Gesto: Mapping UI Events to Gestures and Voice Commands

CHANG MIN PARK, TAEYEON KI, ALI J. BEN ALI, NIKHIL SUNIL PAWAR,
KARTHIK DANTU, STEVEN Y. KO, and LUKASZ ZIAREK,
University at Buffalo, The State University of New York, USA

Gesto is a system that enables task automation for Android apps using *gestures* and *voice commands*. Using Gesto, a user can record a UI action sequence for an app, choose a gesture or a voice command to activate the UI action sequence, and later trigger the UI action sequence by the corresponding gesture/voice command. Gesto enables this for existing Android apps without requiring their source code or any help from their developers. In order to make such capability possible, Gesto combines bytecode instrumentation and UI action record-and-replay.

To show the applicability of Gesto, we develop four use cases using real apps downloaded from Google Play—Bing, Yelp, AVG Cleaner, and Spotify. For each of these apps, we map a gesture or a voice command to a sequence of UI actions. According to our measurement, Gesto incurs modest overhead for these apps in terms of memory usage, energy usage, and code size increase. We evaluate our instrumentation capability and overhead using 1,000 popular apps downloaded from Google Play. Our result shows that Gesto is able to instrument 94.9% of the apps without any significant overhead. In addition, since our prototype currently supports 6 main UI elements of Android, we evaluate our coverage and measure what percentage of UI element uses we can cover. Our result shows that our 6 UI elements can cover 96.4% of all statically-declared UI element uses in the 1,000 Google Play apps.

CCS Concepts: • **Human-centered computing → Ubiquitous and mobile computing systems and tools**; *User interface management systems*;

Keywords: gestures; voice commands; record and replay; mapping UI events;

ACM Reference Format:
Chang Min Park, Taeyeon Ki, Ali J. Ben Ali, Nikhil Sunil Pawar, Karthik Dantu, Steven Y. Ko, and Lukasz Ziarek. 2019. Gesto: Mapping UI Events to Gestures and Voice Commands. In *Proceedings of the ACM on Human-Computer Interaction*, Vol. 3, EICS, Article 5 (June 2019). ACM, New York, NY. 22 pages. https://doi.org/10.1145/3300964

1 INTRODUCTION

Users of mobile devices have long been locked into hierarchical user interfaces displayed on device screens. Users often find themselves clicking through several menus and buttons to perform a given task such as playing music, checking news, closing background apps, and others. This is inefficient, attention-hungry, and ultimately restrictive. It requires users to perform multiple UI actions before finishing a task. It also requires full attention from users since they need to look at their device's screen while interacting with the interface. It ultimately makes it difficult or even impossible to

Authors' address: Chang Min Park; Taeyeon Ki; Ali J. Ben Ali; Nikhil Sunil Pawar;
Karthik Dantu; Steven Y. Ko; Lukasz Ziarek, {cpark22,tki,alijmabe,npawar,kdantu,stevko,lziarek}@buffalo.edu,
University at Buffalo, The State University of New York, Department of Computer Science and Engineering, USA.

2573-0142/2019/6-ART5 $15.00
https://doi.org/10.1145/3300964

use a mobile device when the device is out of reach, its screen is not viewable, and/or when a user cannot completely disengage from her other activities to interact with her device.

Recent developments in gesture-based interfaces [11, 28, 32, 39, 44, 45, 48, 49] and voice commands [12, 14, 22, 35] are rapidly improving this status quo. These new interfaces provide flat, rather than hierarchical, interfaces; users can perform a complicated UI task that requires multiple UI actions with a single gesture or a voice command. These interfaces free their users from the requirement of looking at a device screen, broadening the set of possible interactions between users and their devices.

However, users are still not able to fully enjoy the benefits of gesture-based interfaces and voice commands. This is because it requires support from app developers. Although libraries do exist (e.g., Intel Context Sensing SDK [26], Sensey [46], Google Assistant SDK [23], Alexa SDK [13], Cortana SDK [36], and SiriKit [15]), many existing apps currently do not leverage them. While these libraries provide excellent gesture and voice recognition, their integration to individual apps is left to the respective developers. This means that app developers need to either pre-define a set of tasks that users can perform with gestures and voice, or develop a flexible mechanism for users to define custom tasks to be triggered by gestures or voice. The former restricts users to a limited set of tasks; the latter can potentially be non-trivial. In both cases, it is an extra development burden which many app developers do not venture into.

To simplify such integration, we have developed Gesto, an end-to-end system that enables task automation with gestures and voice commands for existing apps. Gesto allows a user to record a sequence of UI actions, map the sequence to a gesture or a voice command, and later replay the sequence with the mapped gesture or voice command. Gesto provides flexibility to users since they can define customized UI action sequences and map them to gestures or voice commands. Gesto eliminates the additional development burden on the developers as it automatically adds the task automation functionality to an existing app without requiring app source code or developer intervention. Gesto supports a wide range of gestures (e.g., Shake, Chop, Tap) and custom voice commands that users choose. Since it makes use of well-known third party libraries for gesture/voice recognition, this set of gesture/voice commands can be expanded by integrating future third-party libraries that perform these tasks.

Gesto enables such task automation by combining two techniques—bytecode instrumentation and UI record-and-replay. First, Gesto's bytecode instrumentation allows functionality injection without the need for source code. Thus, a developer can run Gesto instrumentation before releasing an app in order to add task automation features. Gesto automatically captures UI elements and frees the developer from cumbersome engineering work. Similarly, a (power) user can download an app, run the app through Gesto on a desktop, and install the app on a mobile device to start using gestures and voice commands to automate custom tasks. Since bytecode instrumentation for UIs is inherently specific to platforms and languages, we have built Gesto for Android apps and have solved Android specific challenges. Gesto does not require root access or a custom OS unlike previous systems [7, 20, 24, 25, 31, 40].

Second, Gesto's UI record-and-replay enables a user to record a sequence of UI actions and later replay it. In order to achieve this, Gesto analyzes an app's bytecode, identifies UI objects and enables recording and replaying of UI object interactions. Since Gesto's goal is task automation, it implements a full UI record-and-replay mechanism rather than a mobile deep link mechanism, where the goal is creating a "bookmark" of a UI page for an app [16, 34]. For example, if a user wants to automate multiple device management tasks at once (all provided by a single app such as AVG Cleaner [37]), e.g., cleaning cache and app history, Gesto is able to automate such a sequence while deep links cannot.

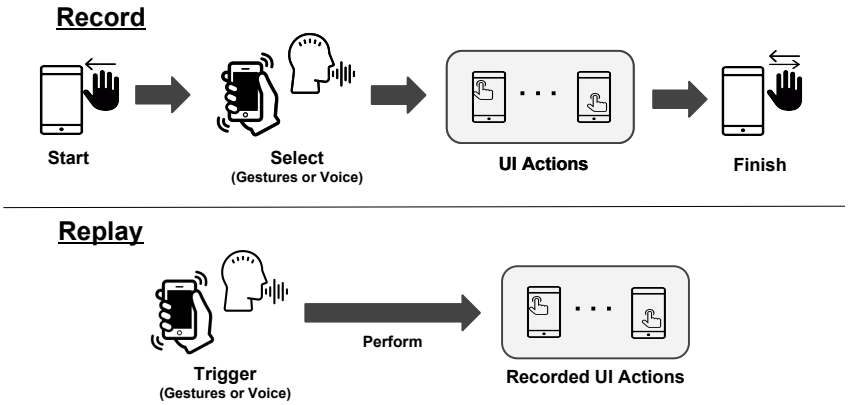


Fig. 1. Gesto Workflow

Although our overall contribution is the design of an end-to-end system, our UI record-and-replay technique also has contributions on its own. First, our technique works at the level of UI objects rather than pixel coordinates. Thus, our technique is not sensitive to device screens; once a recording is done, it can potentially be replayed across different devices that a user has. Second, we address a key problem that arises when recording and replaying at the UI object level on Android—that there has to be a way to uniquely identify a UI object in order to correctly record the use of it and replay it later. It turns out that this is a challenging problem, since Android does not require every UI object to have an ID and real-world apps often do not assign an ID for every UI object. To the best of our knowledge, all previous object-level record-and-replay approaches (e.g., using `AccessibilityService` on Android) require every UI object to have an ID. As a result, they do not correctly work with real-world apps. In Gesto, we implement a mechanism that solves this problem. We detail how previous approaches rely on object IDs in Section 7.

In order to show the applicability of Gesto, we have developed four use cases—(i) playing music on Spotify using a gesture, (ii) searching for keyword-based news (e.g., "latest price of Bitcoin") on Bing using voice, (iii) searching for nearby businesses on Yelp using a gesture, and (iv) performing device clean-up tasks (e.g., cleaning cache and app history) on AVG Cleaner using a gesture. In all these cases, Gesto provides user-defined task automation with modest overhead in terms of memory, energy consumption, and code size increase. We have also evaluated our instrumentation capability by downloading 1,000 popular apps from Google Play and instrumenting them. Our result shows that Gesto can instrument 94.9% of the apps (949 apps).

Overall, our contributions are as follows.

- We develop an *end-to-end* system that enables task automation using gestures and voice into existing Android apps automatically. Our system does not require app source code or developer involvement.
- We develop a UI record-and-replay technique that works at the level of *UI objects* on Android, even when an app does not have an ID for every UI object.
- We show that our system is *applicable* by demonstrating four use cases. We also show that our system is *light-weight* by showing that it incurs modest overhead.

