AndroidRR

Logan Donovan - lrd2127@columbia.edu, Dmitriy Gromov - dg2720@columbia.edu, Riley Spahn - riley@cs.columbia.edu, Deepika Tunikoju - dt2417@columbia.edu

Abstract—We have developed a user interface driven record and replay system for the Android mobile operating system. By modifying the Android View application programming interface we are able to log interactions within the system to be replayed at a later time using a tool built on top of Googles provided MonkeyRunner framework. This type of user interface replay is useful for testing and debugging as well as more novel applications like system auditing. In this paper we discuss the implementation of the system, tests that we ran to evaluate its effectiveness, challenges we faced, and how we would continue to build the system going forward.

1 Introduction

The ability to record and replay actions done on Android device seems incredibly useful. Some attempts at implementing these mechanisms have already been made. However, the existing implementations seem to be used primarily for application testing so they are limited to one single application and function on a very high level. In this paper, we propose a record and replay system implemented as addition to the Android OS and will work across applications and allow its users to keep track of input across all parts of the Android UI.

Since the course had a lot to do with security and auditing one use that we envision for record and replay is multi-user auditing on one device. Although our current implementation does not support it, we can have the record function record on a per user level instead of on a system level. To implement this, we could create a log-in feature and create separate logs for each user. One possible use for this would be while using an android mobile device for course instruction. Often times students will try to tamper with the device in ways that they should not be and safeguards against that do not always succeed. In this way, if a device is ever broken, an instructor could go through and see which of the students using the device actually tampered with it. A even simpler scenario could be if you leave your device on a table and someone tampers with your device and you are unsure what happened. Replaying the recorded actions would reveal to you what was done.

Another useful application would be for testing and debugging. One of the hardest parts about testing applications is having realistic workloads. It is possible to get some users to sit down and actually use the application in question but once they are done you only have the results of their time with you. Using the record mechanism, it would be possible to record the live user actions and then play them back as many times as you need to or even tweak them slightly if the tests demand it. Furthermore, it would be a very useful feature for scenarios like alpha/beta testing. It is much more useful to have the sequence of the events leading up to an error along with an error than just the error.

Finally, there are many studies that are interested in how mobile devices are used. That is, which applications are used, how often, and how. Having the ability to record user events can help such studies and improve the quality of the operating system and software by revealing to the developers how their products are being used.

2 Related Works

2.1 MonkeyTalk

MonkeyTalk is a well developed system that supports record, replay, and test automation across different technologies and frameworks including Android. [3] The system allows you to record and replay user inputs, create automated user tests or run interactive tests through their IDE which is built on top of the popular eclipse IDE. Using MonkeyTalk, one can connect to a virtual or physical device running Android and run their tests on it. From there, most of the user interactions can be recorded and are converted to their specific format including detailed information about the events that occurred and the elements they affected. MonkeyTalk also provides a javascript api which allows you to override event handlers to record custom messages.

While we believe that it is extremely useful to have an abstraction layer that allows cross system utilization, for this project we tried to create a mechanism which would be able to work on a deeper layer of android. The paper further describes how we sacrifice some detail in recording for the trade off of being able to record user interaction at any point and without any addition software besides the modified android system.

2.2 Robotium

Robotium is an Android UI automation framework designed to make programmatic simulation of user actions

on Android devices very simple. [4] It does not support any record or replay functionality as is but provides several mechanisms to ensure sanity in actions taken. For example, when typing into a textbox or clicking on a button it grants its user the ability to check that the desired elements exist and that their data or attributes are correct. We believe that the assertions implemented in Robotium would make our replay mechanism more accurate and plan to support Robotium or similar functionality in further work.

2.3 Deterministic Replay

A lot of research is being carried out in the area of UI testing for mobile apps, many of which involve record and replay. [1] Jason Flinn and Z. Morley Mao from the University of Michigan published a paper [1] about the applicability of deterministic replay for UI testing for mobile devices. Through their research they aimed at studying the challenges posed by implementing replay on phones. They also explored the benefits of replay, especially when it is performed remotely on cloud or cloudlet.

