Versatile yet Lightweight Record-and-Replay for Android

Yongjian Hu   Tanzirul Azim   Iulian Neamtiu
University of California, Riverside
{yhu009, mazim002, neamtiu}@cs.ucr.edu

Abstract

Recording and replaying the execution of smartphone apps is useful in a variety of contexts, from reproducing bugs to profiling and testing. Achieving effective record-and-replay is a balancing act between accuracy and overhead. On smartphones, the act is particularly complicated, because smartphone apps receive a high-bandwidth stream of input (e.g., network, GPS, camera, microphone, touchscreen) and concurrency events, but the stream has to be recorded and replayed with minimal overhead, to avoid interfering with app execution. Prior record-and-replay approaches have focused on replaying machine instructions or system calls, which is not a good fit on smartphones. We propose a novel, stream-oriented record-and-replay approach which achieves high-accuracy and low-overhead by aiming at a sweet spot: recording and replaying sensor and network input, event schedules, and inter-app communication via intents. To demonstrate the versatility of our approach, we have constructed a tool named VALERA that supports record-and-replay on the Android platform. VALERA works with apps running directly on the phone, and does not require access to the app source code. Through an evaluation on 50 popular Android apps, we show that: VALERA's replay fidelity far exceeds current record-and-replay approaches for Android; VALERA's precise timing control and low overhead (about 1% for either record or replay) allows it to replay high-throughput, timing-sensitive apps such as video/audio capture and recognition; and VALERA's support for event schedule replay enables the construction of useful analyses, such as reproducing event-driven race bugs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Reliability, Validation; D.2.5 [*Software Engineering*]: Testing and Debugging—testing tools

General Terms Reliability, Verification

Keywords Mobile applications, Record-and-replay, Google Android, App testing, Event-based races

OOPSLA'15, October 25–30, 2015, Pittsburgh, PA, USA
© 2015 ACM. 978-1-4503-3689-5/15/10...$15.00
http://dx.doi.org/10.1145/2814270.2814320

1. Introduction

The ability to record and replay the execution of a smartphone app is useful in many contexts: reproducing bugs to support debugging [19], recording a user's interaction and replaying it for profiling and measuring [28], generating inputs to support dynamic analysis and testing [27]. While useful, this task has proven difficult: smartphone apps revolve around concurrent streams of events that have to recorded and replayed with precise timing. To keep overhead low, prior record-and-replay approaches for smartphones only capture GUI input [1–3, 11, 13, 17, 19] which hurts accuracy as they cannot replay input from the network or sensors, e.g., GPS, camera, and microphone, which are used frequently by popular apps; or events, to reproduce event-based races [15, 21]. Prior work on record-and-replay for desktop and server platforms [10, 12, 22, 29, 30] has relied on techniques such as hardware changes, VM logging, or system call interception; using such techniques on Android is problematic, due to high overhead and wrong granularity—their word- or object-level granularity can be used for small, timing-insensitive programs, but not for Android apps.

In desktop/server programs input comes from the file system, network, mouse or keyboard; with the exception of drags and double-clicks, large changes to input timing between the record and replay executions are gracefully tolerated by the program. In contrast, on smartphones, input can come concurrently from the network, GPS, camera, microphone, touchscreen, accelerometer, compass, and other apps via IPC. Moreover, the timing of delivering these input events during replay must be extremely accurate, as even small time perturbations will cause record or replay to fail.

To address these challenges, we introduce a novel, *sensor- and event-stream driven* approach to record-and-replay; by focusing on sensors and event streams, rather than system calls or the instruction stream, our approach is effective yet lightweight. We have implemented our approach in a tool called VALERA (VersAtile yet Lightweight rEcord and Replay for Android)[1] that records and replays smartphone apps, by intercepting and recording input streams and events with minimal overhead and replaying them with exact timing. VALERA works for Android, the dominant smartphone and tablet platform [7, 8].

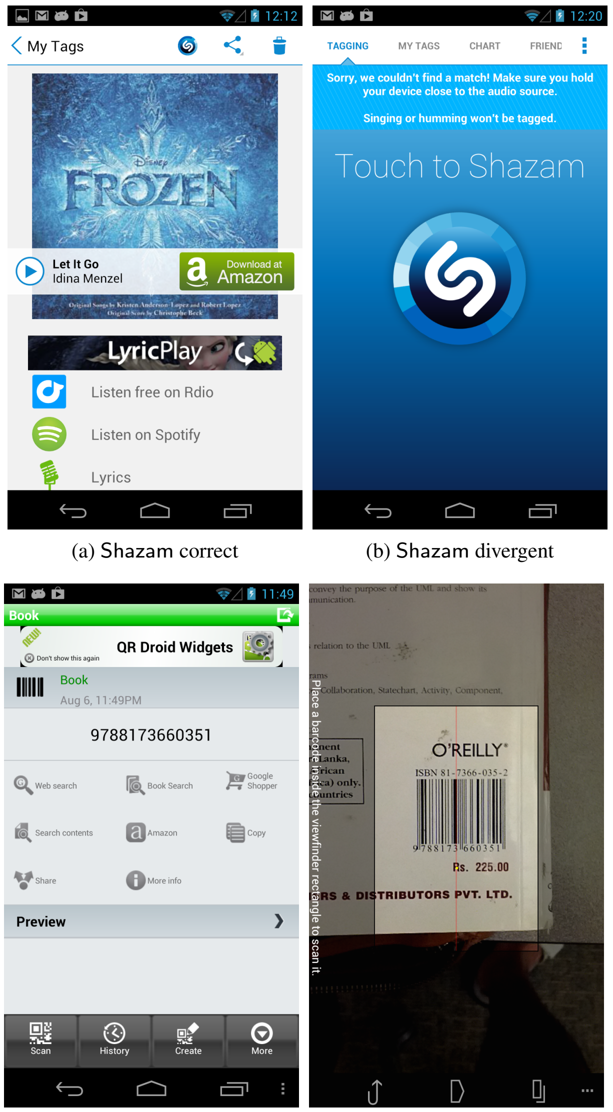
Stream-driven replay hits a "sweet spot" on the accuracy vs. overhead curve: replaying sensor inputs and events with precise timing allows VALERA to be lightweight yet achieve high accuracy. For example, we can replay apps such as Barcode Scanner, Amazon Mobile or Google Goggles, which perform high-throughput video recording and analysis; apps such as GO SMS Pro or Shazam, which perform sound capture and music recognition; apps such as Waze, GasBuddy, TripAdvisor, and Yelp which perform GPS-based navigation and proximity search; IPC-intensive apps such as Twitter and Instagram; finally, we can reproduce event-driven race bugs in apps such as NPR News, Tomdroid Notes, and Google's My Tracks. At the same time, we keep the performance overhead low, on average 1.01% for record and 1.02% for replay.

To show the importance of controlling overhead and timing as well as ensuring schedule determinism, in Section 2 we present three examples of popular apps—Shazam, QR Droid, and Barcode Scanner, where failing to control these aspects during record or replay leads to divergence. We achieve replay fidelity by eliminating *sensor input nondeterminism*, *network nondeterminism*, and *event schedule nondeterminism*. We verify the fidelity by checking the equivalence of *externally-observable app states* at corresponding points in the record vs. replay execution (Section 3).

Note, however, that VALERA does not record all system state: it does not record memory accesses or the VM instruction stream (as other approaches do, albeit not on smartphones [10, 22, 29]) as this state is not externally visible. We made these design choices for two main reasons. First, to keep the approach widely applicable, we avoided hardware modifications. Second, the overhead of recording all memory accesses or the VM instruction stream is too prohibitive: our experience with PinPlay for Android (a whole-system record-and-replay approach) shows that its high overhead perturbs the execution significantly so apps stop being interactive (Section 2.2).

Section 4 describes our implementation. We employ *API interceptors* to intercept the communication between the app and the system to eliminate nondeterminism due to network and sensor inputs, as well as inter-app communication via intents. Second, we introduce *ScheduleReplayer*, an approach for recording and replaying event schedules to eliminate event schedule nondeterminism and allow hard-to-reproduce bugs, such as event-driven races, to be deterministically replayed and reproduced (Section 5).

[1] Available at `http://spruce.cs.ucr.edu/valera/`



(a) Shazam correct (b) Shazam divergent

(c) QR Droid correct (d) QR Droid divergent

Figure 1: Screenshots of correct execution (left) and divergent execution due to imprecise timing (right).

In Section 6 we evaluate VALERA's effectiveness and efficiency. First, we show that VALERA allows versatile record-and-replay; we illustrate this for 50 widely-popular Android apps (most of these apps have in excess of 10 million installs) that use a variety of sensors, and show that VALERA can successfully replay them. Second, experiments show that VALERA is efficient: it imposes just 1.01% time overhead for record, 1.02% time overhead for replay, 208 KB/s space overhead, on average, and can sustain event rates exceeding 1,000 events/second. Third, VALERA's support for deterministic replay of asynchronous events allows us to reproduce event-driven races in several popular apps, that would be very difficult to reproduce manually.

The evaluation has revealed several interesting traits of replaying Android apps: apps that capture video and audio streams have *stringent timing constraints*; apps that use the camera impose the highest *space overhead*; apps using the network intensively impose the highest *performance overhead*; and scheduler events far surpass any other events (e.g., from sensors) in terms of event quantity and event rate.

In summary, our main contributions are:

1. A new, stream-oriented approach for recording-and-replaying Android apps.
2. VALERA, a high-accuracy low-overhead record and replay tool for Android apps running on real phones and without requiring access to the app source code.
3. An evaluation of VALERA on 50 popular Android apps.

2. Motivation

We now illustrate how low overhead, accurate timing, and schedule determinism are critical for successful record and replay of popular Android apps.

2.1 Accurate Timing and Low Overhead

Consider two popular apps, Shazam and QR Droid, that use sensor stream inputs. For each app, we first perform a record with VALERA; thanks to VALERA's low overhead, the support for record has no impact on the execution. Next, we replay the app with VALERA normally, i.e., with precise timing (we call this the "correct" execution as there is no visible difference between the recorded and replayed executions). Then, we replay the app with VALERA again, but alter timing during the sensor replay phase, i.e., deliver events slightly earlier or later than originally recorded. The effect is divergence (the app exhibits different behavior), hence we call this the "divergent" execution. In Figure 2(a) we show a screenshot of the Shazam music recognition app replayed with VALERA with exact audio stream timing, and correctly recognizing a song; Figure 2(b) shows what happens—Shazam fails to recognize the song, per the message on top "Sorry we couldn't find a match"—when we deliberately speed up the audio stream timing by 40%. Figure 2(c) contains a screenshot of the QR Droid barcode scanner app replayed with VALERA with exact frame buffer timing, correctly recognizing a book barcode (97881173660351); Figure 2(d) shows that QR Droid fails to recognize the barcode, per the message on the left side "Place the barcode inside the viewfinder rectangle to scan it", when we deliberately introduce a 200 millisecond delay between frames.

