MobiPlay: A Remote Execution Based Record-and-Replay Tool for Mobile Applications

Zhengrui Qin† Yutao Tang† Ed Novak Qun Li
Department of Computer Science
The College of William and Mary
Williamsburg, VA 23185 USA
{zhengrui, yytang, ejnovak, liqun}@cs.wm.edu
†Zhengrui Qin and Yutao Tang are co-first authors

ABSTRACT

The record-and-replay approach for software testing is important and valuable for developers in designing mobile applications. However, the existing solutions for recording and replaying Android applications are far from perfect. When considering the richness of mobile phones' input capabilities including touch screen, sensors, GPS, etc., existing approaches either fall short of covering all these different input types, or require elevated privileges that are not easily attained and can be dangerous. In this paper, we present a novel system, called MobiPlay, which aims to improve record-and-replay testing. By collaborating between a mobile phone and a server, we are the first to capture all possible inputs by doing so at the application layer, instead of at the Android framework layer or the Linux kernel layer, which would be infeasible without a server. MobiPlay runs the to-be-tested application on the server under exactly the same environment as the mobile phone, and displays the GUI of the application in real time on a thin client application installed on the mobile phone. From the perspective of the mobile phone user, the application appears to be local. We have implemented our system and evaluated it with tens of popular mobile applications showing that MobiPlay is efficient, flexible, and comprehensive. It can record all input data, including all sensor data, all touchscreen gestures, and GPS. It is able to record and replay on both the mobile phone and the server. Furthermore, it is suitable for both white-box and black-box testing.

Keywords

Mobile Application, Record-and-Replay, Google Android, Virtual Machine, Secure Virtual Mobile Platform.

1. INTRODUCTION

Mobile phones have been increasingly popular in recent years with nearly two billion users worldwide [8], and millions of applications (apps) are available in each of the substantial platforms. As new technologies, such as various sensors, and other rich resources are adopted by mobile phones, the user experience is greatly enhanced. However, at the same time, these new features have imposed challenges on application design and testing for developers. Nowadays, running an app may involve multiple input sources: touchscreens (swiping, pinching, zooming, click/tapping), sensors (GPS, accelerometer, compass, gyroscope), and networking (online gaming, websites, bluetooth) to name a few. Therefore, it is challenging for developers to test and debug mobile apps, since it is non-trivial to accurately record the data from all these inputs as well as the interaction among different components involved in the app. Even after the input has been recorded, it is then challenging to replay the app execution using the recorded data. While a handful of record-and-replay tools have been developed for mobile apps, they are far from perfect.

As one of the key technologies in software engineering, the record-and-replay approach to software testing has played an important role in the development of mobile apps. Record-and-replay is a necessary and valuable tool for mobile app development because it allows developers to easily find and recreate elusive and complex bugs, test outlier cases, and increase the speed of testing software by automating the process. Record-and-replay improves software in the testing, debugging, optimization, and upgrading phases. However, we face several challenges in implementing such a system on mobile devices. Considering the rich input capabilities of mobile phones and the real-time interaction between the mobile app and the user, the challenges are as follows. First, it is difficult to accurately record an application's continuous execution instead of some discrete actions. Second, it is hard to record all the input data, which is especially true for sensors such as the GPS. Third, it is preferable that all recorded data is human-readable, such that developers can easily analyze, revise, and re-assemble the recorded data in order to accurately locate and identify bugs or performance bottlenecks. Finally, it may be possible to modify the mobile phone operating system (OS) to achieve record-and-replay functionality. However, modifying the OS requires a device with an unlocked bootloader and an open source operating system. Unlocking the bootloader is impossible on some devices (due to manufacturer obstacles), difficult, and usually erases all user data on the device. Modifying the operating system may introduce bugs, and is difficult in general, requiring access to any proprietary closed-source components

ICSE '16, May 14-22, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2884781.2884854

from the original.

In recent years, researchers have designed and developed several replay tools for mobile apps. However, none of them are able to truly capture all possible input. These tools can be divided into several categories. The first category is tools that obtain the input data by reading /dev/input/event∗ files through the Android SDK *getevent* tool (e.g., RERAN [11] and Mosaic [14]). Although these tools can record continuous gestures on the touchscreen (swipe, move, pinch/zoom), they come with several drawbacks. First, they entirely depend on whether the mobile phone's OS provides interfaces to /dev/input/event∗, which is not always the case. For instance, the Nexus 7 does not push any sensor data into any /dev/input/event∗ file. Second, they are unable to record sensors whose events are made available to applications through system services rather than low-level event interfaces, such as GPS. Third, they can only obtain the event data in low-level hexadecimal codes (e.g., 40-719451: /dev/input/event4: 0003 0035 0000011f), which is not human readable, hindering developers from white-box testing. Fourth, they have potential conflicts with other events occurring during a replay session. Another category is GUI-level tools, such as [1, 29, 13]. They work at a higher level of abstraction by capturing GUI objects, and usually require app modification (e.g., *android:debuggable* = *true*). Though they work well for discrete point-and-click GUIs, they cannot handle continuous touchscreen gestures or customized GUI elements.

A straightforward question is, "Can we solve all the problems and challenges within the mobile phone alone without modifying the operating system?" Unfortunately, the answer appears to be no. For security concerns, mobile operating systems, such as Android, *sandbox* apps in order to provide applications with the guarantee of isolation from other applications on the system. Each application has its own UID that prevents it from doing many things to other applications on the system. If we are to record all the input data for an app, we would have to develop a second app, without using any existing tools, such as the Android SDK *getevent* tool. However, as the Android Application Sandbox has enforced, the second recording app cannot access any data or memory of the app to be recorded. Therefore, we have to introduce other component, rather than the mobile phone alone, to solve this problem.

In this paper, we design a system, called MobiPlay, to record and replay an Android app's execution by introducing a server. The to-be-tested app is actually running on the server, while its GUI is transmitted back to the mobile phone as if the application were running locally on the phone. Although it may seem that latency would be a large concern in this setup, we find that the latency is acceptable due to the high-speed peer connection and the proximity of the server (we evaluate on a LAN connection). We have a video of MobiPlay on YouTube (search MobiPlay). Without modifying the mobile phone's operating system, MobiPlay is able to record all sensor data inputs, for replay later, in the form of high-level events, such as touchscreen gesture, key event, and sensor event. Besides solving the existing problems and challenges we outlined previously, MobiPlay is also able to offer more flexibility than ever before. It can not only record all input data, on both the mobile phone and the server side but it can also replay the app on both sides as well. Furthermore, our system is suitable for both white-box testing and black-box testing.

