AirMochi – A Tool for Remotely Controlling iOS Devices

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 AirMochi, a tool that provides remote access and control of apps by leveraging a mobile platform's publicly exported *accessibility features*. While AirMochi is designed to be platform-independent, we discuss its iOS implementation. We show that AirMochi places no restrictions on apps, is able to handle a variety of scenarios, and imposes a negligible performance overhead. https://youtu.be/rhPz2Hs4Ius https://github.com/nkllkc/air_mochi

ACM Reference Format:
Nikola Lukić, Saghar Talebipour, and Nenad Medvidović. 2020. AirMochi – A Tool for Remotely Controlling iOS Devices. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3324884.3415304

1 INTRODUCTION

The emergence of mobile platforms has resulted in millions of apps that deliver services spanning virtually every human need and endeavor. The providers of mobile platforms—Google's Android and Apple's iOS most notably—have tried to make app building easier for an inexperienced, even untrained, developer. However, this has introduced a host of challenges, both for the consumers and for developers. Namely, apps can have unexpected behaviors, counter-intuitive features, bugs, unpredictable performance, security vulnerabilities, etc. 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.

Existing work has looked at different facets of this problem, such as testing apps' behaviors, analyzing their performance, ensuring adherence to license agreements, identifying and patching vulnerabilities, recovering app designs, and reverse-engineering implementations. Of particular interest to our work are techniques that analyze and/or test apps on a device by remotely accessing and controlling them. Examples include app record-and-replay [10], UI event generation for apps running in an emulator [13], use-case scenario extraction from screen recordings [8], and application of OCR and computer-vision techniques to identify GUI widgets [12].

These techniques have invariably targeted Android, an open platform that provides access to low-level details of an installed app and its execution substrate. By contrast, iOS is a closed platform and is much less amenable to such analyses: no analogous APIs, tools, or emulators exist for it.[1] Two solutions for remote app access/control have been typically adopted on iOS. The first is to override certain kernel-level restrictions by "jailbreaking" a device (e.g., [1]). However, jailbreaking an iOS device introduces security vulnerabilities and results in platform and app versions that are considered illegal by Apple [9]. The second solution is to use the XCUITest black-box testing framework [3] and leverage low-level device-access protocols currently exposed by Apple. However, use of these protocols is not officially sanctioned by Apple and may be disabled in a future iOS version. Additionally, both of these approaches are complex and impose significant engineering burden.

In this paper, we present the design, implementation, and evaluation of AirMochi—Accessibility Interface Runner for Mobile Open Computer-Human Interaction. AirMochi is a remote app access/control tool that only leverages a mobile platform's publicly exported *accessibility features*, which are commonly provided to facilitate the use of a device by persons with vision, hearing, and other physical disabilities. While AirMochi's design is platform-independent, we demonstrate our solution on iOS, the significantly more challenging of the two currently dominant mobile platforms. We demonstrate AirMochi's (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.

2 AirMochi IN ACTION

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

[1] Apple's iOS *simulators* for MacOS require special simulator-based app builds.

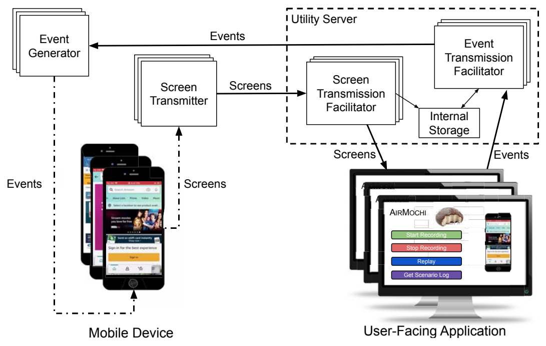

Figure 1: AirMochi's design. The dashed lines to and from the *Mobile Device* are intended to indicate that the *Event Generator* and *Screen Transmitter* components can reside on the device itself or be deployed on a separate hardware node.

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 usage 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*. This allows rapidly bringing a mobile app to a specific execution state, 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.

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), one of our eventual goals is to enable combining different approaches for screen streaming, user-side event generation, and device control. To this end, AirMochi comprises four top-level modules, as shown in Figure 1: (1) *User-Facing Application*, (2) *Utility Server*, (3) *Event Generator*, and (4) *Screen Transmitter*. 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.

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.

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.

