The Android Forensics Automator (AnForA): a tool for the Automated Forensic Analysis of Android Applications

The Android Forensics Automator (AnForA): a tool for the Automated Forensic Analysis of Android Applications[⋆]

Cosimo Anglano[1], Massimo Canonico[1,*], Marco Guazzone[1]

[a]*Computer Science Institute, DiSIT, University of Piemonte Orientale, Viale T. Michel 11, 15121 Alessandria, Italy*

Abstract

Most of our daily activities are carried out by means of mobile applications, that typically generate and store on the device large sets of data. The forensic analysis of these data thus plays a crucial role during an investigation, as it allows to reconstruct the above activities. Manually analyzing these applications is a long, tedious, and error-prone task.

In this paper we present the design, implementation, and evaluation of AnForA, a software tool that automates most of the activities that need to be carried out to forensically analyze Android applications, and that has been designed in such a way to yield various important properties, namely fidelity, artifact coverage, precision, effectiveness, repeatability, and generality.

AnForA is based on a dynamic "black box" approach, in which the application to be analyzed is first installed on a virtualized Android device, and then a set of experiments are carried out, in which actions of interest are automatically performed on the application by emulating a human user that interacts with its interface. During the experiments, the file systems of the device storage are actively monitored, so that the data created or modified by each one of these actions can be located and correlated with that action.

We have devised a proof-of-concept implementation of AnForA, that we use to assess its ability in achieving its design goals, by analyzing through it several Android applications already studied in the literature, so that we can compare AnForA's results against those reported in these papers.

The results of our evaluation confirm that AnForA greatly simplifies the forensic analysis of Android applications, and exhibits all the properties mentioned above, namely fidelity, artifact coverage, precision, effectiveness, repeatability, and generality, to a higher extent than previous studies published in the literature.

Keywords: Digital forensics, mobile forensics, Android applications, digital evidence, automated forensics analysis.

1. Introduction

Mobile devices are an integral part of our everyday lives. More often than not, they play a key role in all our activities, and not only in our interpersonal communications. Most of our daily activities (e.g., eating, sleeping, doing sport, driving, interacting with other people, etc.) are indeed carried out, at least in part, by using suitable apps installed on our mobile devices. These apps generate and store on the device large sets of data, that may be later used to reconstruct the activities carried out by the user on the device. Hence, the forensic analysis of these applications may (and usually does) play a crucial role during an investigation.

To reconstruct user activities starting from the data generated by a given application, the analyst needs to know (a) which data are generated by the application, (b) how these data are encoded, (c) where these data are stored on the device, and (d) the data generated or modified by each operation allowed by the application. In this way, it is indeed possible to establish the causal correlation between the data generated by the application and the user action that led the application generate it. Once these correlations have been established in the general case, it is possible to infer, from the presence of certain data on a given device, whether a given action may have been performed or not on that device.

Unfortunately, gaining the above knowledge is usually a rather complex affair. It indeed involves to perform a set of controlled experiments, in which (a) the analyst carries out each action of investigative interest (e.g., sending a text message), (b) the internal and external storage of the devices are inspected to determine which data are generated by these actions and where they are stored, and (c) these data are analyzed in order to decode them and to assign them their correct meaning [3]. Doing so in a systematic manner, where all the relevant actions allowed by the

[⋆]This research has received the financial support of the Università del Piemonte Orientale.

[∗]Corresponding author

Email addresses: `cosimo.anglano@uniupo.it` (Cosimo Anglano), `massimo.canonico@uniupo.it` (Massimo Canonico), `marco.guazzone@uniupo.it` (Marco Guazzone)

applications are considered, is a long, tedious, and error-prone task.

Moreover, the huge and always growing number of apps available to users, as well as the frequent updates of existing ones, places a hard-to-sustain burden on the analyst, given that a new application (or a new version of an existing application) must be analyzed before the reconstruction of user activity may take place. For these reasons, the idea of automating the forensic analysis of mobile applications has recently received the attention of the scientific community [7, 28, 47].

In order to be adequate, an automated solution for the forensic analysis of mobile applications should provide:

1. *fidelity*, i.e. the ability to reproduce, as faithfully as possible, the interactions that a human experimenter would have with the application under analysis in order to perform the actions of investigative interest;
2. *artifact coverage*, i.e. the ability to identify all the data that are generated/modified by the application (either directly or indirectly, e.g., through another application or a system service) as effect of the above actions and that are stored on the device storage; [1]
3. *precision*, i.e. the ability to include only the data generated/modified by the application (either directly or indirectly);
4. *effectiveness*, i.e. the ability to correlate each user action of interest with the data it modified/generated;
5. *repeatability*, i.e. the ability to provide to a third party the possibility of replicating the same set of experiments and to obtain the same results;
6. *generality*, i.e. the ability of analyzing any mobile application on as many different Android devices as possible (possibly all).

Existing proposals for the automation of the forensic analysis of mobile applications [7, 28, 47] exhibit only a subset of the above features (see the related work in Sec. 2), hence they do not represent a completely satisfactory solution.

[1]This property shall not be confused with the *code coverage* metric used to evaluate the completeness of static and dynamic code analysis techniques [35].

In this paper we fill this gap by proposing AnForA (acronym for *Android Forensics Automator*), a system that automates the forensic analysis of Android applications. AnForA is based on a novel analysis methodology (that extends the approach proposed in [3]), that ensures the achievement of repeatability, and generality. This methodology provides the basis for the design of a software architecture whose components interact among them to achieve fidelity, artifact coverage, precision, and effectiveness. Furthermore, these components fully automate the execution of the experiments required to characterize the behavior of mobile applications, the collection of the results generated in them, and the correlation of each action performed in these experiments with the data they generate.

In particular, AnForA is based on a dynamic "black box" approach, in which the application to be analyzed is first installed on a virtualized Android device, and then a set of *experiments* are carried out, in which actions of interest are automatically performed on the application by emulating a human user that interacts with its interface. During the experiments, the file systems of the device storage are actively monitored, so that the data created or modified by each one of these actions can be located and correlated with that action.

We have devised a proof-of-concept implementation of AnForA that couples off-the-shelf software components already available in the Android ecosystem with components purposely developed by us. We experimentally evaluate the ability of AnForA in achieving the goals mentioned above, by using the above implementation to carry out the forensic analysis of several Android applications already studied in the literature, so that we can compare AnForA's results with those reported in these papers. The results of our evaluation confirm that AnForA greatly simplifies the forensic analysis of Android applications, and exhibits all the properties mentioned above, namely fidelity, artifact coverage, precision, effectiveness, repeatability, and generality, to a higher extent than previous studies published in the literature.

The rest of this paper is organized as follows. In Sec. 2 we discuss related works. Then, in Sec. 3 we present the design and the implementation of AnForA, while in Sec. 4 we illustrate its practical use by using as example the Gmail app. Next, in Sec. 5 we report the results of the experimental validation of AnForA, and in Sec. 6 we conclude the paper and outline future research work.

2. Related work

The forensic analysis of mobile applications has received a considerable attention in the recent literature [1, 2, 3, 22, 25, 29, 31, 44, 45, 46], where a large set of different mobile applications have been analyzed in order to identify and decode the artifacts they store on the device where they run. In all these works, the analysis (i.e., the identification of the artifacts, their location, and their decoding) has been carried out manually. However, the high complexity of the applications makes the manual approach cumbersome, time consuming, and prone to errors. Hence, the need for an automated solution clearly emerges from these works.

The problem of automating the forensic analysis of mobile applications has indeed recently received a significant interest in the literature, where various proposal – focusing on the automatic identification and decoding of the data that are generated by Android applications during their execution – have been published [7, 28, 47].

These approaches rely on the analysis of the application code, that can be performed either statically [21] or dynamically [20] (which are regarded as complementary approaches, as the strengths of one of them corresponds to the weaknesses of the other and vice versa [12]).

Static approaches examine a program without executing it, and are potentially able to reveal all possible paths of execution. As discussed in [21], these approaches have been successfully applied to identify errors and vulnerabilities in programs [26, 5, 14, 13, 24] and for software verification [9, 11]. However, they are prone to issues arising from program whose behavior cannot be determined without running the program. More specifically, in the case of Android applications [35], static analysis is challenged by (a) reflection involving encrypted/obfuscated strings, (b) dynamic loading of code at runtime, (c) the use of native code, and (d) the use of inter-application communication mechanisms. As discussed in [34], this may yield to inaccuracies in those tools based on static analysis.