In summary, in this paper we make the following main contributions:

- We are the first to record input data of mobile apps in the application layer without modifying mobile phone's operating system, which is not achievable with the previous state-of-the-art approaches.
- We have designed and implemented MobiPlay, a system that is able to record and replay the execution of mobile apps. Our system is richer than ever before because it is able to record all sensor data input.
- MobiPlay is able to simulate the same environment on the mobile phone and the server, which fundamentally expands the space of flexibility.
- MobiPlay is flexible in that it can record and replay on both the client (mobile phone) and server side.
- Our system enables white-box testing for app developers because it exposes high-level semantic events, and presents them in a human readable form instead of an encoded stream of raw hexadecimal event data.

The rest of the paper is organized as follows. We present the whole system in Section 2 and describe the implementation in Section 3. Section 4 details the evaluation of MobiPlay. Section 5 briefly discusses the limitations and future work. We review the related work in Section 6 and conclude our paper in Section 7.

2. DESIGN OF MOBIPLAY

In this section, we will first elaborate on the rationale behind our design decisions. Then, we will explain our system design, its general architecture, and the details therein. Finally, we will explain the input data recording and replaying in our system.

2.1 Design Rationale

As mentioned in Section 1, we have to coordinate work between the user's mobile phone and an external server to solve all of our design challenges. Here we further illustrate why the mobile phone **alone** cannot solve this problem, and why a server is indispensable. Figure 1 shows the logical flow of application input data. Suppose the user makes a

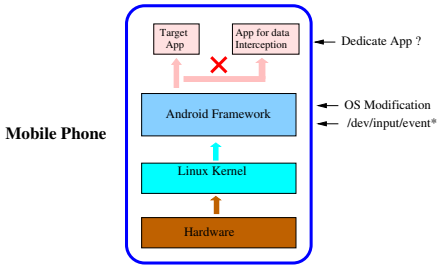

Figure 1: App input data flow, within a mobile phone (no server).

gesture on the touchscreen (tap, swipe, pinch, zoom, etc.). First, the touchscreen hardware captures this gesture, converts it into digital data, and informs the Linux kernel by

sending an interrupt to the CPU. Second, after receiving the interrupt, the OS stops the current job, reads the input data with the corresponding driver, and sends the data to the Android framework. Third, the Android framework packages the data into discrete events (MotionEvent) and sends them to the related service, such as Sensor Service, Input Method Service, or Location Service; at the same time, it sends the data to /dev/input/event∗ as well in the form of hex codes. Finally, the related service sends the discrete events to the application running on the foreground.

RERAN and Mosaic obtain the app input data by reading /dev/input/event∗ files in the Android framework, and OS modification occurs in the Android Framework as well. From Figure 1, we can see that the only other possible location to record app input data, without modifying the OS, is in the application layer. In order to do this, one must develop another application, specifically dedicated to intercepting the input data that is actually destined for the target application. However, this is forbidden by the Android Application Sandbox, as well as the sandbox policy in other mobile operating systems, which guarantee that unsanctioned data sharing between applications is not possible. As a result, the dedicated data intercepting app cannot access any data or memory of the target application. Therefore, it is clear that a mobile phone alone cannot solve the challenges and problems that the current approaches have encountered. To overcome this "isolation" obstacle, we introduce a second component, i.e., a sever.

2.2 Architecture of MobiPlay

In this paper, we design MobiPlay, a client-server system consisting of a mobile phone and a server, as shown in Figure 2. The server and the mobile phone are connected through a high-speed network connection, like 300Mbps to 1Gbps.

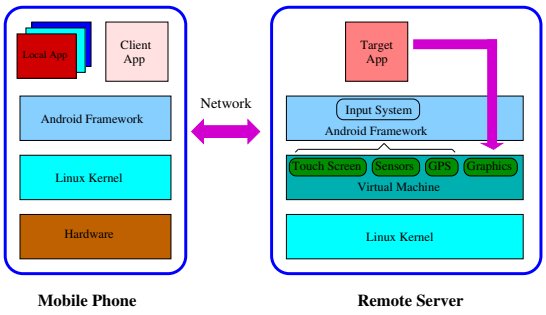

Figure 2: MobiPlay consists of a mobile phone and a server.

In MobiPlay, there are two components associated with the application to be recorded and replayed (the target app, hereafter): a client app on the mobile phone and a virtual machine (VM) on the server. The target app runs on the VM on the server. The client app (the client, hereafter) is a typical Android app that does not require root privilege and is dedicated to intercepting all the input data for the target app. The VM is a "duplicated" mobile phone on the server, which has the same configuration as the physical mobile phone, including screen size, resolution, and all present sensors. The VM runs a modified Android operating system designed for x86 architecture. It is important to note that the tester/user has complete control over the server, including the modified Android operating system; specifically, she has root privileges, has access to modify and recompile the OS source code, is able to make configuration changes, etc.

The basic idea of MobiPlay is that the target app actually runs on the server, while the user interacts with the client app on the mobile phone. The user is not explicitly aware that she is, in effect, using a thin client. At the beginning, we install the target app on the virtual machine on the server, and the client on the mobile phone. The client shows the GUI of the target app in real time on the mobile phone, exactly as if the target app were running on the mobile phone. As a result, the user just needs to interact with the target app as usual, while, under the surface, the client continuously forwards all input data (such as touchscreen gestures, sensor data, and GPS) to the VM on the server. At the same time, the GUI of the target app on the server is forwarded to the client and is then displayed by the client on the mobile phone. The VM on the server injects the input data received to the related OS services, which in turn send it to the target app. The target app runs on the virtual machine with the injected input data exactly same as it would run on the mobile phone. In other words, the user runs the app with exactly the same experience as if the app had been running on the mobile phone. The target app actually runs on the virtual machine on the server, but, with the same environment (inputs, resolution, screen size, etc.) as if it had been running on the mobile phone.

MobiPlay has three modes: *normal*, *record*, *replay*. When the client is opened, the graphical interface presents three buttons to the user, and the user must choose one of the three modes before establishing a connection with the server. In the normal mode, MobiPlay runs without any data recording or data replaying. That is, the user of the mobile phone runs the mobile app on the server, while interacting with the local client app. The mobile user can use this mode to test MobiPlay. Another possible application scenario for this mode might be that the mobile user wants to offload a resource-hungry app to the much more powerful server. In record mode, MobiPlay intercepts all input data through the client app, as the target app runs on the server. The collected data is stored on disk. A user configuration option allows for the data to be stored on the phone, or the server, or both. In the replay mode, MobiPlay first configures where to read data (phone or server), where to replay (phone or server), and the test type (black-box or white-box). Then it reads the input data from disk, injects the input events, in the same order, into the target app running on the chosen device, and replay the target app. During the replay, MobiPlay does not process any local Android service request, such that the target app is replayed with the recorded input only. This avoids any interference from the current state (such as new GPS or accelerometer data).