2.4 GUITAR

GUITAR (Graphical User Interface Testing frAmewoRk) is a test generation and automation framework that can be applied to GUIs of many kinds. [2] It has been extended to android applications as Android GUITAR. Android-Guitar is intended to simplify the testing process of GUIs on the Android platform by invoking GUITAR. A plugin is being developed that allows the GUITAR Ripper and Replayer to communicate with an Android application running on an Android emulator. This plugin is expected to facilitate automated and comprehensive testing of Android GUIs, as well as increase the breadth of GUITAR functionality.

3 Record Architecture/Implementation Details

3.1 Android Application Architecture

Android applications can be segmented into four different types of components: activities, services, content providers, and broadcast receivers. We do not concern ourselves with services, content providers or broadcast receivers because they do not have any user facing assets and only focus on the activities. Each activity represents one screen with which the user will interact and are used to encapsulate application state. As the user progresses through an application and series of activities the previous activities are saved on the activity stack allowing the user to navigate back to previous activities in the application. The activity abstracts away most of the front end application state and later we will explore how the activity stack may provide us with a method for mitigating the application state's impact on user interface divergence during replay. The user interface for each activity is implemented through hierarchies of View objects that represent a single area on the screen and ViewGroups that organize the various views onto the screen. These view objects contain abstractions for the different ways that a user can interact with an application such as touch events on the screen, hardware key events and layout changes. Because the view is a central object for interacting with the user interface this makes it an ideal place for us to record interactions. Another option would be for us insert a code to record interactions at the native handlers layer. This may have provided a performance boost but we would have lost the semantics that the View layer provides. The View object also uses handlers to differentiate between different types of events such as touch and drag events.

3.2 Record System Architecture

The systems recording component has few simple requirements. First, it must be able to log events without degrading the users experience. It must be able to with enough semantic information that the events can be accurately replayed. For example, a touch screen drag event needs to be differentiated from a fast series of discrete touches. Lastly, it needs to be able to record events across multiple applications. To accomplish all of these goals we implemented the record system using a client server model.

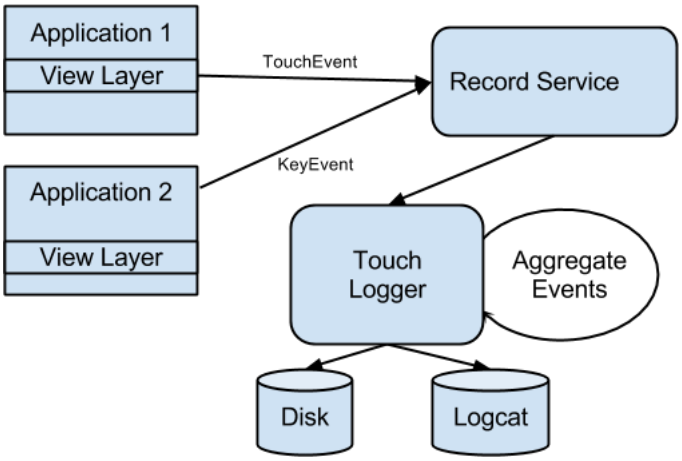

Fig. 1. A visual representation of the backend architecture for the record system.

The clients in this model are bits of code that is inserted into the Andoid APIs View object. Embedding in the View object allows our record system to log events transparently across most applications. Our record system would likely fail in applications that use custom view layers such as games that implement custom interfaces using tools such as OpenGL. To make our system compatible with such applications we would need to log events at a lower level like the native interfaces to the

touch screen. Unfortunately, if we embedded logging at this low a level we would lose the semantics that we use to easily differentiate between different types of events. The View object provides a differentiation interface by using handlers for different types of inputs.

Using a client server model allowed us to accomplish the two remaining requirements, maintaining user experience and cross application aggregation. To maintain the user interfaces responsiveness we simply need to move the logging and aggregation logic off of the UI thread. We could have accomplished this without a separate server program and one of our early prototypes was implemented by just spawning another logging thread. Doing IO on the main thread will prompt many warnings from the developer tools. Only using separate threads in the same program is less useful when trying to implement system wide recording across multiple applications. One early idea was for a logger embedded in each application to write to a common log file. Androids security system makes this difficult. Each Android application must request access to any external media, common files or the network so for each application to be responsible for its own logging all applications would need to have all of these permissions increasing the available attack surface.

