RERAN: Timing- and Touch-Sensitive Record and Replay for Android

Lorenzo Gomez
University of California
Los Angeles, USA
lorenzo@cs.ucla.edu

Iulian Neamtiu
University of California
Riverside, USA
neamtiu@cs.ucr.edu

Tanzirul Azim
University of California
Riverside, USA
mazim002@cs.ucr.edu

Todd Millstein
University of California
Los Angeles, USA
todd@cs.ucla.edu

***Abstract*—Touchscreen-based devices such as smartphones and tablets are gaining popularity, but their rich input capabilities pose new development and testing complications. To alleviate this problem, we present an approach and tool named RERAN that permits record-and-replay for the Android smartphone platform. Existing GUI-level record-and-replay approaches are inadequate due to the expressiveness of the smartphone domain, in which applications support sophisticated GUI gestures, depend on inputs from a variety of sensors on the device, and have precise timing requirements among the various input events. We address these challenges by directly capturing the low-level event stream on the phone, which includes both GUI events and sensor events, and replaying it with microsecond accuracy. Moreover, RERAN does not require access to app source code, perform any app rewriting, or perform any modifications to the virtual machine or Android platform. We demonstrate RERAN's applicability in a variety of scenarios, including (a) replaying 86 out of the Top-100 Android apps on Google Play; (b) reproducing bugs in popular apps, e.g., Firefox, Facebook, Quickoffice; and (c) fast-forwarding executions. We believe that our versatile approach can help both Android developers and researchers.**

Index Terms: Record-and-replay, Google Android.

I. Introduction

Smartphones and tablets have become powerful and increasingly popular platforms for a host of software applications (apps), ranging from office suites to games to social networking tools. The appeal of these new platforms stems in part from the rich capabilities for user interaction via touchscreens, as well as the diverse set of sensors (e.g., accelerometer, compass, GPS) that can be leveraged to drive app behavior. However, these new features also pose important challenges for software development, testing, and maintenance.

In this paper we focus on the challenge of accurately recording an app's execution in order to support automatic replay. Record-and-replay systems can be valuable tools for many software engineering tasks, including program debugging, testing, and understanding. Unfortunately, past approaches to record-and-replay for GUI-based applications are not adequate on smartphones. These tools capture user activity in terms of discrete high-level actions, which cannot easily express complex gestures and do not support capture and replay for events from other sensors.

For illustration, consider the task of recording and replaying the popular game Angry Birds—a fully graphical app with no standard GUI elements, and very timing sensitive. Our approach has no trouble carrying out this task; the reader is invited to watch our YouTube videos [1] showing RERAN doing a record-and-replay of Angry Birds. In fact, our approach can successfully replay 86 out of the Top-100 free Android apps on Google Play. However, existing GUI-level approaches have multiple difficulties achieving record and replay for such apps, as explained next.

First, GUI-level tools (e.g., Android GUITAR [2], Abbot [3], HP WinRunner [4], IBM Rational Robot [5] and GUI crawler [6]) typically employ the *keyword action* paradigm in order to replay user-interface events. In this paradigm, input events are abstracted from the concrete GUI level to a higher-level representation that uses a GUI object's handle or name within the system to interact with it, e.g., `click TextBox1` or `type ''Test123''`. Angry Birds, however, has a single Activity (screen) so there are no GUI elements such as multiple screens, frames, menus, and input boxes that a keyword action tool can "hook" into.

Second, while the aforementioned tools can work well for *discrete* desktop point-and-click GUIs, touchscreen platforms use a much richer GUI paradigm based on *continuous* gestures such as swipe, zoom, and pinch. Even assuming an app has multiple screens and menus that a GUI recorder can hook into, touchscreen gestures are difficult to capture and represent as keyword-actions, since they can occur at arbitrary parts of the GUI (e.g., to zoom in on a map), and/or can require low-level precision in order to accurately replay (e.g., to slingshot a bird in Angry Birds). This precision is required during record and replay both spatially (input coordinates) and temporally (event timing). For example, it would be difficult to capture, represent, and replay a complicated non-discrete gesture such as `circular bird swipe with increasing slingshot tension` in Angry Birds. The problem is even more acute in the presence of multi-touch gestures. To quantify the importance of replaying gestures, in Figure 1, we show the distribution of the number of touchscreen gestures used during a 5-minute run for the 86 apps RERAN can replay: note that 84 out of 86 replayable apps contain gestures and hence would not be fully replayable with prior approaches.

Third, it is insufficient to capture only the user-interface events. Rather, accurate replay requires capturing the multitude of external events that arise from sensors on the device. For example, inputs from the accelerometer and compass are used

978-1-4673-3076-3/13/$31.00 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA

to drive many games and navigation apps, but these sensors are not exposed at the GUI level. One such app is Google Maps Street View, which uses the compass sensor to detect phone orientation (with respect to magnetic North) in order to orient the on-screen map correctly.

Finally, the accuracy of event timing during both capture and replay is crucial. For example, a swipe that is captured too slowly will replay as a sequence of button presses, and the timing between external events and user-interface actions can be similarly crucial. Indeed, we found that delays as short as milliseconds can adversely impact the replay process.

In this paper we present RERAN (REcord and Replay for ANdroid), a record-and-replay tool that addresses these challenges in the context of Android applications. RERAN directly captures and replays the low-level events that are triggered on the device. This approach allows RERAN to capture and playback GUI events, i.e., touchscreen gestures (e.g., tap, swipe, pinch, zoom), as well as those from other sensor input devices (e.g., accelerometer, light sensor, compass). In contrast, prior replay tools for Android [2], [6], [7], which are based on the keyword action paradigm, only replay touchscreen presses and cannot handle touchscreen gestures or other sensor inputs.

RERAN works in a natural,[1] non-intrusive, and low-effort way: during record, a capture component reads the events, with no noticeable impact on execution. This contrasts with many keyword-action-based approaches, which require manual scripting of test cases rather than direct trace collection [3]–[5]. During replay, a phone-based replay agent feeds the events to the phone, and the phone acts as it would if the sensor input came from user or environment interaction. RERAN is carefully designed to manage the large number of low-level events that occur during execution and to record and replay them with precise timings.

RERAN is currently unable to replay sensors whose events are made available to applications through system services rather than through the low-level event interface (e.g., camera and GPS). RERAN also does not capture other sources of nondeterminism in applications, for example use of random-number generators and access to the file system. Providing fully deterministic replay would require a significantly more heavyweight solution, for example, involving modifications to the virtual machine and/or complete interposition between the app and the underlying platform.