In the following, we will describe the data recording and the app replay mechanisms in more detail.

2.3 App Recording

When the user chooses record mode, MobiPlay will be directed to record all input data for replay later. MobiPlay can record all input data for the target app on both the mobile phone and the server. As illustrated in Figure 2, all input data for the target app passes through the client app on the mobile phone. Therefore, the client is able to

intercept all this data. At the same time, since all the input data is transmitted to the server and the user has full control of the server, the data can be intercepted and recorded there as well.

Figure 3 shows how the data is intercepted and stored on both sides. On the phone side, all the data is intercepted in

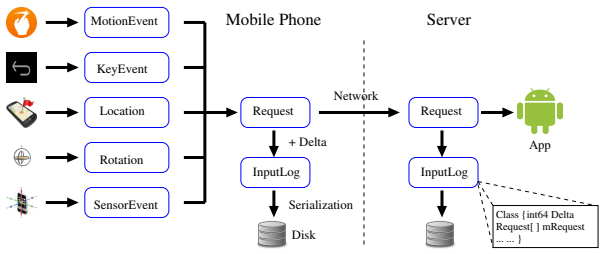

Figure 3: MobiPlay records input data on both the mobile phone and the server.

the form of events, such as motion events, key events, location, rotation, sensor, etc., and each event is an object that contains the input information at a certain point of time. MobiPlay extracts useful information from each event and stores it in a structure called *Request*. MobiPlay also obtains the time intervals between each pair of consecutive events, and stores this information, together with the Requests in order, in the log. The log itself is a class called an *InputLog*, which is a collection of Request objects. Finally the entire InputLog instance is written to disk on the mobile phone. At the same time, all Requests are transmitted through the network to the server, in order to be fed into the target app. Thus, MobiPlay is able to record all input data on the server side as well. Once the target has finished running, i.e, the recording procedure has completed, MobiPlay quits to the GUI with three modes for selection.

One prominent advantage of this architecture is that our recorded data is high-level, readable and revisable. This is incredibly valuable for app developers when trying to recreate and fix bugs in their code caused by specific, infrequent input.

2.4 App Replaying

When the user chooses replay mode, MobiPlay will load the recorded data from the disk and replay the target app. The replay procedure is illustrated in Figure 4. The replay procedure for the white-testing on the mobile phone is similar to that on the server, and neither of them needs root privilege. However, the replay procedure for the black-box testing on the mobile phone is different from that on the server. In case on the phone, the black-box testing still requires root privilege, since the input data has to be injected through /dev/input/event∗ files.

As Figure 4 shows, first, MobiPlay loads the input data from the disk of either the server or the mobile phone. Second, the input data stored on disk as an InputLog class is unpacked into a sequence of Request (i.e., events). Third, (a) for black-box testing on the server, the sequence of events is sent to virtual devices on the VM, which then inject events to the target app for replay; (b) for white-box testing on the server, the input events is injected to the target app through Android testing framework and event emulator; (c) for white-box testing on the phone, it is same as the white-

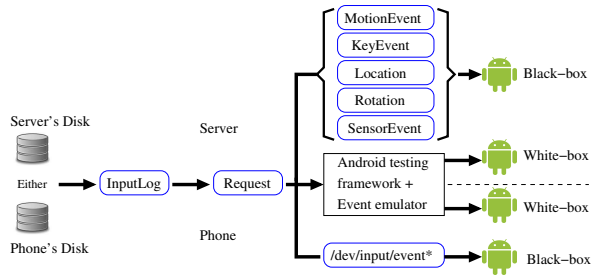

Figure 4: MobiPlay can relay an app on both mobile phone and server, black-box and white-box testing, respectively.

box testing on the server; (d) for black-box testing on the phone, the events is converted into hex codes and fed into the corresponding /dev/input/event∗ files, where the target app read the input data for replay. After the replay has finished, MobiPlay quits to the GUI with three modes for selection.

MobiPlay is advantageous in that app developers can revise the input data as they want to test the app in different scenarios. This is much easier than re-running the app multiple times to collect input data, hoping for good test coverage from all types of input.

3. IMPLEMENTATION

In this section, we present our implementation of MobiPlay. We establish the client-server platform by leveraging the Secure Virtual Mobile Platform (SVMP) [31], and then we build the record and replay approaches on basis of SVMP.

3.1 Physical Devices

In principle, MobiPlay only consists of a mobile phone (or a tablet) and a server by utilizing the existing networking infrastructure. In our implementation, besides the phone and the server, a router is used to set up the wireless connection between the server and the phone. The characteristics of all these devices are listed in Table 1.

Table 1: The devices used in MobiPlay system.

Device	Specification
Android phone Samsung Galaxy S4	quad-core 1.6GHz Cortex-A15 quad-core 1.2GHz Cortex-A7 2GB RAM, 32 GB microSD Android OS, v5.0.1
Android tablet Nexus 7, 2013	quad-core 1.5GHz Krait 2GB RAM, 32GB storage Android OS, v4.1.1
Server Y480 Lenovo laptop	2.4GHz Intel i7-3610QM 8GB RAM, 500 GB HD Ubuntu 14.04
Router TP-Link TL-WR841N	300Mbps

Additionally, the server uses VirtualBox[1] as the VM hypervisor and uses a virtual bridged network adaptor for networking access. The VM configuration allocates 4096 MB

[1]https://www.virtualbox.org/wiki/VirtualBox

RAM, an 8 core processor, and 5GB disk storage space. And the VM runs Android OS v4.4.4.

3.2 The Client-Server Platform

In MobiPlay, the essential component is the client-server platform, which we implement using SVMP. SVMP is a secure mobile application platform developed by MITRE[2], based on thin client technology and cloud computing technology. An open source "virtual smartphone", SVMP runs an Android-based mobile operating system on a cloud platform.

In the big picture, we utilize SVMP to create a virtual machine on the server, where the target mobile app actually runs, and an SVMP client on the mobile phone, where the GUI of the app is displayed in real time. The client and the virtual machine are connected through a wireless network in our setup, which is common for mobile devices.

