Fuzzy and Cross-App Replay for Smartphone Apps

Yongjian Hu
University of California, Riverside
yhu009@cs.ucr.edu

Iulian Neamtiu
New Jersey Institute of Technology
ineamtiu@njit.edu

ABSTRACT

The behavior of smartphone apps is driven by input from sensors such as GPS, microphone, or camera. Hence the ability to construct test inputs, and send these inputs to the app is essential for testing. Leveraging our prior results in recording and replaying sensor inputs in Android apps we have constructed a new approach that helps automate smartphone app testing by capturing the input log (sensor stream) and using this log in two ways. First, we fuzz (alter) the log in a semantically-meaningful way: by applying principled transformations (e.g., changing GPS coordinates or navigation speed), a new input log is constructed, which represents a new test case. Second, we use the log captured in app A to test an app B which offers similar functionality, e.g., GPS navigation or image recognition. We have applied our approach to several widely-used Android apps and found that the approach is effective: it has revealed new bugs in four popular apps; has produced new test cases that increase coverage; and has produced test cases from logs originating in other apps.

Categories and Subject Descriptors

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

Keywords

Mobile applications, Google Android, Record-and-replay, App testing, Physical sensors

1. INTRODUCTION

Smartphones are ubiquitous, with about 2 billion devices in current use [11, 12]. The success of the smartphone platform is due, in no small part, to the plethora of applications ("apps") which leverage sensor-based capabilities. For example, GPS allows apps such as Yelp to provide location-aware services; the camera allows image matching apps like Google Goggles to provide search-by-picture features; the Shazam app can help recognize an ambient song by using the microphone. Our prior study has quantified this reliance on physical sensors: among the top-11 most popular apps across the 25 app categories on Google Play, 60% of the apps use the GPS, 34% use the microphone, and 34% use the camera [18].

Smartphone apps however, pose several testing and verification challenges. First, smartphone apps take the bulk of their input from sensors, in contrast to desktop/server program which tend to take their input from files. This raises obstacles for capturing and delivering the input (the app input cannot be simply redirected to come from a file).

Second, app verification time has to be kept short: to stay competitive and attract users, app developers must create new apps and release app updates at high cadence, but current app testing tools have very limited support for sensors, and testers/developers have little time to construct new test cases. Hence new apps, and new versions of existing apps, tend to be unreliable as they are released with little scrutiny and testing [20, 17, 10, 16].

Third, traditional verification techniques such as static and dynamic analysis are hindered by smartphone apps' idiosyncrasies. Static analysis is hindered by implicit flow, IPC, and the callback-oriented architecture of apps [8, 26]. Dynamic analysis is hindered by the lack of tools that can generate high-coverage testing suites [9]. While testing and test automation tools do exist, they cannot handle sensors properly—they focus only on GUI events [7, 6, 15, 19, 4] or only work on the emulator (which has poor sensor support), not on the phone [23, 3].

The first issue — input capture and replay — has been addressed by our prior work, where we constructed a record-and replay system named VALERA [18]. VALERA performs record and replay by capturing and replaying not only sensor input, but also network input and scheduling events with high accuracy and low overhead.

Our approach in this paper aims to address the other challenges — creating new test cases and running them on the smartphone in a repeatable fashion. Specifically, we use VALERA to record the input stream from an execution (this initial execution can be the result of user interaction or a test case itself) and then alter the input stream to create new test cases for the recording app or a different app.

Our first contribution is *semantic sensor data alteration* (SSDA) in which we manipulate recorded sensor data in a semantically meaningful way, to generate new executions from existing ones (Section 3). We present a set of semantic alteration strategies for geographical location, images, and audio

AST'16, May 14-15, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2896921.2896925

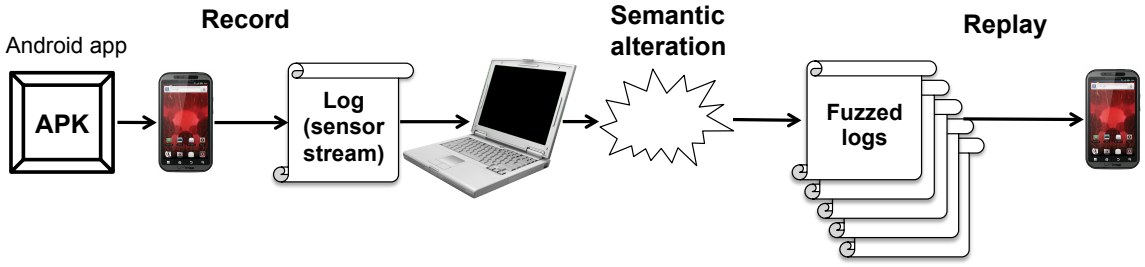

Figure 1: Overview of SSDA.