The importance of overhead during record and replay. The previous examples illustrate how the execution of mobile apps is sensitive to input stream timing, and how timing deviations lead to record or replay errors. We now discuss two more classes of timing errors in Android apps.

First, Android No Response (ANR) error: Android will raise an ANR and kill the app if the UI thread cannot handle an event within 5 seconds or if an IPC event gets no response for 10 seconds. Thus, if recording overhead is high (e.g., recording every memory access), then the system is likely to raise an ANR error, terminating record or replay.

Second, the semantics of UI gestures may change [19]. When the user touches the screen, the hardware generates a series of motion events, starting with ACTION_DOWN and ending with ACTION_UP. The time between ACTION_DOWN and ACTION_UP is crucial for deciding whether the user input is a tap, a click, or a long click. High-overhead recording affects input timing hence the semantic of the gesture changes, e.g., a click becomes a long click.

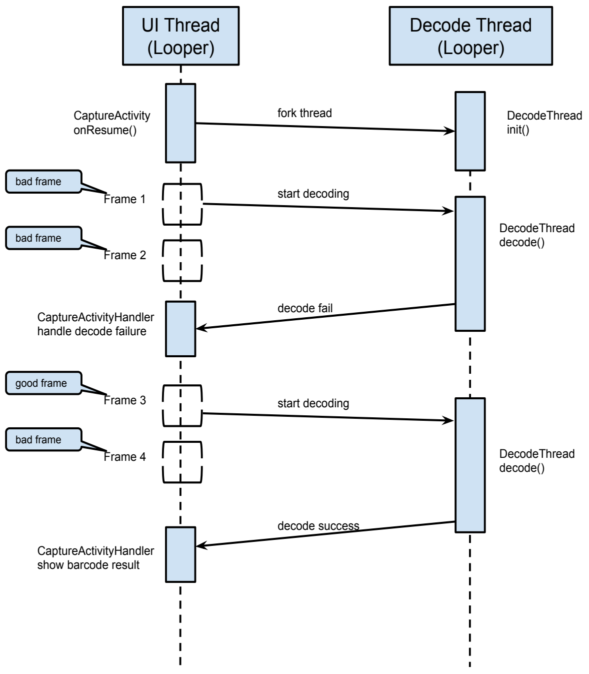
This overhead issue affects the usability of PinPlay [23], a record-and-replay system that has been recently ported to Android. When using PinPlay to record app executions, the overhead is prohibitively high because PinPlay instruments the screen device driver in the OS, the OS code that dispatches the input events, and finally the Android Java code that dispatches events. We have observed that, when performing a touchscreen gesture, apps respond very slowly, showing the ANR error; eventually the OS kills the app. For example, when attempting to use PinPlay to record the Amazon Mobile app, sending multiple clicks triggered time outs and eventually the OS killed the test app after 90 seconds. In contrast, VALERA can handle Amazon Mobile essentially in real-time (1.7% overhead for record and 2.34% overhead for replay).

2.2 Schedule Replay

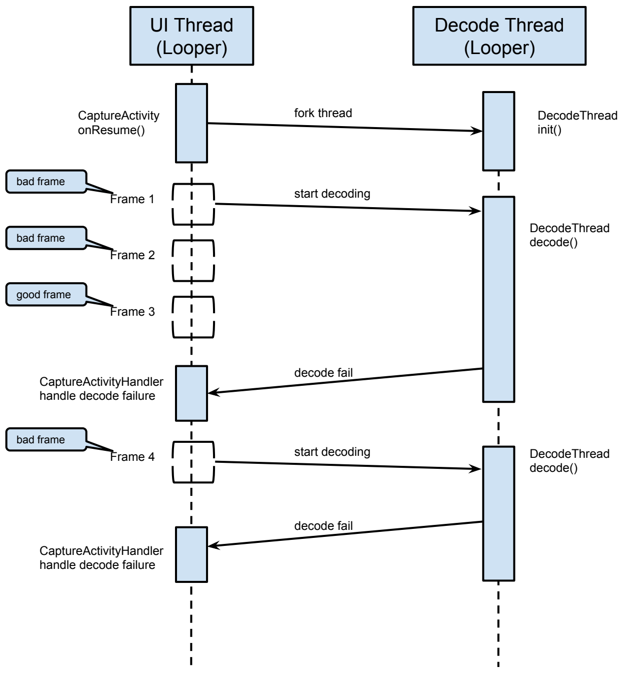
Consider the popular Barcode Scanner app that can recognize various types of barcodes. Figure 2a illustrates the working model of Barcode Scanner. When the user starts the app, the UI thread will load the CaptureActivity screen. In the onResume() method, the UI thread forks a new thread, DecodeThread, that performs heavy-weight computation—decoding the barcode—thus relieving the burden of the UI thread and ensuring the app is responsive. After initialization, it waits to receive events from the UI thread.

In the onResume() method, the UI thread also opens the camera and registers PreviewFrameCallback as a handle for callbacks from the camera; this handler invoked by the hardware periodically. If a frame, say frame1, arrives from the hardware, onPreviewFrame in the UI thread handles this callback by sending a message with the "start decoding" flag together with the frame buffer data to the decode thread. The decode thread starts to use various complex algorithms to decode this frame. The UI thread continues to handle other events. Suppose a second frame, frame2 arrives, but the decode thread has not finished yet. The handler in the UI thread knows that the decoding thread is still working, so it discards frame2.

Suppose the decode thread finishes decoding and fails to recognize the barcode from the given frame. This is a normal case, due to various reasons such as the frame does not contain any barcode, or the frame taken by the camera is blurred. When the UI thread receives the "decode failure" message, it marks the decode thread to be ready for decoding and con-



(a) Barcode Scanner correct



(b) Barcode Scanner divergent

Figure 2: Schedule of correct execution (left) and divergent execution due to imprecise schedule replay (right).

tinues to receive new frames from camera. Say frame3 comes next, and this frame can be successfully decoded; a frame4 would be skipped for the same reason as frame 2. When the UI thread receives a "decode success" message, it updates the UI element to show the barcode result.

To successfully replay Barcode Scanner, the key is to enforce the same event order executed in the looper as in the record phase. Otherwise, the replay may diverge, as showed in Figure 2b and explained next. Suppose that in the recorded four frames, only frame3 can be successfully decoded. Since the Android camera does not guarantee a constant frame rate, the "decode failure" message due to frame1 could arrive after frame3 has been delivered to the UI thread, in which case frame4 will be sent to the decode thread, instead of frame 3. But frame4 could be a poor-quality frame that cannot be recognized; now the replay diverges as the app cannot successfully decode *any of the four frames*—this situation can happen in RERAN but not in VALERA. When using VALERA the "decode failure" message will be delivered after frame 2 not frame 3 since we enforce schedule determinism (Section 5) hence avoiding divergence.

This example illustrates the importance of ensuring event order determinism in the replay of Android apps, as Android apps are mainly event-driven. In fact, as our study in Section 6.2.2 shows, the event stream is far more demanding than other streams, with typical burst rates in excess of 1,000 events/second, whereas the second-most demanding stream, the network, has typical burst rates of 207 events/second.

Table 1: Network and sensor API usage in top-11 apps in each of the 25 categories on Google Play.

	Network	Location	Audio	Camera
percentage	95%	60%	34%	34%

2.3 Network and Sensors

Supporting network and sensors is essential: the success of the smartphone platform is due, in no small part, to on-the-go network connectivity and the sensor-based context-aware capabilities offered by apps. To quantify the need to replay network and high-level sensors, in Table 1 we show their use frequency in top-11 most popular apps across all 25 app categories on Google Play: 95% of the apps use the network, location is used by 60% of the apps, etc. Thus we argue that supporting network and sensor input is necessary for successfully recording-and-replaying Android apps.

3. Overview

We believe that, to be practical, a record-and-replay system for Android should meet several key desiderata:

1. *Support I/O (sensors, network) and record system information required to achieve high accuracy and replay popular, full-featured apps.*
2. *Accept APKs as input—this is how apps are distributed on Google Play—rather than requiring access to the app source code.*
3. *Work with apps running directly on the phone, rather than on the Android emulator which has limited support for only a subset of sensors.*

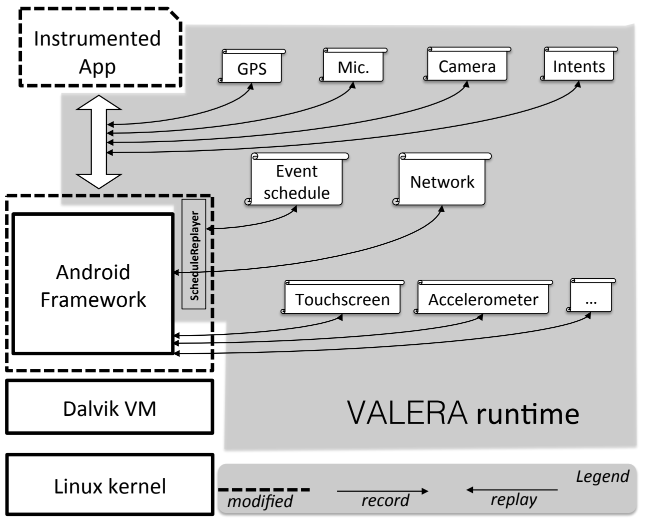


Figure 3: Overview of the VALERA runtime.

4. *Low overhead to avoid perturbing the app's execution.*
5. *Require no hardware, kernel, or VM changes.*

We have designed VALERA specifically to meet these desiderata. Current approaches [2, 3, 11, 19] do not meet one or more of these desiderata (especially #1, which we believe is critical). We now turn to presenting overviews of the Android platform and our approach, then state VALERA's replay accuracy guarantees.

Android overview. The Android software stack consists of apps using the services of the Android Framework ("AF"). Each app runs in its own copy of the Dalvik Virtual Machine (VM)[2] which in turn runs on top of a custom, smartphone version of the Linux kernel. Android apps are typically written in Java and compiled to Dalvik bytecode that runs in the VM. Apps are distributed as APK files, which contain the compressed Dalvik bytecode of the app (`.dex`) along with app resources and a manifest file.

