Record and Replay

Aveek Karmakar, Satyaditya Munipalle, Taeyeon Ki
(aveekkar, satyadit, tki) at buffalo [dot] edu
Department of Computer Science and Engineering
University at Buffalo, The State University of New York

Abstract

Bugs that occur for a single time are hard to reproduce due to the change in the application environment and different use cases. We present a record and replay tool for Android that records the events occurring in the Android application from the user space to a persistent storage (log file). Using the replay system we are able to faithfully replay the saved execution of the recorded application execution. The log files contains data which represent the timestamp and event related information .The replay can be done without requiring the specific APK, libraries or support data. Using this tool we can find the exception paths in the framework and find the source of the exception.

Keywords- Record&replay, Debugging, Soot, Android, AOSP

1 Introduction

There are various types of errors that can occur while a program is being run. These software bugs can be hard to produce due to differences in the application environments, the data being fed as an input to the process or due to some non-determinism. Bug reports and stack traces are inadequate in some scenarios when the application environment in not consistent. Also in order to generate the bug report we have to run the application again. This can be time consuming or there might be a cost that can be associated while performing this kind of repeated analysis. Also for repeated analysis there might be a requirement for a human user to operate the program, which might not be feasible in most of the cases. To avoid the above mentioned scenarios and many more, a record and replay system is necessary for effective debugging purposes. A Record and Replay is a powerful technique for debugging bugs, which are difficult to reproduce with re-execution. By directly recording the execution of the application at selected points and capturing the bug as it occurs, the burden of repeated testing to reproduce the bug is avoided. In the record and replay system we developed, the logging statements are inserted into the Android application during the recording phase. We tested our approach with a custom application we built for this purpose. To simplify the instrumentation and to enable it across any Android appliaction we performed automatic instrumentation using Soot[3], a Java profiling framework. Soot supports reading and writing DALVIK bytecode using two modules - Dexpler and toDex. When an APK is given we profile the application using Soot by analysing the Jimple three address byte code generated by Soot. Despite its potential for simplifying bug reproduction and debugging, the fundamental limitation of previous record-replay approaches is that they require the availability or replication of the production application environment during replay. We simplified our approach by not considering the intricacies involved with various Android platforms, but rather, this tool acts as a simple record and replay tool provided the environments are same while recording and replaying .The replay system works by reading from the log files generated by the recording module and replays the previous application execution faithfully.

2 Motivation

The central theme for this project is to create a record and replay system that records from the user space and the replay is performed from the data generated from the user space .The benefit of having such a system is that reply is not dependent on the source APK . The motivation for this work is derived from various record and replay systems present in the market mainly AppInsight [1] and RERAN [2]. Current systems that exist for Android record the data generated at the lower levels of the frameworks. RERAN[2] is a record and replay tool that captures low-level event stream, GUI and sensors except GPS and Camera, record events in log, replay the events from log. The logging mechanism we used in our project is inspired from the AppInsight paper. Our application records from the user level where as RERANs logging is done at the lower levels of Android.

3 Record

The recorder package consists of the Java classes necessary for logging in the user space of the Android application. Each class in the recorder package will be utilized for case-specific recording. The logger class consists of methods which write the data onto persistent data files which will be used later for the replay. Generally these data files (separate for each use case) are saved in the phone and are uploaded to the remote server upon the completion of recording. Figure 1 shows the various use cases that can be recorded using our recording module are.

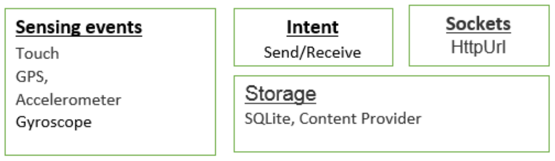


Figure 1: Use cases for recording.

Figure 2 shows the idea behind logging.

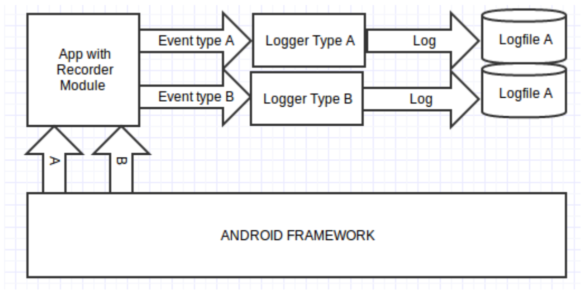


Figure 2: Recording overview.

