Remote Control of iOS Devices via Accessibility Features

Nikola Lukić*
University of Southern California
Los Angeles, USA
nlukic@usc.edu

Saghar Talebipour*
University of Southern California
Los Angeles, USA
talebipo@usc.edu

Nenad Medvidović
University of Southern California
Los Angeles, USA
neno@usc.edu

ABSTRACT

This paper presents an approach for remotely accessing and controlling mobile apps by leveraging a mobile platform's publicly exported *accessibility features*. This approach is implemented in a technique and accompanying tool called AirMochi.While AirMochi is designed to be platform-independent, our current implementation has focused on iOS, the significantly more challenging of the two dominant mobile platforms, for which access to apps' source code is generally not possible. We show that AirMochi places no restrictions on apps it can "plug into" and control, is able to handle a variety of scenarios, and imposes a negligible performance overhead.

ACM Reference Format:
Nikola Lukić, Saghar Talebipour, and Nenad Medvidović. 2020. Remote Control of iOS Devices via Accessibility Features. In *2020 Workshop on Forming an Ecosystem Around Software Transformation (FEAST '20), November 13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3411502.3418427

1 INTRODUCTION

The emergence of mobile computing platforms has resulted in millions of apps that provide services spanning virtually every facet of human need and endeavor. In part, this is because the providers of mobile platforms—Google's Android and Apple's iOS most notably—have tried to make app building easier for an inexperienced developer. This has given consumers a previously unimaginable range of options at relatively low costs. However, it has also introduced new challenges, both for the consumers and for software developers. Namely, individual apps can behave in unexpected ways, include counter-intuitive features, have a variety of bugs, exhibit unpredictable performance, be susceptible to security breaches, and so on. This is magnified by the expectation that apps on a mobile device will interact, not only with back-end servers, but also with one another. The very flexibility of the mobile platforms is thus also a key source of complexity developers must address.

Existing work has looked at different facets of this problem. A large number of techniques have focused on issues such as testing apps' behaviors [22], analyzing their performance [17], identifying [13] and patching [18] their vulnerabilities, recovering their designs [20], and reverse-engineering their implementations [20]. Of special interest to the work described in this paper are techniques for remotely accessing and/or controlling apps running on a device. Examples include app record-and-replay [16], UI event generation for apps running in an emulator [21], use-case scenario extraction from screen recordings [14], and application of OCR and computer-vision techniques to automatically identify GUI widgets [20].

These techniques have invariably targeted Android, an open platform that provides access to low-level details of an installed app and its execution substrate. Furthermore, many of these techniques assume access to at least some parts of an app's source code-level information. For instance, RERAN [16] uses Android's *getevent* and *sendevent* tools [8] to capture and regenerate UI events, while MobiPlay [21] leverages Android emulators to run applications remotely.

By contrast, iOS is a closed platform and is less amenable to such analyses: no analogous APIs, tools, or emulators exist for it.[1] Two solutions have been typically adopted on iOS: override certain kernel-level restrictions by "jailbreaking" a device (e.g., [1]) or leverage low-level protocols currently exposed by Apple. However, jailbreaking an iOS device introduces security vulnerabilities and results in platform and app versions that are considered illegal by Apple [15]. On the other hand, techniques that rely on low-level protocols generally rely on the XCUITest black-box testing framework, and their capabilities are dependent on APIs whose public access is not officially sanctioned by Apple and may be disabled in a future iOS version. For example, this is the case with Facebook's WebDriverAgent [4], an iOS specific implementation of W3C's WebDriver [12], a remote device control interface. Additionally, both of these approaches are complex and impose significant engineering burden. This is also likely the case with commercial tools that target iOS, such as Eggplant [7] , but those tools are based on proprietary technologies, preventing their analysis as well as necessary extensions.

This paper introduces AirMochi, a technique and accompanying tool for remotely accessing and controlling mobile apps. AirMochi only leverages a mobile platform's publicly exported *accessibility features*. Such features are commonly provided to facilitate the use of a device by persons with vision, hearing, and other physical disabilities. AirMochi exploits these features to "plug into" a device

*The authors have contributed equally to this work.

