightweight recording to reproduce field failures," Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, 2013, pp. 362-371.

[43] Cao Y, Zhang H, Ding S. "SymCrash: selective recording for reproducing crashes," Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. ACM, 2014, pp. 791-802.

[44] "F-Droid," https://f-droid.org/.

[45] "ReplayKit," https://github.com/appetizerio/replaykit.

[46] "UI Automator." https://developer.android.com/training/testing/ui-automator.html.