3.2.1 SVMP Client

The SVMP client is installed on the mobile phone as a normal mobile application. The client, simple and unprivileged, is associated with the VM on the server rather than the target app. That is, when we test multiple apps, we just need to install each app in turn on the VM, without making any changes to the client. While MobiPlay runs in *normal* or *record* mode, the client captures native touch screen events, sensor inputs like the accelerometer and gyroscope, location information, and messages such as notification popups and Android "Intents". All these data is packaged under the SVMP message protocol, and is sent from the client to the server in real time. At the same time, the client displays the GUI of the target app transmitted from the server. In a word, the client enables the user to interact with the target mobile app running on the server, in the same way as if the app had been running on the mobile phone.

3.2.2 SVMP Virtual Machine

The SVMP VM is installed on the server. On top of the VM is the Android framework where the target app is installed. The VM provides virtual devices including *Touch Screen*, *Sensors*, *GPS*, and *Graphics*, which can be seen in Figure 2. The first three virtual devices are responsible for feeding the input data, as captured on the client side, to the target app running on the VM. As the target app runs, its GUI is displayed on the virtual display, i.e., Graphics. The VM, in turn, packages whatever is displayed on Graphics and sends it to the client in real time.

It is important for MobiPlay to maintain the same environment on the VM as that on the mobile phone, including the screen size, the resolution, and all the input devices. For instance, if the screen size is different, the touchscreen gestures will be represented with coordinates that are incorrect, or even undefined, on one of the displays. In our implementation, we ensure that no device mismatch occurs in our system to avoid problems like these.

3.2.3 Networking

It is critical to maintain a high-speed network connection between the mobile phone and the server, otherwise the user experience of MobiPlay will be impacted. The network is responsible for transmitting the input data from the client on the phone to the VM on the server and the GUI of the target app from the VM to the client in real time. As the mobile phone does not have a wired-network option, we set up a wireless network using a TP-Link TL-WR841N router, which can provide connectivity with throughput up to 300Mbps. Fortunately, SVMP uses WebRTC[3] to transmit data between the phone and the server, which greatly reduces the latency and the data volume. This is especially useful for situations where there may only be a slower network connection option available. In our implementation, MobiPlay maintains a frame rate of 50 FPS.

3.3 The Record Approach

As mentioned in Section 2.3, MobiPlay is able to record the input data in order to replay it later, on both the mobile phone and the server. Here we detail the implementation of the recording procedure.

On the mobile phone side, the client in MobiPlay intercepts all input data for the target mobile app in the form of events, which are grouped into five categories: **MotionEvent**, **KeyEvent**, **Location**, **Rotation**, and **SensorEvent** (please refer to Figure 3). Each event is an object containing the input information at a certain time. After intercepting an event, the client extracts only the necessary information for replay, and creates a data structure called a *Request* to store it. The client also calculates the time interval between two consecutive events. The event information and the time interval are then logged in an *InputLog* class, which stores a collection of *Request* objects. Finally, the InputLog is serialized and stored on disk.

Note that we define the InputLog class via Google's protocol buffer[4], which is a language-neutral, platform neutral, extensible, and automated mechanism for serializing structured data. The data stored in InputLog can be easily converted to JSON or XML format which is human-readable.

In the following, we will describe the interception of each of the five categories of events.

3.3.1 MotionEvent

For most mobile apps, the most frequent input data is touchscreen events, including tap, press and hold, pinch, zoom, swipe, etc. The Android framework uses the MotionEvent class to record each touchscreen event. MotionEvents describe movement in terms of an action code and a set of axis values. The action code specifies the state change, such as a pointer going down or up. The axis values describe the position and other movement properties.

MotionEvents occur representing all possible touchscreen actions, such as ACTION_DOWN, ACTION_MOVE, ACTION_UP, the time, the coordinates, and any historical event that happens before the current event. For the purpose of efficiency, Android may batch multiple touchscreen events into a single MovementEvent with several movement samples, and an ACTION_MOVE action code. MotionEvents are passed as a paramenter into the onTouchEvent() method, which is triggered by Android framework when a touch gesture happens. The top panel of Figure 5 shows all the fields of the class MotionEvent, which MobiPlay intercepts and records.

3.3.2 SensorEvent

[2]www.mitre.org.

[3]http://www.webrtc.org/

[4]https://developers.google.com/protocol-buffers/docs/overview

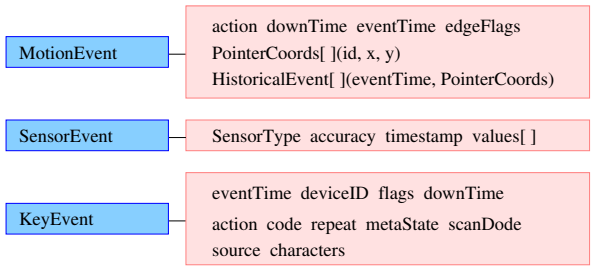

Figure 5: The MotionEvent, SensorEvent and KeyEvent classes along with their associated fields.

In the Android framework, sensor data is represented in the SensorEvent class and as each sample occurs, a SensorEvent instance is sent to the app by the sensor service. Each SensorEvent contains four fields: sensor type, time, accuracy, and the new data value(s), as shown in the middle panel of Figure 5. In our implementation MobiPlay can support the following physical sensors: accelerometer, gyroscope, light sensor, magnetic sensor, pressure sensor, proximity sensor, and virtual sensors (gravity, linear acceleration, orientation, and rotation vector). Sensor events are intercepted by onSensorChanged() method in the client app.

3.3.3 *KeyEvent*

KeyEvent is used to report key and button events, and is intercepted by the dispatchKeyEvent() method. Each key press is described by a sequence of key events. All information for these events is listed in the bottom panel of Figure 5.

3.3.4 *Location*

One advantage of our MobiPlay system is that it can handle location data (i.e., GPS), which other current approaches, such as RERAN and Mosaic, cannot. MobiPlay utilizes the LocationListener class to intercept location information. LocationListener has four methods, each of which has several input parameters, as listed in Figure 6. Whenever changes have been made to the location, Android framework will trigger the LocationListener to notify the target app. At the same time, it intercepts and records all these parameters.

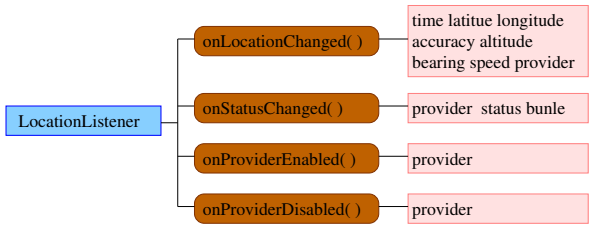

Figure 6: The class LocationListener and its four methods with the corresponding parameters.