Despite these limitations, we demonstrate that RERAN is useful in several ways. First, we were able to record and successfully replay execution traces for 86 out of the Top-100 free Android apps on Google Play (the main Android app distribution channel). Moreover, RERAN replays in nearly real time, with an overhead of just over 1% compared to the original execution.

Second, RERAN is a powerful debugging aid because it can capture and replay event sequences that lead to crashes in apps, which helps developers inspect system state around a crash point. In particular, we demonstrate how, using RERAN, we were able to reproduce and replay bugs in widely-used apps including Firefox, Facebook, K-9 Mail, and Quickoffice.

[1]RERAN requires the device to be "rooted" in order to execute the replay agent, which we believe is a reasonable assumption for the intended users of RERAN, researchers and developers.

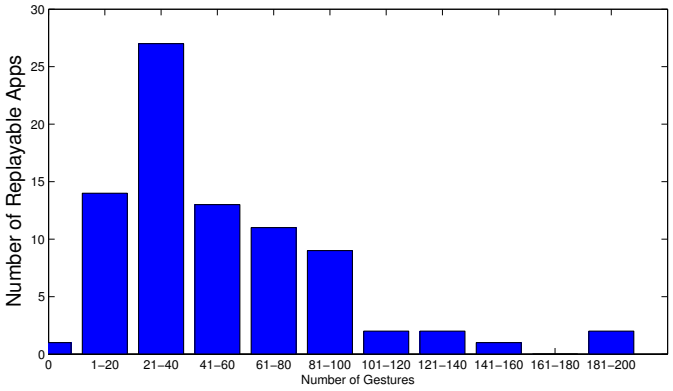

Fig. 1. Distribution of gestures in the 86 apps replayable with RERAN.

Third, RERAN has support for time warping which allows the execution of certain apps to be fast-forwarded in order to reach a specific app state faster than in the original execution, and without any additional manual input. We found that the replay time for time-warpable apps can be reduced by 22 to 68 percent compared to the original execution.

The remainder of this paper is organized as follows. Section II introduces the Android input events and sensor classes, as well as event generation, delivery, and processing. Section III provides an overview of our approach and the challenges that accurate record-and-replay pose for touchscreen-based smartphones. In Section IV we discuss the experimental methodology we followed, the implementation of RERAN, results of running it on Top-100 Google Play apps, and performance overhead. Section V presents three applications of using RERAN for research and development tasks: repeatability, reproducing bugs, and time warping. In Section VI we discuss limitations of our approach and record-and-replay on Android in general. Section VII discusses related work, and in Section VIII we present our conclusions.

II. Android Inputs

In this section we describe the diverse kinds of inputs that Android apps can accept, as well as how these inputs are represented internally as *events*.

A. Touchscreen User Actions

Smartphone apps are distinguished in part by the expressive variety of user actions possible through touchscreens. We briefly describe the most common ones:

Press-and-Release: The primary input action is the simple press, in which the user taps an area on the screen and releases quickly, e.g., clicking a button or typing text with the on-screen keyboard.

Press-and-Hold: The press and hold consists of the user tapping an area on the screen and releasing after holding, in the same position, past some threshold amount of time. This action can be used to access secondary menus or hidden options.

Swipe: Swiping on the touchscreen occurs when a user presses down on the screen from position (x_1, y_1) and, while continuing to hold down on the screen, moves to a new position (x_2, y_2), and releases. Swiping is used in many apps, e.g., to scroll through text in a Web browser or slingshot a bird in Angry Birds.

Zoom-and-Pinch: Multi-touch gestures involve pressing the touchscreen through more than one contact point, e.g., using two or more fingers. The most common multi-touch usages are zooming and pinching; when apps support such gestures, they usually correspond to changing the magnification level of the content being viewed, e.g., in Google Maps zooming and pinching change the level of detail.

Of the four user actions listed above, three are considered gestures: press-and-hold, swipes, and zoom-and-pinch. Gestures require a large group of underlying touchscreen events to provide their perceived functionality, as explained shortly. The press-and-release user action requires a much smaller grouping of events, and, as described in Section VII, has been successfully replayed using other methods. As shown in Figure 1, touchscreen gestures in apps are very frequent; we collected these results based on a 5-minute run of the app from a single user. Note that the majority of apps (60 out of 86) had anywhere from 20 to 100 gestures during the session, whereas only 2 apps did not contain any gestures (Uno Free and Brightest Flashlight Free).

Aside from the touchscreen, users can provide input through physical buttons on the phone. Our test devices did not have a physical keyboard, but they have several buttons with various usages, e.g., lock the screen, turn off the device, increase or decrease the speaker volume.

B. Physical Sensors

Android devices typically have several sensors that generate inputs asynchronously. Sensor data is generated by hardware components that measure changes in the physical properties of the phone. We describe the most common sensors and their uses in Android apps. Note that GPS location is attained from an Android service rather than directly from a physical sensor, so we cannot currently replay its events.

The **accelerometer** allows an application to detect whether the phone has changed its speed and can also be used to detect phone tilting and rotation [8]. This is the mechanism used to change the screen layout from portrait to landscape mode, and some games use the accelerometer as a primary means of user input, e.g., to roll a ball through a maze or shake a magic eight ball.

The **proximity (light) sensor** allows applications to detect how close the user is from the phone [8]. This sensor is primarily used to dim the screen as well as protect the user from accidental button presses when the phone is close to the user's ear during a phone call.

The **compass** reports the phone's orientation relative to the magnetic North. The compass is frequently used in mapping and navigation apps to provide a direction perspective.

C. Android Events

The Android software stack consists of a custom Linux kernel and libraries, the Dalvik Virtual Machine (VM), and apps running on top of the VM. When users interact with an Android app, the Android device's sensors generate and send *events* to the kernel via the `/dev/input/event*` device files. Events have a standard, five-field format (timestamp: device: type code value), where timestamp represents the time elapsed from the last system restart, device names the input device that has created the event, and the remaining fields depend on the specific event type.

Touchscreen gestures are encoded as a stream of touchscreen events in the above format, with no labeling as to which gesture was performed. For example, one such event could look as follows:

40-719451: /dev/input/event4: 0003 0035 0000011f

Here the timestamp indicates the event was generated 40 seconds and 719,451 microseconds since system restart, and the input device is event4, which corresponds to the touchscreen (for our device). The next three columns provide position information: 0035 corresponds to the x position of the event and 0000011f (hex) corresponds to coordinate 287 (decimal) of the screen. However, this single event alone is not enough information for reconstituting the high-level gesture: for instance, a single press usually involves roughly 18 touchscreen events, while a swipe usually involves roughly 50 touchscreen events.