Conversely, dynamic approaches execute a program and observe the results, so they are able to deal with programs whose behavior depend from runtime conditions. As discussed in [20], these approaches have been successfully used for discovering memory access errors and memory leaks [23, 30, 33] as well as for detecting deadlocks and data race conditions [10, 37, 4]. However, they have code coverage problems as they may overlook code paths that are not taken during the execution. More specifically, in the case of Android applications [35], dynamic analysis is challenged by (a) how to monitor and collect data about the runtime behavior of applications, (b) the effective generation of test input, and (c) the impact of system events.

As already mentioned, both static and dynamic analysis techniques have been used to automate the forensic analysis of Android applications. However, as discussed below, these proposals are characterized by various drawbacks, which are due to the challenges mentioned before.

ForDroid [28] and EviHunter [7] combine static code analysis, to discover all the possible execution paths in the application code, with taint analysis, to track the flow of relevant information from their *sources* (i.e., the places in the code where these data are generated) to their *sinks* (i.e., the places in the code where these data are written to the file system). These tools suffer from the following drawbacks:

1. Sources and sinks, that correspond to specific methods of the Android API (e.g., those that are used to obtain GPS coordinates) or to specific variable types (e.g., strings), must be known in advance. However, gaining such knowledge is a non trivial task, as it requires a complex analysis of the Android API, that must be repeated each time the above API changes [7].
2. They achieve partial precision: they do not consider – as starting point of the analysis – the set of user actions chosen as target by the analyst. Hence, they cannot filter the code paths exhaustively generated by static analysis to exclude those that do not originate from target user actions. Therefore, they identify also data that are generated by uninteresting (from the forensic point of view) actions.
3. They lack effectiveness: another consequence of not requiring the specification of target user actions is that these tools cannot correlate each user action with the data it generates.
4. They achieve partial artifact coverage: being based on static analysis, they are unable to deal with code paths that cannot be determined statically (see the challenges discussed above); furthermore, being based also on taint analysis, they have issues with highly obfuscated code [32].

[47] proposes instead a tool that relies on dynamic taint analysis, whereby the application of

interest is executed on a modified version of the ART Android runtime, and the data of interest is tracked as it flows through the various parts of the application code.

This tool, however, suffers from some of the drawbacks affecting EviHunter and ForDroid, namely (a) lack of effectiveness and partial precision (as the analysis is driven by randomly-generated user actions), (b) partial of artifact coverage (it is unable to deal with applications that use native code, or that exhibit implicit data flows [47]), and (c) it requires the knowledge of the sources and sinks in the applications.

To avoid the above drawbacks, AnForA adopts – as already mentioned – a dynamic "black box" approach in which the application is run on a virtualized device without analyzing its code (neither statically nor dynamically), and the data it generates (in response to analyst-defined target user actions) is monitored and collected. In this way, it is able to achieve an artifact coverage level higher than the tools mentioned above. Furthermore, being its analysis driven by analyst-chosen target user actions, (a) it is able to identify only those data that are created or modified as direct consequence of these actions, disregarding instead data generated by other and uninteresting actions (thus achieving an precision level higher than its alternative counterparts), and (b) it is able to correlate each one of the chosen actions with the data it generates (thus achieving effectiveness). It is worth to point out that AnForA's goal is not that of uncovering all possible application paths, but only to uncover those paths corresponding to target user actions, assuming that the analyst is able to conceive all of them.

3. The AnForA System

As anticipated in the Introduction (Sec. 1), AnForA is based on a methodology for the forensic analysis of mobile applications that has been specifically conceived in order to provide repeatability and generality. This methodology provides the basis for its architecture, that encompasses various components that interact among them to provide fidelity, artifact coverage, precision, and effectiveness through the full automation of the execution of analysis experiments, the identification of the location and format of all (and only) the data generated during these experiments, and their correlation with the actions that generated them.

In this section, we first describe the analysis methodology (Sec. 3.1), then we describe the architecture of AnForA (Sec. 3.2) as well as its proof-of-concept implementation (Sec. 3.3), and finally we discuss how AnForA is able to achieve its design goals (Sec. 3.4).

3.1. The Analysis Methodology

The methodology providing the foundations of AnForA is based on the design of a set of experiments, each one focusing on one of the operations allowed by the application under analysis (e.g., sending a text message or a picture), on their systematic execution using the application on a mobile device, on the inspection of the device storage during and after each experiment (so as to identify the data generated during it), and on the analysis of the generated data to determine their meaning and context.

Generality and repeatability, that are the main goals of this methodology, are achieved through the use of virtualized mobile devices in place of physical ones. A mobile virtualization platform makes indeed simple and cost-effective to run experiments on a multitude of different virtual mobile devices (featuring different hardware and software configurations), thus yielding generality. Furthermore, it allows a third-party to easily replicate experiments on the same mobile device models and configurations, as well as to enforce the same operational conditions holding during the experiments, thus yielding repeatability.

The methodology, whose workflow is schematically depicted in Fig. 1, is articulated in a sequence of steps, as discussed below.

First, the functionalities of the application under consideration are analyzed, in order to identify its actions that have a potential investigative interest, and then suitable experiments, aiming at eliciting the generation of the data corresponding to the above actions, as well as their storage on the local memory of the device, are designed.

Next, in the *Application installation* step the application is installed on the device. Then, in the *Application footprint characterization* step, the set of directories that contain data generated either directly or indirectly by the application (the *Analysis Paths*) are identified, so that they can be monitored during the execution of the experiments in order to detect changes to the data they store. Furthermore, the data generated during the installation is also collected and examined, and

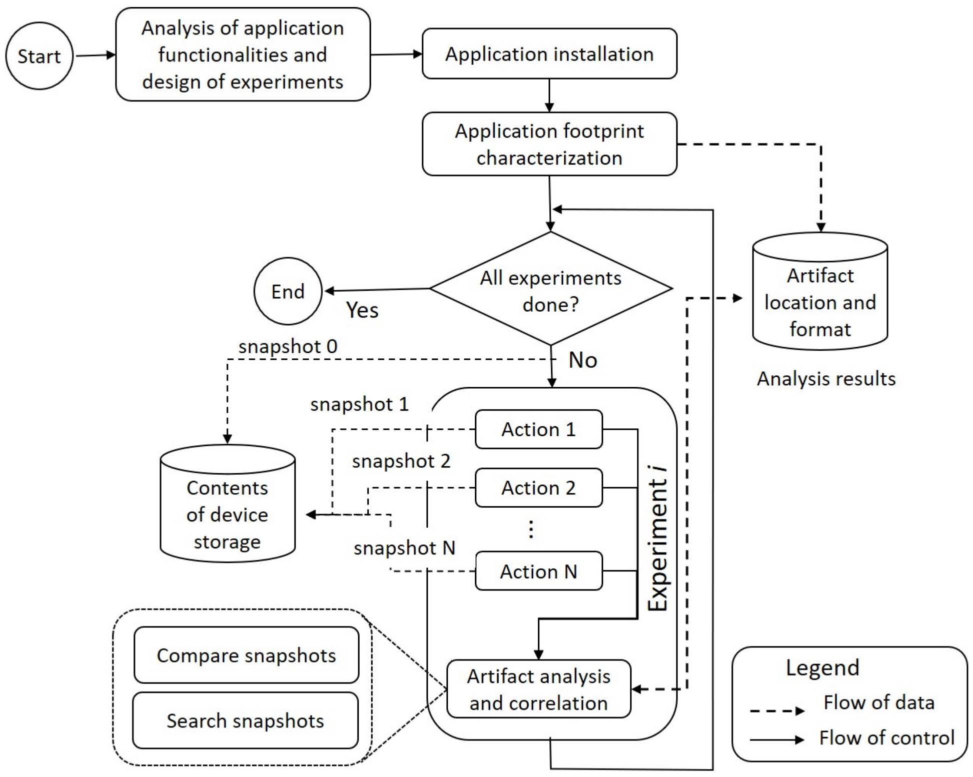


Figure 1: Workflow of the analysis methodology for mobile applications.

the results of this analysis are stored into the *Artifact location and format* database.