2 USAGE MODEL

Before discussing the design and implementation of Gesto, we first describe our usage model to help the reader understand how Gesto operates.

Category	UI Element	Callback Method Example
Button	Button RadioButton ToggleButton	onClick onRadioButtonClicked onToggleClicked
Clickable View	TextView ImageView	onClick onClick
Text Box	EditText	onTextChanged
Check Box	CheckBox Switch	onCheckBoxClicked onCheckedChanged
Bar Type	SeekBar RatingBar	onProgressChanged onRatingChanged
Item Selection	ListView Spinner TimePicker NumberPicker DatePicker	onItemClick onItemSelected onTimeChanged onValueChange onDateChanged

Table 1. User Interactive Element Examples

```
// In a layout XML file
<Button android:text="button"
    ...
    android:onClick="example_method" />

// In an app class
void example_method(View v){
    // Developer Code
}
```

Fig. 2. Statically Declaring a UI Element in a UI XML file and Registering a Callback

```
Button b = (Button) findViewById(R.id.button);
b.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
      // Do something in response to button click
    }
});
```

Fig. 3. Programmatically Registering a Callback

Usage at App Development/Instrumentation Time: Gesto is a static bytecode instrumentation tool that injects new bytecode into an existing Android app and rewrites some parts of the app's bytecode (more details on this in Section 3.6). It takes an existing Android app (a `.apk` file) as input, instruments the app, and produces another version of the app that is now capable of task automation using gestures and voice. A few ways are possible to use this tool: (i) a developer can run it on the developer's machine to transform an app that the developer has written before releasing it to an online app market, (ii) a user can run it on the user's machine to transform an app that the user downloads from an online app market such as Google Play, and (iii) a third party can provide a web service that runs the tool in the cloud.

Once instrumented, a developer can deploy the new version of an app on Google play store, or a user can install the app on a regular Android device just like any other Android app. It does not require any special privilege (such as rooting) to work properly.

Usage at Run Time: Once an app is Gesto-enabled and installed on an Android device, a user can automate custom tasks using gestures and voice commands. In order to create a UI task to be automated for an app, a user needs to (i) open the app, (ii) perform the "start recording" command (currently, a pre-registered gesture), (iii) perform a gesture or record a voice command that will be used as the trigger for replaying, (iv) record a sequence of UI actions to be automated, and (v) perform the "stop recording" command (another pre-registered gesture). In our current prototype, waving once near a device starts recording; waving twice near a device stops recording. Figure 1 depicts these run-time workflows.

After a user finishes mapping a sequence of UI actions to a gesture or voice command for an app, the user can simply perform the gesture or speak the voice command to trigger the action sequence when the app is open. We can further automate app opening by instrumenting the main app launcher of the user's device using Gesto, since app launchers are regular Android apps. However, Gesto currently does not support cross-app task automation. We discuss this more in Section 6.

3 SYSTEM DESIGN AND IMPLEMENTATION

In this section, we discuss how Gesto automatically enables task automation using gestures and voice for existing apps. First, we provide background on Android UI that is necessary to understand how Gesto works. We then discuss how our record-and-replay mechanism works, how we map recorded UI action sequences to gestures or voice commands, and how to inject such funcitonality into an existing app.

3.1 Background on Android UI

We discuss the types of UI elements that Android provides, how to display and handle them, how to identify them, and finally, the main Android app window, `Activity`. The purpose of this discussion is an overview of Android UI pertaining to Gesto; it is not meant to be comprehensive.

Types of UI Elements: Android app developers use two main classes to build UI elements —`View` and `ViewGroup`. `View` is a UI object drawn on a device's screen. Examples include `Button`, `ImageButton` and others. `ViewGroup` is a subclass of `View` but is a special `View`—it is a container that holds multiple `Views` or other `ViewGroups`. Examples include `LinearLayout`, `ListView` and others. Since a `ViewGroup` can contain other `Views` and `ViewGroups`, they can form a `View` tree, where the root is always a `ViewGroup`.

Table 1 lists popular UI elements on Android in different categories. Other less popular UI elements, which are often derived as subclasses from elements listed above, are omitted from the table. For example, `ImageButton` is a subclass of `ImageView` and works in a similar fashion. These elements are all subclasses of `View`. The current version of Android platform APIs (API 27) has 75 UI element classes in total [21].

Displaying a UI Element: To make UI elements visible on a screen, a developer can either declare the elements statically using XML files (e.g., `layout.xml`) or add them dynamically at run time by writing app code. Declaring a UI element in an XML file is straightforward, and a developer can do it by just adding the element's specification in an XML file. In Figure 2, lines 2-4 show an example.

Alternatively, developers can create `Views` and add them at run time in their app —a developer can create a new `View` object (e.g., a `Button`), and add it to a `ViewGroup` (e.g., a `LinearLayout`) by calling `ViewGroup.addView()`. Using this method, a developer can dynamically show different UI elements depending on app logic. For example, a developer can show a list of nearby restaurants using this dynamic method.

Handling UI Events: A developer handles UI events via callback methods. In each callback method, a developer provides app logic that gets invoked when a UI event happens. Android provides two ways to register a callback method. The first way is through an XML file. Figure 2 lines 2-4 show an example. `example_method()` is declared in the file as a method to handle a click event for a particular button (line 4). After a callback method is declared and registered this way, a developer provides app logic that handles the button click (e.g., in lines 7-9) as part of app code. The second way to register a callback method is to programmatically write callback registration app code. For this purpose, Android provides pre-defined classes and interfaces that developers can use to register and implement callback methods. Figure 3 shows an example, where `setOnClickListener()` is used on a `Button` to register a `OnClickListener.onClick()` callback. As mentioned, these methods and classes are all defined by Android, and a developer uses or implements them. Table 1 shows some of the callback methods for UI elements defined by Android. They are customized for different types of UI elements.

Identifying a UI Element: Each `View` *may* have an ID to identify a specific `View` within a `View` tree. However, having an ID is *not* a requirement, and not all developers assign `View` IDs in practice. It turns out that this is a challenge that Gesto needs to address as we detail later. A developer can assign an ID either in an XML file when declaring a UI element, or dynamically by calling `View.setId()`. Also, `View` IDs do not need to be globally unique. It only needs to be unique within the same `View` tree.

As mentioned in Section 7, all existing UI object based approaches [5–7, 9, 10, 16] using `View` IDs cannot correctly handle cases where the `View` elements do not have IDs assigned. In order to understand if this is an actual problem, we have downloaded 324 apps from 34 different categories

Category	# of Apps	# of View Elements		Percentage of ID not Assigned Views
		ID Assigned	ID not Assigned	
Library and Demo	17	88	24	21.43%
Art and Design	17	53	13	19.70%
Events	16	47	19	28.79%
Beauty	15	69	21	23.33%
Social	14	18	13	41.94%
Comics	13	74	40	35.09%
Photography	12	72	20	21.74%
Books and Reference	11	49	13	20.97%
Food and Drink	10	42	23	35.38%
Personalization	7	43	21	32.81%
Others(24)	192	563	97	14.70%
Total	**324**	**1118**	**304**	**21.38%**

Table 2. View ID Percentage for 324 Popular Apps

from Google Play and tested them. We have run the apps for 2 minutes clicking on elements at random and have logged all performed `View` elements' IDs. In order to do this, we use bytecode instrumentation to inject logging statements for all UI callbacks to each app we test. This way, we can print out if a UI object has an ID assigned when used. Table 2 shows the number of apps for each category, the number of visited `View` elements separated by whether each one was assigned an ID or not, and the percentage of view elements not assigned an ID for each category as well as across all apps. The table shows that the overall percentage of views not assigned an ID is 21.38%. This represents 1 out of 5 `Views` that cannot be handled by existing UI object based record-and-replay approaches [5–7, 9, 10, 16]. Also, 111 apps out of total 324 apps contain at least one `View` which is not assigned an ID. Therefore, UI object based record-and-replay tools are not able to replay these apps correctly. We note that our testing approach does not explore all possible paths to use UI objects since it is a random testing. Thus, our results are conservative, i.e., there could be more `Views` without an ID in reality. In Section 3.4, we describe in detail our solution to this challenge.

Activity: An `Activity` on Android is a UI window object that contains `Views`. On a mobile device, it is typically presented as a full window, but smaller window sizes are possible. A typical app creates multiple UI window objects; for example, a chatting app can have a login `Activity`, a contact list `Activity`, and a chatting conversation `Activity`. In order to define an `Activity` and its behavior, an app developer needs to write a subclass of the Android's base `Activity` class. This base `Activity` class has multiple pre-defined callbacks such as `onCreate()`, `onStart()`, `onResume()` and others that are invoked according to the lifecycle of the app. Developers can override these callbacks to customize an app's behavior during those lifecycle events.

3.2 Gesto Design Overview

We now present the design of Gesto from a high-level viewpoint. Unfortunately, discussing a UI record-and-replay approach requires platform-specific details, since each platform has its own way of handling UIs and there are distinctive features one can leverage from different platforms. Due to this reason, we only give a brief overview of our design here and delve deeper into the design with Android-specific details in the rest of this section.

Gesto consists of the following four mechanisms.