In order to solve this problem, we implemented a global service responsible for all logging. The clients we mentioned earlier that are embedded in each View pass all events up to the service. The service we added runs separately from all applications and starts with the rest of the system services at boot time guaranteeing that we will not lose any early interactions unless the service itself crashes. Using a service also localizes all extra access permissions that may be needed for logging. If we chose to implement a network logging interface we would only need to increase the permissions of the service and not the permissions of all applications that are logging interactions.

3.3 Android UI Changes

Since it may not be desirable to record all of the events for the entire time the device is powered on, we implemented a way to turn recording on and off. Modifying the existing development settings application on Android 17, we added a new section that allows a user of the device to turn recording on and off. When recording is turned on, the user interface is sent to the home screen of the device and the service starts recording input. Sending the user to the home screen is done to create a standard start point for each record and replay scenario. The replay functionality described in the next section will also start from the home screen. We believe that having a static start point will minimize discrepancies between record and replay. Also, since the development settings options appeared as they are only in the most recent release of Android, we are limited to using only the newest releases. However, since this provides the only dependency on having the newest version, this feature can easily be implemented in a more backwards-compatible way to allow for older devices to use our system.

4 Replay

4.1 Replay Architecture/Implementation Details

The second part of this system is the replay functionality. This purpose of this part is to seamlessly integrate with the recording part of the system to provide automated playback with the necessary feedback. This feedback can be separated into two parts. First, understanding how successful the replay functionality itself was in performing the actions specified by the log output from the recording system. The system outputs a record of how many of the input actions it was successfully able to put into the MonkeyRunner system. The second part is understanding how well the recorded actions were mirrored by the replay device to achieve the desired results or workflow.

The replay system itself is built on top of a tool call MonkeyRunner, which ships with the Android SDK as a way for users to interact with their devices. It allows Jython (Java python implementation) scripts to interact with an emulator or actual connected device through an API. As it turns out, the allowed set of interactions in MonkeyRunner is actually relatively small, focused on UI interactions for apps. This system seemed extremely well suited for our replay system but it soon became clear that it did not function as expected.

Our replay system consists of a single python script that accepts JSON-formatted replay actions. It checks each one for correct formatting and all required elements for the specific type of input. The replay system currently supports three types of inputs, touch, press, and drags/gestures. Touch events are specified by an x and y coordinate in addition to the period of time for which the press was held down. Press events correspond to keyboard input, like L or Left Shift or Home Button. Drag events are specified by a starting and ending coordinate locations, as well as a time period and number of intervals. The idea is to be able to simulate all types of potential input through the user interface.

4.2 Replay Implementation Challenges

Unfortunately we ran into several problems while implementing the replay system on top of MonkeyRunner. First, the documentation on the system is not very clear about how it actually interacts with the connected device. Even though the MonkeyRunner API supports all of the interactions with an UP and DOWN specification, it converts all input into a single tap event. So a press and hold event is not possible since the DOWN event does not wait for an UP event to release it. Touch events, as a results, cannot be held and are more like location based clicks. Keyboard or button presses similarly cannot be held. Also, since drag events are specified by a beginning

and ending point, so only straight line drags/gestures are possible. Unfortunately, arrays of location tuples are not accepted and the recording of them is actually not always consistent, as not enough points may be captured to correctly draw the shape.

The second problem we ran into with MonkeyRunner is that it does not have the ability to monitor the devices state or current activity, meaning that there is no feedback loop for the replay system. This prevented us from doing some deeper level replay functionality that we had considered as well as unexpected problems with basic replay. One unexpected challenge is that the only timing we can control is the wait time between each event record with a sleep command that specifies a time interval for the device. Any interaction that requires a click and hold to access functionality is cannot be replayed. Since the timing interval is based on static time recording timestamps, actions are synced directly to that system and the corresponding response time. With this consideration, application interactions that are time sensitive rather than just sequential operations may not playback correctly, introducing non-determinism into the system which is supposed to be deterministic.