VALERA ***overview.*** VALERA consists of a runtime component and an API interception component. We first discuss the runtime component, shown in Figure 3 (the grey area on the right)—the interception component will be discussed in Section 4.2. The instrumented app runs on top of the instrumented AF, which in turn runs on top of unmodified versions of the VM and the kernel. App instrumentation, achieved via bytecode rewriting, is used to intercept the communication between the app and the AF to produce log files (values and timestamps) associated with network and high-level sensor input, such as GPS, microphone, and camera; intents are also intercepted at this point. AF instrumentation (which we performed manually) is used to log and replay the event schedule—see the ScheduleReplayer vertical box inside the AF. As the arrow directions indicate, during record the value/timestamp stream flows from left to right (toward the log files), and during replay from right to left (from the log files to the app/AF). To sum up, the VALERA runtime consists of record and replay code and the log files; this code runs inline in the app and AF, with no extra processes or threads needed. Other apps that execute concurrently run in their own address space, on their own VM copies; we omit them for clarity.

Note that, since VALERA uses bytecode rewriting and an instrumented AF, its operation is not affected by either the JIT compiler used in Android versions prior to 5.0 or the runtime/compiler combination (ART/AOT) used in Android versions 5.0 and beyond.

[2] This applies to Android versions prior to 5.0, since VALERA was constructed and evaluated on Android version 4.3.0. In Android version 5.0 and later, Android uses a runtime system (ART) and ahead-of-time compilation (AOT).

Replay accuracy. We define *Externally Visible State* as the subset of app state that might be accessed, or viewed, by the user; currently the EVS includes GUI objects (views, images) and Shared Preferences (a key-value store where apps can save private or public data [4]).

We validated VALERA's replay fidelity via snapshot differencing, as follows: (1) during record, upon entering or leaving each activity (screen) A, we snapshot the EVS into EVS_{recA}; (2) likewise, we snapshot the EVS during replay, into EVS_{repA}; and (3) compare EVS_{recA} and EVS_{repA} to find differences—a faithful replay should show no difference, that is, the user cannot tell the difference between the record and replay executions. Note that record vs. replay differences might still exist in hidden state, e.g., memory contents or the VM stream, but these differences are not our focus. Nevertheless, our fidelity guarantee is stronger than techniques used in prior approaches to compare app executions (which just compared screen contents [14]). The next sections show how we effectively implement record and replay in VALERA to achieve these accuracy guarantees.

4. API Interception and Replay

We now present our *API interception* approach. The infrastructure for interception, record, and replay, is generated automatically through app rewriting, based on an interceptor specification. While VALERA has a predefined set of interceptors, VALERA users can easily change or extend this specification, e.g., to add interceptors for new sensors or API calls, while still guaranteeing *input determinism*.

4.1 Example: Intercepting the Location Services

We first illustrate our technique by showing how we intercept, record and replay the GPS location. The location API provides functionality for finding the current GPS location, as well as receiving notifications when the location changes, e.g., when the smartphone is in motion.

Figure 4 shows an excerpt from a simple app that uses the location services to display the current location on the GUI and update the display whenever the location changes.

```
1  //class LocationActivity extends Activity
2  protected void onStart() {
3     location =mLocMgr.getLastKnownLocation(provider);
4    updateUILocation(location);
5    mLocMgr.requestLocationUpdates(provider,
6        TEN_SECONDS, TEN_METERS, listener);
7  }
8
9  protected void onStop() {
10   mLocMgr.removeUpdates(listener);
11 };
12
13 private LocationListener  listener  = new
        LocationListener() {
14   @Override
15   public void onLocationChanged(Location location) {
16     updateUILocation(location);
17 };};
```

Figure 4: Location API example.

When the app starts (method onStart()) it asks the manager for the last known location (getLastKnownLocation on line 3), updates the current location in the GUI (line 4) and directs the manager to provide location updates every 10 seconds or when the location has changed by 10 meters (lines 5–6). The location updates are provided via a callback mechanism: note how, on line 6, a listener is passed as an argument to requestLocationUpdates. This location listener, initialized on line 13, has a method onLocationChanged() which is the callback that will be invoked whenever the location manager needs to inform the app about a location update—when that happens, the app updates the location on the GUI (line 16).

The getLastKnownLocation method returns a location object containing the last known location obtained from the given provider. We name such API calls *downcalls*: they are initiated by the app and go downwards (i.e., into the AF), run synchronously with the calling thread and return the result. Another kind of API calls are *upcalls*: in this case the lower levels of the system software (i.e., the AF) invoke an app-defined callback hence the direction of the call is upward. The onLocationChanged method is an upcall since after registering the callback, the AF periodically sends back the updated location by invoking onLocationChanged. By overriding this method, the app receives notifications whenever the user's location changes.

VALERA takes API annotations (e.g., upcalls, downcalls) as input and generates the support for interception, record, and replay automatically. In our location example, VALERA records the values exchanged in location API upcalls and downcalls and upon replaying, feeds the app the recorded values—this way we can "trick" the app into believing that the phone's geographical location (or sequence of locations, if in motion), is the same as during the record phase, even though the phone's location could have changed since then. For example, when a developer in New York wants to replay and debug a location-related crash that happened on the phone of a user based in San Francisco, VALERA injects the record-time GPS sequence (the San Francisco coordinates) into the app.

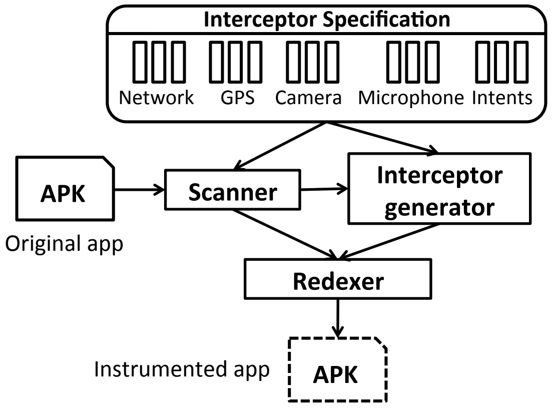


Figure 5: Overview of VALERA's automatic interception.

4.2 Automatic Interception through App Rewriting

Figure 5 presents the VALERA interception component, which performs automatic interception via app bytecode rewriting. While the rewriting is currently performed on a desktop or laptop, VALERA's record-and-replay (runtime component) runs on the phone with no outside system necessary.

We use the bytecode rewriting capabilities of Redexer (an off-the-shelf Dalvik bytecode rewriting tool [18]) along with interceptor specifications to transform an off-the-shelf app into an app with intercept/record/replay capabilities, as explained next.

The *Scanner* takes as input the original app (APK file) along with an *Interceptor specification* and finds all the callsites in the bytecode that match the specification and should be intercepted. The specification consists of a list of API methods along with simple annotations on how the methods and their parameters should be treated from the point of view of intercept/record/replay (explained in Section 4.3). We have a predefined library of such calls for instrumenting commonly-used APIs (Section 4.4); users can expand the library with their own specifications. The Scanner first extracts the Dalvik bytecode file from the APK, then finds matching method invocations (callsites). The *Interceptor generator* produces two parts: a dynamic intercepting module and a stub. The intercepting module is a plug-in for the Redexer that executes on each method that needs to be intercepted (i.e., the Redexer rewrites the call to go to the stub). Finally, the dynamic intercepting modules and stubs are passed on to the *Redexer* that performs the bytecode rewriting to effect the interception, and repackages the bytecode into an instrumented APK that now has interception/playback support.

```
[downcall]
public Location getLastKnownLocation(String provider);

[upcall]
public abstract void onLocationChanged (Location location);
```

Figure 6: Location interceptor example.

```
[upcall]
public abstract void onPictureTaken (
  [xpointer] byte[] data,
  Camera camera);

[upcall]
public abstract void onPreviewFrame (
  [xpointer] byte[] data,
  Camera camera);
```

Figure 7: Camera interceptor example.

4.3 Interceptor Specification

The interceptors specify what to intercept and how to handle the intercepted values upon replay. VALERA supports three simple annotation keywords to describe Android API methods. We borrow this idea from R2, an application-level record/replay framework that allows users to choose the functions to be recorded and replayed [12].

Examples. We first provide examples of how users can easily specify interceptors to achieve record-and-replay, and then discuss the general framework. In the examples, for clarity, we leave out full package names and module annotations—these have to be specified before passing the specification to VALERA but they are straightforward.

Figure 6 shows a specification snippet from our interceptors for the Location API. First, we specify that getLastKnown Location is a downcall and that the provider argument cannot be modified during replay (i.e., it is not subject to record and replay). However, the return value of getLastKnownLocation is subject to record and replay, hence will be recorded and during replay, VALERA will return to the app the location values that were saved during record, rather than the current location. The specification also mentions that onLocationChanged is an upcall, and that the location argument will be filled with the recorded value rather than the value provided by the framework.

We did not find it necessary to support `in` or `out` annotations on method parameters, because the stub implementations in our predefined stub library implement the appropriate `in` or `out` semantics for each API call anyway.

Figure 7 shows a snippet from our interceptor specification for the Camera API: first, we specify that onPictureTaken and onPreviewFrame are both upcalls, and the camera argument is not subject to record and replay. The annotation on data is more interesting: it is an array whose size *varies between record and replay*, hence the xpointer annotation (we will provide details shortly).

Table 2: Annotation keywords.

Annotation	Scope	Description
`xpointer`	parameter	pointer to reference
`downcall`	function	synchronous API call
`upcall`	function	asynchronous callback

General annotation framework. Table 2 lists the annotations VALERA provides for constructing interceptors. There are two categories of keywords: parameter and function. Parameter keywords describe whether the value can change in size from record to replay. Function keywords label each method as downcall or upcall.

`Xpointer` is necessary when objects vary in size from record to replay. For example, in the previously-mentioned camera interceptor, onPictureTaken and onPreviewFrame take a **byte**[] data argument. Let us assume that during the record phase the user takes a 1 MB picture. However during replay, the camera may take a 500 KB picture (while the image width and height do not change between record and replay, the camera captures different images in the two different executions hence due to differences in pixel colors and the use of compression, the image sizes will likely be different). Since attempting to copy the 1 MB data from the log into a 500 KB byte array would cause buffer overflow, VALERA offers an `xpointer` annotation to deal with such cases. Instead of directly accessing the **byte**[] data reference, `xpointer` wraps the reference and provides get and set methods. During replay, the stub updates the `xpointer` reference to point to the recorded stream data.

`Downcall` denotes a synchronous API method call. In Figure 6, getLastKnownLocation is annotated as a downcall method since the caller waits until it returns.

`Upcall` denotes an asynchronous callback. Android apps rely heavily on callbacks to improve system efficiency. For example, onLocationChanged, onPictureTaken, onPreviewFrame in Figures 6 and 7 are callback methods, thus they are marked with `upcall`.