FEAST '20, November 13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8089-8/20/11…$15.00
https://doi.org/10.1145/3411502.3418427

[1] Apple does provide iOS *simulators* for MacOS. However, due to differences in hardware architectures between iOS (ARM-based) and MacOS (Intel-based) devices, it is impossible to run iOS apps acquired through the AppStore on simulators; special simulator-based builds of those apps are required.

via an emulated mouse and keyboard, and then simply use the apps on the device in the usual manner. In other words, AirMochi does not rely on the availability of a mobile app's code and only assumes that the app can be installed and run on a device.

AirMochi records the video stream of the device's UI during the usage, while capturing and time-stamping UI events. This forms a critical foundation for a range of downstream capabilities: accurate replay of any app's usage, back-end analysis of the app's user-facing behavior, extraction of key use-case scenarios, automated widget recognition and reconstruction of app GUIs, streamlining of third-party app testing, automated construction of accurate app-behavior models, and even introduces the possibility of automatic reimplementation of the app itself.

While AirMochi's implementation currently targets iOS, its design can be applied to Android or another platform by making modifications that are isolated within AirMochi's individual components. We demonstrate AirMochi's key properties of (1) effectiveness, (2) efficiency, (3) modularity, and (4) extensibility. Specifically, we show that AirMochi places no restrictions on apps and is able to handle a variety of representative scenarios. We also show that AirMochi imposes a negligible performance overhead even though we have not attempted to optimize its current implementation.

Section 2 provides an overview of AirMochi's intended use by way of a representative high-level scenario. Section 3 describes AirMochi's architecture and provides details of its six principal components. Section 4 details AirMochi's current implementation. Section 5 describes our preliminary evaluation of AirMochi. Section 6 discusses on-going work and concludes the paper.

2 OVERVIEW

The manner in which AirMochi is intended to be used is reflected in Figure 1. AirMochi's user is able to start a remote-execution session in the *User-Facing Application*, e.g., on a desktop computer (bottom-right). During the session, the user is able to remotely control a dedicated *Mobile Device* (bottom-left). Each session consists of a stream of user-generated UI *events* flowing from the *User-Facing Application* to the *Mobile Device*, and video stream comprising device *screens* flowing in the opposite direction.

The user is able to seamlessly control the *Mobile Device*, using only the input peripheral devices on her side (e.g., mouse and keyboard). Once the user generates a particular UI event in the *User-Facing Application*, that event is routed to and executed on the dedicated *Mobile Device*. The resulting video stream is captured from the dedicated device in the form of a sequence of device-screens and displayed, remotely, on the user's side.

During a remote-execution session, the user can opt to record certain scenarios of interest. Both the device-screen video stream and UI event stream are persisted by AirMochi during the recording. AirMochi's user can analyze the event and screen streams side-by-side, e.g., to debug an app. The user can alternatively replay usage scenarios by re-executing the recorded UI events on the *Mobile Device*. The UI event streams are persisted as logs containing coordinates which couple the UI events to specific areas of the device screen. AirMochi translates these coordinate-based logs to GUI-based level scenarios. This translation makes it possible for the user to replay the recorded scenario on any type of device since the recording is no longer dependent on a specific hardware. The replay functionality allows rapidly bringing a mobile app to a specific execution state on any device, enabling further analysis, and testing of the app. The video of the resulting replay and the previously recorded video can also be compared to evaluate the correctness of the replay, to assess whether a given app has changed, to analyze the effects of any server-side changes on the app, etc. AirMochi also incrementally builds a partial behavioral model of the app that has been remotely executed. The generated model shows the transitions between different possible UI states of an application and can be used "downstream" for further app analysis and testing.