In Figure 2 we show a subset of events involved in a single typical gesture—a swipe. The left side of the figure shows the raw stream of 54 events, while the right side shows the high-level semantics of event clusters: press, move, release. The sample event explained previously is actually event 3 in the stream. Note how the timing information reveals that, in order to successfully replay the swipe, the 54 events have to be replayed accurately within 179 milliseconds (between timestamps 40-719421 and 40-898681).

To provide an illustration of the event flux in a real-world app that uses multiple sensors, in Figure 3 we show the stream of events in the GasBuddy app (the app displays gas prices on a map), over the first 80 seconds of a normal run. The x-axis shows the time, in seconds, and the y-axis shows the number of events. The graphs show the number of events at 5-second granularity: the top curve, in red, shows the touchscreen events, while the bottom curve, in light blue, shows events coming from the compass. The text on top indicates the distinct phases in the execution. First, the user inputs the geographical location—Los Angeles—using the touchpad (seconds 0–10); notice how the compass is not used in this phase. Next, the app loads the gas station maps in the vicinity of Los Angeles and uses the compass to orient the map; notice how the user waits at this point as indicated by the absence of user events (seconds 11–25). In the third phase, the map has been loaded and the user navigates on the map using pinch and zoom, indicated by the high number of touchscreen events (seconds 26–45); the compass is still used to keep the correct map

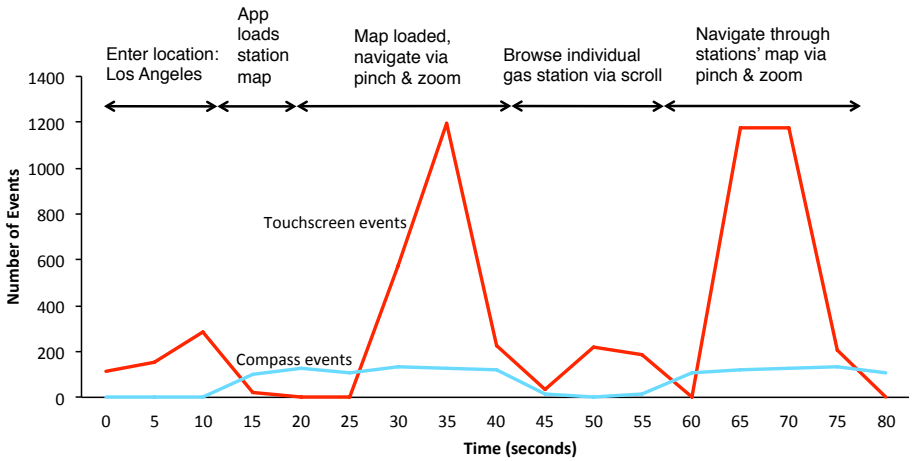

Fig. 3. Time series of events in Gas Buddy.

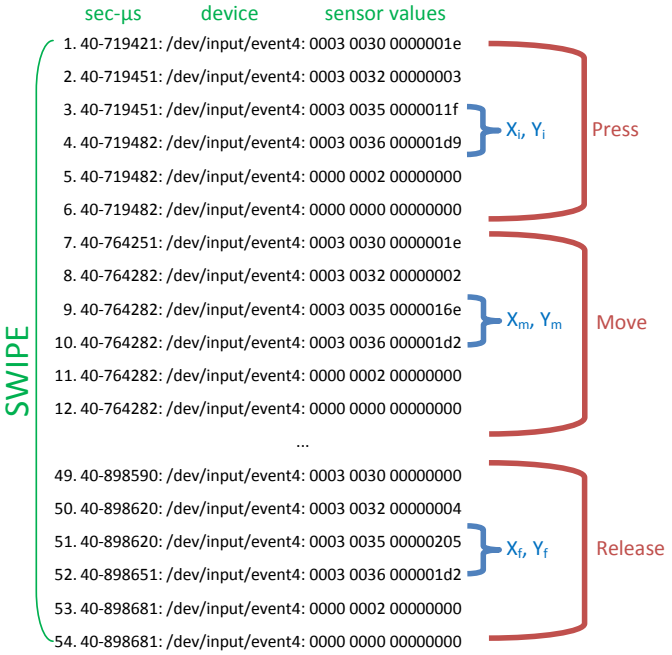

Fig. 2. Series of events representing a swipe; X_i, Y_i are the initial starting coordinates, X_m, Y_m are the coordinates during moving (still holding down), and X_f, Y_f are the final coordinates.

orientation. In the fourth phase, the user browses gas prices for an individual station (seconds 46–60), which requires scrolling and no compass. Finally (seconds 61+) the user navigates again through the stations map using pinch and zoom, which generates both touchscreen and compass events.

The time series reveals several key requirements for successful replay: events from all sensors, not only the touchscreen, must be captured and delivered, as they influence app behavior; and events must be captured and delivered at *high throughput*.

We provide a broader quantitative assessment of typical event volume in Table I. The table shows the total number of events received from three input devices, captured during a 5-minute run of the app listed. Touchscreen events dominate,

TABLE I
NUMBER OF INPUT EVENTS PER INPUT DEVICE DURING A 5-MINUTE RUN.

App name	Touchscreen	Proximity sensor	Accelerometer
Facebook	6,199	701	6
Angry Birds	5,895	675	3
GasBuddy	4,723	273	15
Amazon	4,831	779	4

however there are also a significant number of events from sensors. For all apps the phone was kept flat on a table during execution, which accounts for the small number of reported accelerometer events.

III. Approach Overview

We now present an overview of our approach, its technical challenges, and its advantages over prior record-and-replay approaches. Google provides the Android Software Development Kit (SDK), which contains various tools to aid app developers in implementation, testing, and debugging. Several of these tools were used in the development of RERAN.

The high-level view is presented in Figure 4. Record or replay can be started from either the phone, using the Android Terminal Emulator [9] app available for free on Google Play, or from a computer connected to the phone via a USB cable. The Android Debug Bridge (ADB), accessible through the `adb` command in the Android SDK, and Android Terminal Emulator act as the interface between RERAN and the smartphone. Commands can be issued to the smartphone's OS via these shells. We used this ability both to record events and to replay them.