4.4 Intercepting Events and Eliminating Nondeterminism

We now describe how VALERA intercepts sensor events and eliminates various sources of nondeterminism.

Motion and Key Events. Motion (i.e., touch screen) and Key events are the main sources of events that drive app execution. Prior manual replay tools such as Robotium [11] and Monkey Runner [3] provide programming scripts that allow the user to specify which GUI objects to interact with. However, they only support basic GUI operations such as click or long click, whereas most mobile apps provide rich gestures such as zoom, pinch, and swipe. Our previous record-and-replay system, RERAN, supports these complex gestures by recording the event streams at the OS driver level and replaying them back with precise timing. However, the drawback of RERAN is that it has no knowledge about the app's events order. For example, in the Barcode Scanner example in Figure 2a, suppose that during recording, four frames were recorded but during replay, due to the unexpected nature of the external events, the camera may invoke callbacks at a slower rate and only replay three frames. If the fourth frame is the successful frame, then replay will fail (diverge).

To address these issues, VALERA records motion and key events on the app's side instead. Whenever the Windows Manager Service dispatches the event to the app, VALERA intercepts the dispatchInputEvent method recording the event data and the time since app start. In addition, VALERA records the current window ID of the app because Android dispatches motion and key event to each window and one app may have multiple windows (e.g., Activity and Dialog).

Sensor Events. Mobile devices provide a richer set of sensors than desktop/server machines. They can be classified into two categories: low-level sensors and high-level sensors. Low-level sensors, e.g., accelerometer, gravity, gyroscope, etc., provide streams of events and invoke the app via the SensorManager API. VALERA records and replays the event and its associated data.

High-level sensors such as GPS, Camera, and Audio, are richer, as they provide principled APIs for device access via upcalls and downcalls; we illustrate these using the Location API.

The *Location API* offers the getLastKnownLocation() downcall—the app waits until the system returns the last known location data. The location API also provides an upcall, onLocationChanged(): when the physical location has changed, the hardware GPS sensor invokes this upcall on the UI thread's Looper as an event. VALERA records and replays both downcalls and upcalls.

Camera. Android apps can use the camera to take pictures in three ways. First, apps can use the preinstalled Camera app as a proxy by sending it an intent. The Camera app takes the picture and returns it via the intent mechanism. The intent recording mechanism, which we will describe shortly, ensures that pictures taken via intents will be replayed. The second way is to use the frame buffer, i.e., continuously read from the camera's sensor, similar to using a camera's preview feature. A typical example of such use is in barcode scanning apps, e.g., Barcode Scanner or RedLaser Barcode. These apps read from the frame buffer using the onPreviewFrame upcall, scan the frame for a barcode picture, and direct the user on how to properly expose the barcode so the app can get a good reading. The third way is to take a single picture, e.g., via the onPictureTaken() upcall. VALERA intercepts all necessary camera downcalls/upcalls and intents, hence the input data from the camera can be recorded and replayed.

Audio. Android provides audio services through two main API components: MediaRecorder and AudioRecord. MediaRecorder is high-level, supports audio compression, and automatically saves the audio data to files. AudioRecord is more low-level because it captures raw audio data into a memory buffer and allows processing that data on-the-fly (akin to the camera's frame buffer described previously). Different apps use different methods for audio manipulation: if the app just needs to record the audio, using MediaRecorder is easier, but for apps that require high quality audio information and runtime processing, e.g., for audio recognition, AudioRecord is a better option. VALERA intercepts all necessary methods in the MediaRecorder and AudioRecord API.

Network non-determinism. Previous systems recorded and replayed network activity at the system call level, e.g., send() and recv(). However, in our experience, recording at the system call level is not appropriate on Android because the OS uses a socket pool: which socket is assigned to connect() is not deterministic. Our insight is that, instead, we need to record the network connections and the data transferred. Our implementation intercepts the HTTP/HTTPS protocol APIs[3] as follows. For each HTTP/HTTPS connection, VALERA records the data sent and received, the timing of the network API calls, as well as any exception encountered. During replay, VALERA feeds the app data (as well as error values or exceptions) from the log instead of sending/receiving real network data. Note that reading from a log file is faster than reading from the network. Thus VALERA needs to sleep an appropriate amount of time, according to the network connection log, to preserve precise replay timing.

Another advantage of eliminating network nondeterminism is enabling replay for apps that use dynamic layout (i.e., the GUI layout is determined by the server) which has posed problems in RERAN since RERAN assumes the same GUI layout between record and replay [19].

Random number nondeterminism. The Random number API is another possible source of non-determinism. Android provides two sets of Random API: java.util.Random and java.security.SecureRandom. The former is pseudo-random: VALERA just intercepts the seed, hence subsequent random number calls are deterministic. The latter is a stronger random number API, hence VALERA intercepts all the generated random numbers to ensure accurate replay. If an app implements its own random number library, the corresponding API has to be marked so VALERA replays it; however, we did not find any app, among our 50 examined apps, that defines a custom random number library.

[3] HTTP/HTTPS is the most widely used protocol for Android apps; VALERA can be easily extended to intercept other protocols.

4.5 Intercepting Intents

In Android, sensitive resources and devices are protected by permissions. Apps that want to use a resource directly must obtain an install-time permission, e.g., an app needs the ACCESS_FINE_LOCATION permission to access the GPS or the CAMERA permission to access the camera directly.

Android also allows apps to use devices via a proxy app, in which case no permission is required, as we described in the Camera app example. This is realized by Android's Intent mechanism, as follows. App A constructs an intent object with the ACTION_IMAGE_CAPTURE action. Then A invokes the Camera app by calling startActivityForResult () with the intent object as parameter. After the picture is taken, the result will come back through the onActivityResult () method. We intercept such methods to log the Intent object data (in this case, the picture) and use this data in the replay phase. Thus, to replay intent-based sensor input carried through proxy apps, we must intercept the Intent API, even though intents are not sensors per se.

4.6 Recording and Replaying

We now describe how record-and-replay is effectively achieved via auto-generated stubs, and how we control the timing and delivery of values and exceptions.

Stubs. API call interception is realized by redirecting the original API call to go to a stub. By default, VALERA auto-generates stubs that implement the record-and-replay functionality, as follows: during record, a stub saves parameters and return data in a log file and then passes them through to the original callee; during replay, the stub code feeds the app recorded data from the log file instead of the "fresh" data coming from the sensors. More concretely, for upcalls, the dynamic intercepting module will add pre-function stub code that executes before the intercepted method, and post-function stub code that executes after the intercepted method completes. For downcalls, the invocation instruction will be replaced to redirect to the stub code.

Timing. In addition to logging values associated with API calls, VALERA records the timestamp of each intercepted method, so the method is replayed at the appropriate time. We realized that precise timing control is crucial, because feeding the recorded values too early or too late will cause the replay execution to diverge from the record execution; this is especially true for apps using audio and image streams, as shown in Section 1. During replay, it takes much less time to just read (and feed to the app) the recorded sensor data from the log file, hence during replay VALERA sleeps for a calculated amount of time in the stub function to replicate the precise timing from the record mode.

Exceptions. VALERA has to record and replay any runtime exceptions. If an exception is logged during record, we have to re-throw it during replay.

```
class SyncMessageHandler extends Handler {
 Activity activity ;

  void onSynchronizationStarted() {
    Animation pulse = loadAnimation();
    View dot = activity.findViewById(R.id.sync_dot);
    dot.startAnimation(pulse);
  }
  void onSynchronizationDone() {
    View dot = activity.findViewById(R.id.sync_dot);
    Animation pulse = dot.getAnimation();
    pulse.setRepeatCount(0);
  }
  public void handleMessage(Message msg) {
    if (msg.what == SYNC_START) {
      onSynchronizationStarted();
    else if (msg.what == SYNC_DONE)
      onSynchronizationDone();
  }}
```

Figure 8: Source code of race bug in Tomdroid.

Limitations. VALERA cannot handle nondeterminism in apps that perform customized rendering, e.g., games that do not use the Android UI toolkit; while VALERA will record and replay the app, it does not guarantee that the nondeterminism in the customized rendering part will be eliminated.

5. Event Schedule Replay

In the previous section, we have shown how VALERA eliminates network and sensor nondeterminism. This, however, is not enough, as in Android, another important source of non-determinism is the event schedule. We now describe our approach for eliminating *event schedule nondeterminism*. We first motivate the need for replaying event schedules with a real-world event-driven race bug in the Tomdroid app. Next we provide an overview of Android's event model, then we describe how events are recorded and then replayed in VALERA.

5.1 Example: Tomdroid's Event-driven Race Bug

Tomdroid is an open source note-taking app; it allows notes to be saved on, and synchronized with, a remote server. When the user clicks a 'Sync' button to synchronize notes, a new background worker thread ("sync task") is forked to perform this task. Periodically, the sync task sends back the progress status to the main thread. These status messages are handled by a SyncMessageHandler; each activity ("activity" means a separate GUI screen in Android parlance) has an associated SyncMessagehandler. If the received message is SYNC_START, the main thread invokes the onSynchronization Started method which plays an animation to show that Tomdroid is now syncing data. When the main thread receives a SYNC_DONE, it calls onSynchronizationDone to stop the animation.

Tomdroid has a race condition which can lead to a crash. Suppose the user enters the ViewNote activity and clicks

```
// Schedule of clicking 'Back' after sync is done
...
 Lifecycle event: launch Main activity.
UI event: click ListView to show one note.
 Lifecycle event: pause Main activity.
 Lifecycle event: launch ViewNote activity.
 Lifecycle event: stop Main activity.
UI event: click sync button to sync notes.
Async event: SyncMessageHandler SYNC_START
Async event: SyncMessageHandler SYNC_PROGRESS
Async event: SyncMessageHandler SYNC_PROGRESS
Async event: SyncMessageHandler SYNC_DONE
UI event: click back button.
 Lifecycle event: pause ViewNote activity.
 Lifecycle event: resume Main activity.
 ...
```

```
// Schedule of clicking 'Back' before sync is done
...
 Lifecycle event: launch Main activity.
UI event: click ListView to show one note.
 Lifecycle event: pause Main activity.
 Lifecycle event: launch ViewNote activity.
 Lifecycle event: stop Main activity.
UI event: click sync button to sync notes.
Async event: SyncMessageHandler SYNC_START
Async event: SyncMessageHandler SYNC_PROGRESS
UI event: click back button.
 Lifecycle event: pause ViewNote activity.
 Lifecycle event: resume Main activity.
Async event: SyncMessageHandler SYNC_PROGRESS
Async event: SyncMessageHandler SYNC_DONE
CRASH: Null pointer exception
```

Figure 9: Event schedule of main thread in Tomdroid: normal execution (left) and race leading to crash (right).

the 'Sync' button, waiting until the sync operation is done, then clicks 'Back' to go back to the main activity. The sync operation usually completes quickly, thus in most cases the user clicks 'Back' after the sync has already completed. The left side of Figure 9 shows the event schedule from the main thread in this scenario.