- **Capturing UI events:** Gesto instruments an app so that it can capture UI events at run time. The instrumentation injects logging code into UI event callback methods, which is sufficient for capturing UI events. More details can be found in Section 3.3.

```
void onClick(View v) {
    Logger.logOnClick(v);
    // Original Developer Code
}

void onItemClick(AdapterView parent,
    View view, int position, long id) {
    Logger.logOnItemClick(
        AdapterView parent, View view,
        int position, long id);
    // Original Developer Code
}
```

```
void onFocusChange(View v, boolean hasFocus){
    Logger.logOnFocusChange(
        View v, boolean hasFocus);
    // Original Developer Code
}

void onTextChanged(CharSequence s,
    int start, int count, int after){
    Logger.onTextChanged(
        CharSequence s, int start,
        int count, int after);
    // Original Developer Code
}
```

Fig. 4. Logging UI Callback Examples

- **Recording UI events:** Starting from the main (first) activity of an app, Gesto records four types of information: (i) the activity where an UI event occurs, (ii) the name of a UI event callback method, (iii) the unique ID of a UI element, and (iv) the timestamp of a UI event. More details are in Section 3.4.
- **Mapping UI events:** Gesto stores a sequence of UI events along with a selected gesture or voice command. More details can be found in Section 3.5.
- **Replaying:** Gesto starts replaying from the main (first) activity, and it performs recorded UI actions using the information previously stored. We discuss more details in Section 3.5.

3.3 Capturing UI Events

Capturing UI events is the first component necessary for recording UI events. The basic mechanism that we use is injecting logging code into UI event callback methods via bytecode instrumentation. This way, as long as we identify all callbacks, we can reliably capture UI events.

Care must be taken when identifying callbacks because there are two ways to define callbacks—declaratively in a XML file and programmatically in app code. The difference is that when a developer declares a callback for handling a UI event in the XML file, the developer can use a *custom* name. However, when a developer programmatically registers a callback in app code, the developer needs to implement a known method defined by Android. To identify all custom callbacks declared in UI XML files, we parse the UI XML files and get the list of UI elements and their custom callbacks. To identify callbacks registered in app code, we statically analyze app bytecode and look for known callback names defined by Android.

Once we identify the UI event callbacks in an app, we inject logging code as the first line of each callback method. Figure 4 shows some examples of our code injection. [1] The first example, `onClick()`, is perhaps the most popular way to handle a click event for many different types of user-interactive `Views`, such as `ButtonView`, `ImageView`, `TextView`, etc. The second example, `onItemClick`, is also popular and used for `ListView` that shows a list of clickable items. The third and fourth examples, `onFocusChange()` and `onTextChanged()` are used to handle text input via `EditText`. `EditText` is a text field UI object for entering text. When it receives focus, `onFocusChange()` is called. Then, `onTextChanged()` is called every time text is entered or deleted. As shown in all four examples, our logging code passes all parameters as well, since they are necessary for replay later.

Our current prototype provides support for 6 UI elements, `TextView`, `ImageView`, `ListView`, `Button`, `ImageButton`, and `EditText`. In Section 5.2, we show our coverage analysis with these 6 elements, i.e., we analyze what percentage of UI element uses we can cover with the 6 elements. Our result shows that we can cover 96.4% of all statically-declared UI element uses in 1,000 Google Play apps.

[1] We show Java code for illustration purposes; we do not need source code to inject our code.

3.4 Recording UI Events

With the capturing mechanism described above, we record UI events for replay later. Our recording always starts from the first `Activity` of an app; when a user starts recording (via a pre-defined gesture), we revert to the main (first) `Activity` of the app and let the user perform UI actions to be recorded. We discuss the implementation details of this in Section 3.6.

There are four types of information we record for each UI event—(i) the `Activity` object on which the UI event is performed, (ii) the callback method that handles the UI event, (iii) the UI object information (e.g., its ID), and (iv) the timestamp of the event. We store all four types of information in permanent storage for later retrieval. We discuss each type in detail below.

Activity: As mentioned earlier, An `Activity` on Android is a UI window object that contains `Views`. We record the `Activity` for each UI event for two reasons. One reason is our debugging, since we find it helpful to know how `Activity` transitions are happening during record and replay. The other is a more practical one, that transitioning from one `Activity` to another takes a variable amount of time, which makes it difficult to replay recorded UI actions correctly. For example, if an `Activity` transition takes longer during replay than recording, then there is a chance that we perform a UI action before the new `Activity` is completely drawn. In order to avoid this, we first record each `Activity`, and also record a *relative* timestamp for the UI event, i.e., the duration between when an `Activity` is completely drawn and when the UI event is performed. We discuss in Section 3.6 how we detect when an `Activity` is completely drawn in our implementation.

Callback: We record the callback method that is invoked for a UI event. This information identifies which UI action is performed, e.g., a click, a selection, etc. We use this information to perform that action during replay.

UI Object Information: We record the information of the UI object on which a UI action is performed. For all UI objects, we record IDs. In addition, we record other necessary information depending on UI object types. For example, Gesto records the text in a `EditText` element and the row that was clicked in `ListView`. All this additional information is used for replay.

A major challenge in identifying UI elements was unique IDs. As mentioned earlier, assigning an ID to a UI element is not a requirement, and many developers do not assign IDs for their UI elements. However, in order to identify a UI object and correctly record and replay, we need to have a unique ID for each UI object. Thus, we have developed a mechanism to assign an ID for a UI object ourselves when the original developer does not assign an ID to it. Since there are two ways to register a UI element (i.e., in a UI XML file and in app code), we need to handle both cases. It is not difficult to assign an ID to a UI element declared in a UI XML file; we just need to parse all XML files, find out if there is any UI element that has no ID, and assign an ID using the correct XML attribute.

However, UI objects added at run time via an `addView()` call are more challenging. This is because there are two constraints. The first constraint is that an ID assignment mechanism needs to be deterministic across different runs. If IDs that we assign do not survive across app terminations and restarts, then we cannot record and replay correctly. The second constraint is that Android requires that an ID for a UI object be unique within a single `View` tree.

We considered various properties that a UI object has as ID candidates. For example, we considered using a hash of a UI object's pixel coordinates as an ID. However, there are cases where two different UI objects are drawn on the same position within a single `View` tree (e.g., clicking a button turns into another button). In these cases, we cannot use pixel coordinates as an ID as they are not unique. We also considered a UI object's Java hash (`Object.hashCode()`) as an ID. However, we discovered that a Java hash is a run-time value, and changes across different runs.

```
: Select a gesture type...
: ============ Recording(shake) starts. =============
: Activity: com.spotify.music.MainActivity
: + android.widget.TextView(assigned ID: 2131363048)
: + android.widget.TextView(assigned ID: 2131363049)
: + android.widget.TextView(assigned ID: 2131363050)
: + android.widget.TextView(assigned ID: 2131363051)
: - onItemClick/16908298_2_1/3218
: Activity: com.spotify.music.MainActivity
: + android.view.View(assigned ID: 2131363048)
: + com.spotify.android.paste.widget.HeaderView$2(assigned ID: 2131363049)
: + com.spotify.android.paste.widget.ViewPagerIndicator(assigned ID: 2131363050)
: + android.widget.ImageView(assigned ID: 2131363051)
: + com.spotify.android.paste.widget.HeaderView(assigned ID: 2131363052)
: + com.spotify.mobile.android.ui.prettylist.PrettyHeaderView(assigned ID: 2131363053)
: + com.spotify.mobile.android.ui.prettylist.StickyListView(assigned ID: 2131363054)
: + android.widget.Button(assigned ID: 2131363055)
: - onClick/2131363055/2640
: ================================================
```

Fig. 5. Record Log of Spotify App

```
: ---------- Replaying(shake) starts. -------------
: Activity: com.spotify.music.MainActivity
: + android.widget.TextView(assigned ID: 2131363048)
: + android.widget.TextView(assigned ID: 2131363049)
: + android.widget.TextView(assigned ID: 2131363050)
: + android.widget.TextView(assigned ID: 2131363051)
: - onItemClick/16908298_2_1/3218
: Activity: com.spotify.music.MainActivity#1
: + android.view.View(assigned ID: 2131363048)
: + com.spotify.android.paste.widget.HeaderView$2(assigned ID: 2131363049)
: + com.spotify.android.paste.widget.ViewPagerIndicator(assigned ID: 2131363050)
: + android.widget.ImageView(assigned ID: 2131363051)
: + com.spotify.android.paste.widget.HeaderView(assigned ID: 2131363052)
: + com.spotify.mobile.android.ui.prettylist.PrettyHeaderView(assigned ID: 2131363053)
: + com.spotify.mobile.android.ui.prettylist.StickyListView(assigned ID: 2131363054)
: + android.widget.Button(assigned ID: 2131363055)
: - onClick/2131363055/2640
: ------------------------------------------------
```

Fig. 6. Replay Log of Spotify App

Based on these experiences, we have developed the following mechanism. We first assign all IDs for static UI elements declared in UI XML files. We then pick a number that is greater than all IDs used in the UI XML files. We use this number as the base number for all dynamically-added UI objects.

Whenever a new UI object is added (via `addView()`) without an ID, we increment the base number by one and assign it as the ID for the newly-created object. On Android, there is a single `View` tree per `Activity`, so this mechanism creates a unique ID for each UI object in a `View` tree. Also, since we retrieve the base number every time an `Activity` transition happens, we effectively reset the ID counter for a new `Activity`. This way, our IDs do not increase arbitrarily, and we can avoid potential conflicts with static IDs due to overflow.

