Data Synthesis Is Going Mobile—On Community-Driven Dataset Generation for Android Devices

MARKUS DEMMEL and THOMAS GÖBEL, University of the Bundeswehr Munich, Research Institute CODE, Neubiberg, Germany
PATRIK GONÇALVES, Central Office for Information Technology in the Security Sector (ZITiS), Munich, Germany
HARALD BAIER, University of the Bundeswehr Munich, Research Institute CODE, Neubiberg, Germany

Personal electronic devices such as smartphones and smartwatches have become indispensable daily companions, collecting a multitude of personal and sensitive data. As a result, they are of paramount importance in digital forensic examinations. However, there is a lack of publicly available and ready-to-use digital forensic datasets, especially in mobile forensics. This work presents a concept and an open-source proof-of-concept implementation, which simplifies and automates the creation of mobile forensic datasets within the scope of the Android operating system. In contrast to previous approaches, which populate the most common databases of an Android device, our concept is based on community-driven playbooks and makes use of interaction with the actual smartphone GUI. Hence, we are able to generate coherent and realistic traces as they occur in real-world human usage. Our proof-of-concept implementation is based on the standard Android emulation environment and borrows tools from the user interface testing community. Our evaluation shows that our approach actually generates realistic Android datasets. For instance, we can generate traces that cannot be simulated by gestures (e.g., changing the GPS position or triggering incoming phone calls). Recording the actual data synthesis process allows users to either create and share their own playbooks (i.e., the exact instructions for the data synthesis process rather than having to share the full image) or reproduce Android images with different scenarios using playbooks previously created and shared by the community.

CCS Concepts: • **Applied computing → Evidence collection, storage and analysis**; **Computer forensics**;

Additional Key Words and Phrases: Mobile forensics, Android image, Forensic dataset, Digital corpora, Data synthesis, Data generation, Data synthesis framework, UI testing, Android Emulator, User simulation, Human interaction

ACM Reference format:
Markus Demmel, Thomas Göbel, Patrik Gonçalves, and Harald Baier. 2024. Data Synthesis Is Going Mobile—On Community-Driven Dataset Generation for Android Devices. *Digit. Threat. Res. Pract.* 5, 3, Article 30 (October 2024), 19 pages.
https://doi.org/10.1145/3688807

1 Introduction

Over the last few decades, smartphones and wearables have become integral parts of our daily lives. People interact with these devices by either actively using them or passively wearing them while performing everyday

Authors' Contact Information: Markus Demmel, University of the Bundeswehr Munich, Research Institute CODE, Neubiberg, Germany; e-mail: ma.demmel@outlook.de; Thomas Göbel (corresponding author), University of the Bundeswehr Munich, Research Institute CODE, Neubiberg, Germany; e-mail: thomas.goebel@unibw.de; Patrik Gonçalves, Central Office for Information Technology in the Security Sector (ZITiS), Munich, Germany; e-mail: Patrik.Goncalves@ZITiS.bund.de; Harald Baier, University of the Bundeswehr Munich, Research Institute CODE, Neubiberg, Germany; e-mail: harald.baier@unibw.de.

This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2024 Copyright held by the owner/author(s).
ACM 2576-5337/2024/10-ART30
https://doi.org/10.1145/3688807

tasks. In each case, they collect various data that is stored either locally or in a remote cloud storage. This data typically contains valuable forensic information that might be used as evidence in court.

The forensic community relies on high-quality training and test datasets to ensure that law enforcement officers receive hands-on training and that digital forensics software and tools perform as expected. While real-world datasets contain realistic traces of user interactions with these devices, researchers may be unable or not allowed to obtain datasets from seized devices, as they may contain personal data, be protected by law, or be part of ongoing investigations [10]. According to their survey, only 3.8% of the newly created datasets were published. Therefore, the forensic community has created datasets from their own experiments and published them on various websites such as *Digital Corpora* [2] or *CFReDS* from NIST [13]. However, Gonçalves et al. [9] and Göbel et al. [6] discuss typical problems with these datasets, such as that they often significantly deviate from reality, are of poor quality, lack labels (i.e., a ground truth), contain only a few traces, or are quickly outdated. In a recent survey of published mobile forensic datasets, Gonçalves et al. [9] identified only 31 available datasets in the digital forensics domain and argued that the *dataset gap problem*, which Garfinkel et al. [5] already pointed out in 2009, still exists.

A mobile forensic dataset is usually a snapshot of the current state of a device, realized as a dump of its internal memory, sometimes as a physical, sometimes as a logical backup. These images may contain various kinds of data related to a device, system, and application-related files containing various user-created and modified content such as chat messages, browser history, and geo-tagged files. The available data and information on a mobile device depend on several factors. First, when interacting with a device, the user follows a specific behavior with individual preferences and routines (e.g., selecting a specific messaging app from a range of available apps). Second, using software tools to extract data from a device can lead to different extraction types (e.g., required root access for specific extraction methods). Third, devices run on various operating systems with brand and model-specific builds, which in turn may have different data structures, file systems, and security mechanisms.

At first glance, criminal content is no different from non-criminal content. Only after closely analyzing the exact content of a forensic image can a forensic investigator find the information relevant to a particular case. For smartphone contents, the investigator may find multiple file types (e.g., pictures and databases) from different applications (e.g., messaging, social media, banking), which in turn store other information (e.g., usage history, login data, and temporary files). This makes it necessary to dynamically adapt to different cases to find relevant information to answer a set of questions (e.g., where was the suspect at a particular time?) and is therefore not a trivial task.

The creation of such forensically sound datasets is not trivial either. Therefore, various authors have developed different tools in the past that facilitate the automatic generation of forensic datasets. Examples of recently published works in this area include the data synthesis frameworks `hystck` [8], `TraceGen` [3], and `ForTrace` [7]. However, none of the available tools currently supports the automated creation of reasonably realistic mobile content.

Contribution. The contribution of this article is as follows. We analyze existing approaches for the automated generation of mobile device images. In addition, we explore suitable means for UI testing of smartphone apps that will later be used to perform a fully interactive user simulation, particularly the scripted use of the smartphone GUI. Next, we create a concept and implement a new framework for the automatic generation of Android smartphone images. This includes interfaces to integrate our framework into existing data synthesis frameworks. To validate the framework, we identify the presence and significance of the generated digital traces within the Android image and compare them with a manually created image following the same storyline to ensure the correct functioning of our image generator. Finally, we provide open-source access to pre-configured playbooks and the source code of our framework via the following GitHub project: https://github.com/dasec/ForTrace/tree/android-synthesis-framework.

Article Outline. In Section 2, we discuss prominent data synthesis frameworks for both general desktop and mobile operating systems. Furthermore, we explore relevant mobile data generation tools. Based on our findings, we propose a methodology to create an image synthesizer to generate Android smartphone dumps in Section 3 and subsequently explain in detail our Python-based implementation in Section 4. Next, we create a test dataset to validate our methodology in Section 5. In addition, we use various dataset (generation) properties discussed in the literature to validate our framework and the generated images. Finally, we present the current limitations of the chosen approach and possible future works in Section 6 and conclude our article in Section 7.

2 Related Work

In this section, we review related work in the scope of mobile forensic dataset generation during the last decade between 2013 and 2023. We distinguish between general data synthesis frameworks in the field of digital forensics and data injection tools specifically designed for mobile devices (e.g., Android or iOS smartphones). Based on our literature review, we note that there is no fully automated Android synthesis framework that generates realistic traces that can be used in mobile forensics. We therefore extend our search to the Android testing community, which provides several utility tools for creating test cases (in particular for GUI testing purposes).