However, in case the sync is slow, the user could click 'Back' before the sync is done. Then the 'Back' operation will trigger a switch from the ViewNote activity to the Main activity. When the SYNC_DONE message is processed in the Main activity's handler, the main thread invokes onSynchronizationDone; in that method, dot.getAnimation() returns null because the animation object is created and registered in ViewNote activity's handler. This will cause a null pointer exception that crashes the app; the event schedule is shown on the right side of Figure 9. Note that to faithfully reproduce this bug, the SYNC_DONE must be delivered after the activity transfer events are handled. In the next section, we show how we achieve this by replaying the event schedule.

5.2 Event Handling in Android

The Android platform is event-driven, with the AF orchestrating app control flow by invoking user-provided callbacks in response to user and system events. The AF provides support for events, threads, and synchronization. In Android, threads can communicate with each other in two ways: via shared memory or messages. The former is the same as in traditional Java applications, while the latter is more prevalent. In Android's concurrency model, every application process has a main thread (also called "UI thread"); only the main thread can access the GUI objects, to prevent non-responsive threads from blocking the GUI. To update the GUI, other (non-main) threads can send messages to the main thread. The main thread runs in a loop waiting for incoming messages, and processing them as they come.

Thread kinds. Android provides a Looper class that clients can use to attach message dispatching capabilities to threads. Each thread can attach at most one Looper object. The main thread has one Looper by default when the app launches. Inside the Looper object, there is a MessageQueue. If there is no message in the queue, the thread will block. Otherwise, the Looper removes the message at the front of the queue and processes it. Once the thread begins to process one message event, no other message processing routine can be started until the current one finishes. Hence event handling within each thread is atomic [15]. We will refer to any thread that has an attached Looper as a *looper thread*.

In addition to looper threads, Android supports two other kinds of threads: *binder threads*, created as thread pools when an app is launched and for inter-process; and *background threads*, which are the result of a regular thread fork().

Messages and Handlers. Android also provides a Handler class that allows threads to send and process messages, as well as runnable actions. Each Handler instance is associated with a single thread and that thread's Message Queue. There are two main uses for a Handler: (1) to schedule messages and runnables to be executed at some point in the future; and (2) to enqueue an action to be performed by a thread. After retrieving it from the message queue, the Looper dispatches the message to the corresponding Handler, which will either handle the message or run the messages's runnable action. Messages can be posted in a variety of ways: AtTime(time), i.e., post a message/action at a specific time, Delayed(delay), i.e., post a message/action after a specific time, or AtFrontOfQueue, i.e., post a message/action at the front of message queue. There are two kinds of messages: a post version, which allows Runnable objects to be enqueued and invoked by the message queue; and a sendMessage version which allows Message objects, containing data bundles (what, arg1, arg2 and Object) to be enqueued.

Event posting. Event posting is at the core of the Android programming model. Android events can be divided into two

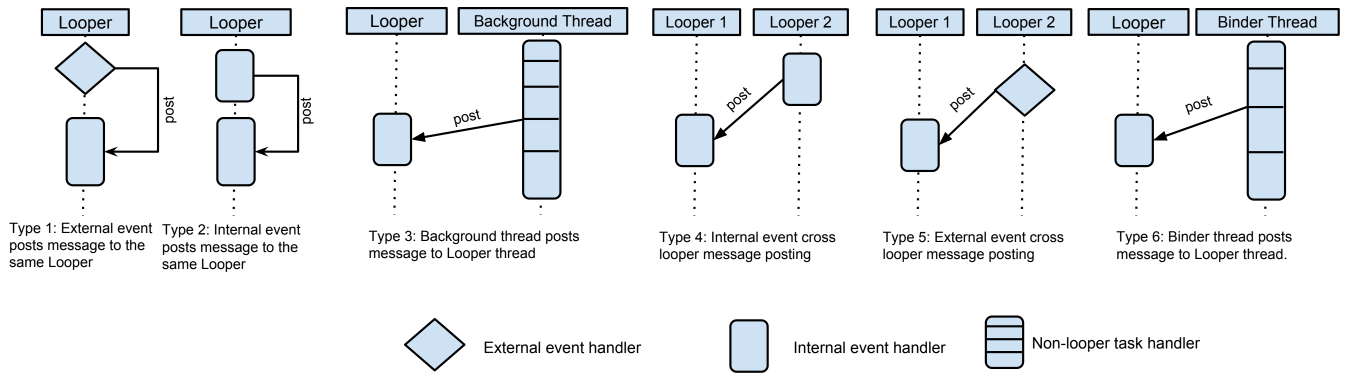

Looper
post
Type 1: External event posts message to the same Looper
Looper
post
Type 2: Internal event posts message to the same Looper
Looper
Background Thread
post
Type 3: Background thread posts message to Looper thread
Looper 1
Looper 2
post
Type 4: Internal event cross looper message posting
Looper 1
Looper 2
post
Type 5: External event cross looper message posting
Looper
Binder Thread
post
Type 6: Binder thread posts message to Looper thread.
External event handler
Internal event handler
Non-looper task handler

Figure 10: Event posting patterns in Android.

categories: *internal events* (messages or callbacks sent to a looper thread) and *external events* (caused by hardware interrupts). We have identified six different event posting types (Figure 10 illustrates them). We first describe each type then discuss how VALERA handles them.

Type 1: an external event posting a message to the same looper. For example, when the user touches a view widget on screen, the touchscreen generates a series of hardware interrupts (motion events). If the motion events are located in a View object that has a listener, the UI thread will post a listener callback to itself.

Type 2: an internal event posting a message to the same looper, i.e., looper posting to itself. One use of this scenario is for implementing timeouts. For example, if a looper wants to limit the time for processing a task, it can use an internal event to post a timeout message to the same looper—when the timeout message is due for processing, the task has reached its time processing limit.

Type 3: background worker thread posting a message to a looper thread. Since Android only allows the main thread to update the UI elements, it is common that background threads post messages or runnable callbacks to update the UI.

Type 4: cross-looper posting of internal events, e.g., when looper thread 1 posts a message to looper thread 2. Type 4 posting is very similar to Type 2, the difference being that in Type 4 the message is posted to another looper. Figure 2 (the Barcode Scanner app) contains one such event posting type: the main thread sends a message with the camera's frame data to the decoder thread.

Type 5: cross looper posting of external events, e.g., looper thread 1 posts a message to looper thread 2. This is similar to Type 4, but the event is external. As we show later (Table 4) this type of posting is rare—only 2 out of 50 examined apps use this posting type.

Type 6: binder thread posting a message to a looper thread. Android apps frequently communicate with background services such as the Activity Manager Service (AMS). For example, when the AMS decides to start an app's activity, it sends a "launch activity" IPC call to the binder of that app and the binder posts an internal event message to the main thread looper. In this scenario, the activity's onCreate lifecycle callback will be invoked.

During replay, the external events no longer come from the hardware, but rather from the replay log. Thus for Types 1 and 5, VALERA programmatically creates the external events based on the logged events and posts them to the looper as internal events. For internal event posting, i.e., Types 2, 3, and 4, each event is assigned a logical order based on which the scheduler will execute each event. (Algorithm 1 and Section 5.4 explain the details). Type 6 is a special case: since VALERA only controls the specific app instead of the whole system, other processes may send IPC events during record but not during replay or vice versa; we call these "missing" and "unrecorded" events, respectively, and in Section 5.4 we discuss how VALERA handles such cases.

In Section 6.2.2 we show the prevalence (number of events and event rate) for each app and each event type: in essence, Types 1 and 2 (self-posting) are the most prevalent, Types 3, 4, and 6 are less common, while Type 5 is rare.

Event-driven races. Since only the main thread is privileged to update the GUI, other threads send messages to the main thread to request GUI updates. The way these messages are handled can be non-deterministic. Although these messages will be put into the main thread's Looper message queue, their order of execution is not guaranteed and can result in violations of the happens-before relationship. This is the cause of *event-driven races* [15, 21]. We devised an algorithm, explained shortly, that can replay the event schedule in a deterministic manner, allowing us to reproduce and replay event-driven races.

Algorithm 1 Deterministic Event Order Replay

Input: Total Order of ScheduleList $ScheduleList$

```
1:  procedure LOOPER.LOOP
2:      while Looper not exit do
3:          $executable \leftarrow true$
4:          $msg \leftarrow$ CHECKPENDINGEVENT()
5:          if msg is null then
6:              $msg \leftarrow$ MESSAGEQUEUE.NEXT()
7:              $executable \leftarrow$ CHECKEXECUTABLE($msg$)
8:          end if
9:          execute msg if executable is true
10:     end while
11: end procedure
12: procedure CHECKPENDINGEVENT
13:     $msg \leftarrow$ PENDINGQUEUE.PEEK()
14:     if No msg from PendingQueue then
15:         return null
16:     end if
17:     if msg times out then        ▷ msg considered as missing
18:         Scheduler.turn++
19:         PENDINGQUEUE.REMOVE()
20:         return null
21:     end if
22:     if Scheduler.turn == event.order then
23:         Scheduler.turn++
24:         PENDINGQUEUE.REMOVE()
25:         return event.msg
26:     else                                  ▷ must wait its turn
27:         return null
28:     end if
29: end procedure
30: procedure CHECKEXECUTABLE(msg)
31:     for all $S_i$ in $ScheduleList$ do
32:         if $S_i$ match msg then
33:             if Scheduler.turn == $S_i$.order then
34:                 Scheduler.turn++
35:                 return true
36:             else
37:                 Add $S_i$ to pending queue
38:                 return false
39:             end if
40:         end if
41:     end for
42:     return true              ▷ let unrecorded event execute
43: end procedure
```

5.3 Recording the Event Schedule

VALERA records the event schedule by logging each message send and message processing operation into a trace file. Every time a thread sends a message, we record this operation as a <etype, eid, pid, tid, type, looper, caller > tuple. Here the etype indicates whether this is an internal or external event, eid denotes the unique event identifier, pid is the process id, tid is the thread id, type shows whether this event is a handler or a runnable action, looper is the target Looper object, and caller records the caller method that has created this event message. When the Looper begins to execute an event or finishes an event, VALERA also saves this information into the event schedule.

We found that certain types of events, e.g., Android's FrameHandler background events, do not affect the correctness of our event replay hence they are not included in the schedule. However, most events are relevant and are included, e.g., Activity's lifecycle events (Activity launch/stop/resume/etc.), user interaction events (e.g., touch or click a button), UI update events (e.g., resize/hide a view object) and the app's own messages.