This mechanism relies on an assumption that all dynamically-added UI objects are created in the same order across different runs. If this assumption does not hold for an app, then we cannot correctly record and replay. Our experience with real apps downloaded from Google Play indicates that this assumption holds for real apps that we use for our use case studies (in Section 4). For example, Figure 5, 6 compares a record run of Spotify to a replay run. Both logs show the order of UI object creations for each run, and they are identical (indicated in blue). As a result, we assign the same IDs across two runs. The red arrow shows that one of the dynamically-created buttons has been clicked and recorded correctly. In Section 6, we further discuss the implications of this mechanism.

Using Sensor/Services	Features
Base Sensor(Accelerometer, Proximity, etc.)	Shake, Chop, Wave, Flip, and 14 more
Touch Gesture	Tap, Swipe, LongPress, and 7 more
SpeechRecognizer(Platform Class)	Dynamic Voice Recognition

Table 3. Mappable Features Provided by Gesto

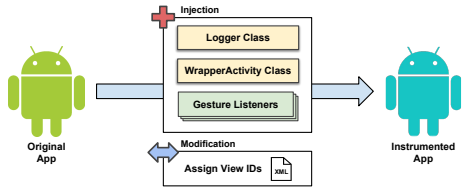


Fig. 7. Overview of Gesto Implementation

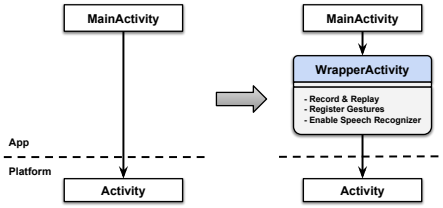


Fig. 8. WrapperActivity Class

Timestamp: The last piece of information we record is a timestamp for each UI event. As mentioned earlier, this timestamp is relative to each `Activity` transition completion time. We record the duration between a new `Activity` completing rendering and when a UI event is performed. Then, Gesto uses it to trigger each UI event at the exact time when replaying. Some UI events require time to load (e.g., waiting for data over the Internet), and the app may either crash or misbehave if it triggers next UI events during the loading. A timestamp is important to prevent this potential problem, and it also gives users the necessary control of when UI actions should occur.

3.5 Mapping UI Events and Replaying

As mentioned in Section 2, recording a sequence of UI actions starts with a pre-registered gesture which signals the start of a recording. After that, a user performs a gesture or speaks a voice command that will be used as the trigger for replaying. Then, the user performs UI actions to be automated and ends the recording with another pre-registered gesture. While the user performs the UI actions, Gesto stores the sequence of the UI actions with the name of a selected gesture or a voice command. This process is essentially storing a mapping, and our system reads and follows the sequence when replaying. Table 3 shows the list of gestures and voice commands Gesto currently supports. We use Sensey [46], an open-source gesture recognition library. We also use `SpeechRecognizer` that Android provides, which allows custom voice commands to be recorded and used. We detail how we inject this code in Section 3.6.

When a user starts replaying, Gesto always goes back to the first `Activity` of the app. Since our recording always starts from the first `Activity`, we replay a UI sequence from the same first `Activity`. For each event, we wait until an `Activity` is completely drawn and perform the event based on the timestamp recorded. When performing the event, we call the appropriate method based on the callback type recorded. For example, for a `onClick()` callback, we use `performClick()`; for a `onItemClick()` callback, we use `performItemClick()`; for a `onFocusChange()` callback, we use `setFocusable()` callback.

3.6 Implementation

Gesto uses Soot [47] to inject custom classes and rewriting bytecode, and apktool [2] for re-building apps with the injected classes and rewritten bytecode. Figure 7 shows an overview of Gesto implementation. Gesto modifies all UI XML files to assign UI object IDs, and injects three components—`Logger` class, multiple gesture listeners, and `WrapperActivity` class. `Logger` has

all the methods for capturing UI events (Section 3.3) and recording them (Section 3.4). Our gesture and voice listeners use Sensey [46] and Android's `SpeechRecognizer` to detect gestures and voice (Section 3.5).

`WrapperActivity` is a class we inject between Android's base `Activity` class and each of the custom `Activity` classes of an app. As mentioned earlier, an app developer needs to extend Android's base `Activity` class and write a custom `Activity` class (e.g., `MainActivity`) to customize the behavior of an `Activity` (e.g., showing a login page). Figure 8 shows our injection.

There are two reasons for this injection. First, it allows us to monitor which state an app is in. As mentioned earlier, Android invokes `Activity` callbacks to signal app's state transitions—`onCreate()` when an `Activity` is first created, `onStart()` when the `Activity` is starting, `onResume()` when the `Activity` is completely drawn and ready to handle user interactions, etc. If we inject `WrapperActivity` between the base `Activity` and an app `Activity`, we can override all callbacks and monitor these state transitions. We mainly override `onResume()` since it signals that an app's `Activity` is completely drawn and ready to handle user interactions. Using `onResume()`, we mark the app state to be "ready" for either record or replay.

The second reason is that Android requires UI interaction capabilities (e.g., gestures and voice) to be registered for each `Activity` that an app creates. In other words, Android requires that a developer define all UI interactions per `Activity`. Once again, `WrapperActivity` allows us to tap into each app `Activity`, and we register our gesture and voice recognition capabilities using `WrapperActivity` for each `Activity` created. `WrapperActivity` also triggers an app to go back to the first `Activity` whenever a recording run or a replaying run starts. This is done by starting an app's main `Activity` (called the launcher `Activity`).

4 USE CASES

We demonstrate the usability of Gesto by describing four use cases. These examples are meant to be representative of Gesto's capabilities.

4.1 Simplifying UI Interaction

There are several circumstances under which users cannot interact with the UI on mobile devices, e.g., when exercising, on a rainy day, and when wearing gloves. Further, disabled people have difficulties in interacting with the UIs of mobile devices. Many approaches attempt to solve this problem; for example, there are touchable gloves and earphones with additional functions for control. However, these approaches require users to purchase additional products, and even when users purchase them, the mileage may vary in their utility. Gesto can solve this problem without additional hardware by instrumenting individual apps.

Consider Spotify music player app as an example. A music player app is typically used to provide background music while a user performs other tasks. Thus, there is a high probability that users are occupied (such as when running or driving) and unable to interact with the app via the UIs of their mobile devices. Although today's earphones come with extra functionalty for play, stop, and volume control, they are limited in many ways. The provided functions are set statically which means users cannot change their mapping or define custom mappings to other actions in an app. Further, they can enable just one action and not a complete sequence. Gesto's instrumentation enables both dynamic mapping and mapping multiple UI actions with single gesture or word.

Figure 9 shows how a user can leverage Spotify instrumented by Gesto. In fact, it is a real screenshot of Spotify after we instrument it with Gesto. We have mapped a shake gesture to a sequence of two UI actions that plays one of the playlists that user have created. There are two logs in Figure 9. First, the recording log shows recorded activities, recorded UI actions with their view ID, information which is used in replay, and time intervals between each UI action. Second, the

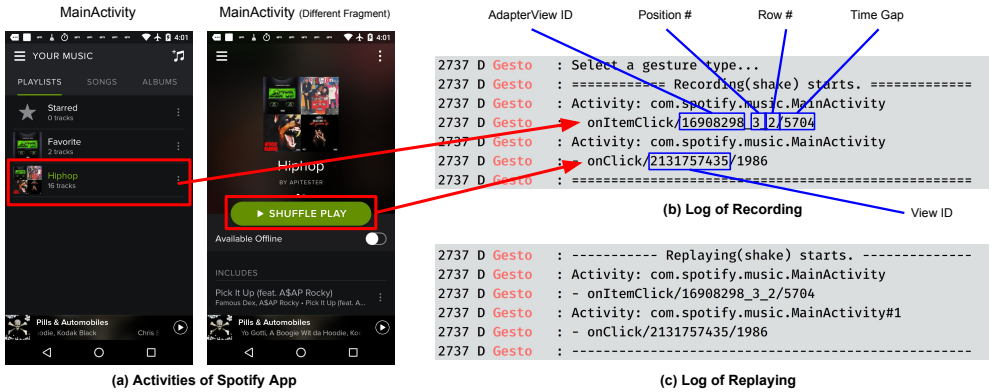


Fig. 9. Record and Replay of Spotify App

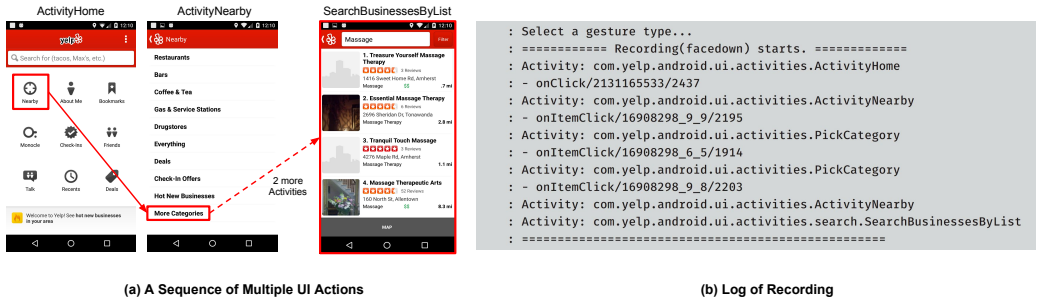


Fig. 10. Record of Yelp App

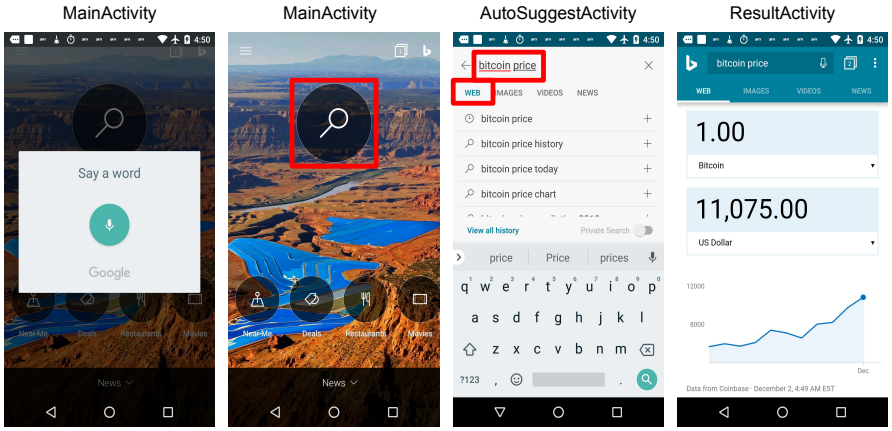
replay log shows that those recorded activities and UI actions are replayed in order. Gesto allows us to use other gestures/voice commands to play other playlists as well. Finally, users are able to perform such mappings dynamically at run time and have these mappings persist across reboots.