A. Record

We use the Android SDK's `getevent` tool, which reads the `/dev/input/event*` files in order to provide a live log (trace) of the input events on the phone, as shown on the top of Figure 4. The performance of the app is unaffected during logging. Once logging has ended, RERAN's record phase translates the captured trace and creates the event trace file that is ready to be replayed. During translation, all the

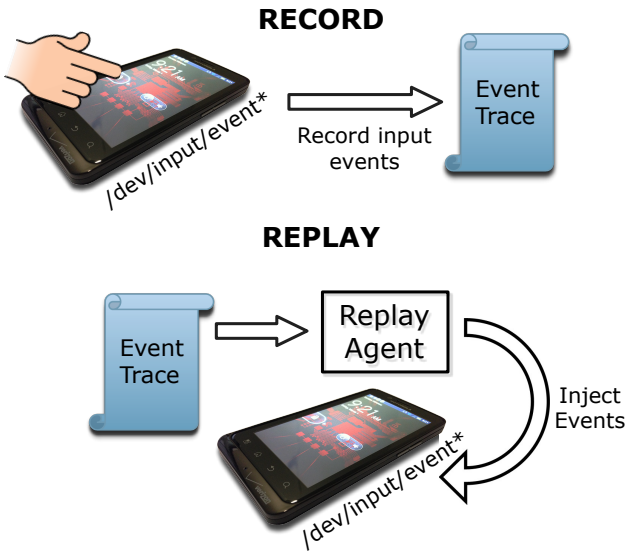

Fig. 4. Overview of RERAN.

captured event information is converted to a concise form and time delays are calculated. For implementation reasons, we currently perform translation offline, on a computer, but we plan to perform it directly on the phone in future work.

B. Replay

The Android SDK also provides a `sendevent` tool, which allows developers to send a single input event to the phone. Unfortunately, we found that `sendevent` has a small lag when used in series, which makes it unable to faithfully replay recorded event streams. There are several negative consequences if accurate timing is not preserved during replay. For example, delays in the middle of a swipe will make the swipe appear to be a series of presses, delaying the release that marks the end of press will make the press act as a press-and-hold, etc.

Therefore, delivering events with exact timings was a key technical challenge. As mentioned earlier, recreating a swipe usually requires 50 or more events to be replayed. Further, each swipe is usually less than a second, and the time between individual events in the swipe is typically less than 60 microseconds.

Because of the sensitive time dependency between events we had to refine our approach to replay. Rather than employ `sendevent`, we implemented our own replay agent that runs on the phone and directly injects events into the phone's event stream (as shown on the bottom of Figure 4). This approach also allowed us to inject multiple events at the same time, thereby seamlessly handling events that occurred simultaneously during recording.

During replay, the replay agent on the phone appears as an external user generating events through the phone's input devices. While in theory there is potential for conflicts with other events occurring during a replay session, in practice we did not find this to be a problem—because of our testing setup (see Section IV-B) the phone gracefully accepts and processes events from all sources, and there was no noticeable difference in behavior for the apps in our test suite.

C. Comparison with Keyword Action Approaches

By performing record-and-replay at the level of individual events, our tool can replay sophisticated touchscreen gestures while being unaware of their high-level action meaning. Furthermore, our approach naturally generalizes to handle new touchscreen gestures that are available in the future, as well as new kinds of sensors. All that is necessary is that these actions can be represented as low-level events.

In contrast, approaches based on the keyword action paradigm represent high-level actions directly. This can be useful in conveying more semantic information to users. However, it requires that the high-level taxonomy be updated each time a new kind of gesture or sensor becomes available. Furthermore, this approach requires a method for inferring the high-level actions from the individual low-level events, which may not always be straightforward. Finally, the many degrees of freedom involved in touchscreen gestures make the keyword action style relatively unnatural. For instance, a swipe's behavior may depend not only on the start and end positions but also by the time taken, the trajectory followed between the two positions, the amount of pressure used, etc.

IV. Implementation Details and Performance

This section provides further details of RERAN's implementation and experimental setup, and evaluate the tool's performance.

A. Physical Devices

The Android devices used to develop the replay tool were Motorola Droid Bionic phones, which have dual-core ARM Cortex-A9 processors running at 1GHz. The phones were released on released September 8, 2011 and run Android version 2.3.4 with Linux kernel version 2.6.35.

B. Testing Environment

Our phones did not have cellular service; we did not use them for related tasks such as sending SMS messages and making phone calls. In order to minimize the impact of poor wireless link quality on apps, we used WiFi in strong signal conditions. If tested apps did not require the device to be physically moved, the phone was placed on a flat surface when collecting traces; this reduces the flux of "always-on" sensor events, e.g., the accelerometer and light sensor, without affecting app behavior.

C. Replay Agent

We wrote the replay agent in C and compiled it with `arm-elf-gcc`. We used the source code of Android's `sendevent` tool as a guide to determine how to interact with the device directly, i.e., write to `/dev/input/event*`. In the early stages of this research, the replay agent was running on the computer as a Java program that used the Android shell command `sendevent` to send a series of input events into

TABLE II
THE 86 OUT OF TOP-100 APPS IN THE U.S. VIEW OF GOOGLE PLAY (AS OF MAY 6, 2012) THAT RERAN CAN REPLAY.

App name					
Adobe AIR	Craigslist Mobile	Feed Your Dino Free	IMDB	Running Fred	Visual Anatomy
Adobe Flash Player	Crime City	Gas Buddy	Instagram	Sincerely Ink Cards	Weather Bug
Adobe Reader	Daily Horoscope	Google Drive	Kindle	Sky Map	Weather Channel
Amazon Mobile	Dance Legend	Google Earth	Maps	Skype	WhatsApp Messenger
Angry Birds Rio	Death Rally Free	Google Maps Street View	MotoPrint	Slacker Radio	Where's My Water
Angry Birds Seasons	Dictionary	Google Play Books	Movies by Flixster	Slot City Machines	Word Search
Angry Birds Space	Documents ToGo	Google Play Music	MP3 Ringtone Maker	Stick Man BMX Stunts	Yellow Pages
Background HD Wallpapers	Drag Racing	Google Play Movies	Myxer	Talking Tom 2	Yelp
Baseball Superstars 2012	Drag Racing-Bike Racing	Google Plus	Netflix	TED	YouTube
BBC News	Draw Something	Google Search	NBA Gametime	Textgram	Zedge
Bible	Easy Battery Saver	Google Translate	One Touch Drawing	TouchNote PostCards	Zinio
BMX Boy	eBay	Groupon	Picsart	Tunewiki	
Brightest Flashlight	ESPN Score Center	HeartRadio	Pool Master Pro	Twitter	
Bubble Shoot	Evernote	How to Read Thoughts	PulseNews	Unblock Me Free	
Color Note	Facebook	iFunny	Recipe Search	Unicorn Dash	

TABLE III
THE PERFORMANCE AND SPACE OVERHEAD OF RERAN ON 4 POPULAR APPS AVERAGED OVER 5 USER SESSIONS.