All of these considerations have lead us to record and replay action sequences on identical devices or emulator configurations to try and eliminate possible sources of error. We have encountered errors from different placement of apps on the app menu, screen sizes, application versions, and missing software keyboards. Based on trials we have also included the specification that all records should start from the home screen so that each interaction is done in the proper context. Another consideration is the state of the device when interacting with it. A common application that we tested on was the calculator app, but its current answer was persistent across sessions, which can affect the outcome of a replay sequence if it was different than in the recording environment. Being able to provide a cached state or clear the cache would allow the system to get around these problems.

5 Evaluation

5.1 Spatial and Temporal Accuracy

For our replay to be effective we need for strings of replayed actions to be spatially and temporally correct. Spatial correctness means that all of the replayed events interact with the same position on the screen as the original recorded event. The spatial drift is defined as the distance the replay point was from the record point. For a replay to be temporally correct all corresponding events in the recording and in the replay must occur at the same time offset from the first event. For example, if the second event, E2A, in the recording occurred one second after the first event, E1A, then in the replay the second event, E2B, needs to also occur one second after the first event in the replay, E1B.

Temporal drift is calculated using the formula (E2A - E1A) - (E2B - E1B). It is the difference of the offsets of each event from the first event in the series. To test spatial and temporal drift in our system we devised an experiment to compare the recording and the replay of the same series of events. For this experiment we used three different applications, Calculator, ColorNotes and Browser. For each application we recorded a series of approximately 45 touches. We then replayed all three series through using our replay tool and recorded those events. We saw discrepancies in the replayed events where events were either lost or doubly recorded. All of the events that were doubly recorded happened when MonkeyRunner tried to dispatch two events from the original recorded log that were close together. The doubly recorded events happened when either interacting with the on-screen keyboard or hardware buttons. This may be due to timing differences and the keyboard not being active. In out analysis we did not compare either missing events or doubly recorded events.

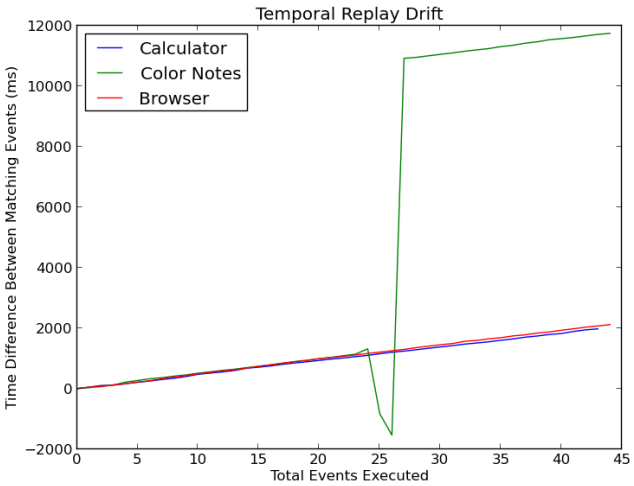

Fig. 2. A graph displaying the results from the temporal drift experiment.

The results of the temporal experiments are shown in figure 2. We do not show the results of the spatial drift because all corresponding points in the recording and in the replay shared the same coordinates. This is not surprising because MonkeyRunner supports a use selecting the exact raw coordinates on which to generate an event. For two of the applications, Browser and Calculator, in the temporal drift graph the graphs shows a roughly linear increase in spatial drift with respect to time. This is due to slight timing discrepancies with each dispatched event in our replay module. These discrepancies occur because we account for the length of and event and the time between events in the recording but we do not account for the time it takes for MonkeyRunner

to dispatch events in the replay. We expand on ways that we can correct this in the future work section. The ColorNotes replay increased linearly until about half way through the replay when there is a sharp increase in drift. This occurred at the same time as two taps on the hardware back button and one of these taps was not recorded in the replay. We do not know the exact cause of the delay but we think that it had something to do with problems dispatching hardware key events.