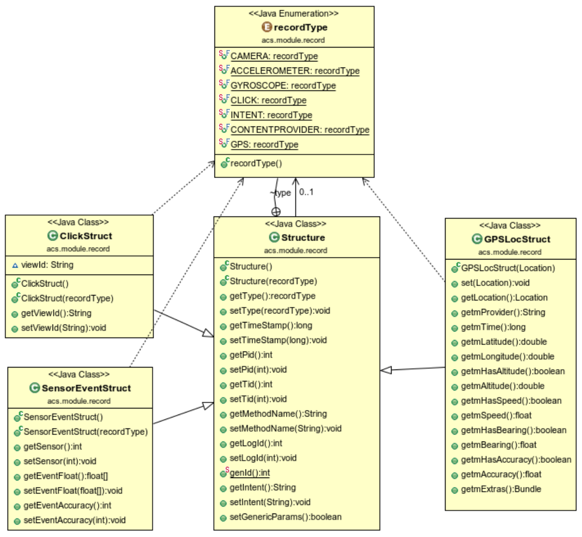


Figure 3: Hierarchy diagram for recorder package.

Figure 3 describes that the hierarchy diagram gives an overview of the package.

The logging or recording leverages on the idea that Android platform is event driven. Every event is associated with the event callback. The event driven programming follows the Observer Pattern.

3.1 Observer Pattern

The observer pattern is designed to allow state changes in a single object to be propagated to a set of Observer objects. The observed object has a set of listeners registered with it. When the observed object changes, the listeners are notified of the change.

In Android, there is more than one way to intercept the events from a user's interaction with the application. When considering events within the user interface, the approach is to capture the events from the specific View object that the user interacts with. The View class provides the means to do so. There are several public callback methods that are useful for the UI events. These methods are called by the Android framework when the respective action occurs on that object. For instance, when a View (such as a Button) is touched, the onTouchEvent() method is called on that object. When the user either touches the item (in touch mode), or focuses upon the item with the navigation-keys and presses the "enter" key, or presses down on the trackball the onClick() from View.OnClickListener is called. For the replay of the clicks the main component is to record the timing at which the user interaction has occurred. At each user click if we save the button name as one of the parameters to the clickStruct class method along with the timestamp of the event occurrence we can recreate the user click event during the replay phase.

3.2 Event Call Back for sensor events

When the listeners for the sensor events are registered in the program, the Android framework calls the onSensorChanged method whenever the sensor reports a new value. When the system invokes the onSensorChangedmethod() it provides a SensorEvent object. A SensorEvent object contains information about the new sensor data, including the accuracy of the data, the sensor that generated the data, the timestamp at which the data was generated, and the new data that the sensor recorded. In order to replay the sensor reading, it is sufficient to record the event float values extracted from this object along with the event timestamps and the type of the sensor reading in the log.

3.3 Ping pong buffering

The logging mechanism implements the Ping-Pong buffering technique. The logs are buffered on to a concurrent queue. Each log record is assigned a unique identifier. The process of assigning this identifier should be synchronized. At any point of time a single buffer is in use. In our application if the queue or buffer size reaches a particular threshold the toggling of queue is done. The new buffer should be empty before it is set as the current buffer. The data from the buffer that is not currently in use will be used for writing to the persistent storage. There are separate loggers present for each use case in our application. All the loggers follow the same ping-pong buffering technique as described above. The log files generated are uploaded to a remote server and will be used by the replay service.

3.4 Automatic Logging

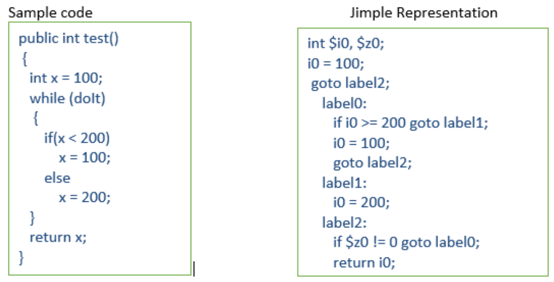
In order to profile any kind of application the profiling has to be done at the byte code level. This instrumentation at bytecode level can be done by using Soot, a Java instrumentation framework.