App name	Run-time overhead			Trace size
	Original *(seconds)*	Replay *(seconds)*	Overhead *(%)*	*(KB)*
Facebook	291.04	294.12	1.05	260.43
Angry Birds	296.81	300.05	1.08	359.76
Dictionary	262.83	264.11	0.48	117.85
Gas Buddy	254.14	256.81	1.04	244.62

the phone via `adb`. However, as described earlier, that solution did not achieve the microsecond accuracy needed for gestures.

RERAN supports *selective replay*, whereby the user can suppress entire classes of events. This can be especially helpful when the user is trying to isolate the root cause of a bug. It can also reduce overhead when replaying an app that never uses a particular sensor input, such as the accelerometer.

The compiled replay agent is uploaded to the phone via `adb` and takes a trace file (event set and their timing) as argument. In order to execute on the phone, system permissions must be changed, which requires the phone to be rooted. The replay agent runs in a separate process from the replayed app.

D. Achievements

Of the Top-100 apps on Google Play, we were able to successfully replay 86; we list the names of these 86 apps in Table II. Note that these apps span a variety of categories from Social Media to Productivity to Games, demonstrating the wide scope of RERAN's practical utility. In Section VI we discuss the reasons why replay is not possible for the remaining 14 apps in Top-100.

E. Time and Space Overhead

We measured time and space overheads for 4 sample apps, using traces from 5 different users. We present the results in Table III. First, we compared the completion time of original executions with the completion time of replayed executions. Columns 2 and 3 show the average run time, across the 5 executions, of the original run and of the replayed run; column 4 shows the time overhead, near 1% in all cases. We believe the replay overhead is largely due to the fact that event injection is not instantaneous. During capture, sometimes an input device reports events as occuring *simultaneously*, i.e., with the same timestamp. We were able to mimic this by writing multiple events into the stream at once. However, because the events are being injected programatically, rather than being generated by the physical device, captured simultaneous events are only replayed within 500 microseconds of each other. Despite the lag, it caused no noticeable differences in app behavior.

The last column shows the average log size of the events captured. As expected, for the more interactive apps that use many gestures, e.g., Angry Birds, the log size is higher than less interactive apps, e.g., Dictionary. All tests were conducted on the physical devices described in Section IV-A.

V. REPLAY USES

The ability to replay a recorded execution has myriad applications. In this section we illustrate several scenarios of how RERAN's record-and-replay capabilities can be beneficial to developers and researchers.

A. Repeatability through Trace Replay

RERAN is able to play back a set of input events of a specific session exactly in the same way as it was originally recorded. This includes touchscreen events, user proximity events, as well as changes in phone orientation sensed by the accelerometer and compass. In contrast, if a user were to manually repeat a session trying to replicate the same actions as the previous run, there could be inconsistencies and abnormalities when compared to the original run due to human error and inconsistent timing; this could in turn lead to unfaithful results or an undesired outcome.

Replay can greatly help developers and researchers who wish to eliminate inaccurate and time-consuming app testing when repetition of a sequence is needed. In fact, in our prior research [10], we used a preliminary version of RERAN to eliminate the need for constant interaction with testers: to profile network traffic and other app characteristics, we examined 10 functionally identical executions of the same app by recording a user's interaction (execution #1) and replaying it 9 times (executions #2 through #10).

TABLE IV
REPRODUCIBLE APP BUGS THAT ARE REPLAYABLE.

Category	App name
File format	Ankidroid 0.7b3 APV PDF viewer 0.2.7 Quickoffice 4.1.80 Soundcloud 1.2.2
Invalid input	K-9 Mail 4.0.0.3
Stress	NPR News 2.1b
Scripts/plugins	Firefox 14.0
App logic	Home Switcher 1.6 Facebook 1.7.1

Varying Environments: Trace replay can also be used to test the effect of various environmental conditions on app behavior. For example, in prior work on network profiling we replayed the same app trace at different times of day to obtain a wider range of app behaviors [10]. Similarly, developers can choose to vary conditions such as network connectivity or phone position during a replayed execution and observe the impact of these variations on app behavior.

B. Reproducing Bugs

Reproducibility is key to bug fixing. RERAN provides significant help for reproducing bugs by recording executions until a bug is encountered and then replaying the trace. We tested RERAN's bug reproducing capabilities using actual bugs in popular Android apps. We first gathered sample bugs[2] from a variety of sources: Google Code, Mozilla's Bugzilla bug tracker, and the Web. We manually tried to reproduce the bug by following the provided steps to reproduce it, while recording our input actions with RERAN. When we were able to reproduce the bug in a manual run, we replayed the collected trace using RERAN. All 9 bugs that we were able to reproduce manually could be replayed, in that during the replayed execution, the bug manifested itself. To ensure the consistency of our results, we replayed the bug trace 5 additional times, and each time the bug was reproduced.

The bug category, along with the names and version of each app, is provided in Table IV. Video demos of RERAN replaying bugs are available on YouTube [1]. We now provide a detailed characterization of the bugs.

1) Bugs stemming from incorrect file formats (e.g., corrupted files, inappropriate file extension, unsupported files, etc.). For example, APV PDF Viewer version 0.2.7 and Quickoffice version 4.1.80 crash if a corrupt PDF file is loaded, while the Soundcloud 1.2.2 app crashes if an unsupported file format (e.g., `.ogg`) is uploaded. The Ankidroid flash card app (version 0.7b3) crashed when it opened a specific custom card template.
2) Bugs that are created by entering invalid or wrong input. For example, in the K-9 Mail 4.0.0.3 app, when IP addresses are inserted in place of the corresponding domain name, and the preferences are saved, the application crashes.
3) Bugs in browser applications, which can be due to unresponsive scripts, erroneous plug-ins, and malformed HTML pages. One example is version 14.0 of the popular browser Mozilla Firefox; it crashes while handling certain XML files and external scripts.
4) Bugs caused by stress testing (i.e., doing the same thing or generating the same sequence of events over and over.) For example, the NPR News reader app (version 2.1b) crashes if the hourly news update feature is exercised repeatedly.
5) Bugs caused by transitioning among app states. The Home Switcher app allows the user to change the system-wide theme of the user interface. Sometimes transitioning from a particular third-party theme to another theme causes the app (version 1.6) to crash. Facebook version 1.7.1 crashed after removing the Internet connection and attempting to log in.