5.2 Slow System Testing

We were concerned about how our record and replay system would perform on heavier weight version of Android such as CleanOS and TaintDroid. The root of our concern is that MonkeyRunner has no knowledge about the state of the system or state transition time. This could manifest itself if an application uses animations between UI states and those animations could take varying amounts of time. Animation time could vary based on system load or replaying a set of interactions on a different variant of Android than the one on which is was recorded. We simulated replaying across heavier weight systems by installing sleeps before events are handled. Our wait times varied between 0 milliseconds to show a replay on the same system and the extreme case of 1000 ms delay to simulate a system under heavy load, a heavy weight variant of Android or an underpowered device. For this experiment we recorded an interaction from the home screen, entered into the app launcher and selected the calculator application. It then proceeded to perform calculations, pause, and then perform further calculations. You can see the results of this experiment in figure 3. In the figure the X axis is milliseconds since the replay started and the Y axis is the amount of delay. Each point in the figure represents one touch event.

In all of the cases except for the 1000ms case the application proceeded through the replay. You can see that there is some temporal drift between the original version and the replayed. The amount of drift looks to remain constant between the systems with different amounts of simulated delay. All of the results except for the 1000ms case replayed through the actions and reproduced the same state. The 1000ms case diverged very early due to the varying animation length issue mentioned earlier. There is an animation when the user selects the application launcher from the home screen which was excessively long in this case. Because MonkeyRunner has no way to know when the system reaches the correct state it tried to replay the touch that selected the application before the animation completed. After the animation finished one of the later taps selected the incorrect application causing UI divergence. This illustrates one of our system's key issues, it has no knowledge about the current system state. If we could incorporate more intelligence into the replay system it could wait for the animation to complete before proceeding.

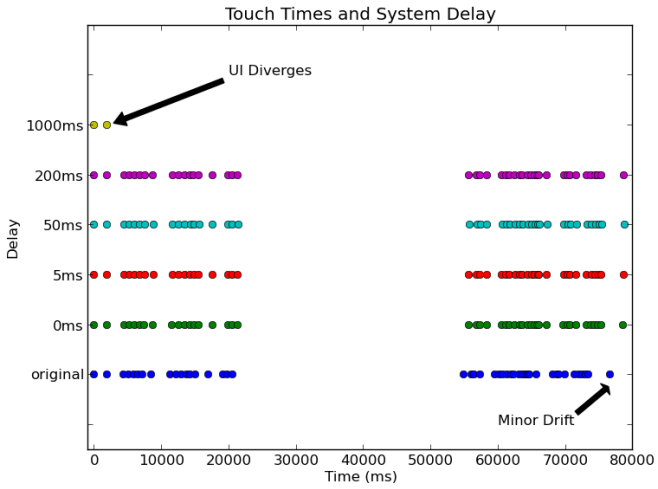

Fig. 3. A graph showcasing the results from the slow system test.

5.3 UI Divergence

Based on the difficulties that we had with the replay implementation, we decided to run tests to see in what contexts the replay system correctly interacts with the system and produces the correct results. We decided to go test two applications: colornote and calculator. The calculator app is one of the main applications we used in the development process for tests so we knew that it worked well with the replay system. It also is primarily dependent on sequential input and does not have any UI element changes that take time to execute.

For the Calculator app experiment, we recorded 222 actions that consisted of inputting numbers and operators to compute the result. In the initial replays, we ran into a bug that caused 12 of the actions to fail. We quickly discovered that our system did not accommodate x,y floating point coordinate locations, so we added a test to for a round points with decimal places. The functions in the application are simple so almost any ordering would work. After recording the sequence. We replayed it again on the same emulator on a different machine. After running the replay 5 times, each trial produced the correct results and had the same sequence of events to get there. Most of the inputs were a continuous calculation so any error would have changed the end result. For the second test, we used an application called colornote. This app allows the user to make notes, change their color, add a title, or make a checklist. In this application the interface has delays when it changes from the main menu to a note or checklist. Other than that, other interactions such as adding text or changing the label color did not require delaying UI changes. The purpose of testing this application was that we believed the more extensive UI interactions and menu changes

would create non-determinism in the system and would be unable to produce the desired result.

During the replay, we initial ran into a few errors with setting up the emulator the same way. On the replay machine, keyboard input for the emulator was provided by the computers keyboard instead of the on screen software keyboard used in the recording. The screen size of the two emulators did not have the same pixel dimensions, so the playback was not able to provide the correct input. Once these problems were rectified, we were able to run the test 5 times with slightly differing results. Three of the trials were the same and produced identical outputs. Two of the trials produced slightly different results. None of the trials however produced the same results as the initial recording.