3.5 Soot

Soot is a free compiler infrastructure, written in Java (LGPL) it was originally designed to analyze and transform Java bytecode. This framework has been developed by the Sable Research Group at McGill University. Soot provides four intermediate representations for analyzing and transforming Java bytecode: Baf (Bytecode (stack-based): a streamlined representation of bytecode which is simple to manipulate. Jimple (Java's simple, typed, 3-addr): a typed 3-address intermediate representation suitable for optimization. Shimple: an SSA(static single assignment) variation of Jimple. Grimp: an aggregated version of Jimple suitable for decompilation and code inspection. More details about Soot can be found in this paper [4]. In this project we relied on Jimple representation of the bytecode generated using Soot for logging purposes as Jimple is easy to understand and gives an idea of how the three address instructions are created. When the APK file is passed to the Soot framework, one of the four IR(intermediate representations) are generated depending on the parameters specified in the SootMain class,.

3.6 Profiling Android application using Soot

In order to profile Android apps,the first step using Soot analysis is to find all the entry points. Soot has inbuilt


Sample code
Jimple Representation

```
public int test()
 {
  int x = 100;
  while (doIt)
   {
     if(x < 200)
        x = 100;
     else
        x = 200;
   }
  return x;
}
```

```
int $i0, $z0;
i0 = 100;
 goto label2;
   label0:
     if i0 >= 200 goto label1;
     i0 = 100;
     goto label2;
   label1:
     i0 = 200;
   label2:
     if $z0 != 0 goto label0;
     return i0;
```

Figure 4: Sample jimple code.

methods for finding all the entry points. The main activity can be found by parsing the Android manifest file as Soot does not provide any mechanism for finding the main activity. The entire instrumentation is done by overriding the internal transform method present in body transformer class in the Soot framework. In Soot method the body is represented as a chain of units. A unit iterator has to be created in order to looped through the units present in the method body. When the appropriate instrumentation points are found the profiling instructions are added. Soot provides snapshot iterator that helps us add new units to the chain. The reason for using the snapshot iterator is that it does not throw concurrent modification exception, which happens with the Java iterators.

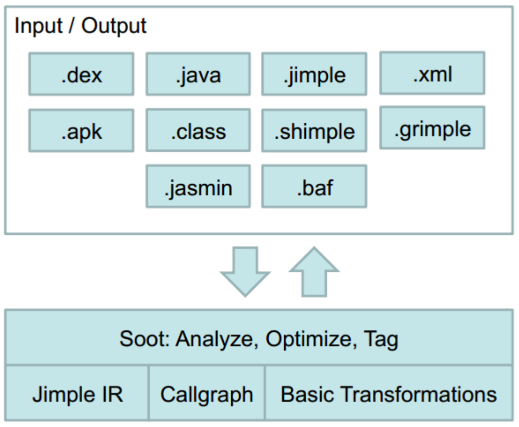
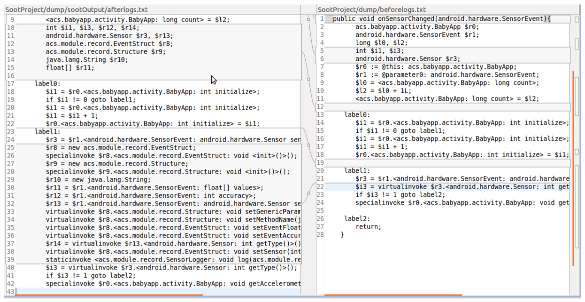


Figure 5: Soot overview.

A new APK will be generated after the instrumentation is done .The new APK needs to signed by a jarsigner in order to be installed as an application to the phone. Figure 6 shows the sample code for onSensorMethod() after instrumentation.