Note that RERAN enables reproducing bugs that contain gestures, whereas other Android testing tools cannot handle gestures. For example, AndroidGUITAR [2] (which we compare with extensively in Section VII) could reach the point of the crash in those apps that did not contain input gestures such as swipes. However, for apps that did contain gestures (e.g., Ankidroid, Quickoffice, APV PDF Viewer, and Soundcloud), AndroidGUITAR is unable to reproduce the bugs.

Debugging: RERAN can be used in combination with debuggers to drives the app to a certain state and then, using a breakpoint, the debugger helps examine that state to facilitate bug finding and fixing. For example, we used RERAN in conjunction with the Eclipse integrated debugger, which communicates with the VM on the smartphone via the Java Debugger (JDB), to inspect app state during execution and just before a crash.

C. Time Warping

With RERAN, we can time-warp the GUI events of an event trace to alter timing during replay while still going through the same execution states. Time warping is accomplished by increasing or decreasing the *time delays* (TD, for short) between consecutive events of a trace without changing the original path captured. We have focused on fast-forwarding, but we imagine that a similar approach could be used to slow down execution, which could be used as another debugging tool in a testing suite. For example, the TDs could be increased to give the developer more time to examine the state of the app before the next interaction occurs.

We do not attempt to manipulate the speed of touchscreen presses or gestures. As we have discussed, such time warping can easily modify the gesture's effect or convert it to a different action or set of actions. However, we have identified two cases when significant TDs occur that can be targeted for fast-forwarding: (1) during data entry and (2) when the user is processing content. *Data entry* refers to the user typing on the

[2]We searched the Google Code and Mozilla Bugzilla bug repositories with the keywords "Steps to reproduce", "Stack trace", "Observed behavior", "Test cases", "Error report", "Error message", "Code sample". Of the returned results that were not device-specific and did not require phone service, we installed the version of the app that contained the bug.

virtual or physical keypad and clicking buttons. These events occur on human timescales and thus have significant room for speed-ups. *Processing content* refers to the user examining content on the screen, and deciding which action to perform next, e.g., press a button. Processing content can also take place after accessing data on external servers via the Internet (e.g., the CNN and Facebook apps). These situations can be fast-forwarded because there is a lag between the time that the content has loaded and the time the user performs the next input action.

We fast-forward using three rules that we have derived empirically for our phones and test apps, but we expect the general principle to hold for other setups.

The rules are as follows: (a) if the TD is less than 0.7 seconds, then we compress the delay to 0.001 seconds; (b) if the delay is greater than 3.0 seconds, we compress the delay to 3.0 seconds, and (c) any delay between 0.7 and 3.0 seconds is unchanged.

Rule (a) tries to identify and fast-forward data entry from the user. This rule, for example, allows data entry to be greatly sped up in apps requiring heavy text input such as Dictionary and Google Translate.

Rule (b) accounts for the long TDs during execution that can occur when a user is viewing content on the screen or deciding where to navigate to next. For example, this optimization can be seen in the gas station finder Gas Buddy, in which the user views a map of gas prices before selecting one, creating long periods of delay.

Rule (c) preserves the speed of the execution in all other situations.

Despite the simplicity of these rules, we were able to use them successfully to time-warp many apps. Video demos of RERAN fast-forwarding are available on YouTube [1].

We present the results of fast-forwarding in Table V. For each app, in column 2 we present the original run-time, in column 3 we present the run-time when fast-forwarding, while column 4 shows the run-time reduction as a percentage. Note that, in general, apps can be sped up anywhere from 4% to 68%. Certain apps, however (Quickoffice, NPR News, Home Switcher), could not be fast-forwarded due to reasons that we will explain next.

Uses of Fast-forwarding: Fast-forwarding can shorten the time to reach a certain point in the app without requiring manual input. For example, while testing a feature added to one of the app's screens, a developer could use fast-forwarding to greatly reduce the time required to get to that screen. Fast-forwarding can also be used in combination with RERAN's ability to reproduce bugs in order to speed up debugging time. By fast-forwarding the replay, the amount of time to reproduce the crash, i.e., replay the steps to reproduce, will be reduced, thus saving the developer time. As shown in Table V, we also fast-forwarded the 9 apps that had reproducible bug crashes. In each, we ran our time-warped replay and found that 5 apps had reduced running time. The remaining 3 apps saw no reduction and kept their previous running time. This is due to the apps not having data entry or long delays (so we could not apply rules (a) or (b)), and instead the TDs stayed in the range of our do-not-alter rule (rule (c)), which kept the original TD.

TABLE V
APP RUN TIME, IN THE ORIGINAL EXECUTION AND REPLAYED USING FAST-FORWARDING. BUGS REPRODUCED ARE SHOWN IN GRAY.

App name	Run-time		Reduction
	Original *(seconds)*	Fast-forwarded *(seconds)*	*(%)*
Facebook 1.9	249.71	162.93	34.75
Dictionary	238.49	144.31	39.49
Gas Buddy	263.06	84.72	67.79
BBC News	277.31	120.32	56.61
Craigslist	308.92	239.54	22.46
ESPN Score	314.22	244.95	22.05
Amazon	298.69	109.15	63.46
Movies	286.16	197.09	31.13
Google Translate	308.66	211.63	31.44
Ankidroid	8.66	8.35	3.58
APV PDF viewer	10.93	8.17	25.25
Firefox	64.22	29.55	53.99
Soundcloud	29.34	25.79	12.10
K-9 Mail	32.23	24.84	22.93
Facebook 1.7.1	30.31	18.69	38.34
Quickoffice	3.12	3.12	0.00
NPR News	9.91	9.91	0.00
Home Switcher	7.99	7.99	0.00

TABLE VI
THE 14 OUT OF TOP-100 APPS THAT RERAN CANNOT REPLAY.