After these preparatory steps, the set of experiments is carried out in a systematic way, until all of them have been completed. As shown in Fig. 1, each experiment consists in a set of actions, carried out by the analyst in a predefined order by interacting with the application user interface. At the beginning of the experiment, and also after each action of the experiment is completed, a snapshot of the contents of the *Analysis Paths* may be collected and stored for subsequent analysis.

After all the actions of a given experiment have been completed, the various snapshots are compared in order to identify which files have been created, deleted and/or updated as effect of each action. Furthermore, snapshots may be searched for known information (e.g., the text of a message that has been sent) to determine the data that have been written in, or deleted from, the above files. These findings, jointly with the association of each artifact with the (set of) action(s) that generated them, are recorded into the *Artifact location and format* database.

3.2. The Architecture

To carry out the analysis according to the above methodology, AnForA couples a mobile virtualization platform, enabling the configuration of, use of, and interaction with a virtual device where the app under analysis is installed and executed, with an *analysis machine*, where its software components run and where the data extracted from the mobile device are stored and analyzed.

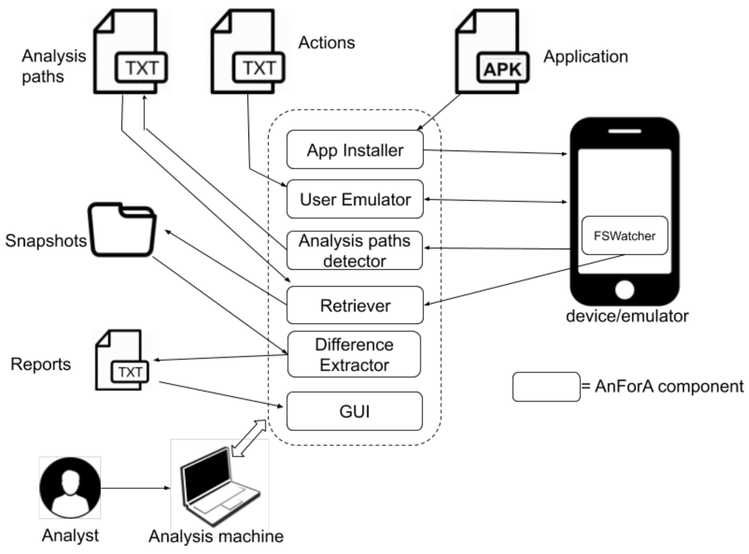


Figure 2: AnForA architecture.

The architecture of AnForA is shown in Fig. 2, while in Fig. 3 we show how its components automate the various steps of the analysis methodology, by annotating each one of them with the corresponding component automating it.

As shown in Fig. 2, AnForA consists of seven components, whose role is detailed below:

1. The App Installer, which installs the application to be analyzed on the virtual mobile device, thus automating the *Application installation* step. In particular, the App Installer takes as input the installation file of the application (the *APK file*), extracts from it the *Manifest* file that lists the permissions the application needs to function properly, and executes suitable commands to install the application, as well as to grant the permissions it needs.
2. The User Emulator, which interacts with the application to perform sequence of actions that make up an experiment by using the application *Graphical User Interface* (or *GUI*, for brevity) exactly as a human user would do. These actions are specified by the analyst into an *Actions*

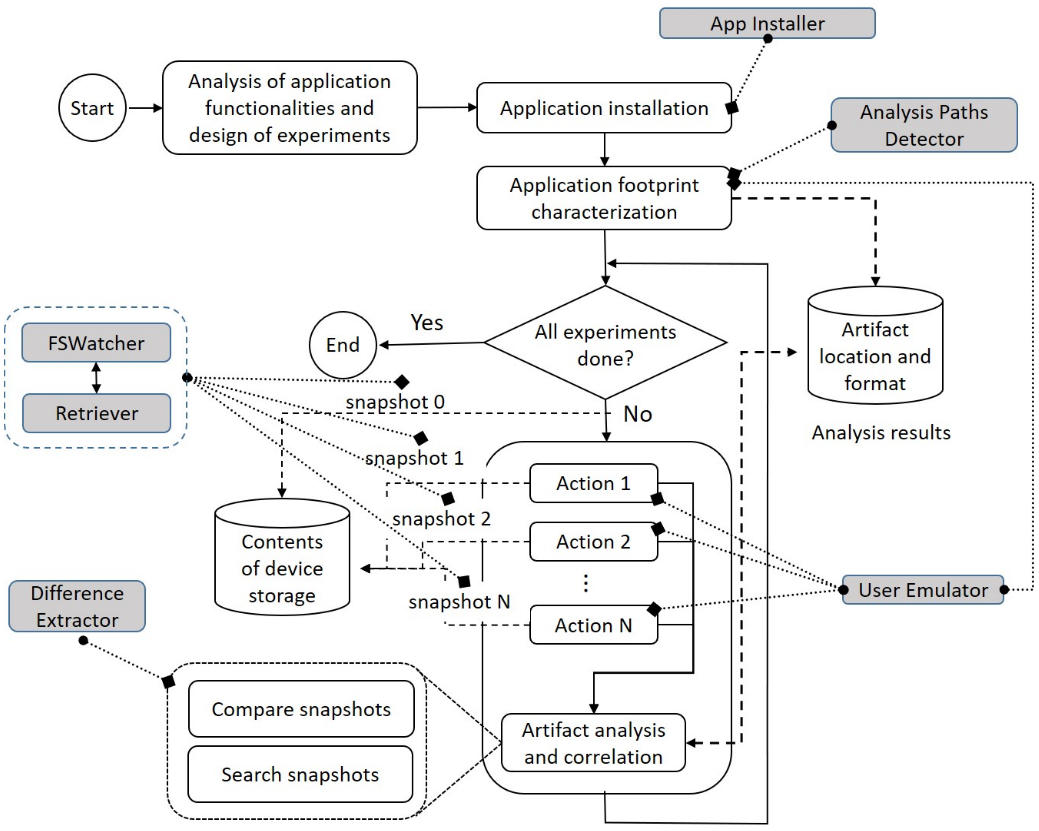


Figure 3: The workflow of AnForA. Each AnForA component is shown as a gray box attached via a dotted connector to the methodology step it automates.

File, using a suitable command language (see Sec. 3.3) As shown in Fig. 3, the User Emulator automates the execution of the experiments.

3. The Analysis Paths Detector, which automates the *Application Footprint Characterization.* In particular, it determines the folders, in the internal and external storage of the device, where the application can write data (the *Analysis Paths*). More specifically, the Analysis Paths Detector identifies, and includes in the *Analysis Paths*, the following folders:
   - the *private data folder* of the application, i.e. the private directory where the application may write data;
   - the *additional folders* of the applications, i.e. public directories where the application may write if it is granted the corresponding write permissions;
   - the *EC private and additional folders*, i.e. those folders where third-party applications or system services (henceforth referred to as *External Components* or *ECs* for brevity), write data on the behalf of the app when it requests service to them.
4. The FSWatcher, which runs on the mobile device, and monitors all the folders in the *Analysis Paths* to detect which files are created/deleted/modified during the experiments. The FSWatcher works in close interaction with the Retriever, from which it receives the start/stop monitoring commands (see below), and to whom it reports the results of the monitoring sessions.
5. The Retriever, which retrieves the set of files reported by the FSWatcher, and creates a snapshot that is stored on the analysis machine. In particular, as shown in Fig. 3, at any time during the experiment, the Retriever may be invoked by the User Emulator (if specified in the *Actions File*), and in this case it (a) stops the FSWatcher, (b) gets a snapshot of the contents of each directory in the *Analysis Paths* , and (c) restart the FSWatcher.
6. The Difference Extractor, which compares two versions of the same file contained in different snapshots, in order to find which data have been added to/removed from/modified in that file, and associates these data with the corresponding action. In particular, as shown in Fig. 3, the Difference Extractor comes into play at the end of each experiment to examine all the pairs of consecutive snapshots to identify all the files that have been created/modified/deleted as

effect of the action corresponding to the second snapshot of each pair. For each one of these files, the Difference Extractor locates the portions that have been modified, decodes them (if it knows their encoding scheme), and writes this information into a report. At the end of the experimental campaign, the report contains – for each individual experiment – the list of the files that have been added/created/modified/deleted by each one of its actions. In this way, the analyst can establish the correspondence between each user action, and the artifacts it generates. Furthermore, the report contains also – for each one of the above files – which data have been modified, and where this data are located within it, thus enabling the analyst to quickly focus the analysis of them.