streams. An immediate application of recording and "fuzzy replaying" app executions is test amplification: generating new test cases from existing ones, as well as boundary testing to attempt to crash an app by feeding legal boundary data. Using SSDA we were able to uncover crash-inducing bugs in popular apps, each having at least one million downloads, such as Yelp and ROUTE 66 Maps. In addition, our experiments show that SSDA can increase coverage by exploring previously-unexplored app methods.

Our second contribution is cross-app testing, i.e., allowing snippets of executions recorded on one app to be replayed on other apps, which significantly expands the reach of app testing, and allowing one record-based test to be executed on a variety of related apps. We demonstrate this functionality on three classes of applications: GPS-based navigation, barcode scanning, and audio recognition. We present details in Section 4.

Our approach fulfills several key desiderata: it does not require access to the app source code, supports popular sensors, creates new test cases without burdening the user, and works with apps running directly on the phone. These features keep the approach flexible, widely applicable, and easy to use.

2. BACKGROUND

Android platform overview. The Android software stack consists of apps using the services of Android libraries and an Android Framework (AF). The AF orchestrates the control flow by invoking app callback in response to sensor events. Each app runs in its own address space, on top of the Dalvik Virtual Machine (on Android versions < 5.0) or ART, the Android Runtime (on Android versions ≥ 5.0). The VM or ART in turn run on top of Android Linux. Android apps are typically written in Java and compiled to bytecode; some apps also employ native code. Apps are distributed in the form of APK files, which bundle the bytecode along with app resources and a manifest file.

Record-and-replay overview. Record-and-replay allows executions to be captured and reproduced. On Android, this capability is particularly useful because apps are sensor-driven (so input can't be fed from a file) which makes test automation difficult and highly concurrent (which means replaying the event schedule is essential). In prior work we have developed a tool named VALERA that has shown to be highly effective at recording and reproducing executions on dozens of popular apps that use a variety of sensors, e.g., GPS, camera, microphone. VALERA's overhead is around 1–2% for either record or replay [18].

VALERA uses two main techniques to achieve record-and-replay: (1) it employs bytecode rewriting to wrap sensor API calls hence allow sensor input to be recorded to log files and then injected back into the app upon replay, and (2) it uses a modified Android Framework that records and replays the event schedule as well as network input.

Record-and-replay systems strive to be deterministic, i.e., the replayed execution should be indistinguishable from the recorded execution; a replay execution that is distinguishable from a recorded one or other replayed ones is said to "diverge". In this work, however, we specifically undermine the determinism goal: we deliberately alter (fuzz) the recorded values to steer the execution away during replay, deliberately causing divergence but in the process creating new executions. These new executions can be put to multiple uses, e.g., test amplification or boundary condition testing.

For our purposes, given an app, we record its execution into a log file on the phone (in practice, a set of log files, one per sensor) using VALERA, modify the log file off-line, and then restart the app and feed the input from the log instead of letting the app read the input from the sensors.

3. SEMANTIC SENSOR DATA ALTERATION

Since mobile app behavior is context-dependent, our idea is to alter the context to induce a change in app behavior. Leveraging VALERA, we start with a recorded execution and then alter sensor readings in a semantically meaningful way (SSDA) so that during replay the altered sensor readings drive the app execution to new states.

An overview of SSDA is provided in Figure 1. We first *record* the execution of the app; the app is exercised in this phase via manual input, e.g., navigating to record a GPS trace, using the camera to record a set of camera inputs, or running an app such as Shazam to record microphone input. The result of the record phase is a *log*, i.e., a stream of sensor inputs. Given the recorded log, we use *semantic alteration* to modify the recorded sensor inputs; this phase is performed off-line, on a desktop/laptop computer. SSDA provides a set of systematic altering methods for each sensor, described later. The result of SSDA is a set of *fuzzed logs* – streams of sensor inputs. We emphasize that the log alteration process is completely automated, avoiding the user the burden of manually creating new logs. Finally, we *replay* the altered logs back into the app, observe the execution, and note differences from the original recorded execution. We now proceed to defining the semantic transformations specific to each sensor.