Non-replayable reason	App name
Requires Android sensor service (e.g., camera, microphone)	Barcode Scanner, Tango Video Calls Voice Search, Voxer Walkie-Talkie
Dynamic or random elements (i.e., nondeterminism)	Pandora, Boxing Game, Fruit Ninja Jewels Star, Shoot Bubble Deluxe Solitare, Temple Run, Tetris Uno Free, Words with Friends

VI. LIMITATIONS

As mentioned in Section IV-D, RERAN could replay 86 of the Top 100-apps on Google Play. However, there were 14 apps we could not replay. These apps fell into two main categories: apps which required an Android sensor service and apps that contain dynamic or random elements, as shown in Table VI. We now discuss the main reasons why these apps cannot be replayed.

Requires Android sensor service: Some apps require sensor input that RERAN does not record. For example, we cannot replay the Barcode Scanner app properly without capturing the identical barcode image that is fed as input from the camera. Similarily, we cannot replay Voxer Walkie-Talkie because the app requires capturing audio data from the microphone. In Android, these types of input come from sensors that are not instrumented in the same way as the touchscreen and accelerometer (that use `/dev/input/event*`). Instead, due to privacy and security issues, these sensors are not openly accessible for capturing and are protected by system permissions, e.g., the device's camera, microphone, and GPS location. Currently, we do not capture such sensors, as tapping into such sources would require a non-trivial extension to our system, a task we leave to future work.

Dynamic or random: Similarly, we do not capture sources of nondeterminism, for example arising through dynamic layouts or popup windows [11], network connectivity changes [12], or internal nondeterminism such as the use of a random number generator. As can be seen in Table VI, the majority of these dynamic or random apps are games, e.g., the popular game Temple Run randomly generates levels each time a user plays, making replay with RERAN impossible. In general, handling these sources of nondeterminism requires capturing the results of system calls as well as calls to nondeterministic APIs and providing customized versions of these APIs for replay [13], which requires instrumenting apps and/or the VM. RERAN's simple design loses this expressiveness but can still replay the vast majority of apps in the Top 100.

VII. Related Work

Record-and-replay techniques have been explored by a rich body of prior work, on a range of platforms including Android. Most closely related to our line of work are approaches that implement record and replay at the GUI level; they usually do so using two methods, a keyword action approach, in which the GUI components are accessed and changed via scripts, and an event-driven approach in which input device events are captured and replayed. We first survey general-purpose GUI testing tools and then Android-specific approaches.

A. GUI Testing for Desktop Applications

Some JUnit-based capture-and-replay tools, e.g., Jacareto [14] or Pounder [15], are event-driven and capture coordinates in a manner similar to RERAN. They record the (x, y) coordinates of mouse clicks and keyboard strokes, and during replay, create new mouse and keyboard events with the captured information. These tools, even though they are at a higher level of abstraction than RERAN, are also susceptible to errors when changes in the GUI occur, e.g., `Button1` changes position between application versions. Some of these tools also offer support for mouse drags [11]. However, mouse drags differ from swipes in that the individual events being sent by other platforms are usually being detected as mouse presses (which already contain positioning information). In contrast, in Android, as shown in Figure 2, a down-press corresponds to a group of 6 single precisely-timed events. Therefore, timing accuracy for mouse drags on other platforms is not as critical as in Android.

Marathon [16], HP WinRunner [4], Abbot [3], and Rational Robot [5] offer support for record and replay for desktop or server applications, but in a different manner. Instead of capturing raw position coordinates they use a keyword action technique that works at a higher level of abstraction by capturing GUI objects. By capturing the GUI components themselves, they are able to refer to objects by name in their traces; however, they rely on the existence of a GUI layout in the first place, an assumption which might not hold, e.g., in the Angry Birds case. Finding appropriate object handles to use [17] can be alleviated with techniques such as GUI mapping [18] or creating unique identifiers based on the thread creating the GUI object [19]. Once a script has been created, the GUI interaction is replayed by accessing the GUI objects and updating their properties.

Tools following the keyword action technique often require users to manually write test scripts rather than capturing user interaction directly as in RERAN, because their focus is creating test cases, rather than replay. This type of testing is not as fragile as coordinate-based testing when changing GUI components; however modifications made to the GUI API, e.g., changes, additions, or deletions of GUI classes, might also require the scripts to be changed [6], [17].

All the aforementioned approaches were built targeting traditional desktop interfaces, and none of them offer support for touchscreens. Attempts for touch-based replay outside of mobile devices, e.g., DART [20] and Microsoft Surface SDK for tabletop applications [21], are usually built to function solely with the hardware and API's of the targeted platform. Recent work has moved towards creating a universal interface for multi-touch devices [22], [23].

B. Android GUI Testing

The Android SDK provides the Monkey tool [24], which can generate random touchscreen presses and gestures and other system-level events, and feed them into an application. Monkey also supports event sequence scripts to be fed into an application, however it is not supported by Google, which lists its purpose solely as a stress-tester. Monkey scripts can contain Android MotionEvents, allowing it to handle presses. However, based on our initial attempts when developing RERAN, scripting presses is very labor-intensive, e.g., besides the (x, y) position, each press requires 11 additional parameters to be specified including pressure, precision, and size. In addition, Monkey scripting does not support touchscreen gestures. RERAN alleviates the need for such complex and time-consuming scripting via recording, that allows the user to interact naturally with an app.

Google also provides a similar tool, Monkeyrunner [25], which brings the keyword action technique to Android by providing an API for replaying scripts that drive the Android device or emulator. The tool allows users to view the ending state as a screenshot. Robotium [26], a framework based on JUnit, provides a similar approach as Monkeyrunner, but also allows for automatic black-box testing of Android applications by invoking GUI elements, e.g., button names and menus, to navigate the application. Both tools heavily depend on the app layout being structured in a way that allows them to easily retrieve GUI components. However, deciding how to structure a layout is very much at the discretion of the individual Android app developers who can either (a) provide an XML layout file, or (b) create layout elements at run-time [21]. RERAN's testing is independent of layouts and limitations of user-defined GUI elements. This enables RERAN to be able to record and replay applications no matter how the layout is structured.