7. The GUI, which provides the analyst with the access to the various functionalities of AnForA, and allows him/her to control the various steps of the analysis workflow by controlling the operations of its components.

It is worth noting that AnForA supports the execution of experiments where multiple users interact among them using the application under analysis, or even different applications. To do so, it is sufficient to start as many instances of AnForA are necessary, either on the same or on different analysis machines, and provide to each instance its specific *Actions File*.

3.3. The Implementation

We have developed a proof-of-concept implementation of AnForA, which relies on a mix of freely available tools and of software components that we developed specifically for it using the Python and the C++ languages, as discussed below.

First of all, for the configuration and use of virtualized mobile devices, AnForA relies on the *Android Mobile Device Emulator* [18], a collection of software tools running on the analysis machine that allows the creation and the execution of the so-called *Android Virtual Devices* (*AVD*s), i.e. emulated mobile devices behaving exactly like real physical devices that can be customized with different hardware characteristics and Android versions.

The components of AnForA running on the analysis machine interact with those running within the virtual device, as well as with its Android operating system, by means of the Android Debug Bridge (ADB) service [15], that is part of the standard Android SDK [16].

In the following, we briefly discuss how each one of the components of AnForA has been implemented.

- The App Installer is implemented as a Python script that uses suitable ADB commands to install the app using its APK file, and to grant it the permissions it needs to function properly. These permissions are automatically extracted from the *AndroidManifest.xml* file, which is contained in the APK file of the application. The App Installer also installs the FSWatcher on the AVD and suitably sets port forwarding on it to enable the FSWatcher – Retriever network communications.

- The User Emulator is implemented in Python by means of the *UI Automator* testing framework [19], that provides a set of APIs to build UI tests that perform interactions on user apps and system apps. This APIs allows to programmatically interact with the various components of the application GUI by emulating via software typical user gestures such as tap, swipe, long tap and so on. In particular, the User Emulator reads from the *Actions File* the sequence of actions it has to perform on the application user interface,and for each one of them calls the appropriate function of the UI Automator API. The User Emulator supports all the actions provided by *UI Automator*, using a syntax like that shown in Table 1 for an example subset of supported actions. To find out the identifier of a widget, or its position on the screen, the

Table 1: A subset of the actions supported by the User Emulator.

Action name	**Description**
`AllApps`	Go to the device *All Apps* screen
`Back`	Tap on the *Back* button
`Dump`	Invoke the Receiver which (1) starts the FSWatcher, (2) extracts from the device the list of files it reports, and (3) then restarts it
`Home`	Go to the device *Home* screen
`SetTxt(widgetId,t)`	Insert text `t` into the widget identified by `widgetId`
`SetTxtXY(x,y,t)`	Insert text `t` into the widget placed on position ⟨`x,y`⟩ of the screen
`TapOn(widgetId)`	Tap on the widget identified by `widgetId`
`TapXY(x,y)`	Tap on point ⟨`x,y`⟩ of the screen

analyst uses the *UI Automator Viewer*, a tool which is part of the UI Automator framework (see Sec. 4 for the discussion on how to use it).

- the Analysis Paths Detector is implemented as a Python script that performs the following two actions:

  1. it parses the APK file of the application to extract the information concerning the corresponding private and additional folders (from the *AndroidManifest.xml* file);
  2. it discovers the EC private and additional folders by performing a "dry run" (i.e., a run in which no data are collected from the device memory) in which the User Emulator carries out all the experiments defined by the analyst. As a matter of fact, ECs can be determined only at run-time, since they depend on specific choices made by the user when the application runs. By performing a dry run, the application is forced to issue all the service requests to the ECs it uses. The information about these requests (called *intents* in the Android jargon) are recorded by Android into a log file that, after all the experiments have been performed, is extracted (through the Android's *dumpsys* tool [17]) and analyzed, so that the Analysis Paths Detector can identify the above ECs. At the end of the dry run, the virtualized device is brought back to a clean state, so that all the modifications it induces are wiped away.

- The FSWatcher is implemented as a `C++` program, and relies on the Linux's *inotify* mechanism [39] for watching file system events under specific root paths. For each given root path to watch, the FSWatcher recursively monitors its entire subtree and can follow all symbolic links found therein. On stopping, the FSWatcher produces a report containing all the changes that took place in the monitored paths since when started. The FSWatcher is installed by he App Installer and it communicates with the Retriever through a socket interface by means of which they exchange JSON messages.

- The Retriever is implemented as a Python script that uses ADB commands to retrieve files from the Android device to the analysis machine, and it is invoked any time a `Dump` action is specified in the *Actions File*.

- The Difference Extractor is implemented as a Python script that uses the *libmagic* library [8] to determine the type of the file it needs to process, and calls the appropriate diffing utility.

In the current implementation, it is able to compute the differences between text and binary files (using the *diff* utility [41]), and SQLite databases (using the *SQLDiff* utility [43]).

- The GUI is implemented in Python and uses the *PyQt* Python bindings for the *Qt* cross-platform framework [42].

3.4. Achievement of Design Goals

As discussed in Sec. 1, the design goals of any solution for the automation of the forensic analysis of mobile applications are the achievement of fidelity, artifact coverage, precision, effectiveness, repeatability and generality. In this section we discuss how these goals have been achieved by AnForA, whose components jointly provide these properties.

- The fidelity property is ensured by the User Emulator which, being based on the Android's UI Automator framework, is able to faithfully reproduce the actions that a user would do on the analyzed application (i.e., by issuing commands like swipe or tap on the screen, as well as by typing on the keyboard). By supporting all the actions provided by *UI Automator*, that in turn supports all the possible actions that a user may perform using the GUI of an application, AnForA fully achieves fidelity.

- The artifact coverage property is ensured by the Analysis Paths Detector, which is able to identify all the device folders belonging to the *Analysis Paths* of the application, and the FSWatcher, which identifies all the files, stored in these paths, that are modified by the application during the experiments. Specifically, the Analysis Paths Detector, through the information extracted from the APK file of the application to be analyzed and the use of the Android's *dumpsys* tool, is able to identify the paths of the device file systems where the applications will store its artifacts generated by a given set of actions to perform on it; whereas, the FSWatcher, through the Linux's *inotify* mechanism, is able to intercept all the changes (that happen at runtime on those file system paths previously identified by the Analysis Paths Detector) which are either direct or indirect consequence of an action executed on the analyzed application. Despite the known limitations of the *inotify* mechanism [39] can adversely impact artifact coverage, we have implemented the FSWatcher in such a way to overcome those that apply to the forensic scenarios considered in this work, and in particular:

  - the limit on the maximum number of notification events that may be queued before the kernel buffer storing them overflows: if the events generation rate (i.e., the changes performed by the analyzed application on the file system) is much faster than the events consumption (i.e., the interception of the file system changes in the FSWatcher), the kernel's event queue may fill up and overflow, thus causing the loss of some events. However, by running the FSWatcher as a real-time process under the *SCHED_RR* scheduling discipline with a careful chosen time slice [40], we ensure that the event queue is drained at a rate sufficient to avoid it overflows, while the impact on other applications running in the emulated device is not significantly affected;
  - the maximum number of paths that can be monitored: this problem may arise if either the number of *Analysis Paths* of the analyzed application is large, or if the directory trees rooted at those *Analysis Paths* are very deep. To avoid this problem, AnForA increases this limit to the maximum between the current kernel's default value and the number of directories in the directory trees rooted at the *Analysis Paths*.

- The precision property is ensured by the ability of the FSWatcher, which identifies only the files modified by the application, and is able to record changes in the device file systems that only happen at runtime, while executing a specific action, thus avoiding to report changes that happened in the same location but at a different time (e.g., because they are a consequence of previously executed actions). This property is fully achieved by AnForA because the FSWatcher, through the Linux's *inotify* mechanism, only watches for file system events occurring in the directory trees rooted at the *Analysis Paths* of the analyzed application.