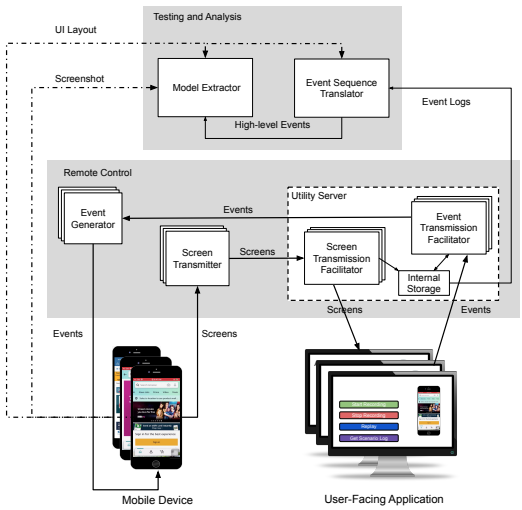

Figure 1: AirMochi's design. The *Event Generator* and *Screen Transmitter* components can reside on the *Mobile Device* or be deployed on a separate node. The dashed lines from the *Mobile Device* to the *Model Extractor* and *Event Sequence Translator* components are intended to indicate that this information can be obtained in different ways. In our current implementation, we use the Appium [6] third-party library.

3 AirMochi's DESIGN

Three of AirMochi's over-arching design objectives are efficiency, modularity, and extensibility. AirMochi's design is primarily driven by *efficiency*. To be usable in practice, AirMochi must support a realistic usage experience in which there is no perceptible performance degradation compared to the direct use of a mobile device. Our solution must therefore minimize the latency when transmitting event streams and device-screen video streams. This is reflected in AirMochi's design in (1) the limited number of software components interceding between the user and the mobile device, thereby avoiding connection and/or processing overloads; (2) the dedicated, separate instances of key components for each execution session, as detailed below; and (3) the minimal online manipulation of the event and device-screen streams between their sources and destinations.

Another design objective for AirMochi is *modularity*. As discussed above, there are several one-off solutions for recording and replaying app executions. Although AirMochi currently targets iOS devices by leveraging their accessibility interfaces in specific

ways (as further detailed in Section 4), our eventual goal is to enable combining different approaches for screen streaming, user-side event generation, and device control. To this end, AirMochi comprises six top-level modules, as shown in Figure 1: (1) *User-Facing Application*, (2) *Utility Server*, (3) *Event Generator*, (4) *Screen Transmitter*, (5) *Event Sequence Translator*, and (6) *Model Extractor*. This allows us to add event generation and screen sharing solutions for different platforms quickly, without significant engineering overhead.

This leads to *extensibility* as the final principal design objective for AirMochi. In addition to leveraging AirMochi's modularity in order to extend it to different platforms as discussed above, its pluggable design makes it a suitable research foundation for easily introducing additional data collection and processing tools. By plugging in components that implement the desired capabilities, AirMochi will allow us perform downstream analyses such as runtime tracking of unwanted app scenarios, mapping of UI events to network requests, detection of information-leaks on a mobile device, etc.

We now describe each of AirMochi's components from Figure 1. We divide the components into two subsets according to their primary purpose: (1) the core capabilities required for remote control of mobile devices and (2) the advanced, "downstream" features for mobile app testing and analysis. We describe each in turn.

3.1 Remote Control of Mobile Devices

AirMochi's support for remotely controlling devices relies on four components: *User-Facing Application*, *Utility Server*, *Event Generator*, and *Screen Transmitter*. We describe each in turn.

User-Facing Application
Each AirMochi user is provided a separate instance of the *User-Facing Application*. During the execution of a user session, the *User-Facing Application* receives a device-*screen* video stream from the *Utility Server*, while capturing and streaming the user-generated UI *events* in the opposite direction. The *User-Facing Application* is in charge of notifying the *Utility Server* of the start and end of stream recording. Finally, when replaying previously recorded sessions, the *User-Facing Application* only displays the *Mobile Device*'s screen video stream, without attempting to capture user events.

Utility Server
The *Utility Server* is the main orchestrator of remote device control. It is in charge of establishing and persisting the connections between users and devices. As shown in Figure 1, the *Utility Server* itself consists of three components: (1) *Screen Transmission Facilitator*, (2) *Event Transmission Facilitator*, and (3) *Internal Storage*.

The *Screen Transmission Facilitator* is in charge of establishing the video stream from a *Mobile Device* and its corresponding *User-Facing Application* instance. As discussed previously, video streams are captured as sequences of device-*screens*. The *Screen Transmission Facilitator* may need to modify the stream to accommodate specific *User-Facing Application* requirements.