3.1 Location

As mentioned previously, the location (GPS) sensor is

used to provide location-sensitive services, and detect changes in the phone's geographical location. We have identified three semantic alterations for the location:

1. *Null location*: we inject a `null` location reference into the logs to simulate scenarios when the underlying GPS module has trouble acquiring a location. Note that `null` is a valid location reply in the GPS API, and it is good programming practice to check the location parameter before using it.

2. *Map shift*: this involves taking the series of coordinates $\langle (x_1, y_1), \ldots, (x_n, y_n) \rangle$ from the recorded execution and changing them by "shifting the map" by a $(\Delta x, \Delta y)$ factor so that during replay the set of coordinates fed to the app is $\langle (x_1 + \Delta x, y_1 + \Delta y), \ldots, (x_n + \Delta x, y_n + \Delta y) \rangle$. The parameters $(\Delta x, \Delta y)$ can be inputed manually by the developer or generated randomly by SSDA.

3. *Speed change*: we replay a route at different speeds, e.g., to simulate driving vs. driving faster (where speed limit alarms can go off). This is achieved by altering the GPS coordinates according to the interval between two consecutive location API upcalls. Note that during replay we cannot alter timing — the app expects the sensor input *at the same time as during record*. Hence we alter the coordinates to simulate a change in speed: we compute the first derivative of position and then during replay alter the coordinates so that the coordinates change faster/slower.

To create valid routes for our SSDA experiments we have downloaded predefined routes from Google Earth [14], used the route's locations as waypoints, then modified the route according to a given speed to simulate the required activity, e.g., driving.

3.2 Camera

In Android, the camera can take pictures one-by-one or continuously, using the frame buffer. We implement several image transformations.

1. *Exposure*: we alter the exposure (i.e., darken or lighten the image) to simulate a day/night swap. Similarly, we alter the color balance to simulate different lighting conditions, e.g., moving from sun to shade.

2. *Size/blur*: we alter image size and blur the image to simulate a poor-quality picture or holding the phone incorrectly w.r.t. the object of the picture.

3. *Rotation:* we rotate the image at various angles; this transformation tests the ability of image recognition algorithms to handle different orientations of objects in the picture.

In all cases, we have performed the semantic picture alteration by using the ImageMagick [1] image manipulation toolkit to alter the recorded picture, e.g., blur, rotate, darken.

3.3 Audio

Android apps use two API classes, MediaRecorder and AudioRecord, to process audio. For SSDA we mainly focus on AudioRecord as it receives the raw audio stream from the microphone and is the preferred way of performing on-the-fly audio processing. We have applied several audio altering techniques:

1. *Sample rate*: we increase and decrease the sample rate (the sample rate determines the quality of the audio stream as it defines the number of sound samples per unit of time).

2. *Noise*: we add background noise, to simulate the situation that the audio is recorded in a noisy environment.

We first used the PCM Recorder to record and save a song in raw audio format. Next, we performed audio transformations by altering the recorded sound data via the SoX open source package [2] and finally sent the altered audio to the app via replay.

Table 1: Test apps.

App	Sensor	# Downloads (millions)
GasBuddy	GPS	10–50
Sygic GPS	GPS	10–50
Waze Social GPS	GPS	10–50
Yelp	GPS	10–50
Scout GPS Navig.	GPS	1–5
Route 66 Maps	GPS	1–5
GPSNavig.&Maps	GPS	0.5–1
NavFreeUSA	GPS	0.1–0.5
Barcode Scanner	Camera	50–100
Google Goggles	Camera	10–50
Amazon Mobile	Camera	10–50
QR Droid	Camera	10–50
CamScanner	Camera	10–50
CamCard HD Free	Camera	1–5
RedLaser Barcode	Camera	1–5
Walmart	Camera	1–5
CardToContact	Camera	0.1–0.5
Business Card Rdr Free	Camera	0.5–1
ScanBizCards Lt	Camera	0.5–1
Shazam	Microphone	50–100
Tune Wiki	Microphone	10–50
PCM Recorder	Microphone	1–5

3.4 Evaluation

We now discuss how we evaluated the utility of our SSDA techniques.