2.1 Digital Forensic Dataset Synthesis Frameworks

The `ForGe (Forensic Test Image Generator)` was introduced in 2015 by Visti et al. [15]. `ForGe` provides a UI and takes instructions in the form of database entries. In addition, the output contains images and information sheets. Although the tool is still available on GitHub, it has not been further developed since 2015 and thus seems no longer to be maintained. Furthermore, `ForGe` does not address mobile device images.

Another work in this area is the `EviPlant` framework by Scanlon et al. [14]. The framework uses a base image as a starting point. The challenges or traces can then be downloaded in the form of *evidence packages*. This has the advantage that large files do not have to be sent multiple times, which is particularly interesting for teaching purposes. The evidence packet must only be injected into the base image and the investigation can be started. However, no base images or injection traces are available for mobile devices.

In 2020, Göbel et al. [8] published `hystck`, a Python-based framework that can create network and hard disk traces. The creation can be automated using Python scripts or YAML (YAML Ain't Markup Language https://yaml.org/) configuration files. Automated synthesis makes it possible to create a wide variety of traces within a virtual machine with little effort, which can be distributed efficiently by defining the contents as changes from a template image. However, it currently only supports traces synthesized on Windows-based systems.

Du et al. [3] developed the framework `TraceGen` written in Python to automatically generate forensic images. They adopt the idea of taking a set of pre-defined user actions and injecting them into virtual disk images. This is done by an emulator that translates high-level actions and simulates user behavior by performing (sub)-operations inside a virtual machine, e.g., using an Internet browser or modifying files on a hard disk. All changes are stored on a disk image and simultaneously logged in a separate file that serves as the ground truth. Although they offer fully automated control over the emulator, their framework currently only supports emulation on Windows-based systems. Furthermore, the source code is unfortunately not published.

To our knowledge, the most recent framework is the `ForTrace` framework, developed by Göbel et al. [7], which builds upon the `hystck` framework [8]. The `ForTrace` framework supports the injection of traces through the simulation of human-computer interactions, and according to the authors, this approach is intended to create more realistic scenarios. Along with the resulting disk image, it also contains a log of the generated events that serves as the ground truth. However, it currently does not support any mobile operating system synthesis.

2.2 Data Injection Tools for Mobile Devices

In our literature review, we did not find a complete data synthesis framework for Android devices, so we extended our review to tools that can inject pre-processed data into (emulated) Android smartphones.

Delgado et al. [1] presented the `FADE` tool. It is a proof-of-concept to inject static traces in an Android-based emulator using the **Android Debug Bridge (ADB)**. The authors modify files and database entries in an Android virtual machine to mimic user-created content. Although they show that it is feasible to manually inject some traces that can be detected by the forensic toolkit, such as Autopsy, the tool provides limited support to inject other app-related information or to automate the injection process. Further, it does not offer the ability to interact with the Android UI.

Another injection tool in the context of mobile devices is `AutoPoD-Mobile` shown by Michel et al. [12]. It is a proof-of-concept framework for generating Android datasets using ADB, APIs from selected apps, and a Google account. As mentioned in the article, the tool only works on a few physical and meanwhile outdated Android devices (e.g., Samsung A50, Huawei Mate 20 Lite). Similar to `FADE` [1], this framework can inject some user-generated traces by automatically inserting a set of pre-defined content, for example, a set of pictures, a list of calendar events, or messages. Although their framework is capable of injecting traces into various smartphone models, it does not provide the ability to synthesize this content through automated interaction with the device's UI and thus does not create important artifacts within the Android OS that would typically occur when interacting through the UI with the Android OS.

Both `FADE` [1] and `AutoPoDMobile` [12] populate devices by injecting information into an Android device by either directly modifying files or using various APIs to mimic device usage. However, human interactions with the device, particularly with its UI, are not addressed, and thus, potential digital traces are not covered. We argue that synthesizing human interactions by creating appropriate events on the UI, e.g., by simulating touch events on the smartphone, is similar to humans interacting with a device. This approach should produce more realistic forensic traces inside the OS instead of simulating API calls or adapting the respective files.

2.3 Android Test Data Generation Tools

Since only the two data generation tools `FADE` and `AutoPoD-Mobile` were found in the field of mobile forensics, we expanded our literature search to the Android testing community. App developers use a wide range of tools and techniques to model and generate various test cases. In general, most techniques try to simulate all possible states on the app- or system-wide levels and observe how inputs might influence an app or the system. These might be used to find faulty behavior or security leaks in applications.

Two notable tools are the `AndroidViewClient (AVC)`[1] and `UI Automator.`[2] Both tools provide similar functionalities in user-friendly environments for creating test scripts for emulated Android devices. Although they provide similar functionalities, the AVC is a Python-based implementation, while the UI Automator is a Java application integrated into Google's Android Studio. Both mimic user inputs by interacting with the emulator or executing pre-processed scripts. Among other functionalities, the scripts can invoke UI events on GUI elements and thus create touch events similar to those that occur during real human interactions. In addition, the AVC provides built-in functions to support Android's UI Automator functions.

Although we recognize more recently published works in the field of test data synthesis and the Android testing community, the current state of the art does not support the synthesis of realistic content as found on typical real-world Android devices. While the testing community's main focus is not on generating datasets, its tools and techniques could still help create a framework to synthesize traces on an (emulated) device. Therefore,

[1]https://github.com/dtmilano/AndroidViewClient
[2]https://developer.android.com/training/testing/other-components/ui-automator

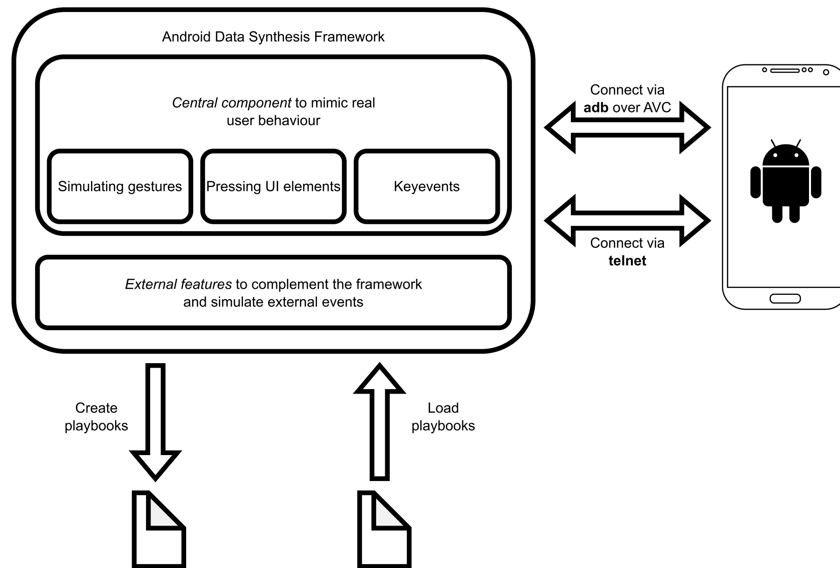


Fig. 1. Structure and workflow of our proposed Android data synthesis framework.

we aim to develop a data synthesis framework that supports simulated user actions to evoke various events on an emulated device and finally address the need for realistic synthetic Android smartphone images.