The *Event Transmission Facilitator* also establishes a connection between a *Mobile Device* and its corresponding *User-Facing Application*, but is in charge of transmitting user *events* from AirMochi to the device. Furthermore, the *Event Transmission Facilitator* is in charge of re-executing recorded execution scenarios.

Finally, the *Internal Storage* component is used by both facilitator components. The *Screen Transmission Facilitator* uses it to store *screen* frames from the device's video stream. The *Event Transmission Facilitator* uses it to read in the scenario that is to be replayed and to persist the UI *event* log when recording an ongoing scenario.

Event Generator
The *Event Generator* supplies UI *events* to the physically connected *Mobile Device*. It receives each UI event using an internal AirMochi representation, translates the event to the device-specific representation, and initiates the event's execution on the connected device. As discussed above, to support efficiency, each device is assigned a dedicated instance of the *Event Generator* component. This component can reside on the device itself, or it can be deployed remotely.

Screen Transmitter
In response to the execution of UI events, the *Screen Transmitter* begins capturing the *Mobile Device*'s video frames as *screen* sequences and pushing them to the *Utility Server* via the connection previously established by the *Screen Transmission Facilitator*. As with the *Event Generator*, there is one instance of *Screen Transmitter* per device, and it can reside on the device itself or be deployed remotely.

3.2 Testing and Analysis of Mobile Apps

Another goal of AirMochi is to use the data obtained via remote control for a range of downstream analyses. For example, we can integrate the Appium testing framework [6] with AirMochi. AirMochi's current support for testing and analysis relies on two additional components: *Event Sequence Translator* and *Model Extractor*.

Event Sequence Translator
The *Event Sequence Translator* is responsible for translating AirMochi's hardware-dependent recording logs, coupled to specific coordinates on the mobile device screen, to event sequences in a UI element-based representation. While useful, low-level events impede the reusability of recorded scenarios by tying them to specific hardware characteristics (e.g., screen size). The new representation is hardware-independent and can be replayed on devices different from the device on which the original session was executed.

The logs recorded by AirMochi contain three types of low-level events obtained from the *Event Generator* component:

L_a: `mousePress(x, y, timestamp)`
L_b: `mouseRelease(x, y, timestamp)`
L_c: `keyboardPress(value)`

In order to translate these to hardware-independent events, we define a corresponding set of high-level UI events as follows:

H_a: `click(x, y, timestamp)`
H_b: `longClick(x, y, timestamp)`
H_c: `swipe(startX, startY, endX, endY, timestamp, duration)`
H_d: `keyboardPress(value)`

We map the low-level events to high-level events using the following algorithm. Each low-level event of type L_a is directly followed by a low-level event of type L_b. Based on this, AirMochi identifies pairs of low-level events $\{a, b\}$ of types L_a and L_b, respectively. Depending on the relationship between the two events' attributes (`x, y, timestamp`), we map each pair $\{a, b\}$ to one of the high-level UI events according to one of the following four cases:

(1) A high-level event of type H_a is identified when the coordinates of events a and b are the same, and the difference

between their timestamps is not greater than *500ms*, which is the default duration for generating long click on iOS [9].
(2) If the coordinates are the same but the difference in the timestamps exceeds *500ms*, then we identify a high-level event of type H_b.
(3) If coordinates of events a and b are different, then the high-level event of type H_c is identified.
(4) Low-level events of type L_c are simply mapped to high-level events of type H_d.

The *Event Sequence Translator* then needs to map the absolute coordinates to the GUI elements on the device on which AirMochi was run. Before the execution of each of the UI events of type H_a or H_b, the *Event Sequence Translator* fetches the UI layout of the currently visible screen. The fetched layout is in the form of a tree, which is then traversed for finding the intended UI element. This is done by finding the element in the deepest layer of the tree whose boundaries contain the coordinates of the to-be-translated UI event. Swipe events (H_c) usually signify gestures generated on the screen itself, rather than a specific UI element. For example, swipe from left to right generates a "back" button functionality. Also, swipes are used for scrolling over views. Thus, in the case of swipes, we are not trying to find specific UI elements but are generating absolute-coordinate events, since those will map to "swipe" functionality regardless of the particular device. When it comes to the H_d events, we are directly generating keyboard events, since they are already hardware-independent and there is no need to do any translation.