Test apps. We chose our test apps directly from Google Play, the main app store for Android, according to several criteria: apps had to be popular (a large number of downloads) and based around various sensors. Apps, along with their popularity and main fuzzed sensor, are described in Table 1.

Platform. The smartphone we used for experiments was a Samsung Galaxy Nexus with Android version 4.3.0, Linux kernel version 3.0.31. The phone has a dual core ARM Cortex-A9 processor running at 1.2GHz. For app rewriting and semantic alteration we used a MacBook Pro laptop (2.5GHz Intel Core i7 processor with 16 GB memory) running Mac OS X 10.10.5.

Table 2: SSDA Evaluation results.

App	Sensor	SSDA	Outcome	Time (sec)	Log Size (KB)	Coverage increase (%)
Yelp	GPS	`Null` location	**CRASH**	25.42	<1	
		Map shift	Different search result shown	35.67	<1	<1
		Speed change	Normal execution	36.10	<1	<1
GPS Navig.& Map	GPS	`Null` location	**CRASH**	26.83	31	
		Map shift	Different map route shown	73.14	72	<1
		Speed change	Different driving speed shown	71.85	70	<1
Route 66 Map	GPS	`Null` location	**CRASH**	21.43	26	
		Map shift	Different map route shown	65.31	64	<1
		Speed change	Different driving speed shown	67.74	65	<1
NavFree USA	GPS	`Null` location	**CRASH**	32.02	32	
		Speed change	Different driving speed shown	64.13	73	<1
		Map shift	Altered execution: "unknown coordinates" error message	15.97	71	3.5
GasBuddy	GPS	`Null` location	Altered execution: "Oops! device cannot find your location" error message	12.73	<1	1.2
		Map shift	Different search result shown	41.71	<1	1.3
		Speed change	Normal execution	42.92	<1	<1
CamCard HD Free	Camera	Blur	Altered execution: fail to recognize picture	36.72	3,576	2
		Darken	Altered execution: "Low Light" error message	36.14	3,684	2
		Lighten	Altered execution: "Blurry Image" error message	36.86	3,630	2
		Rotate	Altered execution: fail to recognize picture	36.58	3,593	2
Barcode Scanner	Camera	Blur	Altered execution: fail to recognize picture	32.76	4,137	<1
		Darken	Altered execution: fail to recognize picture	32.58	4,186	1
		Lighten	Altered execution: fail to recognize picture	32.74	4,238	2
		Rotate	Altered execution: fail to recognize picture	32.87	4,194	5
Google Goggles	Camera	Blur	Altered execution: fail to recognize picture	36.59	5,186	1
		Darken	Altered execution: fail to recognize picture	35.78	5,240	1
		Lighten	Altered execution: fail to recognize picture	36.81	5,159	2
		Rotate	Normal execution	36.04	5,206	<1
Shazam	Audio	Decrease sample rate	Altered execution: fail to recognize song	45.37	2,745	3
		Adding noise	Altered execution: fail to recognize song	45.26	2,833	3

Procedure. We first recorded a normal, non-crashing execution. Then we used SSDA on the recorded logs, replayed the semantically altered logs via VALERA and noted any difference in app behavior compared to the behavior during record—either crash or increased coverage, as explained next.

Location. The "GPS sensor" rows of Table 2 show the SSDA results when traces are altered in two ways: injecting a `null` location into the trace and map-shifting. We discovered that four popular apps crash when presented with a `null` location: GPS Navigation & Map, Yelp, Route 66 Map, Navfree USA; please see Table 1 for the popularity of these apps. GasBuddy handles this situation more gracefully, though the behavior is different compared to the original execution. Navfree USA exhibits a different behavior when presented with a map-shifted list of locations.

Camera. The middle part of Table 2 shows the result of applying image SSDA to three popular apps. CamCard HD Free helps users scan business cards. Barcode Scanner is used to scan barcodes or QR codes. Google Goggles performs image search, by taking a picture and searching Google Images for similar pictures. In all cases we apply the image SSDA techniques (e.g., blur, darken, lighten, rotate). The "Outcome" column of Table 2 shows the results: the app's behavior is different in all cases, compared to the original execution—displaying an error message or failing to recognize the image thus ending up in a different state compared to the original app.