4.2 Executing Long Sequence of UI Actions

As a second example, consider the fact that today's applications are content-rich with an involved, hierarchical UI. While this makes for more compelling applications, navigating to an element of interest requires executing a long sequence of UI actions. Some apps provide a bookmark feature which helps users to get to desired content that has been visited before. However, such bookmarks are limited to static content. Gesto allows a user to define a dynamic mapping to a sequence of UI events.

We have chosen Yelp to demonstrate this use case. Yelp provides information about local businesses along with customer reviews for these businesses. Users typically use Yelp to find a local restaurant or find the best-rated local coffee shop. However, accomplishing such a task requires navigating through the app and performing several UI actions, which can quickly get involved. With Gesto, a user can easily automate the task.

To demonstrate this, we have instrumented Yelp using Gesto and performed a search for a masseur. Figure 10 shows a sequence of UI events mapped to the face down gesture. Figure 10 (b) shows the recorded log of actions. Figure 10 demonstrates an example of executing a sequence of four UI events through five activities using the face down gesture. We note that such a sequence



(a) Activities of Bing

```
7146 D Gesto   : Select a gesture type...
7146 D Gesto   : ============ Recording(Voice[Bitcoin]) starts. =============
7146 D Gesto   : Activity: com.microsoft.clients.bing.activities.ResultActivity
7146 D Gesto   : Activity: com.microsoft.clients.bing.app.MainActivity
7146 D Gesto   : - onClick/2131755949/2933
7146 D Gesto   : Activity: com.microsoft.clients.bing.activities.AutoSuggestActivity
7146 D Gesto   : - onFocusChange/2131755845/true/106
7146 D Gesto   : - onTextChanged/2131755845/b/1876
7146 D Gesto   : - onTextChanged/2131755845/bi/2263
7146 D Gesto   : - onTextChanged/2131755845/bit/2704
7146 D Gesto   : - onTextChanged/2131755845/bitc/3268
7146 D Gesto   : - onTextChanged/2131755845/bitco/3696
7146 D Gesto   : - onTextChanged/2131755845/bitcoi/3956
7146 D Gesto   : - onTextChanged/2131755845/bitcoin/4418
7146 D Gesto   : - onTextChanged/2131755845/bitcoin/4792
7146 D Gesto   : - onTextChanged/2131755845/bitcoin /4816
7146 D Gesto   : - onTextChanged/2131755845/bitcoin p/5375
7146 D Gesto   : - onTextChanged/2131755845/bitcoin pr/6087
7146 D Gesto   : - onTextChanged/2131755845/bitcoin pri/6699
7146 D Gesto   : - onTextChanged/2131755845/bitcoin pric/7369
7146 D Gesto   : - onTextChanged/2131755845/bitcoin price/7819
7146 D Gesto   : - onClick/2131755846/1072
7146 D Gesto   : Activity: com.microsoft.clients.bing.activities.ResultActivity
7146 D Gesto   : ======================================================
```

Voice Recognized

(b) Log of Recording

Fig. 11. Record of Bing App

could be arbitrarily complicated (within memory limits), and therefore offers the potential to significantly reduce cumbersome UI actions.

4.3 Frequently Used UI Events

The third use case demonstrates simplification of frequently used UI events. A user may repeat a set of UI events periodically (potentially many times a day) which could be annoying to the user. Such interaction gets worse if it involves the user entering text into a text box given the ergonomics of the mobile keyboard. Gesto can be used to automate these cases replacing such a sequence with a gesture/voice.

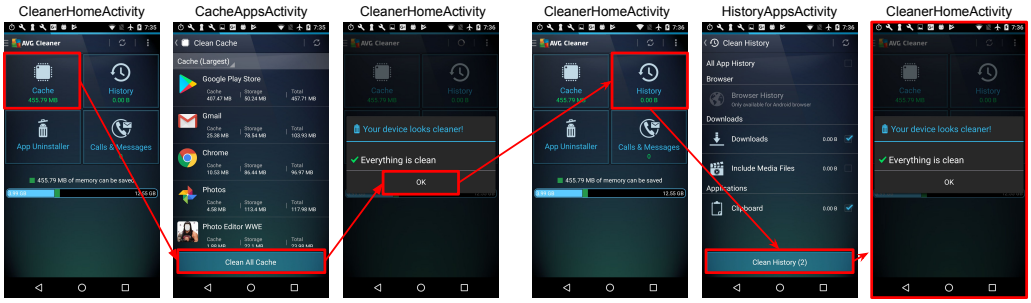


Fig. 12. Multiple Task of AVG Cleaner App

We use the Bing app to demonstrate this use case. Bing is search app from Microsoft that provides a wide variety of search services. Users can search the same keyword at different times to check the latest results such as news on a specific topic, stock price and currency conversion. Shown in Figure 11, we have recorded searching the Bitcoin price with a word "Bitcoin" on Bing app. Figure 11 (a) shows voice recording dialogue at the very first activity. Using a unique gesture (two-finger tap in our case), users can get to voice recording dialogue. Recording starts right after a word is entered by users. The log in Figure 11 (b) shows Gesto's ability to handle text boxes (`EditText`). It captures all text changes and saves each change. In the current example, uttering the voice command "Bitcoin" replays the UI sequence resulting in display of the latest Bitcoin price.

4.4 Performing Multiple Tasks

Since Gesto automates a sequence of UI actions, it allows the user to traverse back to previously visited UI elements and visit other UI elements. Users can record sequences that move forward and backward in an app's activity hierarchy. We highlight this feature because it is beyond the capabilities of previous approaches related to deep links [16, 34].

We demonstrate this feature with AVG Cleaner app [37]. AVG Cleaner is a memory optimizer which cleans the cache, the app history, and the call and message history on a mobile device. Users typically perform several tasks at the same time. As shown on Figure 12, we have recorded two different tasks together—cleaning the cache and cleaning the app history. Figure 12 (a) shows the first task of cleaning the cache. On completion, it goes back to the previous activity and starts to perform another task of cleaning the app history as shown in Figure 12 (b). In our recording, we give ample time during recording for the first task to complete, before continuing to the recording of the second task. This is to accommodate potential delays in executing the first task during future replays. Note that reasoning about time to completion of app tasks is beyond the scope of Gesto as it might require understanding the logic of an app.

5 EVALUATION

In this section, we characterize the overhead of Gesto. To demonstrate the feasibility and practicality of our design, we use 1,000 apps randomly picked from a dataset containing top 100 popular apps from each category in Google Play.

We use LG Nexus 5 running Android 6.0.1 (2GB RAM) for our experiments and Samsung Galaxy S4 (2GB RAM) running Android 7.1.2 with a Monsoon Power Monitor for power measurement. LG Nexus 5 is not suitable to measure energy overhead with a Monsoon Power Monitor because it is a device with a non-removable battery. For our app instrumentation, we run Gesto on a desktop PC

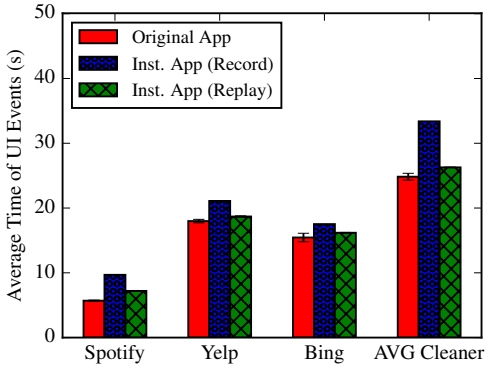


Fig. 13. Latency of Use Case Apps

App Name	Average (J)	Std Dev (J)
Spotify	68.9	5.2
Spotify* (Record)	75.0	5.5
Spotify* (Replay)	78.5	4.8
Yelp	55.5	5.1
Yelp* (Record)	67.8	4.9
Yelp* (Replay)	64.9	5.1
Bing	75.9	4.3
Bing* (Record)	86.1	6.3
Bing* (Replay)	86.9	6.3
AVG Cleaner	56.5	4.7
AVG Cleaner* (Record)	66.0	3.9
AVG Cleaner* (Replay)	63.2	2.4

Table 4. Energy Consumption (*Instrumented apps)

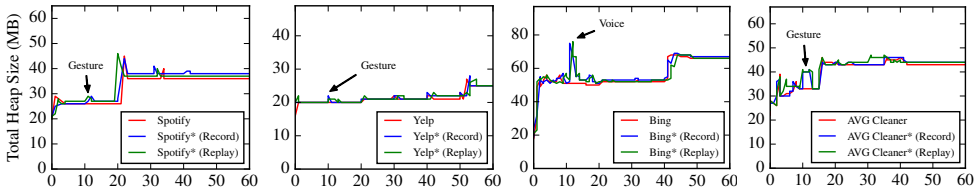


Fig. 14. Heap Usage (*Instrumented apps): The x-axis shows elapsed time in seconds.

with a 3.10 GHz Intel Core i5-2400 CPU, 16GB of RAM, and a single 7200 RPM hard disk. We have measured all our instrumentation-time overhead in this setting.