As its final output, the *Event Sequence Translator* component provides a hardware-independent, UI element-based test-case for each recorded scenario comprising a sequence of low-level event logs.

Model Extractor

The *Model Extractor* component incrementally extracts a partial behavioral model of an app that has been remotely executed using AirMochi. Our work to date has focused on generating the models, in the manner described below. Our on-going work aims to demonstrate that these models can be used for a range of analyses previously developed for ensuring important properties of mobile apps (e.g., correctness, reliability, and security [18]).

The behavioral model of an app extracted by AirMochi is represented in the form of a finite state machine. In this model, each state, which we call a UI state, consists of the information regarding the existing widgets on the currently visible screen, the values for the existing widgets' attributes, UI layout tree of the current screen, and a screenshot from the current state of the app.The edges in the model represent high-level UI events that result in transitions between states (as well as self-transitions). As discussed above, the UI events can be of type `click`, `longClick`, `swipe`, and `keyboardPress`.

The process of model extraction is as follows. An app execution session starts with the app being in its initial state, S_0. The *Model Extractor* component extracts the needed information for defining the UI state S_0. From S_0, executing the high-level ev_1, which is the *Event Sequence Translator*'s first translated high-level event, transitions the app to UI state S_1. Accordingly, in state S_N, the *Model Extractor* extracts the UI state information and by executing event ev_{N+1}, transitions application to the UI state S_{N+1}. After entering each state, the *Model Extractor* determines whether this state has been previously visited or a new node should be generated for the state in the behavioral model. It does so by comparing the current UI state with the existing states in the model. Depending on the desired granularity of the model and the purpose of the analysis, the comparison can look for exact matches or it can allow for approximate matches using a heuristic. In each step, the transition between the two states is labeled by the executed high-level UI event.

Using the above algorithm, the *Model Extractor* creates a partial behavioral model of the app from each of the previously executed scenarios in the form of a state machine. The different state-machine models are then merged using well-known techniques to yield a more complete and more informative model of the given app.

4 AirMochi's IMPLEMENTATION

AirMochi is implemented in ≈3700 SLOC, spanning seven programming languages with modules running on three different platforms. The *Utility Server* from Figure 1 is an HTTP server implemented in NodeJS. For simplicity, this server also hosts the *User-Facing Application* in our current implementation. The *Event Generator* is running on a Raspberry Pi Zero. In the implementation reported in this paper, the *Screen Transmitter* component is running on the *Mobile Device* and is implemented as a native iOS application. More detailed explanations about each of AirMochi's six principal components are presented next.

User-Facing Application

The *User-Facing Application* combines HTML5 and multiple JavaScript frameworks and libraries, mainly: Twilio Video [11], which is an instantiation of the WebRTC real-time communication framework [3]; jQuery; and SocketIO. The *User-Facing Application* captures and displays device-screen streams, captures UI events generated on top of the video stream DOM element, and sends the generated events to the *Utility Server* (recall Figure 1). The *User-Facing Application* is hosted on the same node as the *Utility Server* in our current implementation of AirMochi.

Utility Server

We implemented the *Utility Server* from Figure 1 as a NodeJS application that communicates with AirMochi's *Screen Transmitter*, *Event Generator*, and *User-Facing Application* modules through WebSockets and HTTP requests.

Since we are relying on Twilio Video's WebRTC implementation, the *Utility Server*'s *Screen Transmission Facilitator* component is implemented as an off-the-shelf functionality. The *Utility Server* is in charge of the initial video stream set up, by distributing Twilio Video's access tokens needed for the establishment of the secure connection between the *Screen Transmitter* (discussed below) and the *User-Facing Application*.

The *Event Transmission Facilitator* is implemented as a WebSocket server that mediates messages between the *Event Generator* (discussed below) and the *User-Facing Application*. The UI event messages exchanged in that communication are encoded in JSON.

If session recording is in progress, the JSON messages are persisted in the *Internal Storage*. The *Internal Storage* is implemented as a set of JSON files, one for each recorded session.