Audio. SSDA was effective at altering executions for Shazam as shown in the last row of Table 2. The default sample rate used by Shazam is 44.1KHz, and its buffer size is 4,410 bytes. We changed a song that was successfully recognized during the record phase by altering the song's sample rate to 16KHz, 44.1KHz and 48KHz. The result indicate that Shazam can recognize the song if the recorded audio stream's sample rate is $\geq$ 44.1KHz, but fails when the sample rate is below this value. This behavior is expected, because a low sample rate means low audio quality which hinders Shazam's song recognition efforts.

Surprisingly, we found that, when playing two songs simultaneously, Shazam can recognize one of the songs.

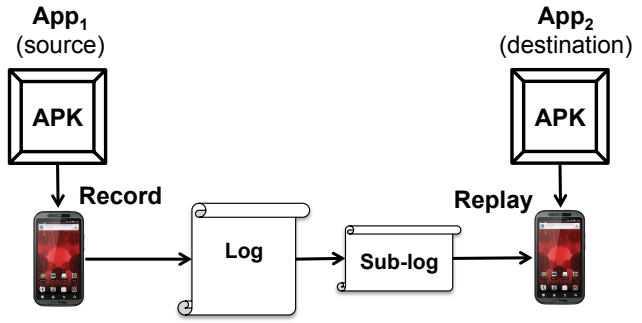

Figure 2: Overview of cross-app testing.

Performance. The "Time" and "Log Size" columns of Table 2 show the running time and log size. Thanks to VALERA's low runtime overhead (about 1%) we did not experience any visible slowdown during execution. In the `null` location case, we inject the `null` object randomly into the middle of the original trace, and remove the trailing data. The four crashing apps exit early, thus their running time is short. The log size is small for GPS: the three navigation apps (GPS Navigation & Map, Route 66 Map, NavFree USA) log around 70 KB data as they continuously use the GPS sensor. Yelp and GasBuddy, on the other hand, log very little data because they use the GPS only when users search by location.

The log sizes of camera apps and audio apps are comparatively much larger: on average, the camera apps log 4 MB of data in 35 seconds, while the audio apps log 2.7 MB in 45 seconds. Though these log sizes are larger compared to GPS, they are very much manageable: modern smartphones' storage capacity (typically 32GB) is large enough to hold the log data.

Coverage increase. We measured the increase in coverage as follows. We leverage Android's default profiler to collect the executed method trace. In each entry and exit of the instrumented sensor API function, the profiler is opened and closed. The trace data is saved into SD card. Then we use Android's `dmtracedump` to dump the executed method trace. Finally, we compare the normal trace and SSDA trace. The coverage increase shows the newly executed functions that do not exist in normal trace. The results (method coverage increase due to SSDA) are in the last column of Table 2. Notice that coverage increase depends on the app. For example, while a `Null` location is effective at crashing Yelp, GPS Navig.& Map, and Route 66 Map the other GPS alterations do not increase coverage substantially in these apps. For NavFreeUSA and GasBuddy the cumulative increases are 3.5% and 2.5% respectively. Camera and audio SSDA, however, are more effective, yielding between 4% to 9% cumulative coverage increase. We believe that these coverage gain figures are acceptable given that the fuzzed logs are created automatically (no user involvement).

4. CROSS-APP TESTING

Cross-app testing involves collecting a trace in one app (source) and replaying snippets of it in another app (destination). For example, one trace snippet can be a route recorded with the navigation app Navfree USA and then replayed on other navigation apps such as GPS Navigation & Maps and Waze Social GPS Maps; the latter apps will behave as if the phone follows the recorded route.

This is somewhat akin to test amplification, though it involves taking a test constructed for one app and using it construct a test for another app. Figure 2 provides an overview of this process: given a log (collected with VALERA) from a "source" app App_1, we extract a sub-log (e.g., only the GPS waypoints) and then replay that log into a "destination" app App_2. The success of cross-app testing relies on two main prerequisites. First, App_1 and App_2 must share the same functionality (e.g., navigation or barcode scanning). Second, the state where record starts for App_1 should be equivalent, at high level, with the replay start state for App_2 so that App_2 can accept the input events in the sub-log.

We believe that cross-app replay has the potential to significantly improve the state-of-the-practice in Android app testing: by collecting a library of sub-logs (trace snippets) specific to each sensor, mobile app researchers and developers can replay the sub-logs from the library on their own apps to augment their testing suites. We now proceed to evaluate the applicability of cross-app testing.

4.1 Evaluation

For evaluation, we used the same platform and apps as those discussed in Section 3.4.