5.1 Use Case Evaluation

Latency Overhead: Since our approach injects additional code as well as wrappers for several classes, it is important to understand how much latency overhead this augmentation incurs. To evaluate the latency overhead, we have measured the previously described use case apps in three contexts—(1) regular UI events on the original app, (2) recording on an instrumented app, and (3) replaying on an instrumented app. For all three tests, we have set the same number of UI events and same time gaps between each UI event. We have tested each sequence 10 times except when recording which needs to be set only once.

Figure 13 shows the average time for UI events. The three contexts for each of the use cases are very comparable. Recording on instrumented app has overhead about 3-8 seconds in comparison to the original app as it requires a gesture to signal the start and end of the recording. This is external to any latency introduced by our instrumentation. The key comparison is between the original use and a replay. Even between these contexts, the overhead during replay includes the latency to execute the gesture that signals replay. Excluding this gesture, the overhead is far less than 1 second which could be considered negligible.

Heap Usage Overhead: To demonstrate practicality of Gesto, we have measured the heap usage of use case apps under the exact same scenarios. Figure 14 shows the heap usage over time. We use Android's `adb` tool to record heap allocation sizes every 0.3s and run each app for 60s. From Figure 14, heap usage graphs for all four apps show that there is no significant difference between the heap usage of the original app in comparison to record and replay events on an instrumented

app. Only noticeable difference on the four graphs is when gesture or voice is detected for record and replay. This results in a brief 5-15% increase in heap usage but it returns to heap usage of the original app in seconds. We believe that this overhead is well within acceptable memory limits on most modern mobile devices.

Energy Consumption Overhead: Since energy is a scarce resource on mobile devices, we characterize the energy overhead of our approach. We run the four apps for one minute ten times and measured energy consumption with a Monsoon Power Monitor. During the measurement, we used the same workload as mentioned in Section 4. Table 4 shows the average and standard deviation of energy consumed across those runs. In the worst case, Gesto incurs 16.9% energy overhead. We attribute this increase to our gesture and voice recognition libraries, which continuously use various sensors such as proximity, accelerometer, and microphone at a high sampling rate. Energy-efficient gesture or voice recognition is currently an active topic of research [27, 39, 41, 43]. We conjecture that the energy consumption of Gesto would be reduced by incorporating some of these techniques. We hope to incorporate some of these ideas in the future.

5.2 Instrumentation Evaluation

The previous section demonstrates the feasibility of our system by measuring overhead and showing that they are acceptable. We now show that Gesto has a wide coverage across apps. We do this by evaluating (1) what percentage of UI element uses we can cover, and (2) what percentage of apps we can instrument successfully.

Android apps create UI elements either statically from layout XML files or dynamically at run time. Accordingly, Gesto can correctly record and replay both statically-declared and dynamic-created UI elements. However, as mentioned in Section 3.3, the current prototype of Gesto supports 6 main UI elements (`EditText`, `TextView`, `ListView`, `Button`, `ImageView`, and `ImageButton`), i.e., it does not support all UI elements. Thus, we measure what percentage of UI element uses we can cover with the 6 elements.

In doing so, we only consider statically-declared UI elements, not dynamically-created elements. This is because a large number of dynamically created UI elements are nested, and it is non-trivial to determine which of the nested UI elements truly interact with users. Thus, we have downloaded 1,000 popular apps from Google play and analyzed all layout XML files to see which UI elements each app statically creates and uses.

Our analysys shows that there are 259,930 statically-declared UI elements from 955 apps, and we cover 250,492 (96.4%) of them. Table 5 shows UI elements sorted by the frequency of use. Gesto supports top 5 (and another from a lower rank). Figure 15 shows that Gesto covers most statically-declared UI elements that each app uses; Gesto perfectly handles all of statically-declared UI elements for 238 apps. Since we cut x-axis below 70%, there are five missing apps (each one at 33%, 42%, 50%, 59%, and 63% respectively). We note that we have removed 45 apps from the results since they did not have layout files.

To evaluate the capability of Gesto instrumentation, we have instrumented the same 1,000 apps. Gesto could instrument 949 apps out of 1,000. Among 51 apps that Gesto could not instrument, the baseline instrumentation tools we leverage (i.e., `apktool`, `Soot`) could not process 27 apps. Gesto could not process the remaining 24 apps.

Table 6 shows our instrumentation overhead results. We have re-categorized 949 apps into eight categories based on Google's categorization to concisely present our results. Our results show average, maximum, and minimum values for instrumentation times, original APK sizes, APK size increase after instrumentation, original lines of code (LoC), and LoC increase after instrumentation.

UI Element	Percent Use	Support
`TextView`	52.8	✔
`ImageView`	29.7	✔
`Button`	8.1	✔
`ImageButton`	3.0	✔
`EditText`	2.8	✔
`CheckBox`	1.0	
`RadioButton`	0.8	
Others (e.g., Spinner, SeekBar)	1.8	

Table 5. UI Element Usage

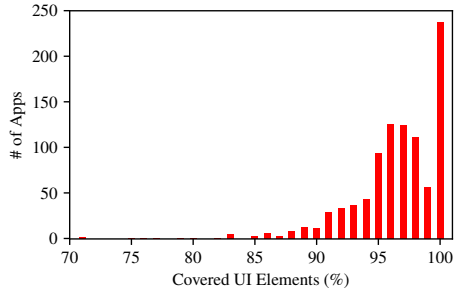


Fig. 15. Histogram of UI element coverage by Gesto. Only 11 of the 955 apps analyzed have fewer than 80% of UI elements covered.

Category (Example)	# of Apps	Inst. Time Avg. (Min./Max.)	APK Size Avg. (Min./Max.)	APK Size Increase Avg. (Min./Max.)	LoC Avg. (Min./Max.)		LoC Increase Avg. (Min./Max.)	
					XML	Jimple*	XML	Jimple*
Game (Color Switch)	95	32.7s (3.7s/60.8s)	12.7M (332.2K/44.4M)	558.8K (5.0K/12.9M)	14.6K (73/54.6K)	442.0K (6.2/792.4K)	640 (0/3.6K)	1.4K (1.3/1.8K)
Entertainment (Roku)	129	34.1s (3.3s/64.7s)	10.5M (174.5K/44.9M)	437.4K (11.1K/1.3M)	15.4K (71/113.2K)	459.0K (8.7/813.4K)	621 (-261/3.6K)	1.4K (1.3/1.8K)
Media (Spotify Music)	132	28.7s (2.1s/65.1s)	9.9M (13.7K/36.6M)	408.4K (-2.1M/4.6M)	11.1K (33/41.4K)	380.1K (18/734.7K)	398 (0/2.4K)	1.4K (1.3/1.9K)
Education (Brilliant)	114	33.0s (5.6s/78.9s)	10.9M (822.2K/48.4M)	718.0K (-117.0K/13.7M)	14.9K (95/79.0K)	440.6K (27.7/868.8K)	698 (0/5.2K)	1.4K (1.3/2.7K)
Personalization (Alarmy)	118	27.7s (2.5s/98.2s)	8.1M (695.0K/49.3M)	364.3K (-72.2K/2.0M)	9.9K (115/57.2K)	366.7K (25/772.7K)	382 (0/3.4K)	1.4K (1.3/1.8K)
Productivity (Evernote)	116	31.5s (3.8s/113.8s)	7.7M (1.1M/41.0M)	647.3K (-761.3K/19.5M)	19.2K (62/127.3K)	414.2K (7.6/831.8K)	707 (0/5.0K)	1.4K (1.3/1.7K)
Business (Venmo)	91	33.8s (2.2s/81.8s)	10.4M (47.0K/36.5M)	435.8K (-2.4M/5.2M)	18.2K (32/103.4K)	436.3K (168/856.0K)	1013 (-139/5.1K)	1.5K (1.3/2.3K)
Social (Yelp)	154	31.9s (3.4s/90.6s)	10.4M (207.6K/46.5M)	675.7K (-1.2M/25.5M)	17.4K (101/193.0K)	425.1K (6.3/792.7K)	736 (0/9.6K)	1.4K (1.3/2.1K)
Total	949	31.6s (2.1s/113.8s)	10.0M (13.7K/49.3M)	534.3K (-2.4M/25.5M)	15.0K (32/193.0K)	419.5K (18/868.8K)	638 (-261/9.6K)	1.4K (1.3/2.7K)

Table 6. Instrumentation Results for 949 Popular Apps (*Jimple LoC is roughly equivalent to Android bytecode LoC.)