While watching the replays, the first source of error was the UI changes. The sequence of events was to open colornote, make a new note, change the color a few times, add some text and a title, then go back and make a checklist called MyList that ends up having the elements One, Three, and Five. The replay system misses the back track from the note to the main menu in order to make the checklist. Instead it stays in the note and performs all of the corresponding actions in the note, which explains the final results. The color changes and most of the text in the note are correct for the first part of the sequence before the checklist is created. The UI menu changes would be better monitored by waiting for the system state instead of waiting for a specified amount of time. In some cases exact timing may be the focus of the test, which would show inconsistencies or differences in how applications run on different system configurations. It is also possible that while watching the tests a mouse click was registered which could have affected the out come of one of the trials. In future implementations the emulator or device should have the option to be locked so as only to accept output from the replay system while it is running.

The next two sources of error are related to the on screen keyboard. The keyboard itself does not always seem to respond to given input, which accounts for some of the inconsistencies in the titles for two of the trials. The trials were run with the system settings on where touch input were shown on screen my a dot where they were pressed. This is how we monitored the keyboard not responding to all given input. The other problem is the predictive text engine. Since the input is pushed together in the note it no longer form words in the English language and can be subjected to automatic changes from the predictive text engine. It may not have changed text in these trial runs but it definitely pops up when typing. Without understanding more about how the correction engine works, we cannot be sure that it will always give the same suggestions in each case. Custom dictionary entries for example may influence what is accepted. The primary suggestion can also be forced as a change depending on whether a space is entered or the system clicks out of the word.

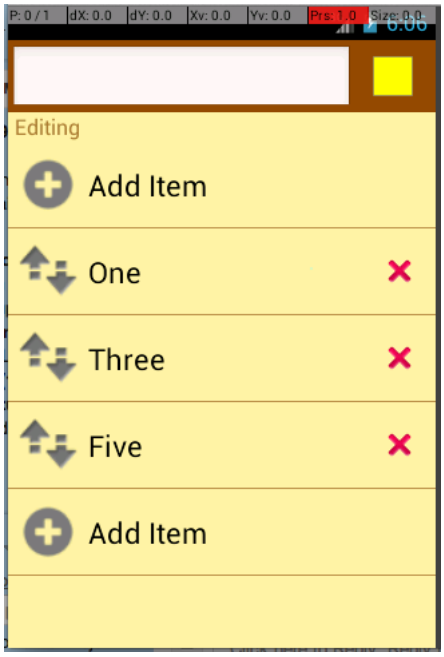

Fig. 4. A screenshot from the end of the recording for the colornote test.

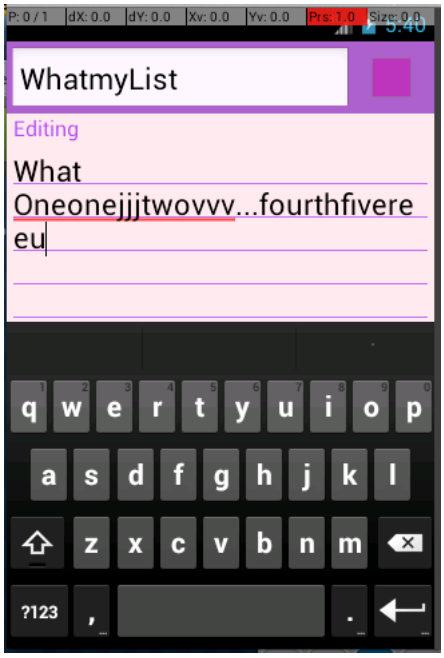

Fig. 5. A screenshot of the final result of a replay of the colornote sequence.