- The effectiveness property is provided by the Difference Extractor, which is able to (a) precisely locate, in the file storing them, the data generated by each action, and (b) correlate the above action with the corresponding data. More precisely, effectiveness is partially ensured by AnForA; specifically, AnForA can always ensure a relaxed version of this property, representing the ability to correlate a group of user actions of interest performed between two successive dumps with the data they modified/generated. To fully achieve the effectiveness property, the analyst either can put a dump instruction just after each user action in the *Actions File*, or it can use the timestamp associated with each recorded change to identify to which action that change is related to.

- The repeatability property is ensured by the use of AVDs, which allow to reproduce the same execution environment as many times as needed, and of *Actions Files*, which allow to repeat the same set of user actions (and in the same order) on the analyzed applications, thus yielding the same results. This property is fully achieved by AnForA because two or more AVDs representing the same device and running the same version of Android are functional equivalent devices as they are created from the same system image file.
- The generality property is ensured by the use of AVDs, which allow to analyze an application over a myriad of devices equipped with different hardware characteristics and running different Android versions. This property is fully achieved by AnForA because it is possible to create AVDs representing very different types of devices, including Android phones, tablets, Wear OS, Android TV, and Automotive OS devices, and running every version of Android supported by the Android Emulator platform.

4. Gmail application use case

To illustrate how to use AnForA, in this section we discuss how to set up and run an experiment in which the *Gmail* email app is used to compose and send a message to a specific destination address.

First of all, we need to specify the sequence of actions, that need to be carried out to compose and send the message, using the User Emulator language. More specifically, this sequence consists in the following actions:

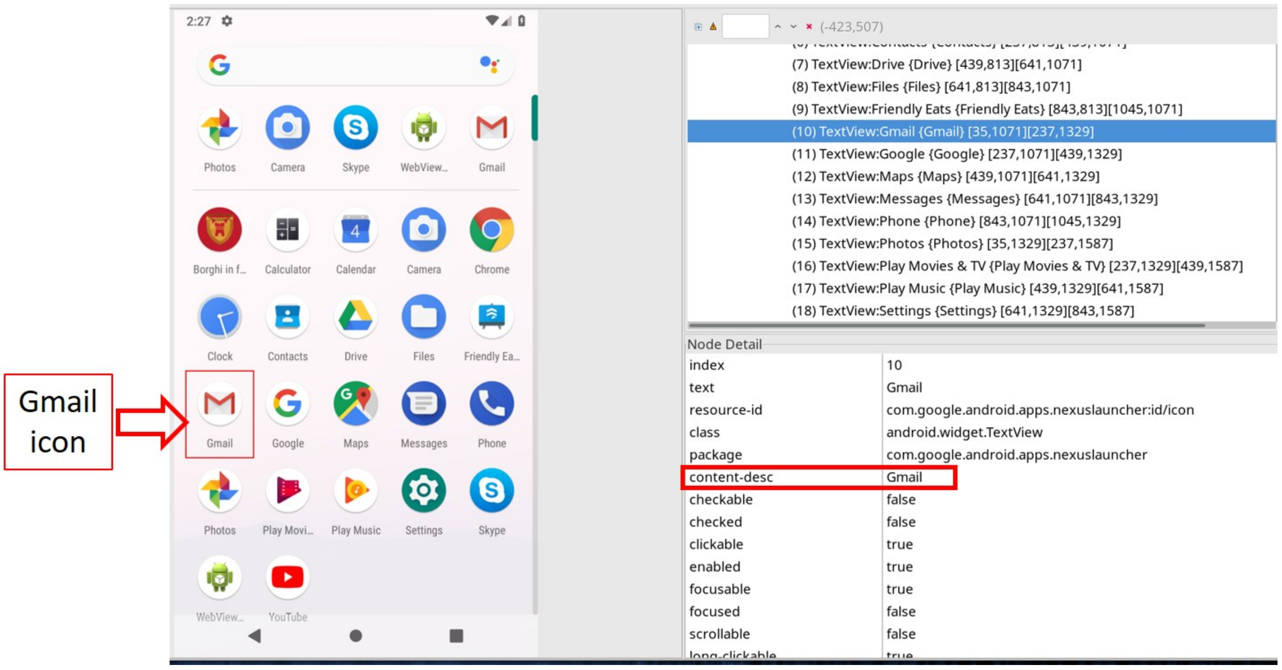
1. bring the device to the “All Apps” screen by swiping up on the “Home” button of its interface, so that the icons of all the installed apps are shown;

2. start the Gmail app by tapping on the corresponding icon;
3. open the “compose” window by tapping on the `Compose` button of the Gmail GUI;
4. fill the `to`, `subject`, and `message body` text boxes shown on the Gmail GUI;
5. send the message by tapping on the `send` button of the Gmail GUI.

As discussed in Sec. 3.3, for each one of these activities, it is necessary to determine either the identifier of the corresponding widget, or its position on the screen. The identifier of a widget is stored into either one of two *properties* of the widget (i.e., attributes that can store values), namely either the content-desc or the resource-id property, while its position on the screen is stored in the bounds property.

As discussed in Sec. 3.3, both information can be retrieved by loading in the *UI Automator Viewer* the window of the GUI where the widget of interest is placed, and by using it to inspect its properties.

Fig. 4 shows how to identify the icon corresponding to the Gmail app using the *UI Automator Viewer*, which is shown in the left side pane. When the analyst selects this icon, its properties – and in particular `content-desc` – are shown in the bottom-right pane. From this property it can



```
(7) TextView:Drive {Drive} [439,813][641,1071]
(8) TextView:Files {Files} [641,813][843,1071]
(9) TextView:Friendly Eats {Friendly Eats} [843,813][1045,1071]
(10) TextView:Gmail {Gmail} [35,1071][237,1329]
(11) TextView:Google {Google} [237,1071][439,1329]
(12) TextView:Maps {Maps} [439,1071][641,1329]
(13) TextView:Messages {Messages} [641,1071][843,1329]
(14) TextView:Phone {Phone} [843,1071][1045,1329]
(15) TextView:Photos {Photos} [35,1329][237,1587]
(16) TextView:Play Movies & TV {Play Movies & TV} [237,1329][439,1587]
(17) TextView:Play Music {Play Music} [439,1329][641,1587]
(18) TextView:Settings {Settings} [641,1329][843,1587]
```

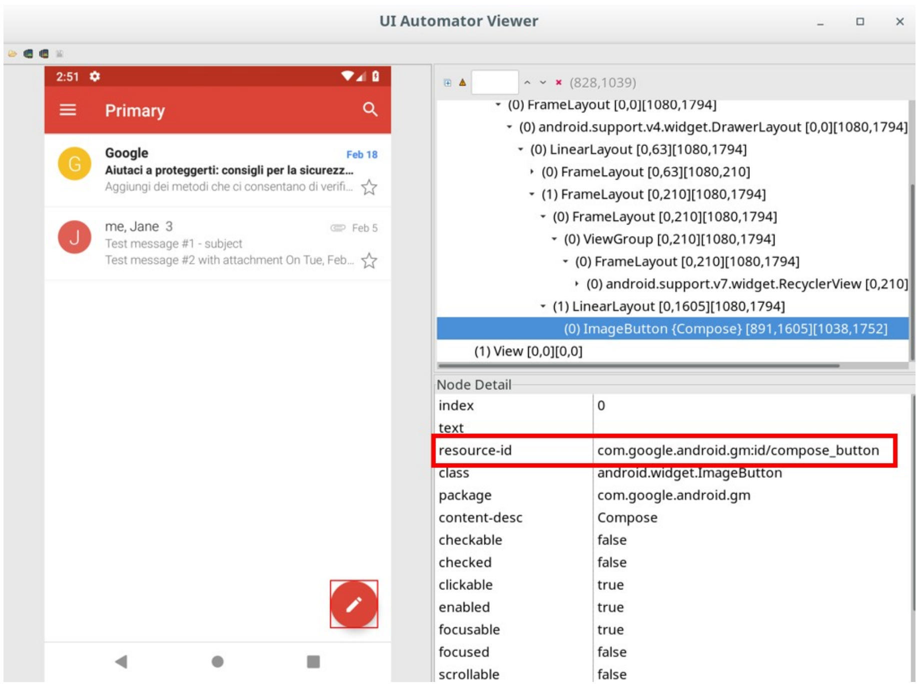
Node Detail

index	10
text	Gmail
resource-id	com.google.android.apps.nexuslauncher:id/icon
class	android.widget.TextView
package	com.google.android.apps.nexuslauncher
content-desc	Gmail
checkable	false
checked	false
clickable	true
enabled	true
focusable	true
focused	false
scrollable	false

Figure 4: Identification of the Gmail icon.

be seen that the identifier of the Gmail icon is "`Gmail`", so the User Emulator action that launches the Gmail app is set to `TapOn(Gmail)`.