3 Concept and Methodology

In this section, we describe our goal to implement and evaluate a synthetic framework for realistic images of Android devices utilizing tools commonly used in UI testing. We present a concept using the AVC test framework that can replicate real human interactions and thus synthesize realistic app data for Android smartphones on an emulated device. To better understand our data synthesis approach, Figure 1 illustrates our framework's exact structure and workflow. In the following, we will explain each component in detail.

Human interactions with a smartphone can be simplified down to a few basic actions. These include (1) pressing elements on the device's UI, for example, to write a message, take a picture, etc., (2) navigating through the OS using various gestures such as touching, swiping, and long tapping, and (3) passively interacting with the device, for example by wearing the device while walking and thus invoking different events from various device's sensors. By replicating these events using a synthetic framework, we can create artificial user input and thus digital artifacts on the device without knowing the apps' internal data structures or databases. In direct contrast to `FADE` [1] and `AutoPoD-Mobile` [12], this approach is less prone to changes in non-UI software updates. Moreover, we do not have to deal with the apps' internal functions and data structures. Therefore, it is essential to understand that we follow an entirely novel approach to synthesize data for digital forensic purposes by directly interacting with the device's UI in an automated way rather than simply populating the most common Android databases.

In the following subsections, we first demonstrate how to mimic the most essential Android user interactions, in particular, executing gestures, pressing UI elements on the display, and executing special key events, e.g., pressing the physical power button, that are bundled in our *central component*. The simulation of human interactions is complemented by other relevant functions that affect smartphone usage, such as incoming calls, text messages, and changes in GPS settings.

3.1 Mimicking Gestures

In our context gestures mean all non-singular user input, i.e., that use one or multiple fingers (or other media) that change its position. People use gestures to navigate on the smartphone, such as scrolling, changing views between multiple pages, and much more. The use of synthetic gestures is one of the three key features of the *central component*. In order to synthesize a specific gesture, we need to either track the actual path of the gesture event (i.e., storing the actual coordinates of the finger(s)) or offer a set of pre-calculated gesture events in the execution of the framework. The first approach may need constant adaptations of the original gesture, for example, to adapt to different screen sizes and altering content. Thus, we decided to offer the most basic swipe gestures in one direction only, i.e., moving the position of a finger towards the screen edge. In most cases, this simplification might be sufficient to synthesize most UI inputs to navigate the Android system.

To realize this, we need to dynamically calculate the starting and ending position of the swipe gesture according to the device's screen size. The AVC provides methods to determine the properties of the current emulated device. This serves as a basis for defining eight basic swipe gestures along both coordinate axes, i.e., top, down, left, and right. In addition to the direction, we label "light" and "hard" swipes to create small and respective big swipe variations differing in length. To execute a gesture, we would again use the AVC's *drag()* method to simulate the gesture.

3.2 Key Events

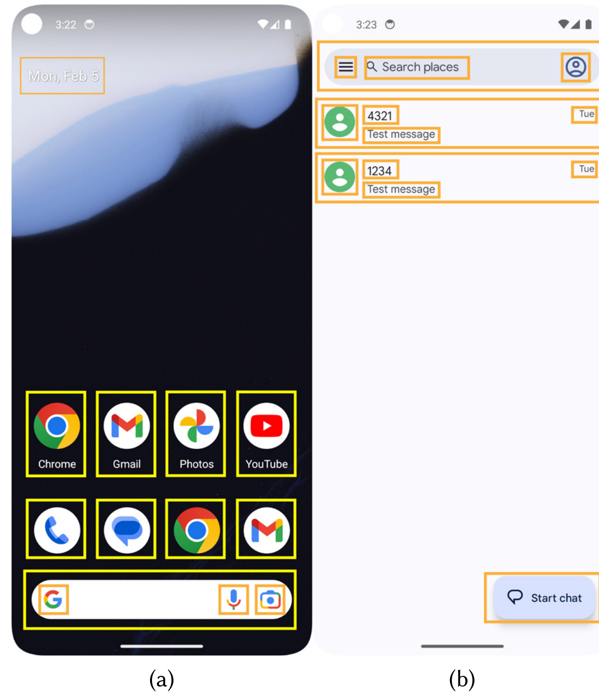
Since the release of Android 10 in 2019, users can use gestures for system navigation. That is, the common home, return, and app overview buttons can be replaced by using gestures. To replicate those gestures more reliably and faster, we decided to complement the gestures using *key events*.[3] Key events are mapped to certain actions and can be used for device interactions, device input, and the simulation of button events (e.g., home, volume button).

3.3 Replicating the Pressing of UI Elements

While gestures describe a user input that follows a specific path, we also have events that (almost) do not change its position. Most commonly, this describes touch events to click on buttons and other UI elements. Similarly, as described in the previous Section 3.1, we also suggest dynamically calculating the position of the UI element and thus where the desired touch events should be evoked. To determine the interactable UI elements in a current view, we use the AVC *dump()* function. This method uses Android's UI Automator[4] and parses its XML output, resulting in a hierarchical view of all UI elements. We use the pre-processed hierarchical view of the displayed UI elements as a starting point before processing the available information. This view depicts all UI elements with their corresponding attributes, particularly an **identifier (ID)**, a label, and a description, all of which are required to trigger touch events on certain elements on the emulated device. Exemplified in Figure 2(a) and (b), we have encircled most of the UI elements that the AVC processes and returns with its attributes so that we can further process the given information. In Figure 3, we can see an excerpt of the attributes belonging to one of the framed Google Chrome applications on the home screen from Figure 2(a).

Once we have processed the hierarchical view elements, we list every possible element with which the user can interact. This enables identifying the desired element, either by a unique ID or by interpreting the contents of the element's attributes, e.g., a label containing 'search places' to browse through available conversations as depicted in Figure 2(b). Consequently, the framework filters out an element based on its given attributes (such as ID and label) rather than using X and Y coordinates, which improves the re-usability on different devices with different screen sizes or build versions. Lastly, the identified element is executed using the AVC's *touch()* method.

[3] https://developer.android.com/reference/android/view/KeyEvent
[4] https://developer.android.com/training/testing/other-components/ui-automator



(a) (b)

Fig. 2. Example of clickable UI elements on an Android home screen and in Google's Messages app. The colored rectangles depict the bounding boxes for the elements with which the user can interact.


```
'index' = {str} '4'
'text' = {str} 'Chrome'
'resource-id' = {str} ''
'class' = {str} 'android.widget.TextView'
'package' = {str} 'com.google.android.apps.nexuslauncher'
'content-desc' = {str} 'Chrome'
'checkable' = {str} 'false'
'checked' = {str} 'false'
'clickable' = {str} 'true'
'enabled' = {str} 'true'
'focusable' = {str} 'true'
'focused' = {str} 'false'
'scrollable' = {str} 'false'
'long-clickable' = {str} 'true'
'password' = {str} 'false'
'selected' = {str} 'false'
'bounds' = {tuple: 2} ((641, 1691), (842, 1859))
'uniqueId' = {str} 'id/no_id/9'
__len__ = {int} 18
```


Fig. 3. Example of the AVC's *dump()* method showing all attributes for the Google Chrome shortcut.