Event Generator

iOS 13 introduced a new *accessibility feature* that allows users to plug in standard peripheral devices—mouse and keyboard—and

control iPhones, iPads, and iPod touches. As shown in Figure 2, we implement the *Event Generator* as a USB peripheral emulator that physically connects to the iOS device, and is seen by the device as a keyboard and a mouse. On the other end, the connection between the *Event Generator* and the *Utility Server* is established through WebSockets, where the *Event Generator* is a WebSocket client.

As mentioned above, we selected the Raspberry Pi Zero as the *Event Generator*'s hardware platform. We did so because of its ability to act as a USB peripheral and the availability of an of-the-shelf Python implementation of the WebSocket client library for its Raspbian operating system [10]. As shown in Figure 2, the *Event Generator* is implemented using three layers: (1) *Message Receiver*, (2) *Message-to-Event Mapper*, and (3) *Event Executor*.

The *Message Receiver* component implements a WebSocket client and receives JSON-formatted event messages from the *Utility Server*. Since those JSON messages do not have a 1-to-1 correspondence with the mobile device events, they need to be translated by the *Message-to-Event Mapper* component. For example, the iOS only supports a relative mouse device. This means that we cannot generate a "screen touch" event at specific (x, y) coordinates on a device; instead, we need to issue a series of low-level "move pointer" events that will relocate the pointer from its original location to (x, y), and follow it by a "click" event. Finally, the *Event Executor* component executes thus generated events by writing byte arrays of specific sizes to a binary file. This file represents the emulated USB device's buffer, from which the iOS device reads.

Screen Transmitter

After an extensive search for the solution that yields the best video capture and streaming performance, we opted for developing the *Screen Transmitter* as a native iOS app that relies on several different frameworks. For increased flexibility, we have implemented the *Screen Transmitter* app in both Swift and Objective-C.

Device-screen capture is based on Apple's native ReplayKit framework [2]. Specifically, it is implemented as a Broadcast Upload application extension. This is Apple's recommended way of implementing screen-sharing functionality, since it is the only way an app running in the background can acquire the screen. The transmission of acquired video frames is achieved through WebRTC, a real-time communication framework. Since the generic implementation of WebRTC lacks off-the-shelf support for iOS screen sharing, we used a specific instantiation of WebRTC, Twilio Video [11].

On application start-up, the *Screen Transmitter* acquires the Twilio Video access token from the *Utility Server* (recall Figure 1) and tries to connect to the video stream with the received token. If the token is valid and the network connection stable, a stream is opened. At this point, iOS starts publishing video frames that we are capturing using the above-discussed Broadcast Upload extensions's callback. WebRTC allows control over the frame sizes. For example, since newer iOS devices are of very high resolution, AirMochi can downscale the frames' sizes when used on slower networks to ensure adequate user experience. Finally, the captured frames are pushed to the video stream using the Twilio Video WebRTC API.

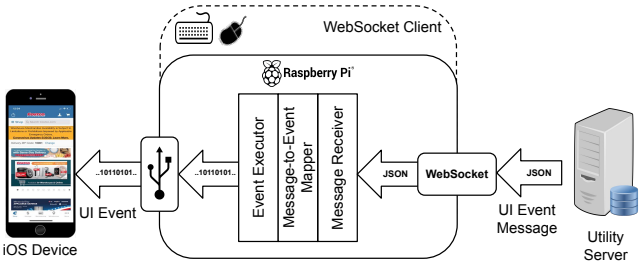

Figure 2: AirMochi's *Event Generator* is implemented by leveraging the iOS Accessibility Features.

Event Sequence Translator

The *Event Sequence Translator* component is written in Python and uses the Appium test automation framework [6]. Appium is a cross-platform open-source tool for automating native, mobile web, and hybrid applications on iOS, Android, and Windows desktop platforms. The *Event Sequence Translator* relies on Appium for extracting the UI layout tree of the current device screen, via Appium's `getPageSource()` method. The outputs of the *Event Sequence Translator* component are GUI-based, hardware-independent test cases for closed-source iOS apps. Specifically, the *Event Sequence Translator* generates Appium test-cases written in Python.