In the same way, the "compose" window is opened by tapping on the `Compose` button of the application GUI, whose identifier is stored – as shown in Fig. 5 – in the resource-id property (note that the content-desc property could have been used equivalently). Hence, the User Emulator action that opens the "compose" window is `TapOn(com.google.android.gm:id/compose_button)`.



UI Automator Viewer
2:51
Primary
Google
Feb 18
Aiutaci a proteggerti: consigli per la sicurezz...
Aggiungi dei metodi che ci consentano di verifi...
me, Jane 3
Feb 5
Test message #1 - subject
Test message #2 with attachment On Tue, Feb...
(828,1039)

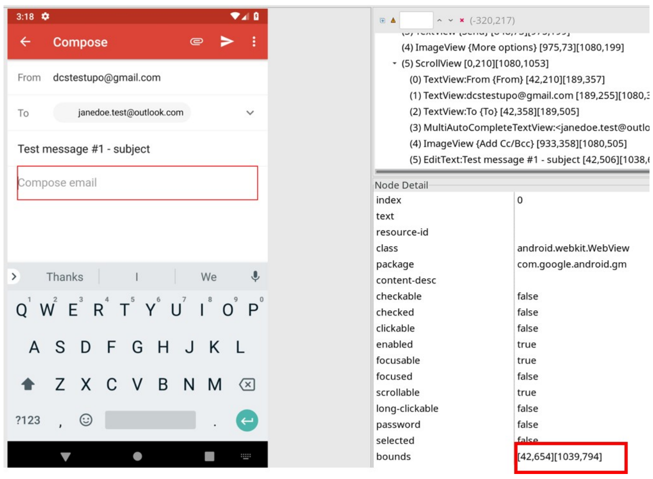
```
(0) FrameLayout [0,0][1080,1794]
  (0) android.support.v4.widget.DrawerLayout [0,0][1080,1794]
    (0) LinearLayout [0,63][1080,1794]
      (0) FrameLayout [0,63][1080,210]
      (1) FrameLayout [0,210][1080,1794]
        (0) FrameLayout [0,210][1080,1794]
          (0) ViewGroup [0,210][1080,1794]
            (0) FrameLayout [0,210][1080,1794]
              (0) android.support.v7.widget.RecyclerView [0,210]
        (1) LinearLayout [0,1605][1080,1794]
            (0) ImageButton {Compose} [891,1605][1038,1752]
  (1) View [0,0][0,0]
```

Node Detail

index	0
text	
resource-id	com.google.android.gm:id/compose_button
class	android.widget.ImageButton
package	com.google.android.gm
content-desc	Compose
checkable	false
checked	false
clickable	true
enabled	true
focusable	true
focused	false
scrollable	false

Figure 5: Identification of the Gmail Compose button.

To compose and send a message, the User Emulator needs to place suitable textual information into the destination address, the subject, and the email body text boxes, and to click on the send button; hence, the corresponding widgets need to be identified. For the first two text boxes as well as for the send button, we proceed exactly as described above, i.e. we use either the content-desc or the resource-id properties, so we do not discuss it here again. However, as shown in Fig. 6, the position on the screen of the email body text box needs to be used, since the above two properties are empty. The screen position of this widget, which is stored in the bounds property, corresponds to the rectangle whose opposite edges correspond to points $(42, 654)$ and $(1039, 794)$. To identify the widget, any point falling inside this rectangle – e.g., $(50, 700)$ – may be used. Hence, the


3:18
Compose
From dcstestupo@gmail.com
To janedoe.test@outlook.com
Test message #1 - subject
Compose email
Thanks
I
We
?123
(-320,217)

```
(4) ImageView {More options} [975,73][1080,199]
(5) ScrollView [0,210][1080,1053]
    (0) TextView:From {From} [42,210][189,357]
    (1) TextView:dcstestupo@gmail.com [189,255][1080,3
    (2) TextView:To {To} [42,358][189,505]
    (3) MultiAutoCompleteTextView:<janedoe.test@outlo
    (4) ImageView {Add Cc/Bcc} [933,358][1080,505]
    (5) EditText:Test message #1 - subject [42,506][1038,
```

Node Detail

index	0
text	
resource-id	
class	android.webkit.WebView
package	com.google.android.gm
content-desc	
checkable	false
checked	false
clickable	false
enabled	true
focusable	true
focused	false
scrollable	true
long-clickable	false
password	false
selected	false
bounds	[42,654][1039,794]

Figure 6: Identification of the email body text box.

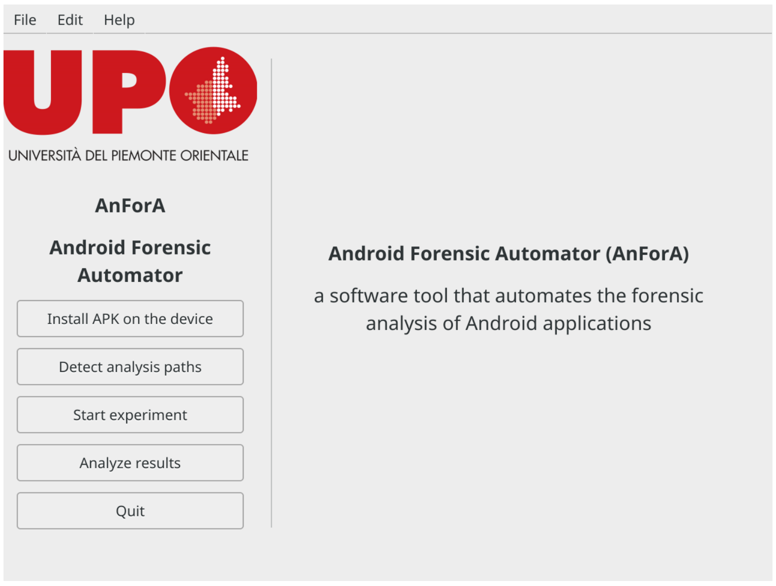
User Emulator action that fills the email body text box is `SetTxtXY(50,700,"Test Message #1 - body")`.

The resulting *Actions File*, that will be provided as input to the User Emulator, contains the following actions:

1. `AllApps`
2. `TapOn(Gmail)`
3. `TapOn(com.google.android.gm:id/compose_button)`
4. `SetTxt(com.google.android.gm:id/to,"janedoetest@outlook.com")`
5. `SetTxt(com.google.android.gm:id/subject,"Test message #1 - subject")`
6. `SetTxtXY(50,700,"Test Message #1 - body")`
7. `TapOn(com.google.android.gm:id/send)`
8. `Dump`

Now, the experiment may be carried out by means of the AnForA's GUI, which is shown in Fig. 7, and provides various buttons, each one corresponding to a specific step of the analysis methodology.

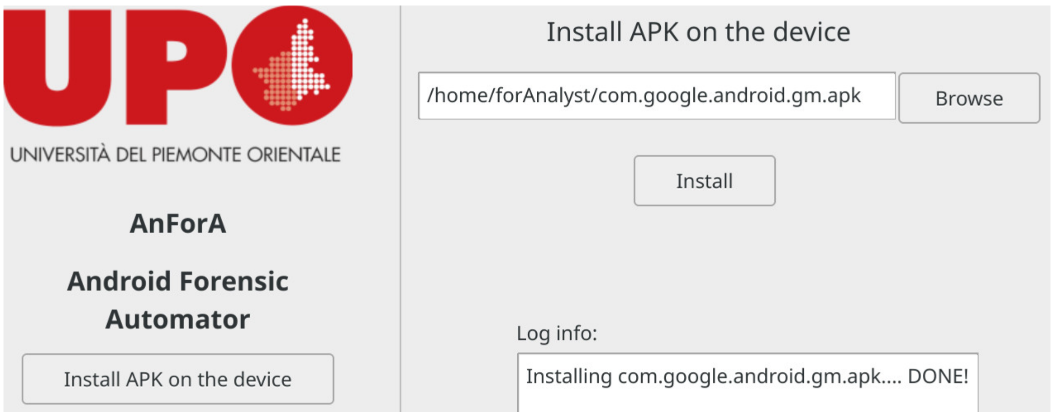
In particular, the app is installed via the *Install APK on the device* button, which prompts the analyst to specify the path name of the APK file, and to click on the *Install* button (see Fig. 8). At the end of the installation, AnForA notifies the user with the outcome of the operation.