Category 1: Navigation Apps. Navigation is a widely-used application of smartphones, hence app marketplaces contain numerous navigation apps. Although each navigation app has its own feature such as voice command, turn-by-turn directions and social connection, their common functionality is to provide routes and guide users. Without our approach it would be tedious to generate location traces for each navigation app. With our API interception support, we can collect a location trace from one app and reuse it for others. Since apps receive location updates via the standard Location API, we could easily port the trace from one app to many others without any modification. As Table 3 indicates (the GPS row), in our experiments, we collected a routing trace from Navfree USA and have successfully replayed it on five other popular navigation apps.

Category 2: Barcode Scanning Apps. Barcode and QR code scanner apps are another popular application of smartphones, e.g., when users are shopping, as the app can recognize the item by its barcode and compare its price on different online shops. QR codes are widely used to entice users to scan the code and visit a certain URL. Table 3 (row "Camera (Frame buffer)") shows the result of cross-app testing for apps in this category. We collect the camera's frame buffer trace from the Barcode Scanner app, and successfully replay it back to other five apps that are also able to read barcodes.

However, unlike for the navigation apps above, we had to modify the trace data before cross-app replay because different apps set different camera parameters. For instance, Barcode Scanner sets the camera preview size to 1280x720 pixels while QR Droid sets it to 864x480. While we could not reuse the trace "verbatim", we found out that modifying the trace data to fit the current app could be automated and performed on-the-fly during replay by simply examining the parameters with which the app has invoked the API, in this case the preview size. By modifying the preview size, we could successfully replay the Barcode Scanner trace on the other five apps.

Table 3: Source and destination apps for cross-app replay.

Sensor	Recording app	Recording Time (seconds)	Log Size (KB)	Replaying app	Replaying Time (seconds)
GPS	NavFreeUSA	72.13	68	GPS Navigation & Maps	68.72
				Sygic GPS	75.48
				Waze Social GPS Maps	83.17
				Scout GPS Navigation	69.44
				Route 66 Maps	78.69
Camera (Frame buffer)	Barcode Scanner	32.55	4,168	QR Droid	38.50
				RedLaser Barcode	41.72
				Goggle Goggles	37.25
				Amazon Mobile	43.52
				Walmart	39.69
Camera (Take picture)	CamCard HD Free	45.81	3,610	CardToContact	41.02
				Business Card Reader Lite	48.60
				ScanBizCards Lite	41.77
				Google Goggles	38.96
Audio	PCM Recorder	52.65	2,395	Shazam	34.28

Category 3: Business Card Recognition Apps. These apps use pattern recognition to extract contact information after taking a picture of the card. Unlike the barcode scanning apps studied above, business card recognition apps mainly use the camera's picture taking feature instead of frame buffers. We found that pictures are saved in a common format (JPEG) that makes it easy to reuse trace snippets from one app to another. As showed in Table 3 (row "Camera (Take picture)") we take pictures of business cards from CamCard HD Free and can successfully replay the trace in four other apps.

Category 4: Audio Apps. The cross-app testing for audio-related apps is interesting: we chose two apps, PCM Recorder and Shazam, which on the surface are not in the same category as the former allows users to record an audio stream while the latter recognizes songs. Since both use the AudioRecord API, it is possible to replay the audio data trace from one app to the other. In our experiment, as illustrated in the last row of Table 3, we collect a song's audio stream in PCM Recorder and can successfully play it back on Shazam.

Cross-app testing time. The "Recording Time" and "Replaying Time" columns of Table 3 show the time it took to record the log in App_1 and replay the sub-log in App_2, respectively; note that these times are low (less than 80 seconds in all cases).

5. RELATED WORK

Android test automation. Recent works on Android test automation mainly focus on the GUI part. Tools like Android Guitar [22, 4], Robotium [15], Troyd [19] 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 supporting complex gestures (e.g., swipe, zoom, and pinch) or sensors. We eliminate this manual effort by recording actual executions, and supporting sensors. However we do not offer scripting facilities as other tools do, since we assume there is a record phase.

Other tools [23, 9, 3] automatically explore GUI events to generate high-coverage testing suites. Our focus is different: we require a base execution and aim to alter behavior slightly via SSDA; our approach cannot construct executions and inputs from scratch as the aforementioned approaches do. Also, these tools do not support sensors (GPS, camera, audio) as we do, but sensors play a crucial role in app execution. Finally, our approach can help developers expand their testing suites by generating test cases via cross-app testing, i.e., collecting a trace of sensor data from one app and playing it back on another app if the target app uses the same API.