3.3.5 *Rotation*

MobiPlay can record device rotation changes as well. If the app runs on the mobile phone, whenever any rotation change is detected, the Android framework will notify the app and trigger the GUI to change from *portrait* to *landscape* or vice versa. The advantage of this design is that the app itself does not need to monitor the orientation of the gravity sensor, which eases work for developers who only need to maintain the state of their GUI when transitioning between view orientations. In MobiPlay, however, the target app will not obtain any rotation event because the target app is running on the server and the Android framework on the server will not detect any rotation change. To solve this problem, MobiPlay collects all rotation events by leveraging onOrientationChanged() in the client app on the mobile phone. Besides black-box testing, these events are also useful in white-box testing, because we can use these events to change the screen orientation. The rotation event has only one parameter, *orientation*, ranging from 0 to 359 degrees. Specifically, 0 degrees means that the device is oriented in its natural position, 90 degrees means its left side is at the top, 180 upside down, and 270 indicates the right side is on top. When the device is nearly flat (parallel with the ground), the orientation cannot be determined, and ORIENTATION_UNKNOWN will be returned. For efficiency, MobiPlay does not record this undetermined case. It is worth pointing out that RERAN cannot handle rotation events.

3.4 The Replay Approach

MobiPlay is able to replay a mobile app on both the server and the phone, for both black-box testing and white-box testing. In the following, we will first present the procedure of the replay on both sides. Note that in the replay duration, MobiPlay does not process any new input from local Android services in order not to interfere the replay; though it still can receive them from local Android services, it does not send it to the app. We then will describe an event-sampling technique to revise the replay data as the tester wants.

3.4.1 *Replay on the Server*

The replay procedure for black-box testing on the server is quite similar to what happens on the server in the *normal* mode except (1) the input is read from the log stored on disk, instead of from the client directly. Also, (2) all concurrent input from mobile phone is discarded. The input data can be injected from either the disk of the mobile phone or that of the server.

At the beginning, the user sets the mobile phone in replay mode. When the client detects that the mobile phone is currently in replay mode, it will load the corresponding input data from disk and store it in an InputLog class, which has the same form as the class used in data recording, by calling the *parseFrom*() method in the Google protocol buffers. The events are simply injected into the system via the virtual devices. To replay the app correctly, it is of the utmost importance to keep the events in the correct order. As mentioned in the data recording procedure, the recorded data includes all the information of every event and the inter-arrival time between all pairs of consecutive events. We need to adjust the time information in the recorded data according to the current time. Specifically, for the first event, MobiPlay changes the event time to the current time, and adjusts the subsequent events accordingly. Then, MobiPlay packages the events into a Request, the same structure used in data recording. After sending the first Request to the target app, MobiPlay will wait in order to maintain the correct inter-arrival time between events. Then the next Request

object is sent and the process is repeated. This continues until all input data has been read from disk and injected. After the replay has finished, MobiPlay will automatically switch back to the GUI with three modes for choosing. Algorithm 1 summarizes the whole procedure.

Algorithm 1: The replay procedure of black-box testing on the server.

```
/* InputLog Class {int64 δt
   Request[] mRequest}                          */
Input: disk=phone/serve;
       device=server;
       test=black-box;
if disk==phone then
    read input data from phone's disk;
else
    read input data from server's disk;
store data into InputLog by call parseFrom();
while Request is not empty do
    (event, δt)=getNextRequest(InputLog);
    change the time of event to current time;
    inject event to the app;
    sleep(δt);
Return to the GUI for mode selection;
```

Since white-box testing is more meaningful on the phone than on the server, and the procedure is similar on the server and on the phone, we defer white-box testing details to Section 3.4.2.

3.4.2 Replay on the mobile phone

For some tests, we may want to conduct the replay on the mobile phone itself. As we know, MobiPlay records all input data when the target app is actually running on the virtual machine on the server. Since the virtual machine offers the same environment as the mobile phone, including screen size and resolution, the recorded data can be replayed on the mobile phone. However, in this case, the tester does not have full control of the mobile phone. Therefore, the replay procedure is different. In the following, we will describe the replay of black-box testing and that of white-box testing, respectively.

Black-box testing. For black-box testing, we cannot make any change to the target app. As analyzed in the introduction, the Android Application Sandbox forbids one application from injecting data into any other application. The only way to inject the input data for replay is to leverage the *sendevent* tool, which requires root privilege. Therefore, for replay on the mobile phone, MobiPlay runs into the same limitations as RERAN does. As the recorded input data in MobiPlay is high-level events, we need to convert it back to hex codes first, and then inject it by writing into the corresponding /dev/input/event∗ files.

White-box testing. Here, we want to inject the replay data directly into the target app assuming that we have the source code of the target app, with the goal of modifying the target app as less as possible. In a normal Android system, the Android framework communicates with the app through API, and sends the input data to the app in the form of events. Therefore, in white-box testing, the recorded input data also has to be sent to the app in the form of events. With the recorded data, we need to recreate all five events: MotionEvent, KeyEvent, SensorEvent, Location, and Rotation. Unfortunately, we cannot create a SensorEvent object as we do for the other four events because SensorEvent( ) is not public in *android.hardware.SensorEvent* class. Thus, we define a new class, NewSensorEvent, to carry sensor data (i.e., SensorType, accuracy, timestamp, and an array of values).

Depending on whether to modify the app, the five events are grouped into two categories. The first category, MotionEvent and KeyEvent, does not need to modify the app. Android provides its own testing framework called "the Android testing framework", which is well integrated into the Android SDK tools. It offers powerful and easy-to-use tools that help developers test their applications at every level, from unit to framework. We use the instrumentation class in this testing framework to inject MotionEvent and KeyEvent through SendPointerSync() and SendKeySync(), respectively. The second category, SensorEvent, Location, and Rotation, requires to modify the app since the Android testing framework does not provide corresponding APIs. We have to manually inject these events into the target app. Specifically, for Location, we call four methods under the Android testing framework: onLocationChanged(), onStatusChanged(), onProviderEnabled(), and onProviderDisable(); for Rotation, we call onRotationChanged(); for SensorEvent, we overload onSensorChanged() method and then call it. Table 2 summarizes how these five types of events are injected.

3.4.3 Event Sampling

The input data for replay is a sequence of events, each of which has a timestamp. We can consider these events as samples, and re-sample them at times different from those at which they are originally captured. In the InputLog, we record the events and the time interval of each pair of consecutive events. What we need to do is to change the time intervals and the event time accordingly without affecting the correct execution of the app.