Model Extractor

The *Model Extractor* component is also written in Python. Since the *Event Sequence Translator* relies on Appium's features, the *Model Extractor*'s current implementation also relies on Appium's `getPageSource()` method to extract the information needed to define a UI state. As its output, the *Model Extractor* component generates a graph which is described in a graph description language (DOT) that represents an app's UI states and the transitions between them in the form of a graph.

5 PRELIMINARY EMPIRICAL EVALUATION

The primary objective of our work to date has been to explore different technologies that can be leveraged to build a solution that relies only on publicly available device-accessibility features. We discussed above how AirMochi's design has aimed to address our goals of efficiency, modularity, and extensibility. This section describes our empirical evaluation of AirMochi's effectiveness and efficiency. We focus on two aspects of effectiveness: (1) applicability to different mobile apps and scenarios and (2) accuracy. We evaluate efficiency in terms of the latency introduced by AirMochi. We note that the results we report here include but do not focus on the specific impacts of the *Event Sequence Translator* and *Model Extractor* components. These components are more recent additions to AirMochi and their evaluation is part of our on-going work.

As a demonstration of AirMochi's *applicability*, we selected ten of the most widely used Apple App Store apps: Amazon, Costco, Facebook, Instagram, Messenger, Netflix, Snapchat, TikTok, YouTube, and Zoom. We executed a variety of usage scenarios on these apps, ranging from 12 to 48 UI events. To measure AirMochi's device-screen streaming performance, we monitored the video stream *latency*, i.e., the time elapsed between capturing a frame on the *Mobile Device* and displaying it in the *User-Facing Application* (recall Figure 1). AirMochi's video stream latency was on average *248ms*, with all samples falling between 200ms and 300ms.

From the end-user's perspective, AirMochi's latency is virtually imperceptible since the use of our subject apps involves a fair amount of "user think time" [19], which is on the order of seconds.

However, AirMochi would need to reach near-real-time responsiveness if we wanted to use it with highly interactive applications such as games [5]. We believe that this is achievable since AirMochi's current, "proof of concept" implementation has not been optimized. We see opportunities for performance improvements by tailoring AirMochi's streaming protocol to fit the nature of specific use cases, by targeted uses of image downsampling, and by employing unidirectional video streaming rather than the general video conferencing currently supported by Twilio Video.

To evaluate AirMochi's *accuracy*, we define `True Positives` as events that are generated in the *User-Facing Application*, received by the *Event Generator*, and executed successfully on the *Mobile Device* (recall Figure 1). `False Negatives` are events generated in the *User-Facing Application* and received by the *Event Generator*, but not successfully executed on the device. Finally, `False Positives` are either events that are never generated in the *User-Facing Application* but are somehow executed on the device, or events that are executed out of the original order in which they were generated. We have not come across either of the `False Positives` cases throughout our use of AirMochi, meaning that AirMochi's `Precision` is 100%.

On the other hand, AirMochi's `Recall` is not perfect. We manually generated over 200 events across the ten subject apps. The events were of different types, such as keyboard inputs, taps, double taps, and swipes. We found that a small number of generated events were not executed on the *Mobile Device*, yielding the `Recall` of just above 96%. The events in question tended to be dropped regardless of whether they were generated manually in the *User-Facing Application* or by the *Utility Server* when replaying a scenario.

We have identified the low-level event processing in AirMochi's implementation (recall Figure 2) as the likely cause of the dropped events. Namely, certain events are simpler than others. For example, keyboard inputs result in just one byte being written by the Raspberry PI Zero, which acts as the *Mobile Device*'s USB peripheral. Keyboard inputs did not result in `False Negatives` in any of our tests. On the other hand, events such as taps and swipes are represented as series of bytes and do result in occasional `False Negatives`. We believe that the hardware limitations of the Raspberry Pi Zero are the potential reason for the dropped events. We continue to explore hardware platforms that may give us better results without sacrificing AirMochi's other desired properties.

6 CONCLUSION

Remote access to mobile devices is attractive for a range of reasons. Different solutions have tended to trade-off certain objectives and resulting properties for others. This has been especially the case with iOS, where the available solutions use various strategies to bypass the tight controls imposed on the platform by Apple. AirMochi has demonstrated that it is viable to use the public accessibility APIs to control an iOS device.