Test amplification. Test amplification is a useful testing strategy as it generates new test suites by systematically amplifying existing cases. Recent works on concolic testing [24, 13, 21, 5] are examples of amplification. These techniques obtain program execution traces from concrete test cases, then apply symbolic execution to steer the program to visit new paths, thus maximizing code coverage. Test amplification has also been applied on Android testing. Yang et al. [25] test the responsiveness of Android apps by randomly injecting time delays in time-consuming calls such as file I/O, network operations and database operations. Their goal is to check whether developers employ such heavyweight operations in the main (UI) thread which may cause poor responsiveness. Zhang and Elbaum [27] use test amplification to validate exception handling code. Their approach exhaustively explores the space of exceptional behavior of an external resource that is exercised by a test suite, then apply these amplified test cases to check whether the app handles exceptions correctly. We achieve test amplification in another direction: alter the sensor input data, then observe differences in execution. Our cross-app testing is another kind of test amplification as it collects input from one app and applies it to another app which uses the same API. Unlike [25] and [27] which require access to the source code hence have only been applied to open-source apps, we directly instrument code at the bytecode level, which allows the approach to be used on a much wider set of apps.

6. CONCLUSIONS

We have presented two approaches for constructing new test cases for Android apps, based on the observation that

sensor input plays a fundamental role in the construction and execution of smartphone apps. First, new test cases are constructed by taking existing, recorded sensor streams and altering sensor data in a semantically meaningful way; this can outright crash the app or increase coverage. Second, in a cross-app testing approach, one app's execution becomes a test case for many other apps.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1064646.

7. REFERENCES

[1] ImageMagick. http://www.imagemagick.org/script/index.php.

[2] SoX - Sound eXchange. http://sox.sourceforge.net/Main/HomePage.

[3] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In FSE '13.

[4] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, 2012.

[5] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 59:1–59:11, 2012.

[6] Android Developers. MonkeyRunner, August 2012. http://developer.android.com/guide/developing/tools/monkeyrunner_concepts.html.

[7] Android Developers. UI/Application Exerciser Monkey, August 2012. http://developer.android.com/tools/help/monkey.html.

[8] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, 2014.

[9] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 641–660, 2013.

[10] P. Bhattacharya, L. Ulanova, I. Neamtiu, and S. C. Koduru. An empirical analysis of bug reports and bug fixing in open source android apps. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, CSMR '13, pages 133–143, 2013.

[11] J. Callaham. Google says there are now 1.4 billion active Android devices worldwide, September 2015. http://techcrunch.com/2015/09/29/android-now-has-1-4bn-30-day-active-devices-globally/.

[12] H. Dediu. When will there be one billion iOS devices in use?, November 2013. http://www.asymco.com/2013/11/25/one-billion-ios-devices/.

[13] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, 2005.

[14] Google. Google Earth. http://earth.google.com.

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

[16] J. Guyn. Facebook users give iPhone app thumbs down. *Los Angeles Times*, Jul 21 2011. http://latimesblogs.latimes.com/technology/2011/07/facebook-users-give-iphone-app-thumbs-down.html.

[17] C. Hu and I. Neamtiu. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 77–83, 2011.

[18] Y. Hu, T. Azim, and I. Neamtiu. Versatile yet lightweight record-and-replay for android. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 349–366, 2015.

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

[20] A. K. Maji, K. Hao, S. Sultana, and S. Bagchi. Characterizing failures in mobile oses: A case study with android and symbian. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*, ISSRE '10, pages 249–258, 2010.

[21] R. Majumdar and K. Sen. Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 416–426, 2007.

[22] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon. Guitar: An innovative tool for automated testing of gui-driven software. *Automated Software Engg.*, 21(1):65–105, Mar. 2014.

[23] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: Automatic security analysis of smartphone applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 209–220, 2013.

[24] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, 2005.

[25] S. Yang, D. Yan, and A. Rountev. Testing for poor responsiveness in android applications. In *Engineering of Mobile-Enabled Systems (MOBS), 2013 1st International Workshop on the*, pages 1–6, May 2013.

[26] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 89–99, 2015.

[27] P. Zhang and S. Elbaum. Amplifying tests to validate exception handling code. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 595–605, 2012.