5.4 Energy Testing

We wanted to perform energy testing on the system to determine the overhead of the record system on a device. The standard benchmark for this program is powertutor,

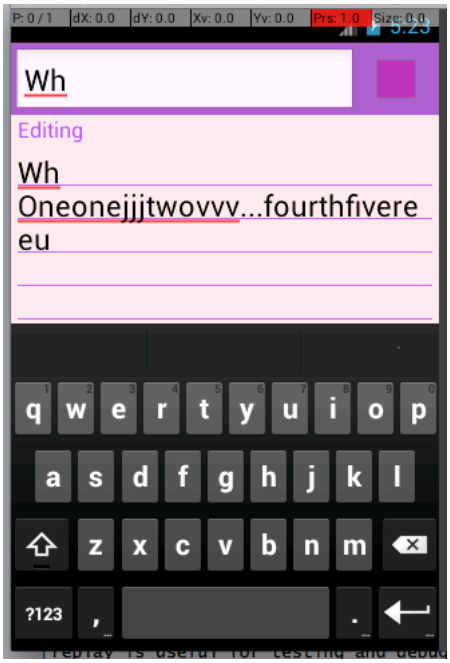

Fig. 6. A screenshot of one of the alternative final result of a replay of the colornote sequence.

which we could not get to work on our current testing configuration. We believe that the overhead for the system would be low, approximately 1-2% but we were not able to empirically confirm this.

We made several attempts at running our code on a Nexus 7 tablet but were unable to get the system to boot using our generated images. We believe that there were issues with compilation configurations. Since we are only supporting Android version 17 and higher we could not use an older device and had no other new enough devices available to us at the time of testing.

6 Future Work

6.1 Application State

One of the biggest limitations to any record and replay system is replicating the system state and handling non deterministic inputs. Our pure UI based replay system is especially vulnerable to state change because the output of the system can be affected by the internal data state, non deterministic IO and layout of the user interface which can be influenced the 2 previous. Researchers have developed other record and replay systems to try and mitigate problems associated with state change. One such system is Chronicler. Chronicler analyzes and modifies java bytecode to capture all input associated with the system that may cause non deterministic output. When the program is replayed the logged input fed back into the application to replicate this state. Another record and replay system, Dora, takes a similar approach. Dora captures non deterministic system calls and replays them allowing for potential differences in the system call order. In addition to these two techniques we could employ Android specific structures. In the future we may be able to take advantage of the Android activity stack to ensure that an applications is advancing through and identical series of activities and potentially force it back to the same stack trace. Some types of nondeterminism may be difficult to replicate even with the previously mentioned systems such as simulating the same network latency. This could be important in an application that might warn of network delays changing the activity stack.

6.2 Device Agnostic UI Replay

A useful direction for future work on this record and replay system would be device agnostic replays. Right now our system is limited to running on devices with exactly the same screen dimensions with exactly the same layout with exactly the same user interface. This is very limiting because something as simple as installing a new application could reorganize the application launch screen causing the entire replay to crash. The simplest and most naive solution is to linearly scale the location of the replay touch with the size of the screen. This would not work because even the default views through the Android API act in a much smarter way by filling the screen with more contents rather than scaling the contents that are already present.

To achieve device agnostic replay we need more semantic information about the application itself. This may be possible because many android application user interfaces are laid out according XML documents with tagged user interface elements included in the distributed application .apk file. If we could tell what the user was trying to do rather than just the locations where they pressed then it would allow phone records to replayed on the same piece of software on a tablet, given that the tablet implementation was the same with a different layout, if buttons were missing it would prevent this approach from working. It would be able to function in cases where the GUI was redesigned from a graphical perspective, like inserting a new CSS file onto a webpage, only the look and feel changes. If there were a way to specify certain actions were only to be run if a condition, like the presence of a specific button or field, were met, then the system could potentially run a subset of tests across multiple versions of an application.

Another consideration with device agnostic UI replay is the performance of the system, since not all devices have the same hardware specifications. As noted in the testing, our current system is able only control the time in between actions, which is based on the running of a specific configuration. If we could monitor the activity of the system and save state information as the recording was running, the system could wait until a certain action was performed or state changed before taking the next step, instead of an arbitrary amount of time. This would allow for playback on slower or faster devices.