File
Edit
Help
UNIVERSITÀ DEL PIEMONTE ORIENTALE
AnForA
Android Forensic Automator
Install APK on the device
Detect analysis paths
Start experiment
Analyze results
Quit

Android Forensic Automator (AnForA)

a software tool that automates the forensic analysis of Android applications

Figure 7: The AnForA main screen.

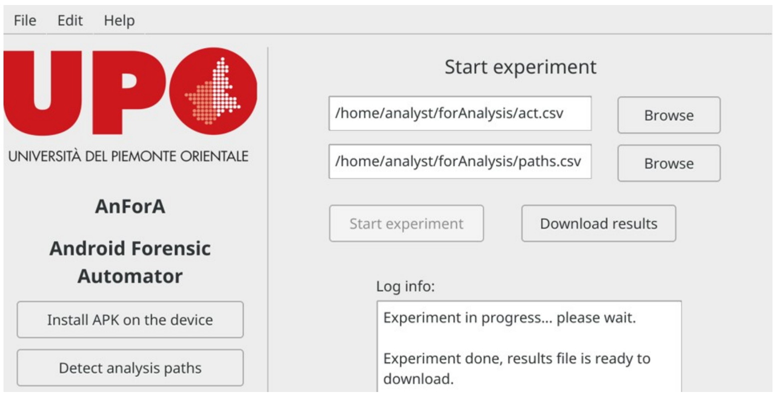

UNIVERSITÀ DEL PIEMONTE ORIENTALE
AnForA
Android Forensic Automator
Install APK on the device
Install APK on the device
Browse
Install
Log info:

/home/forAnalyst/com.google.android.gm.apk

```
Installing com.google.android.gm.apk.... DONE!
```

Figure 8: The APK installation menu. The figure has been cut to show only the information of interest.

The other actions, namely the detection of the *Analysis Paths*, and the execution of the experiment, are carried out by means of the *Detect analysis paths* and *Start experiment.* When the experiment is done, AnForA notifies the analyst (see Fig. 9), and enables him/her to download the results and to analyze them using the *Download results* button.



/home/analyst/forAnalysis/act.csv

/home/analyst/forAnalysis/paths.csv

Experiment in progress... please wait.

Experiment done, results file is ready to download.

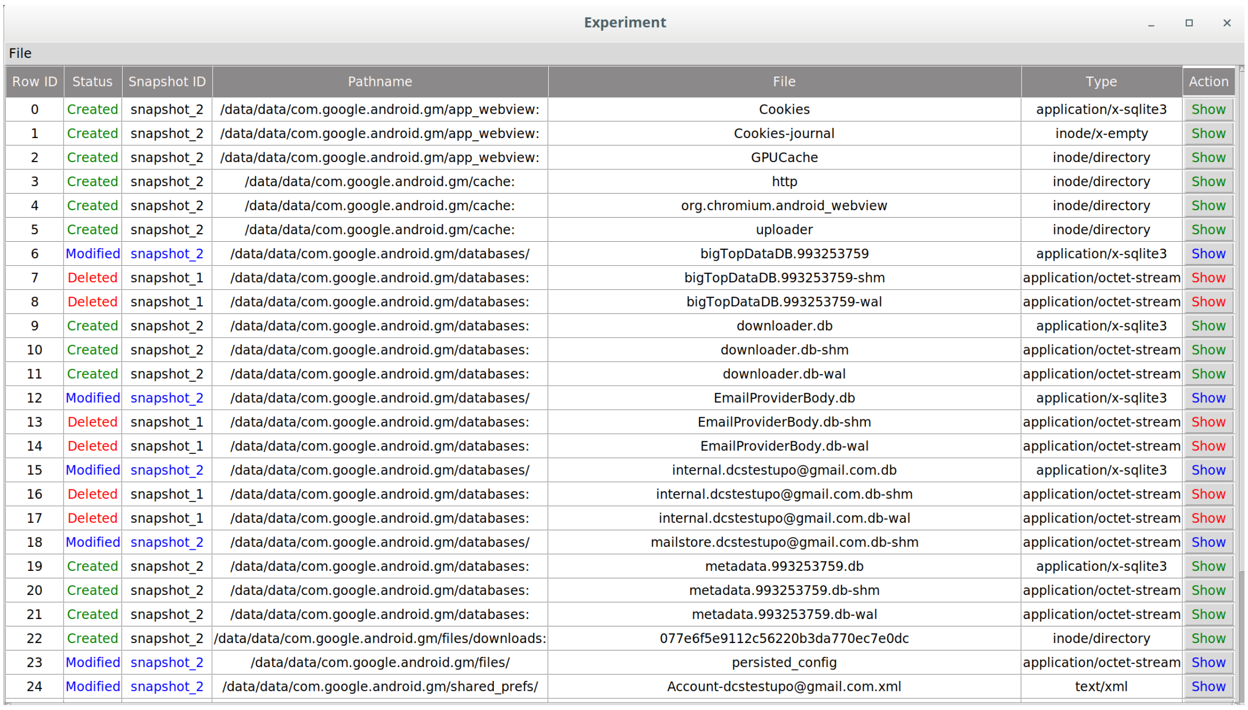
Figure 9: The experiment has been completed. The figure has been cut to show only the information of interest.

At this point, once the analyst clicks on the *Analyze results* button, (s)he is presented with the set of snapshots generated as consequence of the various Dump actions in the *Actions File*. By selecting two different snapshots, the analyst is presented with the list of files that have been created, modified or deleted in the second snapshot with respect to the first one (Fig. 10).

As shown in the figure, for each file AnForA reports its status (one of Created, Modified, or Deleted), the identifier of the snapshot containing the file, its pathname on the device file system where it is stored, its name, and its type, and allows the analyst to visualize the file (in case it has been created or deleted), or the changes occurred in this file as a consequence of the actions corresponding to the last snapshot.

For instance, from Fig. 10 we see that file in row 5 has been created in `snapshot_2` (i.e., it was not present in `snapshot_1`), while file in row 7 has been deleted (hence, it is contained in `snapshot_1` only).

By clicking on the `Action` column of each file, AnForA shows the changes occurred in that file from the first to the second snapshot. Fig. 11 shows the changes occurred in the text file `Account-dcstestupo@gmail.com.xml` as consequence of sending an email message using the Gmail app. As can be seen from the figure, the contents of that file in the two snapshots are shown