First, we can use event sampling to cancel the latency introduced by the server. There is inevitably a latency from the point of view of the mobile phone, since the app actually runs on the server, even though the latency is small. To cancel the latency, we can shrink the time interval between every pair of consecutive events, say event a followed by event b, by the amount of event b's latency; please refer to Section 4.2 where the latency of different types of input has been measured. To do this, MobiPlay carefully examines the events and identifies which are affected by the latency and which are not, and only adjusts the time intervals associated the former. Note that MobiPlay adjusts the event time of each event as well according to the shrunk time intervals.

Second, we can replay an app in fast mode with MobiPlay by adjusting the time interval between two consecutive activities, similar to the technique used in RERAN. For instance, imagine that the user zooms in and then clicks a button on the screen, we can shorten the time interval between the two activities when replaying the app.

4. EVALUATION

In this section, we will evaluate MobiPlay. First, we demonstrate that MobiPlay can record and replay a variety of mobile apps. Then, we measure the latency introduced by the server, and the time/space overhead. Finally, we will test the event sampling technique.

Table 2: Details of data injection in white-box testing.

Type	Recreated?	Modify app?	Injection description	Injection method
MotionEvent	Yes	No	Use instrumentation class in Android testing framework.	SendPointerSync( )
KeyEvent	Yes	No		SendKeySync( )
SensorEvent	No	Yes	Developers need manually inject these data to the target application.	Call onSensorChanged( )
Location	Yes	Yes		Call four functions
Rotation	Yes	Yes		Call onRotationChanged ( )

4.1 Usability

For usability, MobiPlay currently does not support mobile apps that require ARM-based third party libraries, since the server in MobiPlay is x86-based. MobiPlay does not support camera or 3D acceleration either. As listed in Table 3, we randomly tried 52 apps from Google Play in different categories, including games, tools, news, health & fitness, lifestyle, education, shopping, etc., none of which requires ARM-based third party libraries or 3D acceleration libraries. We have successfully recorded and replayed all of the 52 apps (the replay is done on the server side multiple times).

4.2 Latency

The client-server model introduces latency. From the view of the mobile user, she cares about how long it takes to get a response for her input. For instance, suppose she clicks a button on the client, then how long does it take till she notices that the click really happens? Thus, we define the latency as the time interval between the time the input occurs at the mobile phone and the time the input takes effect on the mobile phone; that is, the round-trip time of the input between the mobile phone and the server. To measure the latency, we have designed an app such that the screen turns red when an input event finishes. Then we run the app on MobiPlay and record the time when an event occurs on the mobile phone and the time when the screen turns red. For instance, for the event of *click*, we record the time when ACTION_UP of click occurs and the time when the click spot turns red.

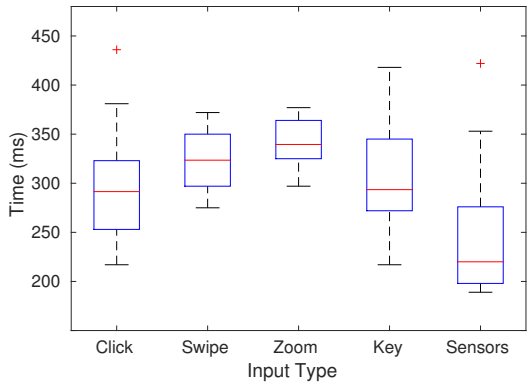

Figure 7: Round-trip time for different types of input.

We have evaluated the latency for five different types of input, each with 10 rounds. Figure 7 show the box plot of the results. As we can see, all the latency is less than 450 milliseconds, and the average is below 350 milliseconds, which does not affect the continuity of app execution and is acceptable to most testers. Individually, the *sensor* input has the shortest latency; the reason is that MobiPlay only needs to intercept four fields of data, as listed in Figure 5. The inputs of *swipe* and *zoom* have longer latency; one reason is that MobiPlay has to intercept more data including historical data (refer to Figure 5) and SVMP also batches a sequence of actions. The input of *click* and that of *key* have nearly the same latency, shorter than that of swipe and zoom but longer than that of sensor. Even though they belong to different categories of events (MotionEvent and KeyEvent, respectively), they share the same action with an ACTION_DOWN and an ACTION_UP, leading to similar latency.

4.3 Time and Space Overhead

We have measured the time overhead. Table 4 shows the result of 8 apps, which are either touch-intensive or sensor-intensive. Column 2 is the original run time, which is the time of an app running in the record mode. Column 3 is the replay time, which is the time of the app running in the replay mode with the corresponding recorded data. Column 4 is the time overhead. As we can see, the time overhead ranges roughly from 2% to 4%. We believe that at least the following three factors contribute to the overhead. First, during the replay, after injecting the first event of a pair of consecutive events, MobiPlay waits a period of the time interval of the two events before injecting the second event by utilizing the *thread.sleep* method; however, *thread.sleep* is inaccurate, and operation in parallel can lead to excessive sleep. Second, it takes time for MobiPlay to adjust the time information of an event on basis of current time. Third, reading the input data from the disk needs time as well.

We also have measured the size of the recorded input data for the same 8 apps. Additionally, we have recorded the number of events for each of five categories. In Table 4, ME, KE, SE, R, and L stand for MotionEvent, KeyEvent, SensorEvent, Rotation, and Location, respectively. The first six are touch-intensive, and the rest two are sensor-intensive. As the table illustrates, the more events, the larger the data size; and each motion event occupies more space than each sensor event, since the former has more parameters than the latter.

4.4 Event Sampling

Our evaluation here focuses on the event re-sampling on touchscreen gestures themselves, i.e, the MotionEvent. RERAN has conducted similar test, but it has focused on time warping during data entry (such as shrinking time interval between two button presses) and content processing (such as reading a story) instead of touchscreen gestures. RERAN states that manipulating the speed of touchscreen gestures can easily modify the gesture's effect or convert it to a different action or set of actions. We have found that our re-

Table 3: The apps that MobiPlay has recorded and replayed successfully.