6.3 Additional Functionality

There are some functions built into MonkeyRunner that we were not able to implement successfully, like automatic screenshots. The current implementation of the system can take screenshots, just not in a controlled or useful way, so the functionality was disabled for the final release. While the script is able to run all actions sequentially, it processes input asynchronously of the device playback. For this reason, screenshots cannot be synced with specific parts of the replay sequence, only at arbitrary points. Once this was done, we could have screenshots taken during corresponding record and replay sections and do an automatic diff comparison of the images to show what, if any, were the differences at that point. The automatic comparison functionality would be less useful on differ devices where the images or UI was different, but could still be examined by hand.

Based on all of the things we have learned from this project and the shortcomings of the current tools, we believe that future implementations of the replay system would need to be built as a custom system service rather than on top of the tools that ship with the SDK. There is another testing tool called UIAutomator, which gives you the XML of the current application, but beyond that is still similarly limited in its access to the device it is interacting with.

Continuing on, we would try to implement out replay system using Robotium instead of MonkeyRunner. Based on an over view of the projects source code it seems that it would allow us to overcome some of the issues we faced using MonkeyRunner. For example, Robotium makes no assumptions about the lengths of button presses and will emulate holding a button down if such an action is requested through it's API.

7 Conclusion

The scope of the project was determined by the amount of time we had to complete it, approximately one month. As a result, we had to scale our expectations and project according what we could realistically accomplish in this timeframe. We decided that the best approach was to build the recording tool as a system service and the replay system on top of the MonkeyRunner tool. We ran into implementation problems on both fronts. The recording system is functional and has access to all of the necessary activity on the device. We were unable to successfully capture all types of planned input, but this would be possible given more time. The system successfully outputted the actions as JSON objects for the replay system to use. The replay system is able to accept and run all three types of planned input, touch, press, and drag. Basic screenshot taking was tested but due to the implementation of the MonkeyRunner system this and other aspects of the playback were not able to function according to the specifications we had decided on. After gaining a better understanding of MonkeyRunner though testing and watching the system activity when it was running, we were able to determine that it is not a sufficient platform for a comprehensive replay system. Further implementation would need to be done at a deeper level, perhaps as a second system service.

We were able to evaluate the system on many fronts and discovered that the replay is correct with regard to location and type of action, but the timing is not completely correct. Given this consideration, we are able to still use the system on applications with very sequential input without long UI menu or element changes. Given out time restrictions and implementation problems the resulting system contains a lot of the planned functionality.

8 Work Distribution

8.1 Record/Replay System

RILEY & DMITRIY: Record and Android System Services
LOGAN: Replay Implementation

8.2 Evaluation

RILEY: Spatial and Temporal Accuracy, Slow System Test, Attempted Energy Testing
LOGAN: UI Divergence Testing, Attempted Energy Testing
DMITRIY: Attempted Energy Testing

8.3 Paper

LOGAN: Section 4 - Replay, Section 5.3 - UI Divergence Evaluation, 5.4 - Energy Testing, Section 6.2 - Device Agnostic UI Replay, Section 6.3 - Additional Functionality
RILEY: Section 3 - Architecture/Implementation Details, Section 5.1 - Spatial and Temporal Accuracy, Section 5.2 - Slow System Testing, Section 6.1 - Application State, Section 6.2 - Device Agnostic UI Replay
DMITRIY: Section 1 - Introduction, Section 2.1 & 2.2 - Related Works, Section 4.2 - Replay Implementation Challenges, Section 7 - Conclusion
DEEPIKA: Section 2.3 & 2.4 - Related Works

Acknowledgment

The authors would like to thank Professor Roxana Geambasu for her guidence and advice on this project.

References

[1] Jason Flinn and Z. Mao, *Can Deterministic Replay Be an Enabling Tool for Mobile Computing?*, Pervasive Computing Research at Michigan. University of Michigan. http://notrump.eecs.umich.edu/papers/hotmobile11.pdf

[2] Nguyen, Bao, Bryan Robbins, and Ishan Banerjee, *GUITAR - A GUI Testing Framework* Event Driven Software Lab - University of Maryland. http://sourceforge.net/projects/guitar/

[3] Gorilla Logic, *MonkeyTalk*. https://www.gorillalogic.com/monkeytalk

[4] Renas Reda, *Robotium - The World's Leading Android Test Automation Framework*, User Scenario Testing for Android. https://code.google.com/p/robotium/.