5.4 Replaying the Event Schedule

We now present our algorithm for deterministically replaying events; in Figure 3 the algorithm is implemented in the *ScheduleReplayer*. We illustrate event replay on the Looper, though VALERA tracks and delivers other events as well.

Each event, either internal or external, is assigned a Lamport timestamp (logic order number [20]) in the schedule. At app startup time, we load the schedule into ScheduleReplayer's linked list. Loopers run an infinite loop waiting for events and dispatching them to the target handler. In the infinite loop, the looper checks with the ScheduleReplayer to see if there is any recorded event whose Lamport timestamp indicates it is next; if there is such an event, it is replayed, and the current (replay) logic order number is increased; otherwise it is saved in a pending queue, waiting for its turn to be executed. If the current event has not been recorded before, the ScheduleReplayer simply dispatches it.

The pseudocode is shown in Algorithm 1. The input is a schedule file indicating the execution order of the recorded events on this looper. Each event is assigned a logic order number. Every time the looper tries to fetch a new event, it first checks whether there is any event in the pending queue (line 4). An event is added to the pending queue if it matches the event in the schedule, but its turn to execute has not come yet. If there is no event in the pending queue, the looper fetches the event from its message queue as usual (line 6), then checks whether this event is executable or not.

In the CHECKPENDINGEVENT procedure, ScheduleReplayer first checks whether there is any event in the pending queue; if there is no event in this queue, the Looper will check its message queue. Otherwise, if an event exists, ScheduleReplayer checks the event's logic order number with the scheduler's current turn number. If they match (i.e., it is the pending event's turn to execute), the event is popped from the pending queue and returned (line 25). The scheduler's global turn number is increased to indicate next available executable event.

In the CHECKEXECUTABLE procedure, the input parameter is the message event from the looper. ScheduleReplayer iterates through the schedule list and matches the given event. An event is matched with the recorded schedule if the tuple described in Section 5.3 matches. If the event matches a schedule and the global turn matches its logic order, then the procedure returns true indicating that this event can execute. Otherwise, the event is added to the pending queue (line 37). Note that if the event does not match any recorded schedule, ScheduleReplayer returns true to allow this event to run (line 42).

Handling external event replay. During replay, external events are delivered from the recorded log, instead of the underlying hardware; VALERA implements a controller for this purpose. The controller is a background thread which continuously sends internal events to the looper. The internal event wraps the logged external event data with a what field indicating its type (e.g., touchscreen, sensor, GPS or camera event). The *ScheduleReplayer* knows the logic order number of every external event and executes it in its turn. This way, event non-determinism such as the example showed in Figure 2 is eliminated. After the current event is consumed, the controller will fire next.

Handling missing and unrecorded events. While VALERA records and replays the events coming into or going out of the subject app, it cannot control the behavior of other apps (for that, a system-wide approach would be needed). That might pose a problem if an external app sends an event during record but not during replay (or vice versa). For example, the system's Activity Manager Service (AMS) can send a $TRIM\_MEMORY$ event and invoke the app's onTrimMemory() callback if the AMS detects that the system is low on memory. Since VALERA does not control the AMS, the AMS might send a $TRIM\_MEMORY$ event during record but not during replay. To handle this situation, VALERA assigns a timeout value for each event in the schedule list. If the waiting time exceed the timeout limit (line 17), VALERA regards the event as missing and removes it from the schedule list. Conversely, the $TRIM\_MEMORY$ event could come during replay without appearing in the record log. VALERA handles this case by allowing execution of any unrecorded event (line 42). In both of these cases, VALERA logs the missing or unrecorded events, and the user can decide how they should be handled.

6. Evaluation

We now describe our experimental setup, then evaluate the effectiveness and efficiency of VALERA.

Environment. The smartphone we used for experiments was a Samsung Galaxy Nexus running Android version 4.3.0, Linux kernel version 3.0.31, on a dual core ARM Cortex-A9 CPU@1.2 GHz.

Setup. The experimental setup involved three scenarios, i.e., three different executions for each app: baseline, record, and replay. To establish a baseline, we first ran each app without involving VALERA at all, that is a human user ran the original, uninstrumented app; while the app was running, we were recording touchscreen events using RERAN (as RERAN has already been shown to have a low overhead, at most 1.1%). We call this scenario the *baseline execution*. Next, we ran the VALERA-instrumented versions of apps, with RERAN replaying, to inject the same touchscreen inputs as in the baseline run, while VALERA was recording—we call this the *record execution*. Finally, we ran a *replay execution* in which VALERA was set to replay mode (of course, no user interaction was necessary).

The user interacted with each app for about 70 seconds, since prior research has shown that the average app usage session lasts 71.56 seconds [6]. The user was exercising the relevant sensors for each app, e.g., scanning a barcode for the Barcode Scanner, Amazon Mobile and Walmart apps; playing a song externally so apps Shazam, Tune Wiki, or SoundCloud would attempt to recognize it; driving a car to record a navigation route for Waze, GPSNavig.&Maps, NavFreeUSA. To record intents, in apps Twitter, Instagram, PicsArt, Craigslist we took pictures by invoking Android's default Camera app (invoking the Camera app and returning the resulting picture is achieved via intents); for Google Translate, eBay, and Dictionary we used speech recognition which is also achieved via intents.

6.1 Effectiveness

We evaluate the effectiveness of our approach on two dimensions: (1) *Is* VALERA *capable of recording and replaying highly popular real-world apps?*, (2) *Is* VALERA *capable of recording and replaying high-throughput stream-oriented apps?*, and (3) *Is* VALERA *useful for reproducing event-driven races?*

Recording and replaying popular apps. To demonstrate the importance of recording and replaying sensor data, we ran VALERA on a wide range of popular apps. The apps were chosen based on the following criteria: (1) the app must use at least one of the sensor APIs described in Section 4.4, (2) apps must come from a variety of categories, such as business, productivity and tools. Note that these apps were downloaded directly from Google Play, the main Android app marketplace, which does not provide the app's source code.

Table 3 lists the 50 apps that we chose for our evaluation. To answer questions (1) and (2) above, we chose a mix of highly-popular apps and high-throughput stream apps.

The first column contains the app name, the second column indicates the popularity of the app, i.e., number of downloads (installs), while the "Streams" grouped columns show the streams used in that app.

For example, Sygic GPS, a popular navigation app with more than 10 million downloads, could be replayed with VALERA because VALERA replays GPS and network inputs. Similarly, popular apps with more than 50 million down-

Table 3: VALERA evaluation results: apps, popularity, streams and overhead.

App	**# Downloads**	**Streams**						**Time**					**Space**	
		GPS	Camera(fbuf.)	Camera(pic.)	Microphone	Network	Intent	Baseline Time	Record Time	Record Overhead	Replay Time	Replay Overhead	Log size	Log rate
	(millions)							(sec.)	(sec.)	(%)	(sec.)	(%)	(KB)	(KB/s)
Booking.com	10–50	●				●		74.38	75.26	1.18	75.86	1.99	1,246	16.56
GasBuddy*	10–50	●				●		83.47	84.02	0.65	84.77	1.56	206	2.45
Sygic: GPS N.&M.	10–50	●				●		94.11	97.46	3.56	98.03	4.17	6,538	68.08
TripAdvisor	10–50	●				●		69.39	71.34	2.81	71.56	3.13	1,328	18.62
Waze Social GPS	10–50	●				●		86.30	87.91	1.87	88.12	2.11	4,719	53.68
Yelp*	10–50	●				●		75.40	76.13	0.97	76.24	1.11	867	11.50
Flixster*	10–50	●				●		78.31	79.45	1.46	80.01	2.17	1,147	14.65
Hotels.com	5–10	●				●		84.50	85.17	0.79	85.66	1.37	1,563	18.35
Priceline	1–5	●				●		82.18	83.45	1.55	83.12	1.14	2,313	27.72
Scout GPS Navig.	1–5	●				●		66.39	68.11	2.59	68.47	3.13	5,312	77.99
Route 66 Maps	1–5	●				●		88.79	89.23	0.5	89.89	1.24	4,108	46.04
Restaurant Finder	1–5	●				●		71.46	72.18	1.01	73.45	2.78	918	12.72
GPSNavig.&Maps	0.5–1	●				●		72.19	73.58	1.93	73.45	1.75	5,177	71.71
Weather Whiskers	0.5–1	●				●		65.43	65.67	0.37	66.01	0.89	31	0.47
NavFreeUSA	0.1–0.5	●				●		63.81	64.37	0.88	65.11	2.03	75	1.17
Barcode Scanner	50–100		●			●		69.29	71.43	3.01	71.37	3.00	145,271	2,033.75
Google Goggles	10–50		●			●		73.10	74.12	1.40	74.87	2.42	106,121	1,451.72
Pudding Camera	10–50			●				61.26	61.38	0.20	61.91	1.06	7,488	121.99
Evernote*	10–50			●		●		74.12	75.00	1.19	75.19	1.44	2,317	30.89
Amazon Mobile*	10–50		●			●		85.31	86.77	1.71	87.31	2.34	41,071	473.33
QR Droid	10–50		●			●		79.46	81.55	2.63	82.39	3.69	114,812	1,407.87
CamScanner	10–50			●				62.01	62.76	1.21	62.87	1.39	2,612	41.62
CamCard Free	1–5			●				61.49	62.38	1.45	62.82	2.16	4,501	72.15
RedLaser Barcode	1–5		●			●		72.47	74.05	2.18	74.87	3.31	91,191	1,231.48
Walmart	1–5		●			●		85.65	86.78	1.32	86.86	1.41	157,129	1,810.66
Camera Zoom Fx	1–5			●		●		56.37	57.11	1.31	57.32	1.69	6,328	110.80
Horizon	1–5			●		●		64.39	65.71	2.05	66.10	2.66	5,413	82.38
Shazam	50–100				●	●		91.28	92.73	1.59	92.41	1.24	6,186	66.71
GO SMS Pro	50–100				●	●		58.12	59.33	2.08	59.87	3.01	101	1.70
Tune Wiki*	10–50				●	●		84.10	85.27	1.40	86.31	2.63	7,192	84.34
SoundCloud	10–50				●	●		64.38	65.87	2.31	66.12	2.70	1,206	18.31
Ringtone Maker	10–50				●			67.30	68.11	1.20	68.73	2.12	2,490	36.56
musiXmatch	5–10				●	●		73.28	74.01	0.99	74.35	1.46	651	8.80
Best Voice Changer	5–10				●			58.45	59.17	1.23	59.83	2.36	108	1.85
Smart Voice Rec.	5–10				●			51.39	53.12	3.37	53.81	4.71	97	1.89
PCM Recorder	1–5				●			46.28	48.12	3.98	48.73	5.23	2,418	52.25
RoboVox Lite	0.05–0.1				●	●		68.10	68.95	1.25	69.27	1.72	2,617	37.96
Diktofon	0.01–0.05				●	●		62.47	63.71	1.98	64.05	2.53	2,102	32.99
Twitter*	100–500					●	●	81.19	83.45	2.78	84.57	4.16	835	10.01
Google Translate*	100–500					●	●	69.36	70.48	1.61	71.02	2.39	49	0.70
Instagram*	100–500					●	●	55.47	55.98	0.92	56.13	1.19	872	15.58
PicsArt	100–500						●	64.21	64.32	0.17	64.55	0.53	12	0.19
eBay*	50–100					●	●	96.37	97.24	0.90	97.98	1.67	1,354	14.05
Bible*	10–50					●	●	73.91	74.63	0.97	75.38	1.99	871	11.67
Craigslist*	10–50					●	●	65.28	66.33	1.61	66.91	2.50	1,672	25.21
Dictionary*	10–50					●	●	58.31	59.23	1.58	59.88	2.69	164	2.77
GO SMS Pro Emoji	10–50					●	●	54.17	55.67	2.77	55.90	3.19	76	1.37
Weibo	5–10					●	●	90.46	91.87	1.56	92.44	2.19	3,182	34.64
1Weather	5–10					●	●	45.61	46.00	0.86	46.02	0.90	318	6.91
Weather	5–10					●	●	87.31	88.45	1.31	88.19	1.01	673	7.61
Mean								*71.36*	*72.49*	*1.01*	*72.92*	*1.02*	*15,101*	*208.32*