SootProject/dump/sootOutput/afterlogs.txt

```
        <acs.babyapp.activity.BabyApp: long count> = $l2;
        int $i1, $i3, $r12, $r14;
        android.hardware.Sensor $r3, $r13;
        acs.module.record.EventStruct $r8;
        acs.module.record.Structure $r9;
        java.lang.String $r10;
        float[] $r11;

     label0:
        $i1 = $r0.<acs.babyapp.activity.BabyApp: int initialize>;
        if $i1 != 0 goto label1;
        $i1 = $r0.<acs.babyapp.activity.BabyApp: int initialize>;
        $i1 = $i1 + 1;
        $r0.<acs.babyapp.activity.BabyApp: int initialize> = $i1;
     label1:
        $r3 = $r1.<android.hardware.SensorEvent: android.hardware.Sensor sen
        $r8 = new acs.module.record.EventStruct;
        specialinvoke $r8.<acs.module.record.EventStruct: void <init>()>();
        $r9 = new acs.module.record.Structure;
        specialinvoke $r9.<acs.module.record.Structure: void <init>()>();
        $r10 = new java.lang.String;
        $r11 = $r1.<android.hardware.SensorEvent: float[] values>;
        $r12 = $r1.<android.hardware.SensorEvent: int accuracy>;
        $r13 = $r1.<android.hardware.SensorEvent: android.hardware.Sensor se
        virtualinvoke $r8.<acs.module.record.Structure: void setGenericParam
        virtualinvoke $r8.<acs.module.record.Structure: void setMethodName(j
        virtualinvoke $r8.<acs.module.record.EventStruct: void setEventFloat
        virtualinvoke $r8.<acs.module.record.EventStruct: void setEventAccur
        $r14 = virtualinvoke $r13.<android.hardware.Sensor: int getType()>()
        virtualinvoke $r8.<acs.module.record.EventStruct: void setSensor(int
        staticinvoke <acs.module.record.SensorLogger: void log(acs.module.re
        $i3 = virtualinvoke $r3.<android.hardware.Sensor: int getType()>();
        if $i3 != 1 goto label2;
        specialinvoke $r0.<acs.babyapp.activity.BabyApp: void getAcceleromet
```

SootProject/dump/beforelogs.txt

```
public void onSensorChanged(android.hardware.SensorEvent){
        acs.babyapp.activity.BabyApp $r0;
        android.hardware.SensorEvent $r1;
        long $l0, $l2;
        int $i1, $i3;
        android.hardware.Sensor $r3;
        $r0 := @this: acs.babyapp.activity.BabyApp;
        $r1 := @parameter0: android.hardware.SensorEvent;
        $l0 = <acs.babyapp.activity.BabyApp: long count>;
        $l2 = $l0 + 1L;
        <acs.babyapp.activity.BabyApp: long count> = $l2;

     label0:
        $i1 = $r0.<acs.babyapp.activity.BabyApp: int initialize>;
        if $i1 != 0 goto label1;
        $i1 = $r0.<acs.babyapp.activity.BabyApp: int initialize>;
        $i1 = $i1 + 1;
        $r0.<acs.babyapp.activity.BabyApp: int initialize> = $i1;

     label1:
        $r3 = $r1.<android.hardware.SensorEvent: android.hardware
        $i3 = virtualinvoke $r3.<android.hardware.Sensor: int get
        if $i3 != 1 goto label2;
        specialinvoke $r0.<acs.babyapp.activity.BabyApp: void get

     label2:
        return;
    }
```

Figure 6: The sample code for onSensorMethod().

4 Replay

4.1 The Main Idea

The idea is to recreate the events from the given data in the log files. We explored broadly two high level ideas to have a mechanism in place which will recreate the respective event objects from the log File and call the callback methods implemented by the user in an exact sequence in which they actually happened in the record-run of the application in question.

We will take the examples of SensorEvent class and the Location (GPS) class in the framework. The data components of SensorEvent class that is both sufficient and necessary to recreate a SensorEvent object are as follows.

Array of float values containing the actual sensor data (3-4 values depending on the sensor), the Sensor object associated with a particular sensor, the accuracy of the event and the timestamp of the event. We have created a wrapper class around the System SensorEvent Class so that we can recreate the object back. We get an instance of the Sensor object from the SensorManagerService by querying with the sensor type recorded previously. The data components of Location are as follows - GPS Provider
- Latitude
- Longitude
- Altitude
- Speed
- Bearing
- Timestamp
- Accuracy
- Bundle Extras

We are creating back the Location objects from the data. The Bundle can be read as a byte array and un-marshalled into a parcel and the Bundle thus read back. Now that we have a mechanism to recreate Event objects from log files, we will now look at two mechanisms to create a faithful replay.