3.4 Complementing the *Central Component* with External Features

In the previous sections, we discussed how user interactions can be reliably simulated using gestures, key events and touch events. In addition, one of the main features of smartphones is that modern devices contain a variety of sensors to measure incoming signals sent by the device's surroundings. Most smartphones include a microphone (audio signals), a gyroscope (motion signals), a GPS module (GPS satellite signals), and an antenna (networking signals), among other sensors. In contrast to direct interactions on the UI (e.g., touch events), these indirect

device-related events originate from external sources that the user cannot fully control, as they can be modified by some noise signals of the device's environment. For example, the GPS module converts multiple satellite signals to pinpoint the exact geographical location but can be affected by noise, such as buildings and obstacles. Since these features are crucial for a realistic synthetic framework, we need appropriate functions that complement the *central component*. As external sources trigger these events, we call the addition to the *central component* "external" features. Our prototype of the synthetic framework should support the simulation of the most basic *external features*. These are incoming calls that can be declined or accepted, receiving and **sending text messages (SMS)**, and manipulating GPS coordinates. To implement the *external features*, we use additional tools supplied with the Android Software Development Kit. For instance, using a `telnet` client,[5] we can connect to an emulator and send special emulator commands that allow us to influence external factors.

3.5 Scenario Creation and Simulation

Now that we have explained the main methods of the synthetic framework to simulate human behavior, we demonstrate the creation of an actual synthesis scenario. As a result, we want to show how to create the so-called playbook. The playbook contains a detailed description consisting of an ordered set of individual actions necessary to recreate a scenario. An action can be a gesture, a simple UI click event, or an *external feature*. The information within the playbook should be human-readable and adaptable as some information (e.g., the content of messages, phone numbers of incoming calls, etc.) may be easily changed in the playbook. A significant advantage of our approach emphasizes the exchange of community-driven playbooks that can be easily adapted to a customized scenario. As a result, we chose to store our configuration files in the YAML format as depicted in Listing 1. The indentation-sensitive YAML format can be exchanged quickly within the community, allowing us to store various actions as key-value pairs.

```
1  Name: Test
2  Description: Test description
3  Gesture: SWIPE UP
4  Pressing UI Element: Google Chrome
5  External: CALL 1234
```

Listing 1. Abstract Example of a Playbook.

3.6 Device Platform

To improve the availability and reproducibility of forensic datasets, our framework design and structure focus mainly on smartphone emulators, although the approach would also be applicable to physical, rooted devices. In detail, we adapt for Google's **Android Virtual Device (AVD)**[6] Manager in the emulation of Android devices. Although other emulators (e.g., Genymotion) can also be used as long as the used image supports a connection via ADB and `telnet`, AVD is already included in Google's *Android Studio*. The AVD Manager offers a list of previously published Android versions and provides support for a wide range of virtual devices, models and builds. Emulated devices enhance reproducibility and usability in contrast to physical devices. The AVD manager also supports different Android versions containing Google Play services, versions supporting Google API usage, and versions for the Android Open Source Project. Root access is necessary as some commands require higher privileges (e.g., using ADB to acquire a logical copy from the emulated device). However, there is no way to gain root access to images that run Google Play services. Therefore, we use a build without those services enabled and without a pre-installed Google Play Store. Although this may limit the availability of some services and apps, other app stores with similar functions and apps are still available via the emulator and can be installed.

[5] https://developer.android.com/studio/run/emulator-console
[6] https://developer.android.com/studio/run/managing-avds

3.7 Extraction of the Synthesized Data

To forensically analyze the resulting data after the data synthesis process, we need to extract the generated data first. In most cases, this includes information stored in Android's *userdata* partition, whose extraction using the `adb pull` command requires root privileges. Only a logical extraction is possible for the *userdata* partition, as it is protected by full disk or file-based encryption in modern Android versions.

4 Implementation

In this section, we describe how we implemented the concepts from Section 3 to support several human interactions included in our *central component*, i.e., the simulation of gesture events, key events, and touch events on UI elements. We also describe how we complement our framework with additional *external features*. Next, we describe how we can create a scenario with a set of actions to generate abstract configuration files that can be shared with others. Furthermore, we show how to generate forensic artifacts using previously created community-driven playbooks that support a wide range of Android OS versions and variants. To enhance the framework's usability, we created a simple **command-line interface (CLI)** for control mechanisms and to read and create playbooks easily.

4.1 Implementing Gestures

As mentioned in Section 3.1, we offer the user a pre-defined set of the most common swipe gestures. To calculate the exact position required to perform a swipe gesture, we initially determine the display size of the emulated device. After the device is connected to the framework via *Viewclient.connectToDeviceOrExit()*, We use the AVC's *Viewclient.display()* function to get the boundaries of the actual device size. The code Listing 2 illustrates the calculation of the "soft" and "hard" left swipe gestures.

```
1  left_swipe_coordinates = (0, 0, 0, 0) # right to left
2  height = self.get_display_height()
3  width = self.get_display_width()
4  # x, y; top left coordinate is (0, 0)
5  mid_left = width * 0.2, height / 2
6  mid_right = width * 0.8, height / 2
7  mid_center = width / 2, height / 2
8  # Set coordinates
9  left_swipe_coordinates = (mid_right, mid_center)
10 left_swipe_long_coordinates = (mid_right, mid_left)
```

Listing 2. Example for Determine the Start and End Coordinates for the Normal and Long Left Swipe Gesture.

If the user decides to perform a gesture, we present the set of possible gestures to the user. In addition, each entry is provided with a consecutive number. Such a gesture can be executed by stating the number of the gesture to the framework using the CLI.

4.2 Implementing Key Events

Similar to swipe gestures, we offer key events to the user (as shown on Android for Developers[7]). The most common key events are shown in Listing 3. While the aforementioned swipe gestures are mainly used for (horizontal and vertical) scrolling through apps, key events can be used for reliable navigation through the system using software-based or physical buttons.

[7] https://developer.android.com/reference/android/view/KeyEvent

```
1  KEYCODE_HOME # Value 3 => adb shell input keyevent 3
2  KEYCODE_BACK # Value 4 => adb shell input keyevent 4
3  KEYCODE_VOLUME_UP # 24 => adb shell input keyevent 24
4  KEYCODE_VOLUME_DOWN # 25 => adb shell input keyevent 25
```

Listing 3. A Short List of the Most Common Key Events and the Resulting Command. The User Can Chose from All Key Events.