*=VALERA *can replay network, camera, GPS, microphone, intents, and schedule, while* RERAN *cannot*

loads, such as Barcode Scanner, Shazam, Google Translate, and Twitter could be replayed thanks to VALERA's support for replaying camera, microphone, and intent inputs.

Table 4: VALERA event streams: number of events and burst event rate (events/second); a '-' indicates that the app did not use that stream during our recording.

App	Touchscreen		GPS		Camera (fbuf.)		Camera (pic.)		Audio		Network		Intent		Scheduler events (types 1–6)						
	#Evs.	Rate	#Evs.	Rate	#Evs.	Rate	#Evs.	Rate	#Evs.	Rate	#Evs.	Rate	#Evs.	Rate	1	2	3	4	5	6	Rate
Booking.com	356	62	20	62	-	-	-	-	-	-	162	500	-	-	1,224	8,474	147	973	0	427	>1,000
Gasbuddy	606	58	5	58	-	-	-	-	-	-	256	500	-	-	1,943	713	82	17	0	369	>1,000
Sygic: GPS N.&M.	307	47	81	100	-	-	-	-	-	-	35	333	-	-	1,628	523	57	11	0	416	500
TripAdvisor	517	55	13	71	-	-	-	-	-	-	137	333	-	-	982	1,268	168	249	0	344	>1,000
Waze Social GPS	253	41	73	200	-	-	-	-	-	-	51	200	-	-	1,847	202	471	82	0	316	>1,000
Yelp*	493	62	11	166	-	-	-	-	-	-	93	333	-	-	1,764	897	763	65	0	311	>1,000
Flixster*	390	55	7	166	-	-	-	-	-	-	123	333	-	-	1,571	1,056	354	113	0	276	>1,000
Hotels.com	503	66	15	125	-	-	-	-	-	-	188	250	-	-	1,603	841	504	59	0	384	>1,000
Priceline	652	62	9	142	-	-	-	-	-	-	138	90	-	-	1,357	2,514	93	37	0	405	500
Scout GPS Navig.	207	62	78	90	-	-	-	-	-	-	36	71	-	-	1,438	298	539	0	0	361	>1,000
Route 66 Maps	197	52	94	111	-	-	-	-	-	-	46	125	-	-	1,883	3,617	836	18	0	314	>1,000
Restaurant Finder	468	47	12	100	-	-	-	-	-	-	127	333	-	-	1,695	817	596	47	0	325	>1,000
GPSNavig.&Maps	296	52	42	166	-	-	-	-	-	-	12	47	-	-	1,605	192	758	0	0	413	>1,000
Weather Whiskers	541	58	19	100	-	-	-	-	-	-	27	166	-	-	894	1,056	74	81	0	366	>1,000
NavFreeUSA	303	43	84	58	-	-	-	-	-	-	8	200	-	-	726	207	160	0	0	401	500
Barcode Scanner	64	41	-	-	57	5.95	-	-	-	-	5	333	-	-	364	189	31	107	19	144	>1,000
Google Goggles	51	166	-	-	52	5.95	-	-	-	-	16	166	-	-	307	216	15	52	0	158	>1,000
Pudding Camera	103	35	-	-	-	-	5	0.46	-	-	-	-	-	-	798	341	45	96	0	230	500
Evernote*	315	66	-	-	-	-	3	0.23	-	-	23	250	-	-	1,158	589	130	244	0	363	>1,000
Amazon Mobile*	590	58	-	-	32	12.50	-	-	-	-	64	500	-	-	2,005	775	194	9	0	181	>1,000
QR Droid	83	43	-	-	55	6.06	-	-	-	-	6	66	-	-	513	115	20	73	0	160	>1,000
CamScanner	119	45	-	-	-	-	2	0.01	-	-	-	-	-	-	439	312	37	52	0	118	500
CamCard Free	76	55	-	-	-	-	5	0.01	-	-	-	-	-	-	882	436	50	31	0	126	500
RedLaser Barcode	93	62	-	-	41	5.95	-	-	-	-	8	83	-	-	375	231	25	66	0	132	>1,000
Walmart	139	62	-	-	86	3.68	-	-	-	-	35	200	-	-	611	152	55	149	0	155	>1,000
Camera Zoom Fx	86	38	-	-	-	-	3	-	-	-	5	62	-	-	460	287	69	41	0	113	500
Horizon	73	55	-	-	-	-	2	-	-	-	13	83	-	-	512	319	79	53	0	146	>1,000
Shazam	27	71	-	-	-	-	-	-	560	71	33	333	-	-	224	6,617	125	272	0	255	>1,000
GO SMS Pro	18	71	-	-	-	-	-	-	68	52	14	333	-	-	128	117	25	12	0	212	>1,000
Tune Wiki*	86	83	-	-	-	-	-	-	386	66	36	200	-	-	386	1,253	267	88	0	172	>1,000
SoundCloud	93	66	-	-	-	-	-	-	419	90	41	142	-	-	513	420	86	77	0	269	500
Ringtone Maker	125	71	-	-	-	-	-	-	897	83	-	-	-	-	756	138	217	93	0	315	>1,000
musiXmatch	119	62	-	-	-	-	-	-	288	71	39	250	-	-	1,124	683	153	113	0	367	>1,000
Best Voice Changer	65	45	-	-	-	-	-	-	167	62	-	-	-	-	335	517	80	155	0	381	>1,000
Smart Voice Rec.	35	55	-	-	-	-	-	-	260	62	-	-	-	-	297	513	85	98	0	285	500
PCM Recorder	26	50	-	-	-	-	-	-	613	66	-	-	-	-	414	397	52	18	0	415	>1,000
RoboVox Lite	52	55	-	-	-	-	-	-	302	62	15	166	-	-	326	238	47	56	0	248	500
Diktofon	69	62	-	-	-	-	-	-	286	41	13	90	-	-	257	366	38	89	0	325	500
Twitter*	417	62	-	-	-	-	-	-	-	-	64	250	7	0.20	973	652	318	49	0	405	>1,000
Google Translate*	217	66	-	-	-	-	-	-	-	-	36	83	8	0.39	549	572	28	110	0	139	>1,000
Instagram*	536	71	-	-	-	-	-	-	-	-	12	166	12	0.17	1,839	416	150	217	0	315	>1,000
PicsArt	303	45	-	-	-	-	-	-	-	-	-	-	6	0.15	905	531	234	68	0	357	>1,000
eBay*	200	58	-	-	-	-	-	-	-	-	64	250	11	0.22	1,545	377	59	158	0	306	>1,000
Bible*	471	58	-	-	-	-	-	-	-	-	15	142	5	0.15	1,560	603	76	333	138	143	>1,000
Craigslist*	271	55	-	-	-	-	-	-	-	-	48	71	7	0.12	1,147	521	83	267	0	268	>1,000
Dictionary*	318	62	-	-	-	-	-	-	-	-	41	125	9	0.17	1,468	699	103	251	0	375	>1,000
GO SMS Pro Emoji	102	62	-	-	-	-	-	-	-	-	12	66	6	0.25	314	215	34	62	0	236	>1,000
Weibo	486	71	-	-	-	-	-	-	-	-	115	200	5	0.15	1,532	748	428	93	0	386	>1,000
1Weather	275	47	-	-	-	-	-	-	-	-	18	142	8	0.22	948	817	42	88	0	306	500
Weather	183	45	-	-	-	-	-	-	-	-	9	142	7	0.17	829	543	25	46	0	268	500

Several apps, e.g., Amazon Mobile*, are marked with an asterisk. For those apps, the most powerful Android record-and-replay system to date, RERAN, could only replay the GUI interaction, but not the high-level sensors, network, or events. For example, Amazon Mobile allows users to search by scanning a barcode or taking a picture of the item; RERAN cannot replay either of these actions. We discuss the rest of the columns in Section 6.2 where we examine VALERA's efficiency.

Thus we can conclude that VALERA is effective at recording-and-replaying widely popular Android apps, which are drawn from a variety of app categories and use a variety of sensors.

Reproducing event-driven race bugs. We used VALERA to reproduce event-driven races in several open source apps. The races, due to cross-posting of events and co-enabled events, were discovered by Maiya et al. [21]. Note that current Android record-and-replay tools cannot reproduce these races as they cannot preserve event ordering due to non-deterministic thread scheduling.

NPR News. While loading, this app checks the time of the last news list update. Concurrently, when new stories are added, a Runnable thread from NewsListAdapter.addMoreStories makes a post update call to the main thread which updates the lastUpdate variable in an asynchronous manner. These two events are non-deterministic and not ordered by a happens-before relationship. We reproduced the race by alternating the order and replaying the app in that specific order.

Anymemo. This app helps users learn new words in different languages using flash cards. After the user finishes one set of cards, the app creates a background thread which calculates the score and updates list of items to be shown on the UI. Usually the calculation is fast for a small working set and the updated result will show before the user switches back to the list view. However, the calculation and update operation are not ordered by happens-before. Hence if the user switches back before the calculation is done, the update operation cannot get the result and will throw a null pointer exception. In the latest version of this app, the authors have fixed the race bug by simply ignoring the race with a try-catch block. Although this fixes the crash, the UI view show the incorrect result.