4.2 Replaying from the User Space

We have an Iterator to read logs one at a time from the Log Files. We have one background service for every kind of sensor. The services are started by the App to be replayed, on creation, and the start time of the record run is passed to the services. The services computes the time gap starting from the start time and then onwards between the events from the recorded timestamps and broadcasts the log to the activity which in turn calls the necessary callback methods. The idea behind having multiple services is so that different events can be generated in parallel, while still keeping the relative time differences between events intact from the record run. This potentially might not adhere to inter-event happened before relations. But this potential problem could easily be solved if instead of having separate loggers for every event type in the recorded App, we only maintain one event queue and one single log file for all the events. The happened before is evident from the recorded timestamps and in addition to that, we can synchronize the generation of log id to guarantee preservation of happened before relations across different types of events. In such a scenario only one service in the replay framework will suffice. In fact the first implementation that we had of the replay mechanism from the user space did just that. One more component that needs to be mentioned is the identification of non-deterministic framework specific calls that the Appmight make. e.g getLastKnownLocation() in case of GPS. For that, we have a token that is broadcasted and received by concerned classes like GPSLocationProvider and LocationServiceManager. on receipt, they cut off 1) the sensor readings between the framework and HAL interface (e.g reportLocation() in GPSLocationProvider) and 2) when system calls such as getlastKnownLocation() are made, they return the recorded concerned object.

4.3 Replaying from the framework

The record&replay system we have designed is primarily to cover for non-determinism in cyclic debugging of event driven programming models. So while the above approach in theory should be sufficient to do exactly that, one concern might be that we are not accounting for the things that the framework does while it gets an event across from the HAL to the user space. So we also did implement a system which replayed GPS data from the framework. The idea here is to have a master service which could read the logs and maintain the relative time differences between events. And this service could trigger the method which the HAL under normal circumstances would trigger. Like before we cut off the data generation by HAL when the replay mode starts. So in this model of replay, we maintain the happened before relations and relative time difference between events through the service and a custom method is in place instead of the method which usually the HAL calls.This implementation needed a little more research as it would be necessary to determine whether the framework, while transporting the event, itself depends on other environmental conditions of the system at hand and in that case we also have to look whether those conditions can be recorded from the user space as otherwise it will not be a faithful replay.

5 Conclusion

The record&replay framework discussed here provides a great platform for cyclic debugging of bugs like data races, deadlocks, and other synchronization issues related to event driven programming. The replay framework can be easily extended to do a step by step replay of the App, only preserving the happened before relations.It is also possible to modify the replay framework to set breakpoints in the log file according to time elapsed and replay till that point. Exceptions could be caught using Android UncaughtExceptionHandler and a precise Exception path could be generated either showing an event graph or call graph. The log files themselves could be used to do many kinds of analysis, for example finding bottlenecks like downloading Web content through http URL vs doing some heavy lifting in a background service. Recording from the application space has some limitations. Like for example, if the application spawns multiple threads which share a common resource, the thread execution events like creation, entering monitor to get a lock, thread termination cannot be captured from the application and thus will not be reproducible during replay.

Acknowledgments

We would like to take this opportunity to thank our professors Dr. Steve Ko and Dr. Karthik Dantu for coming upon this idea and for guiding us through out the process of implementing this project.

References

[1] Lenin Ravindranath,Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, Shahin Shayandeh ”AppInsight: Mobile AppPerformance Monitoring in the Wild ,” *Microsoft Research.*

[2] Lorenzo Gomez,Iulian Neamtiu,Tanzirul Azim,Todd Millstein, ”RERAN: Timing- and Touch-Sensitive Record and Replay for Android ,”

[3] Patrick Lam, Eric Bodden,Ondrej Lhotak and Laurie Hendrenx ,”The Soot framework for Java program analysis: a retrospective,” *University of Waterloo,Technische Universitat Darmstadt , McGill University.*

[4] Alexandre Bartel, Jacques Klein, Yves Le Traon, Martin Monperrus, ”Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot” .

[5] Raja Vallee-Rai Laurie J.Hendren, ” jimple: Simplifying Java Bytecode for Analyses and Transformations.