Since Gesto modifies XML files and Android bytecode in Jimple [2], we present the LoC results for XML and Jimple separately. App instrumentation finished in 2 minutes or less for all apps. The original APK sizes ranges from 13.7KB to 49.3MB. On average, the APK size increases by 7.2%, the LoC for XML by 3.5% and the LoC for Jimple by 35.0%. Gesto has a slightly high percentage increase in average LoC for Jimple. This is because tiny apps in our data set have a far fewer number of LoC compared to the LoC added by Gesto. However, it is 2.7K LoC in the worst case. Interestingly, we have noticed that Gesto has even *reduced* the size of an app after instrumentation—this is because we use *apktool* [2] that provides better compression and optimization for some of the apps. sizes (Blue Bible-Productivity, Jet Blue-Social) have been increased by 19.5MB and 25.5MB for each, but this is because of *apktool*'s compression. Size increases just for *apktool* are 19.1MB and 24.9MB for these two apps. Excluding the app size increase by *apktool*, Gesto increases the app size modestly by about 0.5 MB.

6 DISCUSSION AND FUTURE WORK

Cross-app Automation: Gesto currently records and replays UI events within an app but does not support cross-app automation. That requires handling Android's messaging mechanism (`Intent`) across different apps. We could potentially instrument `Intents` to pass around recording and replaying information, which we leave as future work.

[2]Jimple is the primary intermediate representation of Soot [47].

Similarly, Gesto currently does not support apps that use `WebView`, an Android system library that displays web pages. `WebView` essentially works as if it were a different app; and because of that reason, we cannot record and replay `WebView` events.

Limited UI Events Handled In Gesto: As mentioned in Section 3.3, our current prototype of Gesto handles six UI elements, `TextView`, `ImageView`, `ListView`, `Button`, `ImageButton`, and `EditText`. According to our analysis of statically-declared UI elements from 1,000 apps (in Section 5.2), these UI elements cover 96.4% of all statically-declared UI elements used in those apps. While Android API lists several more elements, we believe that the current Gesto framework can perform reasonably with this coverage. Supporting more UI elements is a matter of engineering using the same mechanism that we use now.

Dynamic UI Object ID Assignment: Section 3.4 discusses our ID assignment mechanism where the assumption is that when an app populates an `Activity` with UI objects, it always adds dynamic UI objects in the same order. Based on our experience with real apps downloaded from Google Play, this seems to be a reasonable assumption to make since UI objects, even when they are dynamically-added, need to show a consistent look and feel to users. This implies that app code that adds dynamic UI objects needs to be deterministic across different runs. In the future, we will quantify the effect of this assumption on a larger set of apps.

API Version Sensitivity: Since we capture UI events at each callback, if new Android API versions add or change the callbacks, our approach needs to know the additions or changes in order to handle them correctly. However, we believe that this can be automated to a large extent; since all UI element classes are subclasses of `View`, we could detect any addition or change by monitoring the API documentation.

Scalability of Gestures: Gesto currently provides 21 gestures and voice recognition features using Sensey [46] and Android's `SpeechRecognizer`. Since we are leveraging third-party libraries, as they get better, we can incorporate new capabilities.

Change in UI Object Ordering: After recording UI events, several factors can change the order of UI objects. For example, sensor input such as current location can affect the order of UI objects, e.g., a list of nearby restaurants. Also, some user actions (e.g., updating a music playlist) may change the order of UI objects. These changes will affect the correctness of replaying previously recorded UI events. However, this is a general problem that most of the UI record-and-replay systems suffer, including Gesto. In order to address this, a UI record-and-replay system needs to record non-deterministic input (e.g., the current location), potentially continuously (e.g., to record all prior user input), which Gesto does not implement at this time. In practice, we believe that we can communicate this limitation to users, so that they are aware of it when using Gesto.

Supporting a Server for Instrumentation Service: Currently, app developers or users can instrument an app by downloading and using Gesto. In the future, we plan to support instrumentation through a server. This would have two parts—a server that runs Gesto and an app that runs on a smartphone or a tablet of a user that communicates with the server to instrument an app. This way, our app would even be able to track app updates and automatically re-instrument updated apps—the app could monitor installed apps on Google Play and whenever an installed app has a new version, it could automatically contact the server to re-instrument the updated app.

7 RELATED WORK

In this section, we compare previous work related to Gesto in three categories.

7.1 Gesture Detection and Voice Recognition

Gesture and voice recognition have been used for a wide range of applications in mobile systems [32, 39, 44] including mobile devices to imitate a mouse in the air [49], to write on surface or in the

Name	UI Object Based	View ID Insensitive	Timing Sensitive	State-agnostic	Nonintrusive
Gesto	✔	✔	✔	✔	✔
appetizer [3]		✔	✔		✔
Bot-bot [4]	✔	Partial		✔	✔
Culebra [5]	✔				✔
Espresso [6]	✔			✔	Source Code
MobiPlay [40]		✔	✔	?	Custom OS
monkeyrunner [8]		✔			✔
Mosaic [24]		✔	✔		Root Access
Ranorex [9]	✔		✔		✔
RERAN [20]		✔	✔		Root Access
Robotium [10]	✔		✔	✔	✔
Valera [25]		✔	✔	✔	Custom OS, Root Access
SUGILITE [31]	✔				Root Access
Frep [7]	✔		✔		Root Access

Table 7. Comparison Table of Record and Replay Tools for Android [29]. "?" indicates unable to verify this characteristic.

air [11, 48], to share files between multiple devices [17], and even to detect smoking gestures to help users quit smoking [38]. Gesto uses gesture detection and voice recognition to trigger record and replay of UI actions integrating Sensey [46] library into Gesto and Android Speech Recognition service [33] for voice.

7.2 Deep Links

Deep links are special types of links that point to any UI location within an app or website [42]. Aladdin [34] presents a tool that can be used to generate app's APK file with deep links feature enabled. uLink [16] proposes a mechanism to create deep links in mobile apps. Analyzing dependencies between UI elements, a link (deep link) can be provided to navigate to that UI location at a later stage akin to hyperlinks.

Most closely related is uLink, but its techniques have differences and limitations if they are used to provide task automation. A deep link created using uLink can only handle a single task and replay only the shortcut to the target UI location, not the exact sequence that has been recorded. In Gesto, instead of creating deep links which requires tracking dependencies, we dynamically map a sequence of intercepted UI events to specific gesture/voice which can enable replaying the exact sequence at a later time. Further, Gesto allows users to combine multiple tasks in a single record and replay instance. In addition, uLink implements a record-and-replay mechanism similar to Gesto. However, it requires every UI object to have an ID, which does not always happen in real-world apps. Section 3.1 explains this challenge in detail along with how Gesto solves this issue.

7.3 Record and Replay

Record-and-replay has been studied in various platforms. For instance, Improv [18] and CoScripter [30] enable automation of Web tasks. However, mobile platforms have unique UI characteristics causing different challenges such as handling a lifetime of an activity, the existence of UI elements in the foreground, and a power consumption. In this section, we mainly compare with UI automation tools on mobile platforms.

Flinn et al. [19] and Lam et al. [29] discussed several record and replay tools, use cases, and challenges of using this technique on mobile phones. One particular use case of record and replay, that is closely related to Gesto, is record and replay of UI actions. Table 7 compares Gesto to existing UI record-and-replay tools. The table uses the metrics similar to the ones used by Lam et al. [29] that succinctly capture desired functionality for mobile UI record and reply.

Object-Based: There are many tools [3, 8, 20, 24, 25, 40] that record UI events based on coordinates of the events on a device screen. Although it supports WebView events that Gesto can not (discussed in Section 6), this approach is limited in two ways. First, recorded UI event sequences cannot be replayed on different devices/orientations due to different screen dimensions. Second, it cannot handle dynamic UI elements i.e., when the UI element changes its position across runs. For example, on an image searching app, positions of buttons under an image can change depending on the displayed image size.

Other tools [4–7, 9, 10, 31] record UI events based on UI objects (e.g., buttons, text fields). This approach is more robust than coordinates sensitive because it identifies UI elements by their information, not positions. On Android, each UI element contains much information, but `View` ID is the only unique information distinguishable. Bot-bot [4] identifies UI elements only with text of the UI elements, hence is not reliable. The remaining approaches [5–7, 9, 10, 31] use `View` IDs to identify the UI elements but do not work correctly when the UI elements do not have IDs assigned. Gesto records and replays UI events based on UI objects, and it successfully handles `Views` with no IDs.

SUGILITE [31] and Frep [7] use `Accessibility Service` [1] to enable record and replay for UI events. `Accessibility Service` intercepts UI events based on the UI objects. Therefore, this approach can not handle the UI elements which have not been assigned IDs. Again, detailed discussion of this shortcoming is in Section 3.1. Also, the service is originally built to assist users with disabilities, and is now widely used for other purposes due to its ease of use. It allows for programmatic interception of UI interaction. This may cause significant security issues (e.g., intercepting login information on banking apps). Therefore, Google suggests not to use `Accessibility Service` for non-disabled users. Also, users need to allow apps to use `Accessibility Service` in the device settings.

State Agnostic: There are many tools [3, 5, 7–9, 20, 24, 31] that can replay only when the device is in the same state as the state where UI events were recorded. In these tools, users manually need to put an app in the same state before each reply. Gesto supports replay in any state of an app.

Automatic Timing: Many tools [3, 7, 9, 10, 20, 24, 25, 40] automatically set timing between recorded events. However, other tools [4–6, 8, 31] require users to manually set the timing between events, which is tedious and error prone. Gesto supports automatic timing between UI events and replays each event on exact time.

Noninstrusive: Many tools take an intrusive approach where they require OS modifications [25, 40], rooting [7, 20, 24, 25, 31], or source code [6]. Gesto does not have such requirement.