Name	Category	Name	Category
Exploration Lite	Adventure & Creativity	Cartwheel by Target	Lifestyle
Bible	Book & Reference	Instructables	Lifestyle
Amazon Kindle	Book & Reference	MyChart	Medical
Bing Search	Book & Reference	NBC news	News & Magazines
Concur	Business	BBC News	News & Magazines
Square Register	Business	CNN News	News & Magazines
Kids Doodle	Casual	Reddit is fun	News & Magazines
ZingBox Manga	Comics	Flipboard: News Magazine	News & Magazines
Crunchyroll Manga	Comics	Hola Launcher	Personalization
TeachersPayTeachers	Education	Iron Man 3 Live Wallpaper	Personalization
Math Expert	Education	photo editor	Photography
Bing Dictionary (ENG - CHN)	Education	Emoji Keyboard	Productivity
Chase Mobile	Finance	Evernote	Productivity
Mint: Personal Finance & Money	Finance	Onet Connect Fruit	Puzzle
Bank of America	Finance	Amazon for Tablets	Shopping
Tic Tac Toe Free	Game	Best Buy	Shopping
Bubble Shooter Classic	Game	Meetup	Social
Crush Eggs	Game	NFL Fantasy Football	Sports
Word search	Game	Sensor Box for Android	Tools
Chinese Checkers Wizard	Game	Sensors	Tools
Pedometer	Health & Fitness	File Manager	Tools
Calorie Counter - MyFitnessPal	Health & Fitness	Shell Terminal Emulator	Tools
Noom Walk Pedometer	Health & Fitness	Clock	Tools
Cardboard	Libraries & Demo	Adobe AIR	Tools
Always Positive -Daily Quotes	Lifestyle	Amber Weather	Weather
DIY Garden Ideas	Lifestyle	The Weather Channel	Weather

Table 4: The time and space overhead and number of events in each category.

App name	Running time (seconds)			Data size (KB)	# of ME	# of KE	# of SE	# of R	# of L
	Original	Replay	Overhead						
KidsDoodle	192.74	199.98	3.7%	381.5	4571	4	0	0	0
Bible	155.17	159.58	2.8%	171.6	1989	8	0	0	5
Bing Dictionary	134.45	138.03	2.7%	78.9	909	20	0	0	0
Bing Search	174.98	179.80	2.8%	123.8	1409	12	0	0	9
BBC News	156.65	161.18	2.9%	127.8	1452	6	0	0	0
Amazon Kindle	201.93	207.74	2.9%	99.4	1180	22	0	0	0
Pedometer	207.84	212.57	2.3%	880.7	141	6	22744	0	0
Sensor Box	256.76	263.16	2.5%	1100.6	39	14	28656	19	0

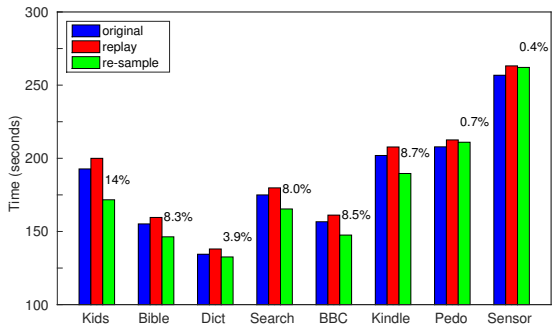

Figure 8: Re-sampling reduces the replay time.

play approach can speed up the touchscreen gestures without any error. Specifically, we sped up the touchscreen gestures twice. For instance, a *swipe* consists of an ACTION_DOWN, a sequence of ACTION_MOVE, and an ACTION_UP. We shrunk the time interval between each pair of ACTION_MOVE by half. Figure 8 shows the results of the same 8 apps in Section 4.3. The numbers above the re-sample column is the percentage of time that has been reduced by re-sampling. As we can see, the more the number of motion events, the more the time reduced (please refer to Table 4).

5. LIMITATIONS AND FUTURE WORK

One limitation of MobiPlay is that the server in our current implementation is x86-based, preventing MobiPlay from running apps that need ARM-based third-party libraries, such as 3D apps. Fortunately, there are ARM-based servers available now, and both KVM [21] and Xen [5] offer exten-

sions for ARM architecture. We leave the implementation of MobiPlay with an ARM-based server as our future work.

There are also several other directions for the future work. In principle, MobiPlay should be able to replay an app on a mobile device with the input data recorded from another device with distinct device configuration, such as resolution, screen size, etc. Therefore, one direction is to test and evaluate the cross-device portability on MobiPlay. In addition, it is worth improving MobiPlay such that it can support camera and microphone, which it currently does not.

6. RELATED WORK

A large body of research has been conducted in record-and-replay techniques, including desktop, server, and mobile phone applications. In this section, we review the most relevant works from recent literature.

Desktop and Server Applications There are bunch of record-and-replay tools in the last decade [17, 28, 19, 1, 18, 22, 29, 2, 20, 25, 33, 9]. Among them, some are event-driven, such as [17, 28]; these tools record the (x,y) pixel coordinates of mouse clicks as well as keyboard strokes, and replay this recorded information by creating new mouse and keyboard events later. Some systems utilize the keyword action technique, such as [22, 1, 29]; they work in a higher level of abstraction by capturing GUI objects. Even though some of these tools [17, 28, 19] record mouse move and mouse drag information, they cannot record and replay gestures on mobile phones like swipe, pinch and zoom due to the added complexity of multi-touch.

There are several other works in this line of research. [6] has presented a tool called Timelapse for quickly recording, reproducing, and debugging interactive behaviors in web applications. [7, 12] have designed an approach for generating test cases for web service applications. [16] has presented an approach called PUPLE to provide automated support for capturing and replaying configuration decisions. [23] has presented a tool to learn how to interact with the application under testing and stimulate its functionality by working at the system level and interacting only through the GUI. And [35] presents a system for automating bug reproduction.

Mobile Phone Applications The android SDK offers a tool called Monkey [4], which can be run on any device or emulator instance. It can generate pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. Monkey supports event sequence scripts to be fed into an app. It can also handle presses but scripting presses is labor-intensive. Furthermore, Monkey scripting does not support touchscreen gestures.

Google has also provided several tools [24, 30, 34, 10]. Monkeyrunner [24] provides an API for writing programs that control an Android device or emulator,and allows a developer to externally exercise an app. Robotium [30] fully supports native and hybrid Android apps and makes it easy to write automatic black-box tests. UI Automator [34] provides a set of APIs to build UI tests that perform interactions on user apps and system apps. Espresso [10] provides a set of APIs to test user flows within an app. All these tools hook into the app source code, which is a limitation as source code is not always available. There is another framework called GUITAR [13] for Java and Windows apps. It has been ported to Android by extending the Monkeyrunner tool to allow users to generate test cases. However, it does not support touchscreen gestures and many sensors typically found on mobile phones. MobiGUITAR [3] presents a tool for automated GUI-driven testing of Android apps, based on observation, extraction, and abstraction of the run-time state of GUI widgets.