GUITAR [27] is a GUI testing framework for Java and Windows applications. GUITAR generates event sequence-based test cases for GUI applications. Their technique can generate test cases automatically using a structural event generation graph. While GUITAR primarily targets Java desktop applications, it has recently been ported to Android by extending Android SDK's Monkeyrunner tool to allow users to create their own test cases with a point-and-click interface that captures press events. However, GUITAR does not support touchscreen gestures, e.g., swipe and zoom, or other input devices, e.g., accelerometer and compass. In the past, navigating through a program by pointing and clicking was sufficient and useful—outside of the mobile world, the primary input devices captured have been mouse movement and keyboard keystrokes. However, as smartphones become more advanced, offering greater features and interactivity through added device sensors and their uses, we argue that in order to faithfully replay user interaction *all* these sensors must be accounted for, not only a subset of one device, i.e., touchscreen presses.

C. Deterministic Replay

A substantial body of work has focused on deterministic replay by capturing and replaying events at the hardware [28], operating system [29], or virtual machine [30] levels. These approaches log events that might introduce non-determinism (I/O, thread scheduling, memory accesses, etc.) and deliver the events in the right order during replay to ensure that the replayed execution is (quasi) identical to the original run. Logging overhead varies depending on the nature of the benchmark, and adding record and replay capabilities to an existing system might be quite intrusive. We explore a different point in the design space: we do not aim to achieve deterministic replay, but instead focus on an effective approach and tool that uses standard software, hardware, and is minimally intrusive, yet nevertheless can replay 86 out of the Top-100 popular apps.

VIII. Conclusions

We have presented RERAN, an approach to record-and-replay for the Android platform. Our research was motivated by the novelty and popularity of touchscreen-based platforms and apps, the unique challenges associated with replaying these apps, and the broad applicability of record-and-replay techniques when tackling research and practical tasks on Android. By directly capturing the low-level event stream on the phone and replaying it with precise timing, RERAN can easily reproduce complex GUI gestures as well as other sensor inputs. The result is a noninvasive yet very effective record-and-replay approach that works for the vast majority of popular, real-world apps in Google Play's Top-100. Moreover, we have demonstrated that the approach can be applied successfully for repeatability, bug reproducibility, and execution time-warping.

Acknowledgments

We thank Shashank Kothapalli for his assistance in collecting user traces. This work was supported in part by National Science Foundation awards CNS-1064646, CNS-1064844, and CNS-1143627, and by a Google Research Award.

References

[1] "RERAN: Record and Replay for Android Video Demo," August 2012, http://www.youtube.com/user/RERAN2012.
[2] SourceForge, "Android GUITAR," August 2012, http://sourceforge.net/apps/mediawiki/guitar/index.php?title=Android_GUITAR.
[3] Timothy Wall, "Abbot framework for automated testing of Java GUI components and programs," August 2012, http://abbot.sourceforge.net.
[4] Hewlet-Packard Company, "HP Functional Testing," August 2012.
[5] IBM, "Rational Robot," August 2012, www.ibm.com/software/awdtools/tester/robot/.
[6] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "A gui crawling-based technique for android mobile application testing," in *ICSTW'11*, pp. 252–261.
[7] Nodobo, "Nodobo Capture: Mobile Data Recording for Analysing User Interactions in Context," 2011.
[8] Android Developers, "SensorEvent," August 2012, http://developer.android.com/reference/android/hardware/SensorEvent.html#values.
[9] Google Play Android apps, "Android Terminal Emulator," August 2012, https://play.google.com/store/apps/details?id=jackpal.androidterm.
[10] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Profiledroid: multi-layer profiling of android applications," in *MobiCom'12*, pp. 137–148.
[11] M. Jovic, A. Adamoli, D. Zaparanuks, and M. Hauswirth, "Automating performance testing of interactive java applications," in *AST'10*, pp. 8–15.
[12] J. Flinn and Z. M. Mao, "Can deterministic replay be an enabling tool for mobile computing?" in *HotMobile '11*, pp. 84–89.
[13] J. Mickens, J. Elson, and J. Howell, "Mugshot: deterministic capture and replay for javascript applications," in *NSDI'10*, pp. 159–174.
[14] Daniel Herding and Christian Spannagel, "Jacareto," August 2012, http://sourceforge.net/apps/mediawiki/jacareto/.
[15] Matthew Pekar, "Pounder - Java GUI Testing Utility," August 2012, http://pounder.sourceforge.net/.
[16] Dakshinamurthy Karra, "Marathon," August 2012, http://www.marathontesting.com/.
[17] P. A. Brooks and A. M. Memon, "Automated gui testing guided by usage profiles," in *ASE'07*, pp. 333–342.
[18] A. C. R. Paiva, J. a. C. P. Faria, N. Tillmann, and R. A. M. Vidal, "A model-to-implementation mapping tool for automated model-based gui testing," in *Proceedings of the 7th international conference on Formal Methods and Software Engineering*, pp. 450–464.
[19] J. Steven, P. Chandra, B. Fleck, and A. Podgurski, "jrapture: A capture/replay tool for observation-based testing," in *ISSTA'00*, pp. 158–167.
[20] L. B. Kara and T. F. Stahovich, "An image-based, trainable symbol recognizer for hand-drawn sketches," in *Computers & Graphics, vol. 29, no. 4, pp. 501-517*, 2005.
[21] Android Developers, "Layouts," August 2012, http://developer.android.com/guide/topics/ui/declaring-layout.html.
[22] S. H. Khandkar, S. M. Sohan, J. Sillito, and F. Maurer, "Tool Support for Testing Complex Multi-Touch Gestures," in *ACM International Conference on Interactive Tabletops and Surfaces*, pp. 59–68.
[23] S. H. Khandkar and F. Maurer, "A Domain Specific Language to Define Gestures for Multi-Touch Applications," in *Proceedings of the 10th Workshop on Domain-Specific Modeling*, pp. 2:1–2:6.
[24] Android Developers, "UI/Application Exerciser Monkey," August 2012, http://developer.android.com/tools/help/monkey.html.
[25] ——, "MonkeyRunner," August 2012, http://developer.android.com/guide/developing/tools/monkeyrunner_concepts.html.
[26] Google Code, "Robotium," August 2012, http://code.google.com/p/robotium/.
[27] Atif Memon, "GUITAR," August 2012, guitar.sourceforge.net/.
[28] S. Narayanasamy, G. Pokam, and B. Calder, "Bugnet: Continuously recording program execution for deterministic replay debugging," in *ISCA '05*, pp. 284–295.
[29] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou, "Flashback: a lightweight extension for rollback and deterministic replay for software debugging," in *USENIX Annual Technical Conference*, 2004, pp. 29–44.
[30] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Revirt: enabling intrusion analysis through virtual-machine logging and replay," in *OSDI'02*, pp. 211–224.