4 AIRMOCHI's IMPLEMENTATION

AIRMOCHI is implemented in ≈2500 SLOC, spanning six programming languages with modules running on three different hardware 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 single-board computer. In the implementation reported in this paper, the *Screen Transmitter* component is running on the *Mobile Device* itself and is implemented as a native iOS application; recall from the above discussion that the *Screen Transmitter* can also be remote from the device. More detailed explanations about each of these modules are presented next.

User-Facing Application
A screenshot of the *User-Facing Application* is shown in Figure 2. The *User-Facing Application* combines HTML5 and multiple JavaScript frameworks and libraries, mainly: Twilio Video [7], which is an instantiation of the WebRTC real-time communication framework [4]; 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 3, 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 [6]. As shown in Figure 3, 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

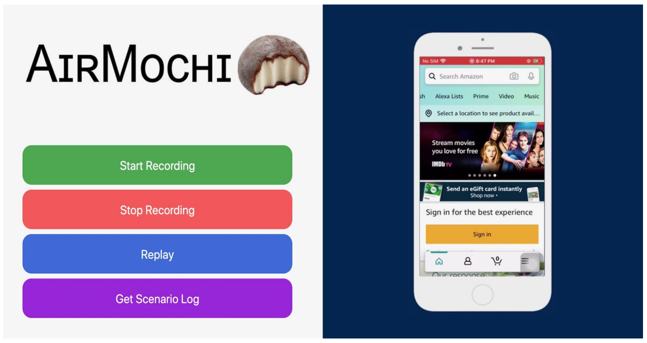
AIRMOCHI
Start Recording
Stop Recording
Replay
Get Scenario Log
Search Amazon
Alexa Lists
Prime
Video
Music
Select a location to see product avail...
Stream movies you love for free
Send an eGift card instantly
Sign in for the best experience
Sign in

Figure 2: Screenshot of the *User-Facing Application* remotely controlling an iPhone 7s running the Amazon app.

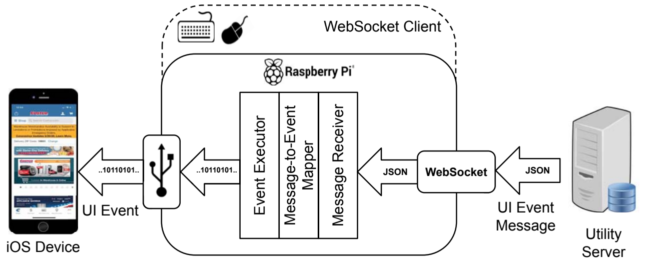

Figure 3: AIRMOCHI's *Event Generator* is implemented by leveraging the iOS Accessibility Features.

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 [7].

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.

5 PRELIMINARY 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.

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 across the different scenarios and apps 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" [11], 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 and used them in a large number of scenarios. 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 3) 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 only 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 above. 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. This will allow us to rapidly analyze an app's dynamic properties, extract use-cases from the acquired execution data, pinpoint the root causes of bugs, build high-fidelity behavior models of closed-source apps, and ultimately reverse-engineer entire 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] 2017. User Interface Testing - Apple Developer. https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/testing_with_xcode/chapters/09-ui_testing.html

[4] 2018. WebRTC. https://webrtc.org/

[5] 2019. Wowza - 2019 Video Streaming Latency Report. https://www.wowza.com/blog/2019-video-streaming-latency-report

[6] 2020. Raspberry Pi OS (previously called Raspbian). https://www.raspberrypi.org/downloads/raspbian/

[7] 2020. Twilio Video: Video SDKs for iOS, Android, JavaScript and web-based video. https://www.twilio.com/video

[8] C. Bernal-Cárdenas et al. 2020. Translating Video Recordings of Mobile App Usages into Replayable Scenarios. *arXiv preprint arXiv:2005.09057* (2020).

[9] D. Geist, M. Nigmatullin, and R. Bierens. 2016. Jailbreak/Root Detection Evasion Study on iOS and Android. *MSc System and Network Engineering* (2016).

[10] Wing Lam et al. 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*. 854–859.

[11] J. W. Mickens, J. Elson, J. Howell, and J. Lorch. 2010. Crom: Faster Web Browsing Using Speculative Execution. In *2010 USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Vol. 10. 9–9.

[12] 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.

[13] 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.