RERAN [11] provides a record-and-replay tool to capture low-level event streams on the mobile phone, including GUI events and sensor events. However, it is not able to record and replay data from the GPS and microphone devices, because Android provides data for these devices through specified services. Furthermore, it has a potential concern regarding time dependence of events. Finally, because of the design of the replay agent there may be conflicts with other events occurring at the same time. Mosaic [14] provides a virtual screen to handle the differences across different devices. It maps a set of touchscreen events from a particular device into a set of virtualized user interfaces that can be retargeted and injected into another device. Both RERAN and Mosaic utilize *getevent* to record app data and *sendevent* to inject the recorded data for app replay. Therefore, it retains some of the problems present in RERAN. Selendroid [32] presents a test framework based on Android instrumentation framework and good for white-box testing. However, it does not provide recording functionality.

Record and replay functionality can also aid security researchers. Work that makes use of physical sensors on the Android platform, such as microphones, accelerometers, and speakers for security purposes, such as [15, 26, 27, 36], can benefit greatly from MobiPlay. Using our system, researchers can debug their applications in a traditional way. And, they can also perform sophisticated security analysis, such as monitoring sensor use, and exploring the feasibility of replay attacks.

7. CONCLUSION

In this paper, we have designed a client-server system, called MobiPlay, which allows users to record and replay mobile application executions. MobiPlay runs the the target mobile application on a server, while displaying the app GUI in real time on the mobile phone, such that the mobile phone user has exactly the same experience as if the application were running on the mobile phone. We are the first to build such a system, which records the input data at the application layer, instead of the Android framework, or the Linux kernel. This allows us to solve many difficulties the current approaches have encountered.

MobiPlay is comprehensive, flexible and efficient. First, it is able to intercept all input data, including all touchscreen gestures, and data from all sensors, better than current state-of-the-art approaches. Second, it is able to record and replay a mobile app on both the mobile phone or the server. Third, it is suitable for both white-box and black-box testing. We have implemented MobiPlay on Android and evaluated it with tens of popular applications, with supportive and convincing results.

8. ACKNOWLEDGMENTS

The authors would like to thank all the reviewers for their helpful comments. This project was supported in part by US National Science Foundation grant CNS-1320453.

9. REFERENCES

[1] Abbot Java GUI Test Framework. http://abbot.sourceforge.net/doc/overview.shtml.

[2] D. Amalfitano, A. R. Fasolino, and P. Tramontana. A gui crawling-based technique for android mobile application testing. In *Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 252–261. IEEE, 2011.

[3] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. Ta, and A. Memon. Mobiguitar–a tool for automated model-based testing of mobile apps. 2014.

[4] Android Monkey. http://developer.android.com/tools/help/monkey.html.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[6] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive record/replay for web application debugging. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, pages 473–484. ACM, 2013.

[7] K. M. Conroy, M. Grechanik, M. Hellige, E. S. Liongosari, and Q. Xie. Automatic test generation from gui applications for testing web services. In *IEEE International Conference on Software Maintenance (ICSM)*, pages 345–354. IEEE, 2007.

[8] S. Curtis. Quarter of the world will be using smartphones in 2016. http://www.telegraph.co.uk/technology/mobile-phones/11287659/Quarter-of-the-world-will-be-using-smartphones-in-2016.html, Dec 2014.

[9] 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. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.

[10] Espresso. https://developer.android.com/tools/testing-support-library/index.html.

[11] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. Reran: Timing-and touch-sensitive record and replay for android. In *35th International Conference on Software Engineering (ICSE)*, pages 72–81. IEEE, 2013.

[12] M. Grechanik, Q. Xie, and C. Fu. Creating gui testing tools using accessibility technologies. In *International Conference on Software Testing, Verification and Validation Workshops*, pages 243–250. IEEE, 2009.

[13] GUITAR. http://sourceforge.net/p/guitar/wiki/Home/.

[14] M. Halpern, Y. Zhu, R. Peri, and V. J. Reddi. Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 215–224. IEEE, 2015.

[15] H. Han, S. Yi, Q. Li, G. Shen, and E. Novak. Amil: Localizing neighboring mobile devices through a simple gesture. In *INFOCOM, 2016 Proceedings IEEE*, April 2016.

[16] W. Heider, R. Rabiser, and P. Grünbacher. Facilitating the evolution of products in product line engineering by capturing and replaying configuration decisions. *International Journal on Software Tools for Technology Transfer*, 14(5):613–630, 2012.

[17] Jacareto. http://sourceforge.net/projects/jacareto/.

[18] jfcUnit. http://jfcunit.sourceforge.net/.

[19] M. Jovic, A. Adamoli, D. Zaparanuks, and M. Hauswirth. Automating performance testing of interactive java applications. In *Proceedings of the 5th Workshop on Automation of Software Test*, pages 8–15. ACM, 2010.

[20] S. H. Khandkar, S. Sohan, J. Sillito, and F. Maurer. Tool support for testing complex multi-touch gestures. In *International Conference on Interactive Tabletops and Surfaces*, pages 59–68. ACM, 2010.

[21] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. Kvm: the linux virtual machine monitor. *Proceedings of the Linux Symposium*, 1:225–230, 2007.

[22] MarathonITE. http://marathontesting.com/.

[23] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro. Autoblacktest: Automatic black-box testing of interactive applications. In *proceedings of the Fifth International Conference on Software Testing, Verification and Validation (ICST)*, 2012.

[24] Monkeyrunner. http://developer.android.com/tools/help/monkeyrunner-concepts.html.

[25] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 284–295. IEEE Computer Society, 2005.

[26] E. Novak and Q. Li. Near-pri: Private, proximity based location sharing. In *INFOCOM, 2014 Proceedings IEEE*, pages 37–45, April 2014.

[27] E. Novak, Y. Tang, Z. Hao, Q. Li, and Y. Zhang. Physical media covert channels on smart mobile devices. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pages 367–378, New York, NY, USA, 2015. ACM.

[28] Pounder. http://pounder.sourceforge.net/.

[29] Rational Robot. http://www.ibm.com.

[30] Robotium. https://code.google.com/p/robotium/.

[31] Secure Virtual Mobile Platform (SVMP). http://www.mitre.org/research/technology-transfer/technology-licensing/secure-virtual-mobile-platform-svmp.

[32] Selendroid. http://selendroid.io.

[33] 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, General Track*, pages 29–44. Boston, MA, USA, 2004.

[34] UI Automator. https://developer.android.com/tools/testing-support-library/index.html.

[35] M. White, M. Linares-Vásquez, P. Johnson, C. Bernal-Cárdenas, and D. Poshyvanyk. Generating reproducible and replayable bug reports from android application crashes. In *23rd IEEE International*

Conference on Program Comprehension (ICPC), 2015.

[36] S. Yi, Z. Qin, E. Novak, Y. Yin, and Q. Li. Glassgesture: Exploring head gesture interface of smart glasses. In *INFOCOM, 2016 Proceedings IEEE*, April 2016.