4.3 Implementing Touch Events on UI Elements

To synthesize touch events on various UI elements, i.e., by pressing an interactive element visible on the display, we use the AVC's *dump()* and *touch()* method. While the *dump()* method lists all interactable UI elements and their attributes, the *touch()* method simulates a touch event on the display, as previously described in Section 3.3. The Listing 4 depicts a selected part of the view shown in Figure 2(b) created by the AVC's *dump()* method and post-processed by our framework. It shows a hierarchical view on all UI elements in the current view. Each node is dynamically numbered with an ID and the class name it belongs to, e.g., ID 0 identifies the root node. However, reloading the view in a more dynamic environment may change the order of the elements and thus the numbering. Therefore, using the ID to identify the same element in subsequent actions may not be sufficient. In addition, the dump provides the attributes associated with a node, particularly a non-unique app-specific ID (*R_ID*).

```
1  ID: 0, CLASS: android.widget.FrameLayout [...]
2  .ID: 1, CLASS: androidx.drawerlayout.widget.DrawerLayout[...]
3  ..ID: 2, CLASS: android.view.ViewGroup [...]
4  ...ID: 3, CLASS: android.view.ViewGroup [...]
5  ....[...]
6  ...ID: 9, CLASS: android.view.ViewGroup [...]
7  ....ID: 10, CLASS: android.view.ViewGroup [...]
8  .....ID: 11, CLASS: android.view.ViewGroup [...]
9  ......ID: 12, CLASS: android.widget.EditText [...]
10 .......[...]
11 ......ID: 15, CLASS: android.support.v7.widget.RecyclerView
12 .......ID: 16, CLASS: android.view.ViewGroup
13 ........ID: 17, CLASS: android.widget.TextView, R_ID: com.google.android.apps.messaging:id/conversation_name, TEXT: 4321,
       ↪ TAG: None, clickable: false, CONTENT_DESCRIPTION: None, com.google.android.apps.messaging
14 ........ID: 18, CLASS: android.widget.TextView, R_ID: com.google.android.apps.messaging:id/conversation_snippet, TEXT: Test
       ↪ message, TAG: None, clickable: false, CONTENT_DESCRIPTION: None, com.google.android.apps.messaging
15 ........ID: 19, CLASS: android.widget.TextView, R_ID: com.google.android.apps.messaging:id/conversation_timestamp, TEXT: Tue,
       ↪  TAG: None, clickable: false, CONTENT_DESCRIPTION: None, com.google.android.apps.messaging
16 .......ID: 20, CLASS: android.view.ViewGroup
17 ........ID: 21, CLASS: android.widget.TextView, R_ID: com.google.android.apps.messaging:id/conversation_name, TEXT: 1234,
       ↪ TAG: None, clickable: false, CONTENT_DESCRIPTION: None, com.google.android.apps.messaging
18 ........ID: 22, CLASS: android.widget.TextView, R_ID: com.google.android.apps.messaging:id/conversation_snippet, TEXT: Test
       ↪ message, TAG: None, clickable: false, CONTENT_DESCRIPTION: None, com.google.android.apps.messaging
19 ........ID: 23, CLASS: android.widget.TextView, R_ID: com.google.android.apps.messaging:id/conversation_timestamp, TEXT: Tue,
       ↪  TAG: None, clickable: false, CONTENT_DESCRIPTION: None, com.google.android.apps.messaging
20 ........[...]
```

Listing 4. Excerpt of the Tree-Like Structure.

Although identification through the use of the app-specific ID works well for simple views, identification can be challenging for views that list similar items. As depicted in Figure 2(b) we have two conversations that contain similar UI elements. An excerpt of the associated dump of the hierarchical view is listed in Listing 4, where the children of ID 16 relate to contact *4321* and ID 20 to contact *1234*, respectively. The child's attributes, in particular, the IDs, do not differ for both contacts, e.g., the TextView class with `id:/conversation_name` holding the contact's name. Only by interpreting the attribute's contents can we safely determine to which conversation a UI element belongs, e.g., the contents of the `TEXT:` attribute from IDs 17 and 21. A similar problem arises in the home screen view from Figure 2(a) where the Google Chrome application can be started by clicking either on

one of the two icons: at the left and on the favorite app bar's icon. Thus, we need to consider the context of the UI elements to synthesize the correct touch input reliably. Therefore, we decided not only to match for attributes but also to store the complete hierarchy of a single element to identify the element correctly. In the example from Listing 4, we need to store the information as depicted in Listing 5 to simulate a touch event to open the conversation for contact *1234*. In the execution of the playbook, we can then compare the attributes of the given UI element and its context, i.e., text, resource-id, class, package, and content description. While we can use this method on any virtual device for the recording phase of our playbook, we are limited to similar devices during the synthesis phase because the view may differ between different states of a device, thus not being identical. Therefore, we recommend using a newly set-up device for the recording and the simulation.

```
1 android.view.ViewGroup
2 ID: 0, CLASS: android.widget.FrameLayout [...]
3 .[...]
4 .......ID: 20, CLASS: android.view.ViewGroup
5 ........ID: 21, CLASS: android.widget.TextView, R_ID: com.google.android.apps.messaging:id/conversation_name, TEXT: 1234,
      ↪ TAG: None, clickable: false, CONTENT_DESCRIPTION: None, com.google.android.apps.messaging
6 ........ID: 22, CLASS: android.widget.TextView, R_ID: com.google.android.apps.messaging:id/conversation_snippet, TEXT: Test
      ↪ message, TAG: None, clickable: false, CONTENT_DESCRIPTION: None, com.google.android.apps.messaging
7 ........ID: 23, CLASS: android.widget.TextView, R_ID: com.google.android.apps.messaging:id/conversation_timestamp, TEXT: Tue,
      ↪  TAG: None, clickable: false, CONTENT_DESCRIPTION: None, com.google.android.apps.messaging
8 ........[...]
```

Listing 5. Information Stored When Pressing the Second Message.

4.4 Implementing Additional External Features

In our prototype, we already support some *external features*. That is, changing the GPS coordinates, initiating an incoming call (i.e., accepting or rejecting the incoming call) and receiving an SMS message. In addition to the actual action in the simulated case, these *external features* are also relevant for the generation of appropriate background noises. In Listing 6, we can see the commands used and their purposes.

```
1 Change GPS coord.: geo fix {longitude} {latitude} {altitude}
2 Initiate incoming call: gsm call {number}
3 Accept incoming call: gsm accept {number}
4 Cancel call: gsm cancel {number}
5 Receive SMS: sms send {number} {message}
```

Listing 6. *External Features* to Complement the *Central Component*.

While these features can also be accessed via the Google's AVD Manager's GUI, it may not be suitable for simulating complex scenarios. Thus, we initiate a `telnet` connection with the target device and send various commands that can be found on the corresponding Android developer website.[8] For example, the `gsm` command simulates inbound phone calls and establishes and terminates data connections. The user can decide which commands to execute by interacting with the tool's CLI.

4.5 Recording a New Scenario

In the previous sections, we described the functions of the *central component* and its complementing *external features*. The set of actions to execute a playbook, as proposed in Section 3.5, is stored in a configurable YAML file that describes the steps to execute a customized scenario. With Algorithm 1, we demonstrate the general process of recording and storing a scenario. To facilitate the recording process, we provide a CLI implementation to the user. The user is guided through the recording by prompting the available set of actions.

[8] https://developer.android.com/studio/run/emulator-console

Algorithm 1: Recording a New Scenario

```
Ensure: key event HOME inserted
  scenario_name ⟵ user_input()
  scenario_description ⟵ user_input()
  while scenario_recording do
      list_available_actions(ui_elem, gestures, external)
      action ⟵ user_input
      if action == ui_elem then
          ui_elem_list ⟵ dump_curr_view_elements
          list_all_ui_elements
          selected_ui_elem ⟵ user_input(chosen_ui_elem)
          if ui_elem requires user_input then
              text ⟵ user_input(text)
              press the corresponding ui_elem
              insert text in ui_elem
              press key event enter
              store selected_ui_elem in playbook
              store text in playbook
          else
              press the corresponding ui_elem
              store selected_ui_elem in playbook
          end if
      else if action == gesture then
          list possible gestures
          selected_gesture ⟵ user_input(gesture)
          execute selected_gesture
          store selected_gesture in playbook
      else if action == key_event then
          list possible key events
          selected_key_event ⟵ user_input(key event)
          execute selected_key_event
          store selected_key_event in playbook
      else if action == external then
          list possible externals
          selected_external ⟵ user_input(external)
          parameters ⟵ user_input()
          execute selected_external
          store selected_external in playbook
          store parameters in playbook
      end if
  end while
```