While AirMochi has been designed and implemented as a generally applicable proof-of-concept, we believe that it can be tailored and optimized for a host of specific scenarios. We identified several such scenarios in this paper. Our future work will also include combining AirMochi's record-and-replay capabilities with image processing, to identify an app's UI elements and support automated testing without having to rely on Appium.

AirMochi can also be used for analysis on extracted app models, including formally verifying a device's susceptibility to security attacks. Since AirMochi's model is in the form of a state machine—i.e. a graph—it is amenable to analysis by different graph traversal algorithms and formal verification frameworks. For instance, a vulnerability can be embodied in a state or a combination of states in the model. A formal analysis framework can then be used to look for paths on the behavioral model that satisfy the conditions needed for the existence of that specific vulnerability type.

Furthermore, AirMochi's platform-independence makes it useful for generating similar models for different platforms. For example, our work provides a promising opportunity to perform studies on comparing security vulnerabilities in iOS and Android apps.

7 ACKNOWLEDGMENTS

This work is supported by the U.S. National Science Foundation under grants 1717963 and 1823354, and U.S. Office of Naval Research under grant N00014-17-1-2896.

REFERENCES

[1] 2014. Veency, Cydia. https://cydia.saurik.com/info/veency/
[2] 2017. ReplayKit: Apple Developer Documentation. https://developer.apple.com/documentation/replaykit
[3] 2018. WebRTC. https://webrtc.org/
[4] 2019. WebDriverAgent on GitHub. https://github.com/facebookarchive/WebDriverAgent
[5] 2019. Wowza - 2019 Video Streaming Latency Report. https://www.wowza.com/blog/2019-video-streaming-latency-report
[6] 2020. Appium: Mobile App Automation Made Awesome. http://appium.io/
[7] 2020. Eggplant Software. https://www.eggplantsoftware.com/
[8] 2020. Getevent: Android Open Source Project. https://source.android.com/devices/input/getevent
[9] 2020. LongPressIos. https://developer.apple.com/documentation/uikit/uilongpressgesturerecognizer/1616423-minimumpressduration
[10] 2020. Raspberry Pi OS (previously called Raspbian). https://www.raspberrypi.org/downloads/raspbian/
[11] 2020. Twilio Video: Video SDKs for iOS, Android, JavaScript and web-based video. https://www.twilio.com/video
[12] 2020. W3C WebDriver. https://w3c.github.io/webdriver/
[13] Steven Arzt et al. 2014. 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* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299
[14] C. Bernal-Cárdenas et al. 2020. Translating Video Recordings of Mobile App Usages into Replayable Scenarios. *arXiv preprint arXiv:2005.09057* (2020).
[15] D. Geist, M. Nigmatullin, and R. Bierens. 2016. Jailbreak/Root Detection Evasion Study on iOS and Android. *MSc System and Network Engineering* (2016).
[16] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. 2013. RERAN: Timing- and touch-sensitive record and replay for Android. In *2013 35th International Conference on Software Engineering (ICSE)*. 72–81.
[17] Heejin Kim et al. 2009. Performance testing based on test-driven development for mobile applications. In *Proceedings of the 3rd International Conference on Ubiquitous Information Management and Communication*. 612–617.
[18] Y. K. Lee, J. Y. Bang, G. Safi, A. Shahbazian, Y. Zhao, and N. Medvidovic. 2017. A SEALANT for Inter-App Security Holes in Android. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 312–323.
[19] J. W. Mickens et al. 2010. Crom: Faster Web Browsing Using Speculative Execution. In *2010 USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Vol. 10. 9–9.
[20] T. A. Nguyen and C. Csallner. 2015. Reverse Engineering Mobile Application User Interfaces with REMAUI (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 248–259.
[21] Z. Qin, Y. Tang, E. Novak, and Q. Li. 2016. MobiPlay: A Remote Execution Based Record-and-Replay Tool for Mobile Applications. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 571–582.
[22] Wenyu Wang et al. 2018. An empirical study of android test generation tools in industrial cases. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 738–748.