8 CONCLUSIONS

In this paper, we have presented Gesto, a task automation system for Android apps using gestures and voice. Gesto enables a user to record a sequence of UI actions for an app, map the sequence to a gesture or a voice command, and later trigger the sequence with the mapped gesture or voice command. In doing so, Gesto combines bytecode instrumentation and UI record-and-replay. Gesto's bytecode instrumentation enables functionality injection into an existing Android app without app source code or developer intervention. Gesto's record-and-replay works at the level of UI objects rather than pixel positions, making it possible to record once and replay it on different devices. For this, Gesto analyzes an app's bytecode, identifies all UI elements, and enables recording and replaying of all UI interactions. Our experience with four use cases (Spotify, Bing, Yelp, and AVG Cleaner) shows that Gesto provides functional task automation for real apps with modest overhead. Our evaluation with 1,000 apps downloaded from Google Play shows that Gesto's instrumentation is able to cover a wide range of apps (94.9%). Overall, we show that Gesto provides applicable and practical task automation using gestures and voice.

9 ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported in part by the generous funding from the National Science Foundation, CNS-1350883 (CAREER) and CNS-1618531.

REFERENCES

[1] Cited May 2018. Accessibility Services. https://developer.android.com/reference/android/accessibilityservice/AccessibilityService.html.
[2] Cited May 2018. Apktool. https://ibotpeaches.github.io/Apktool/.
[3] Cited May 2018. appetizer-toolkit. https://github.com/appetizerio/appetizer-toolkit.
[4] Cited May 2018. Bot-bot. http://imaginea.github.io/bot-bot/index.html.
[5] Cited May 2018. Culebra. https://github.com/dtmilano/AndroidViewClient/wiki/culebra.
[6] Cited May 2018. Espresso Test Recorder, 2017. https://developer.android.com/studio/test/espresso-test-recorder.html.
[7] Cited May 2018. Frep. http://strai.x0.com/frep/.
[8] Cited May 2018. monkeyrunner. https://developer.android.com/studio/test/monkeyrunner/index.html.
[9] Cited May 2018. Ranorex. http://www.ranorex.com/mobile-automation-testing.html.
[10] Cited May 2018. Robotium Recorder. https://robotium.com/products/robotium-recorder.
[11] Sandip Agrawal, Ionut Constandache, Shravan Gaonkar, Romit Roy Choudhury, Kevin Caves, and Frank DeRuyter. 2011. Using mobile phones to write in air. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM, 15–28.
[12] Amazon. Cited December 2017. Alexa. https://developer.amazon.com/alexa.
[13] Amazon. Cited December 2017. Alexa SDK. https://developer.amazon.com/alexa-voice-service/sdk.
[14] Apple. Cited December 2017. Siri. https://www.apple.com/ios/siri/.
[15] Apple. Cited December 2017. SiriKit. https://developer.apple.com/sirikit/.
[16] Tanzirul Azim, Oriana Riva, and Suman Nath. 2016. uLink: Enabling User-Defined Deep Linking to App Content. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (In MobiSys)*. ACM, New York, NY, USA. http://dl.acm.org/citation.cfm?id=2906416&CFID=758371532&CFTOKEN=77725750
[17] Ke-Yu Chen, Daniel Ashbrook, Mayank Goel, Sung-Hyuck Lee, and Shwetak Patel. 2015. AirLink: sharing files between multiple devices using in-air gestures. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, 565–569.
[18] Xiang 'Anthony' Chen and Yang Li. 2017. Improv: An Input Framework for Improvising Cross-Device Interaction by Demonstration. *ACM Trans. Comput.-Hum. Interact.* 24, 2, Article 15 (April 2017), 21 pages. https://doi.org/10.1145/3057862
[19] Jason Flinn and Z Morley Mao. 2011. Can deterministic replay be an enabling tool for mobile computing?. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*. ACM, 84–89.
[20] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. 2013. Reran: Timing-and touch-sensitive record and replay for android. In *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 72–81.
[21] Google. Cited December 2017. Class `android.view.View`. https://developer.android.com/reference/android/view/View.html.
[22] Google. Cited December 2017. Google Assistant. https://assistant.google.com/.
[23] Google. Cited December 2017. Google Assistant SDK. https://developers.google.com/assistant/sdk/.
[24] Matthew Halpern, Yuhao Zhu, Ramesh Peri, and Vijay Janapa Reddi. 2015. Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*. IEEE, 215–224.
[25] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. 2015. Versatile yet lightweight record-and-replay for android. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 349–366.
[26] Intel. Cited December 2017. Intel Context Sensing SDK | Intel Software. https://software.intel.com/en-us/context-sensing-sdk.
[27] Bryce Kellogg, Vamsi Talla, and Shyamnath Gollakota. 2014. Bringing Gesture Recognition to All Devices. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Berkeley, CA, USA.
[28] Sven Kratz and Jason Wiese. 2016. GestureSeg: Developing a Gesture Segmentation System Using Gesture Execution Phase Labeling by Crowd Workers. In *Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '16)*. ACM, New York, NY, USA.
[29] Wing Lam, Zhengkai Wu, Dengfeng Li, Wenyu Wang, Haibing Zheng, Hui Luo, Peng Yan, Yuetang Deng, and Tao Xie. 2017. Record and replay for Android: are we there yet in industrial cases?. In *Proceedings of the 2017 11th Joint Meeting*

on Foundations of Software Engineering. ACM, 854–859.
[30] Tessa Lau, Julian Cerruti, Guillermo Manzato, Mateo Bengualid, Jeffrey P. Bigham, and Jeffrey Nichols. 2010. A Conversational Interface to Web Automation. In *Proceedings of the 23Nd Annual ACM Symposium on User Interface Software and Technology (UIST '10)*. ACM, New York, NY, USA, 229–238. https://doi.org/10.1145/1866029.1866067
[31] Toby Jia-Jun Li, Amos Azaria, and Brad A. Myers. 2017. SUGILITE: Creating Multimodal Smartphone Automation by Demonstration. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 6038–6049.
[32] Jiayang Liu, Lin Zhong, Jehan Wickramasuriya, and Venu Vasudevan. 2009. uWave: Accelerometer-based personalized gesture recognition and its applications. *Pervasive and Mobile Computing* 5, 6 (2009), 657–675.
[33] Google LLC and Open Handset Alliance. Cited December 2017. SpeechRecognizer | Android Developers. https://developer.android.com/reference/android/speech/SpeechRecognizer.html.
[34] Yun Ma, Xuanzhe Liu, Ziniu Hu, Dian Yang, Gang Huang, Yunxin Liu, and Tao Xie. 2017. Aladdin: automating release of Android deep links to in-app content. In *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 139–140.
[35] Microsoft. Cited December 2017. Cortana. https://www.microsoft.com/en-us/windows/cortana.
[36] Microsoft. Cited December 2017. Cortana SDK. https://developer.microsoft.com/en-us/cortana.
[37] AVG Mobile. Cited December 2017. AVG Cleaner. https://play.google.com/store/apps/details?id=com.avg.cleaner&hl=en.
[38] Abhinav Parate, Meng-Chieh Chiu, Chaniel Chadowitz, Deepak Ganesan, and Evangelos Kalogerakis. 2014. Risq: Recognizing smoking gestures with inertial sensors on a wristband. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM, 149–161.
[39] Taiwoo Park, Jinwon Lee, Inseok Hwang, Chungkuk Yoo, Lama Nachman, and Junehwa Song. 2011. E-gesture: a collaborative architecture for energy-efficient gesture recognition with hand-worn sensor and mobile devices. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*. ACM, 260–273.
[40] Zhengrui Qin, Yutao Tang, Ed Novak, and Qun Li. 2016. MobiPlay: A Remote Execution Based Record-and-replay Tool for Mobile Applications. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA.
[41] Giuseppe Raffa, Jinwon Lee, and Lama Nachman. 2010. Don't slow me down: Bringing energy efficiency to continuous gesture recognition *(Wearable Computers (ISWC), 2010 International Symposium)*. IEEE.
[42] Margaret Rouse. Cited November 2017. What is deep link? http://searchmicroservices.techtarget.com/definition/deep-link.
[43] Haichen Shen, Aruna Balasubramanian, Anthony LaMarca, and David Wetherall. 2015. Enhancing Mobile Apps to Use Sensor Hubs Without Programmer Effort. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '15)*. ACM, New York, NY, USA.
[44] Sheng Shen, He Wang, and Romit Roy Choudhury. 2016. I am a Smartwatch and I can Track my User's Arm. In *Proceedings of the 14th annual international conference on Mobile systems, applications, and services*. ACM, 85–96.
[45] Lucio Davide Spano, Antonio Cisternino, Fabio Paternò, and Gianni Fenu. 2013. GestIT: A Declarative and Compositional Framework for Multiplatform Gesture Definition. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '13)*. ACM, New York, NY, USA.
[46] Nishant Srivastava. 2016. GitHub - nisrulz/sensey: [Android Library] Play with sensor events & detect gestures in a snap. https://github.com/nisrulz/sensey.
[47] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*.
[48] Chao Xu, Parth H Pathak, and Prasant Mohapatra. [n. d.]. Finger-writing with smartwatch: A case for finger and hand gesture recognition using smartwatch. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*. ACM, 9–14.
[49] Sangki Yun, Yi-Chao Chen, and Lili Qiu. 2015. Turning a mobile device into a mouse in the air. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 15–29.

Received October 2018; revised December 2018; accepted February 2019