To ensure a global starting point, we must enforce that each recording starts at the home screen by first evoking the corresponding key event, i.e., the home button. Each action from the configuration file is then successively executed according to its type: actions related to gestures, key events, UI elements, or *external features*. Each action is incrementally numbered and subdivided by indenting the associated commands and additional information. For instance, Listing 7 shows the contents of the playbook to simulate actions opening the Google Chrome application (action number 0), then receiving a SMS message (action number 1), changing the current GPS position (action number 2) and returning to the home screen (action number 3). For UI element-related actions, we first need to identify the element by providing the associated parameters containing contextual information, as previously stated in Section 4.3. Therefore, we need to store the complete contextual information under PARAMETERS: for identifying the target element. Solely, the parameter in the line 28 containing TEXT_TO_INSERT: can be customized to insert additional information, e.g., entering an HTML address. After identifying the target element, the algorithm further checks for additional user input, e.g., a text field for inputting an HTML address. In this case, the algorithm selects the text field, inserts the provided text, and finalizes it by executing the key event for *Enter*. The intermediary steps and the element's context information are stored in the playbook. In all other cases, only the element's context and the execution of a touch event are stored. For gesture events, the framework provides a list of possible events through the CLI with the associated code. In this case, the playbook contains a type describing the gesture, e.g., left swipe. The actual positional information is dynamically calculated during

the simulation. Similar to the gesture, the user can decide to add a key event in the same manner by providing the associated ID as described in Section 4.2. As for the recording of an *external feature*, the user can choose between the selection mentioned in code Listing 6 by interacting with the CLI of the framework. Depending on the selected feature, the user is required to insert additional information (GPS coordinates, phone number, etc.) to be stored as parameters in the configuration file.

```
1  name: android-example
2  description: Open Google Chrome, receive SMS, change GPS data, return to home screen
3  0:
4    TYPE: VIEW
5    SUBTYPE: android.widget.TextView
6    PARAMETERS:
7          [...]
8          TEXT_TO_INSERT: None
9  1:
10   TYPE: EXTERNAL
11   SUBTYPE: SMS
12   PARAMETERS:
13     MESSAGE: '"Test message"'
14     NUMBER: '+123456789'
15 2:
16   TYPE: EXTERNAL
17   SUBTYPE: GEO
18   PARAMETERS:
19     LONGITUDE: '10'
20     LATITUDE: '20'
21     ALTITUDE: '30'
22 3:
23   TYPE: KEYEVENT
24   SUBTYPE: KEYCODE
25   PARAMETERS:
26     KEYCODE: 3 # Pressing the home button
```

Listing 7. Example Playbook That Opens Google Chrome, Receive SMS, Changes the GPS Position and Returns to Home Screen.

4.6 Simulation of a Given Scenario Using a Playbook

We need to pass one or multiple playbooks to the framework to simulate a scenario. Each playbook consists of one or more actions. As the configuration files are executed one after the other, both the sequence of actions within a configuration file and the sequence of the configuration files themselves are important. For example, we must first open a browser and download a file to be able to open the file manager and install the downloaded file.

Reproducing an image based on an existing playbook is simple and is illustrated in Algorithm 2. Depending on the type of action, the framework either interacts with a UI element, performs one of the possible gestures, or executes a key event or one of the external commands to complement the feature set of the framework.

4.7 Extracting the Synthesized Data

After finalizing the synthesis, the framework can create extractions of the `userdata` partition using ADB's `adb shell`. This tool enables us to use a Unix shell within a (emulated) device and is able to run various commands. It allows the creation of a logical backup of the encrypted *userdata* partition by using `adb shell pull /path/to/partition/ /destination/path`. Complete physical extraction of the *userdata* partition is not possible due to file-based encryption.

5 Validation

In this section, we validate the framework in three steps. First, we compare the data synthesized with our framework to manual interactions on a physical device. Second, we determine whether we can find any

Algorithm 2: Simulation of a Scenario Using a Playbook

```
load playbook
insert keycode HOME
while configuration_file has item left do
    action ← next item from configuration file
    if action type of ui_ACTION then
        if subtype requires user_input then
            text ← stored input_text
            click ui_elem
            insert stored text
            press keycode enter
        end if
    else
        click ui_elem
    end if
    if action type of gesture then
        perform gesture
    else if action type of keyevent then
        execute keyevent
    else if action type of external then
        read parameters from configuration file
        execute external
    end if
end while
if extraction wanted then
    extract userdata partition
end if
```

framework-specific traces on the emulated device when using the supported functions of the *central component* and the *external features*. Both are covered in Section 5.2. Third, we validate the current state of our framework in general against other synthesis approaches in the field using established properties in Section 5.3.

5.1 Test Scenario

To mimic a real user and determine the scope of functionality, we created a proof-of-concept scenario that reflects a wide range of user interactions that can be expected when using the device in real life. We created a full-day scenario consisting of the following actions: browsing websites, taking photos, sending e-mails, installing applications, writing and receiving messages, making and receiving calls, using Google Maps for navigation, using a third-party application for communication, and taking some notes. An excerpt from the corresponding playbook can be seen in Listing 7. The emulated device running on Google's latest *API level 34, Android 14* is reset before running a playbook. In addition, each app is launched once to skip any tutorial pop-ups that might interfere with the synthesis process. Using the same procedure, we reset the physical device to factory settings and launch each app to create a common base.

5.2 Comparison of the Two Test Datasets and Validation of the Functionality of the Framework

To compare the two generated images fairly, we mainly used the SANS Institute cheat sheets by Epifani [4] and Mahalik et al. [11], as well as common forensic tools such as *Autopsy*, *DB Browser for SQLite*, and *ExifTool*. As we can only extract a logical image due to Android's protection mechanisms, we focus on the generated data of the individual apps located in the *userdata* partition, which is mounted under */data*. Table 1 shows the activities performed in our test scenario described in Section 5.1. We also indicate where the corresponding artifacts are located and whether the artifacts are presented just like on a physical device.

As we visited various websites on both devices, we took into account the browser activity, i.e., we analyzed the associated databases *History* and *Cookies* in Android and the files containing the tabs in */app_tabs/0/**, which allows us to confirm the indistinguishability between the data from the manually operated device and the synthesized content.

Table 1. Individual Activities Performed in Our Test Scenario, Artifacts Generated in Each Case, and an Indication of Whether the Generated Traces Met Our Expectations

Activity	Most relevant analyzed artifacts and its location in /data	**Fulfilled**
Browse websites	**data/com.android.chrome:**	✓
	↪ app_chrome/Default/History	
	↪ app_chrome/Default/Cookies	
	↪ app_tabs/0/*	
Take photos	**media/0/Pictures/***	(✓)
Send e-mails	**data/com.google.android.gm:**	✓
	↪ databases/bigTopDataDB.<USER_ID>	
	↪ databases/peopleCache_EMAILADDRESS_com.google_11.db	
Install applications	**data/com.android.vending:**	(✓)
	↪ databases/localappstate	
	↪ data/app*	
	↪ data/system/packages.list	
Write and receive messages	**data/com.android.messaging:**	✓
	↪ databases/bungle_db	
	data/com.android.providers.telephony:	
	↪ databases/mmssms.db	
Make and receive calls	**data/com.android.providers.contacts:**	✓
	↪ databases/calllog.db	
	data/com.google.android.dialer:	
	↪ databases/dialer	
Navigation with Google Maps	**data/com.google.android.apps.maps:**	(✓)
	↪ cache/image_manager_disk_cache	
	↪ databases/gmm_storage.db	
	↪ files/saved_directions.data.cs	
Chatting using WhatsApp	**data/com.WhatsApp:**	✓
	↪ databases/media.db	
	↪ databases/wa.db	
	↪ files/*	
Creating some notes	**data/com.google.android.keep:**	✓
	↪ databases/keep.db	
	↪ shared_prefs/*	

Legend: ✓ = fulfilled, (✓) = partially fulfilled, ✗ = not fulfilled.