My Tracks. This app exhibited a different type of race. Whenever users try to record their location, the app sends a bind request to the Binder thread, and the thread eventually binds it to a recording service. The problem is that when the service is registered with the request, the field providerUtils is updated. When the service thread gets destroyed this field is set to null. But there is no happens-before order between the field update and the service destruction. If the service is destroyed before the registration is executed, the recording process will attempt to dereference a null pointer. This is a harmful race which can be reproduced and replayed using VALERA.

Tomdroid. This race, which leads to a null pointer exception (Section 5.1), was also successfully reproduced.

6.2 Efficiency

To quantify the efficiency of VALERA we measured: (1) the time and space overhead that VALERA imposes when recording and replaying our test apps; and (2) the streaming requirements, in terms of events and event burst rate.

6.2.1 Time and Space Overhead

The "Time" and "Space" columns in Table 3 present the results of the measurements in the "Baseline", "Record", and "Replay" scenarios; for the record and replay scenarios, we also show the overhead, in percents, compared to the baseline. The last row shows geometric means computed across all 50 apps.

Based on the results in Table 3 we make several observations. First, note that record overhead is typically 1.01%, and replay overhead is typically 1.02% compared to the baseline, uninstrumented app run. This low overhead is critical for ensuring that sensor input, especially real-time streams, e.g., video/audio capture or complex touchscreen gestures, is delivered with precise timing so the recorded and replayed executions do not diverge. Second, note that several apps, e.g., Sygic, have overheads of around 4.17%: upon investigation, we found that the cause is record and replay of heavy network traffic. We also performed experiments without network replay (we omit the detailed results for brevity) and found the overhead to be much lower: at most 1.16%, typically 0.5%–1.0%. VALERA allows users to turn off network replay, e.g., if users wish to reduce overhead or let the app interact with "live" severs and services.

The "Space" grouped columns show the space overhead of our approach: the size of the recorded log (VALERA stores the log data on the phone's SD card), and the required log rate. As the table shows, apps that use the frame buffer have the largest space overhead, e.g., Barcode Scanner's log size is 145 MB, collected during an execution lasting 71.43 seconds. The large log size is due to the frame buffer from the camera sensor continuously sending back image data (in this case, it fills the 1.3 MB buffer every 250 milliseconds). Walmart, RedLaser Barcode, Google Goggles and QR Droid have large logs for the same reason. For the audio sensor experiments (e.g., PCM Recorder, Shazam), the log size is determined by user actions and the duration of the execution. Similarly, GPS-based apps (e.g., Navfree USA, GasBuddy, or TripAdvisor) have smaller logs, as saving GPS coordinates and network traffic takes less space than video or audio streams. The smallest-footprint logs are observed for intent replay—unlike GPS and other sensors, intents are small in size and sparse. We do not expect the logging to be an issue in practice as long as the log file fits onto the SD card: the log rate (last column, always less than 1.8 MB/s) is well within modern SD cards' throughout capabilities.

6.2.2 Streaming Rate

We now present a quantitative characterization of the streams that have to be replayed. In Table 4 we show the number of events for each sensor and the scheduler, as well as the *burst event rate*,[4] in events per second. The burst event rate indicates the burden on the record-and-replay system during periods of high load.

The table suggests a natural grouping of sensors by burst rate. First, the camera (in "taking pictures" mode) has the lowest burst rate, since the user must press the shutter which naturally limits the picture-taking rate. Similarly, the intent event rate is low since intents are usually triggered in response to user input. The camera (in "frame buffer" mode) has a low burst rate as well—at most 12.5 events per second; this is due to the app having to perform frame-buffer processing, namely image recognition, which is computationally intensive.

The touchscreen, GPS and audio have moderate burst rates, 41–200 events/second. The network's burst rate is higher, 207 events/second on average. Note, however, that the touchscreen and network are used by most apps, so their burst rates might need to be accommodated simultaneously.

[4] The event rate fluctuates during an execution. For each sensor's events, we identified the burst periods as the top 25% intervals by event rate, and took the median event rate of those periods.

Finally, the scheduler has the highest burst rate, typically in excess of 1,000 events/second (our timer had millisecond granularity, so for events separated by less than 1 millisecond, we rounded up to the ceiling value of 1 millisecond). More specifically, Type-1 and Type-2 events were the most prevalent, indicating that most messages are self-postings (the sender and receiver thread are the same, per Section 5.2).

Thus we can conclude that, with respect to our chosen apps, VALERA is efficient at record-and-replay in terms time overhead, space overhead, and supporting high-rate burst events.

7. Related Work

Record-and-replay has been widely studied and implemented on various platforms.

On the smartphone platform, the most powerful, and most directly related effort is our prior system RERAN [19], which has been used to record and replay GUI gestures in 86 out of the Top-100 most popular Android apps on Google Play. RERAN does not require app instrumentation (hence it can handle gesture nondeterminism in apps that perform GUI rendering in native code, such as Angry Birds) or AF changes. Mosaic [13] extends RERAN with support for device-independent replay of GUI events (note that our approach is device-independent as well). Mosaic has low overhead, typically less than 0.2%, and has replayed GUI events in 45 popular apps from Google Play. However, RERAN and Mosaic have several limitations: they do not support critical functionality (network, camera, microphone, or GPS), required by many apps; they do not permit record-and-replay of API calls or event schedules; their record-and-replay infrastructure is manual, which makes it hard to modify or extend to other sensors.

Android test automation tools such as Android Guitar [1, 5], Robotium [11], or Troyd [17] offer some support for automating GUI interaction, but require developers to extract a GUI model from the app and manually write test scripts to emulate user gestures. In addition to the manual effort required to write scripts, these tools do not support replay for sensors or schedules.

On non-smartphone platforms, record-and-replay tools have a wide range of applications: intrusion analysis [10], bug reproducing [22], debugging [26], etc. Hardware-based [22, 29] and virtual machine-based [10, 25] replay tools are often regarded as whole-system replay. Recording at this low level, e.g., memory access order, thread scheduling, allows them to eliminate all non-determinism. However, these approaches require special hardware support or virtual machine instrumentation which might be prohibitive on current commodity smartphones.

Library-based approaches [9, 16, 24, 30] record the nondeterminism interaction between the program libraries and underlying operating system with a fixed interface. R2 [12] extends them by allowing developers to choose which kinds of interfaces they want to replay by a simple annotation specification language. VALERA borrows this idea from R2 (which targets the Windows kernel API) but applies it to sensor-rich event-based Android.

CAFA [15] and Droidracer [21] are dynamic race detection tools for Android. Our work is complementary and, we hope, useful to the authors and users of these tools, for the following reason: these tools report a possible race, but cannot capture and replay an execution that deterministically reproduces the race. With VALERA, once a race is captured, it will be reproduced.

8. Conclusions

We have presented VALERA, an approach and tool for versatile, low-overhead, record-and-replay of Android apps. VALERA is based on the key observation that sensor inputs, network activity and event schedules play a fundamental role in the construction and execution of smartphone apps, hence recording and replaying these two categories is sufficient for achieving high-accuracy replay. Experiments with using VALERA on popular apps from Google Play, as well as replaying event race bugs, show that our approach is effective, efficient, and widely applicable. VALERA's accuracy and low runtime overhead make it suitable as a platform for applications such as profiling, monitoring, debugging, testing, or dynamic analysis. We believe that stream-oriented replay could be applied in other contexts besides smartphones, e.g., replay of time-sensitive or stream-processing programs on desktop/server platforms.

Acknowledgments

We thank the anonymous referees for their helpful comments. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1064646.

References

[1] D. Amalfitano, A. Fasolino, S. Carmine, A. Memon, and P. Tramontana. Using gui ripping for automated testing of android applications. In *ASE'12*.

[2] Android Developers. UI/Application Exerciser Monkey, . `http://developer.android.com/tools/help/monkey.html`.

[3] Android Developers. MonkeyRunner, . `http://developer.android.com/guide/developing/tools/monkeyrunner_concepts.html`.

[4] Android Developers. SharedPreferences, . `https://developer.android.com/reference/android/content/SharedPreferences.html`.

[5] Atif Memon. GUITAR, August 2012. `guitar.sourceforge.net/`.

[6] M. Böhmer, B. Hecht, J. Schöning, A. Krüger, and G. Bauer. Falling asleep with angry birds, facebook and kindle: a large scale study on mobile application usage. In *MobileHCI'11*.

[7] CNET. Android dominates 81 percent of world smartphone market, November 2013. `http://www.cnet.com/news/android-dominates-81-percent-of-world-smartphone-market/`.

[8] CNET. Android beat Apple in tablet sales last year – Gartner, March 2014. `http://www.cnet.com/news/android-beat-apple-in-tablet-sales-last-year-gartner/`.

[9] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Jockey: a user-space library for record-replay debugging. In USENIX ATC'06.

[10] 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*.

[11] Google Code. Robotium, August 2012. `http://code.google.com/p/robotium/`.

[12] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *OSDI'08*.

[13] M. Halpern, Y. Zhu, and V. J. Reddi. Mosaic: Cross-platform user-interaction record and replay for the fragmente d android ecosystem. In *ISPASS'15*.

[14] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *CCS'11*.

[15] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn. Race detection for event-driven mobile applications. In *PLDI'14*.

[16] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jrapture: A capture/replay tool for observation-based testing. In *ISSTA'00*.

[17] Jinseong Jeon and Jeffrey S. Foster. Troyd, January 2013. `https://github.com/plum-umd/troyd`.

[18] Jinseong Jeon and Kristopher Micinski and Jeffrey S. Foster. Redexer. `http://www.cs.umd.edu/projects/PL/redexer/index.html`.

[19] L. Gomez, I. Neamtiu, T.Azim, and T. Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *ICSE '13*.

[20] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM'78*, 21(7):558–565.

[21] P. Maiya, A. Kanade, and R. Majumdar. Race detection for android applications. In *PLDI'14*.

[22] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA '05*.

[23] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs. In *CGO '10*.

[24] M. Ronsse and K. D. Bosschere. Recplay: A fully integrated proctical record/replay system.

[25] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX ATC'05*.

[26] 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 ATC'04*.

[27] T.Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *OOPSLA'13*.

[28] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. ProfileDroid: Multi-layer Profiling of Android Applications. In *MobiCom'12*.

[29] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA '03*, pages 122–135.

[30] Y. Saito. Jockey: a user-space library for record-replay debugging. In *AADEBUG'05*.