When analyzing the camera application, we noticed a difference compared to the real device. Although both devices created a picture in the *media/0/Pictures/** folder, it was limited to the picture's content. While you can take a picture of any object with a real device, the emulator uses a camera application specifically for the emulated device that creates a random sample picture. The pre-installed camera application does not create any interesting traces (e.g., fill databases, cache, etc.) in its home directory. So far, we cannot use another camera application on the emulated device (e.g., Google's default camera application) and produce traces with them. However, we can use a workaround to at least upload an arbitrary picture using `adb push /source/path /destination/path`.

Next, we turn to the forensic traces of the *Gmail* app. Our analysis does not reveal differences between the two devices, as similar traces are present on both the emulated and the physical device in *com.google.android.gm*, i.e., the *bigTopDataDB.<USER_ID>* and *peopleCache_<EMAILADDRESS>_com.google_11.db*.

Due to the limitations of Android Studio's emulated devices, we cannot initially use the Google Play Store. However, we can either install Google Play Services manually or use third-party app stores such as *APKMirror*, *F-Droid*, *Aurora Store*. In our approach, we used *APKMirror*[9] to install applications successfully, but unfortunately this only creates traces within the corresponding home directory *com.apkmirror.helper.prod* of the application instead of the Google Play Stores *com.android.vending*.

Next, we analyzed the traces we created when writing and receiving SMS. We looked at the *bungle_db* and *mmssms.db* and compared their content for both devices. We found no differences between the two images.

We also looked at the incoming and outgoing calls in *com.android.providers.contacts*. Among other files, we examined the *calllog.db* and could not find any differences, as we were able to find all initiated and incoming calls in the files mentioned.

To simulate navigation, we used *Google Maps* to start navigation and update GPS coordinates. In this way, we were able to create similar traces to real navigation. However, we noticed less data in the *gmm_storage_table* table of *gmm_storage.db* in the synthesized image, probably due to the different frequency of position updates.

Finally, we analyzed the communication via *WhatsApp* and the creation of notes with *Google Keep*. We examined the most common databases and files of *WhatsApp* by analyzing the *media.db*, *wa.db* and the data located in *files/**, which resulted in no difference between devices. By examining the *keep.db* of Google Keep and the data in the *shared_prefs* folder, we can confirm the indistinguishability of this app-generated data, too.

After comparing the app-generated data on both images, we also looked for traces created by the synthetic framework itself. We want to discover if we are creating other unwanted traces that do not make sense. On the bright side, we could only find limited traces related to `adb` on which the complete synthesis approach is based, as some of the key indicators for an `adb` connection were missing in */data/adbroot*, */data/misc/recovery/last_log*. We believe this might be because the emulated device in Android Studio is designed to connect via `adb`.

5.3 General Framework Validation with Given Properties

With the available activities listed in Table 1, we can already mimic a significant part of the real user behavior on Android smartphones. To validate data synthesis frameworks in this area in general, Göbel et al. [7] proposed a set of properties that such a framework and the resulting datasets should meet to increase acceptance in the forensic community. Table 2 lists the properties and indicates whether our approach meets them. Each property is explained in more detail below.

Free and Open Source Availability. This property is given because our framework and its publicly available source code are explicitly intended for the community to share playbooks for mobile forensic data synthesis.

Holistic Quality. As we currently cannot extract the emulator's RAM and network capture, we do not yet fulfill this property. However, this feature can be added in the future.

Realistic Quality. By synthesizing an image using a playbook and comparing the outcome with a similar manual data creation approach, we have shown that we can create indistinguishable traces for certain activities.

Templates. Templates are provided by the framework in the form of playbooks. These are used to (1) create dynamic and configurable scenarios and (2) in the form of pre-configured playbooks to generate realistic background noise.

Customization. Without having to re-record an entire scenario, the user can freely edit, delete, or add details for the execution of various *external features* and gestures (e.g., GPS coordinates, top swipe, etc.). However, for pressing UI elements we can still improve our current approach by outsourcing recurring parts from the playbook.

Uniqueness. Currently, the framework does not support a function to generate completely random images, resulting in a deterministic behavior. However, during the synthesis process, the user can choose between different pre-defined playbooks (e.g., to initiate multiple calls, browse different websites, etc.) to be executed. This can be

[9] https://www.apkmirror.com

Table 2. Reviewing the Properties the Community Expects from a Data Synthesis Framework

Property	Fulfilled
Free and open-source availability	✓
Holistic quality	✗
Realistic quality	(✓)
Templates	✓
Customization	(✓)
Uniqueness	(✓)
Ground truth	✓
Usability	(✓)
Timeliness	✓
Activity	✓

Legend: ✓ = fulfilled, (✓) = partially fulfilled, ✗ = not fulfilled.

used to generate background noise. However, the order of execution must be considered. Further ideas on how this can be improved are given in Section 6.

Ground Truth. A playbook can not only be used to synthesize data with the framework but can also serve as a basis for the underlying ground truth since every action executed during a scenario is described in the respective YAML configuration file. However, to get a better understanding of the exact execution time of each action, we also use a separate ground truth file that contains both the action itself from the configuration file and the execution time. In addition, we plan to integrate relevant triggered events on the emulated Android device that are logged by Android's `logcatmonkey`[10] circular buffers for log messages.

Usability. Currently, interaction with the prototype can be done via the framework's CLI. Although the interaction is straightforward, selecting the desired UI element can be difficult occasionally (e.g., when browsing websites with a lot of dynamic content).

Timeliness. By supporting AVD Manager, we can synthesize content for a wide range of devices with different builds and OS versions. As AVD Manager is highly customizable, data synthesis should also be possible on newer devices.

Activity. We support different Android OS versions and emulated devices right from the start, which enables the exchange of playbooks in the community. In addition, the framework can be easily maintained in the future due to its open-source policy.

In summary, as shown in Table 1, our proof-of-concept already synthesizes various activities. Moreover, we fulfill most of the key properties of a data synthesis framework by generating indistinguishable traces in certain areas within a community-driven project.

6 Limitations and Future Work

Although we could create many indistinguishable traces with our framework, we still encountered some limitations, so we could not fulfill all the properties mentioned in Table 2. One limitation is that we cannot create retrospective data, as altering the emulator's system time can lead to undesired behavior in some apps (e.g., WhatsApp, outdated certificates, websites, etc.).

Since we rely on a real-time GUI-based approach based on simulated human interactions when using community-shared playbooks, the framework must ensure that the target device has a similar environment to the original

[10] http://developer.android.com/tools/help/logcat.html

setting in which the playbook was recorded (i.e., similar build version, pre-installed applications required for the scenario, same shortcuts on the home screen). In general, the framework is more resistant to changes. Especially background changes (e.g., databases) are not relevant to our framework. While our approach can handle UI-elements rotation (e.g., landscape vs. portrait mode), it cannot operate with removed or newly added UI elements. However, it can be adapted by creating a new recording of the playbook. For future work, we plan to bundle all pre-requisites to the configuration file to customize the target device automatically, pre-install all required applications, and initially start applications to skip possible tutorials. Additionally, we can use Google's `monkey`[11] to start applications regardless of their location on the device's screen, resulting in a more reliable behavior of the framework, and thus skipping initial startups containing potential tutorial pop-ups.

Furthermore, some limitations exist due to Google's protection mechanisms, such as a missing Android production image that includes the Google Play Store. Even if alternative app stores can mitigate the impact of the missing Google Play Store, we will never be able to produce the same traces as an image containing these services.

Since the key for file-based encryption in Android is part of the memory contents, acquiring the physical image on most modern devices (even with root access) is not easy. Since we could not examine the entire physical image, additional traces of the data synthesis process within the OS (e.g., traces in some logging files, metadata, file system, user credentials, etc.) may have been overlooked.

Although we may be able to generate forensic traces on multiple devices, the current approach does not automatically synchronize multiple devices, e.g., sending messages between two instances of emulated devices.

Another improvement would be to make the emulator's memory and network capture available to the user. Although we are able to synthesize near-realistic data, we still need to bypass some of the emulated device's natural limitations, like using the camera and the GPS-related functions. Even though playbooks are fully customizable, we intend to improve them further by minimizing non-editable overhead and thus making them more feasible. The main focus of this framework was to record and execute community-driven playbooks with known information. In real device usage, we find additional background noise that complements any case-relevant information. The `monkey` tool may be used to create random touch events while navigating websites or using apps. As the `monkey` tool cannot create meaningful content, future work should also consider synthesizing randomized realistic background noise, e.g., while navigating websites or generating GPS tracks for navigation apps, to create more sophisticated scenarios with unique content.

As for ground truth data, the Android command `logcat`, which reads several circular buffers for log messages and can also be executed in the ADB, can be used to display all events in the Android operating system in real-time. To determine if a triggered synthesis instruction was executed in the Android system, we plan to integrate `logcat` to check if everything worked and to integrate relevant feedback of the command into the ground truth file.

Last but not least, integrating our approach into existing data synthesis frameworks such as `ForTrace` [7] would be conceivable to enable synthesis on different platforms.

7 Conclusion

We conclude that our framework can generate traces for most everyday activities with an Android smartphone, similar to traces from real-world device usage. Due to our proof-of-concept scenario, we demonstrated the potential of our data synthesis framework in general, the automatic generation of traces using playbooks, and their indistinguishability for most cases. We believe the novel approach of simulating human interactions rather than populating the most common databases and files will generate more realistic forensic traces. Although we have shown that the general approach works and that data synthesis using our *central component* and its

[11] https://developer.android.com/studio/test/other-testing-tools/monkey

complementary *external features* can be used to synthesize realistic app data, the framework is still limited in certain cases, e.g., using the emulator's built-in camera app.

A more abstract validation of the framework led to several future working tasks, the most crucial part of which was the ability to generate background noise more easily. Several workarounds were discussed to overcome an emulated device's natural limitations. In trying to fulfill the outstanding properties, our goal is to integrate our approach into a data synthesis framework such as `ForTrace` [7], providing the community with a synthesis tool with a broader scope. Moreover, we believe that our and similar approaches will play an essential role in closing the *dataset gap* in mobile forensic datasets previously identified by [9, 10].

Acknowledgments

The authors would like to thank Mathias Mitterhofer for the valuable discussions and suggestions on this article. In addition, the authors would like to thank the anonymous reviewers for their feedback, which helped to improve this article.

References

[1] Alberto A. Ceballos Delgado, William B. Glisson, George Grispos, and Kim-Kwang Raymond Choo. 2021. FADE : A forensic image generator for Android device education. *WIREs Forensic Science 4,* 2 (June 2021), e1432. DOI: https://doi.org/10.1002/wfs2.1432

[2] Digital Corpora. 2023. Digital Corpora - Producing the Digital Body. Retrieved January 20, 2023 from https://digitalcorpora.org/

[3] Xiaoyu Du, Christopher Hargreaves, John Sheppard, and Mark Scanlon. 2021. TraceGen: User activity emulation for digital forensic test image generation. In *Proceedings of the 1st Annual DFRWS APAC Conference*.

[4] Mattia Epifani. 2021. Android Third-Party Apps Forensics–Reference Guide. [Online]. Poster-Online Ressource. Retrieved February 4, 2023 from https://www.sans.org/posters/android-third-party-apps-forensics/

[5] Simson Garfinkel, Paul Farrell, Vassil Roussev, and George Dinolt. 2009. Bringing science to digital forensics with standardized forensic corpora. *Digital Investigation* 6 (2009), S2–S11.

[6] Thomas Göbel, Harald Baier, and Frank Breitinger. 2023. Data for digital forensics: Why a discussion on "how realistic is synthetic data" is dispensable. *Digital Threats* 4, 3 (October 2023), Article 38, 18 pages. DOI: https://doi.org/10.1145/3609863

[7] Thomas Göbel, Stephan Maltan, Jan Türr, Harald Baier, and Florian Mann. 2022. ForTrace–A holistic forensic data set synthesis framework. *Forensic Science International: Digital Investigation* 40 (2022), 301344. DOI: https://doi.org/10.1016/j.fsidi.2022.301344

[8] Thomas Göbel, Thomas Schäfer, Julien Hachenberger, Jan Türr, and Harald Baier. 2020. A novel approach for generating synthetic datasets for digital forensics. In *Advances in Digital Forensics XVI*. Gilbert Peterson and Sujeet Shenoi (Eds.), Springer International Publishing, Cham, 73–93.

[9] Patrik Gonçalves, Klara Dološ, Michelle Stebner, Andreas Attenberger, and Harald Baier. 2022. Revisiting the dataset gap problem–On availability, assessment and perspective of mobile forensic corpora. *Forensic Science International: Digital Investigation* 43 (2022), 301439. DOI: https://doi.org/10.1016/j.fsidi.2022.301439

[10] Cinthya Grajeda, Frank Breitinger, and Ibrahim Baggili. 2017. Availability of datasets for digital forensics–And what is missing. In *Proceedings of the 17th Annual DFRWS Conference*, S94–S105.

[11] Heather Mahalik, Domenica Lee Crognale, and Mattia Epifani. 2021. The Most Relevant Evidence per Gigabyte. [Online]. Poster-Online Ressource. Retrieved February 4, 2023 from https://sansorg.egnyte.com/dl/ljmbARD8io

[12] Margaux Michel, Dirk Pawlaszczyk, and Ralf Zimmermann. 2022. AutoPoD-mobile—Semi-automated data population using case-like scenarios for training and validation in mobile forensics. *Forensic Sciences* 2, 2 (March 2022), 302–320. DOI: https://doi.org/10.3390/forensicsci2020023

[13] NIST. 2023. The CFReDS Project. Retrieved January 20, 2023 from https://www.cfreds.nist.gov/

[14] Mark Scanlon, Xiaoyu Du, and David Lillis. 2017. EviPlant: An efficient digital forensic challenge creation, manipulation and distribution solution. Digital Investigation 20 (Mar 2017), S29–S36. DOI: https://doi.org/10.1016/j.diin.2017.01.010

[15] Hannu Visti, Sean Tohill, and Paul Douglas. 2015. Automatic creation of computer forensic test images. In *Computational Forensics*. Springer, 163–175.

Received 21 April 2024; revised 21 April 2024; accepted 3 July 2024