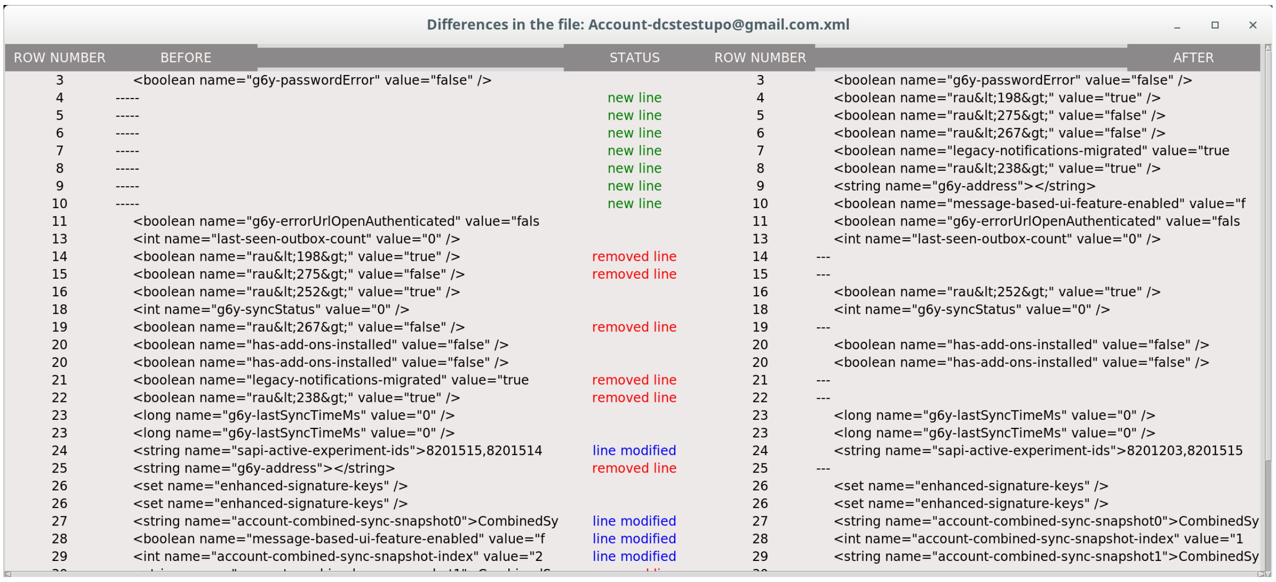


Experiment

File

Row ID	Status	Snapshot ID	Pathname	File	Type	Action
0	Created	snapshot_2	/data/data/com.google.android.gm/app_webview:	Cookies	application/x-sqlite3	Show
1	Created	snapshot_2	/data/data/com.google.android.gm/app_webview:	Cookies-journal	inode/x-empty	Show
2	Created	snapshot_2	/data/data/com.google.android.gm/app_webview:	GPUCache	inode/directory	Show
3	Created	snapshot_2	/data/data/com.google.android.gm/cache:	http	inode/directory	Show
4	Created	snapshot_2	/data/data/com.google.android.gm/cache:	org.chromium.android_webview	inode/directory	Show
5	Created	snapshot_2	/data/data/com.google.android.gm/cache:	uploader	inode/directory	Show
6	Modified	snapshot_2	/data/data/com.google.android.gm/databases/	bigTopDataDB.993253759	application/x-sqlite3	Show
7	Deleted	snapshot_1	/data/data/com.google.android.gm/databases:	bigTopDataDB.993253759-shm	application/octet-stream	Show
8	Deleted	snapshot_1	/data/data/com.google.android.gm/databases:	bigTopDataDB.993253759-wal	application/octet-stream	Show
9	Created	snapshot_2	/data/data/com.google.android.gm/databases:	downloader.db	application/x-sqlite3	Show
10	Created	snapshot_2	/data/data/com.google.android.gm/databases:	downloader.db-shm	application/octet-stream	Show
11	Created	snapshot_2	/data/data/com.google.android.gm/databases:	downloader.db-wal	application/octet-stream	Show
12	Modified	snapshot_2	/data/data/com.google.android.gm/databases/	EmailProviderBody.db	application/x-sqlite3	Show
13	Deleted	snapshot_1	/data/data/com.google.android.gm/databases:	EmailProviderBody.db-shm	application/octet-stream	Show
14	Deleted	snapshot_1	/data/data/com.google.android.gm/databases:	EmailProviderBody.db-wal	application/octet-stream	Show
15	Modified	snapshot_2	/data/data/com.google.android.gm/databases/	internal.dcstestupo@gmail.com.db	application/x-sqlite3	Show
16	Deleted	snapshot_1	/data/data/com.google.android.gm/databases:	internal.dcstestupo@gmail.com.db-shm	application/octet-stream	Show
17	Deleted	snapshot_1	/data/data/com.google.android.gm/databases:	internal.dcstestupo@gmail.com.db-wal	application/octet-stream	Show
18	Modified	snapshot_2	/data/data/com.google.android.gm/databases/	mailstore.dcstestupo@gmail.com.db-shm	application/octet-stream	Show
19	Created	snapshot_2	/data/data/com.google.android.gm/databases:	metadata.993253759.db	application/x-sqlite3	Show
20	Created	snapshot_2	/data/data/com.google.android.gm/databases:	metadata.993253759.db-shm	application/octet-stream	Show
21	Created	snapshot_2	/data/data/com.google.android.gm/databases:	metadata.993253759.db-wal	application/octet-stream	Show
22	Created	snapshot_2	/data/data/com.google.android.gm/files/downloads:	077e6f5e9112c56220b3da770ec7e0dc	inode/directory	Show
23	Modified	snapshot_2	/data/data/com.google.android.gm/files/	persisted_config	application/octet-stream	Show
24	Modified	snapshot_2	/data/data/com.google.android.gm/shared_prefs/	Account-dcstestupo@gmail.com.xml	text/xml	Show

Figure 10: List of files that differ among two snapshots.



Differences in the file: Account-dcstestupo@gmail.com.xml

ROW NUMBER	BEFORE	STATUS	ROW NUMBER	AFTER
3	<boolean name="g6y-passwordError" value="false" />		3	<boolean name="g6y-passwordError" value="false" />
4	-----	new line	4	<boolean name="rau&lt;198&gt;" value="true" />
5	-----	new line	5	<boolean name="rau&lt;275&gt;" value="false" />
6	-----	new line	6	<boolean name="rau&lt;267&gt;" value="false" />
7	-----	new line	7	<boolean name="legacy-notifications-migrated" value="true
8	-----	new line	8	<boolean name="rau&lt;238&gt;" value="true" />
9	-----	new line	9	<string name="g6y-address"></string>
10	-----	new line	10	<boolean name="message-based-ui-feature-enabled" value="f
11	<boolean name="g6y-errorUrlOpenAuthenticated" value="fals		11	<boolean name="g6y-errorUrlOpenAuthenticated" value="fals
13	<int name="last-seen-outbox-count" value="0" />		13	<int name="last-seen-outbox-count" value="0" />
14	<boolean name="rau&lt;198&gt;" value="true" />	removed line	14	---
15	<boolean name="rau&lt;275&gt;" value="false" />	removed line	15	---
16	<boolean name="rau&lt;252&gt;" value="true" />		16	<boolean name="rau&lt;252&gt;" value="true" />
18	<int name="g6y-syncStatus" value="0" />		18	<int name="g6y-syncStatus" value="0" />
19	<boolean name="rau&lt;267&gt;" value="false" />	removed line	19	---
20	<boolean name="has-add-ons-installed" value="false" />		20	<boolean name="has-add-ons-installed" value="false" />
20	<boolean name="has-add-ons-installed" value="false" />		20	<boolean name="has-add-ons-installed" value="false" />
21	<boolean name="legacy-notifications-migrated" value="true	removed line	21	---
22	<boolean name="rau&lt;238&gt;" value="true" />	removed line	22	---
23	<long name="g6y-lastSyncTimeMs" value="0" />		23	<long name="g6y-lastSyncTimeMs" value="0" />
23	<long name="g6y-lastSyncTimeMs" value="0" />		23	<long name="g6y-lastSyncTimeMs" value="0" />
24	<string name="sapi-active-experiment-ids">8201515,8201514	line modified	24	<string name="sapi-active-experiment-ids">8201203,8201515
25	<string name="g6y-address"></string>	removed line	25	---
26	<set name="enhanced-signature-keys" />		26	<set name="enhanced-signature-keys" />
26	<set name="enhanced-signature-keys" />		26	<set name="enhanced-signature-keys" />
27	<string name="account-combined-sync-snapshot0">CombinedSy	line modified	27	<string name="account-combined-sync-snapshot0">CombinedSy
28	<boolean name="message-based-ui-feature-enabled" value="f	line modified	28	<int name="account-combined-sync-snapshot-index" value="1
29	<int name="account-combined-sync-snapshot-index" value="2	line modified	29	<string name="account-combined-sync-snapshot1">CombinedSy

Figure 11: AnForA shows what has been changed in file Account-dcstestupo@gmail.com.xml.

side-by-side, and the lines that have been added (labeled as `new line`), removed (labeled as `removed line`), or modified (labeled as `line modified`) are suitably identified.

In addition to text files, in its current implementation AnForA is able to identify and show the differences also in SQLite databases, while for file whose encoding is not supported it only pinpoints the blocks of data that differ between two snapshots.

5. Validation

In order to validate the results generated by AnForA, and in particular its ability to provide artifact coverage and precision, we use it to perform the forensic analysis of Android apps already analyzed in the literature, so that the results it yields can be compared against those that have been already published.

More specifically, we consider the following apps:

- *Google Gmail*: we analyze version 8.5.6.199637500, and compare AnForA's results against those reported in [6, 38] (both referring to an unspecified version);
- *Microsoft Skype*: we analyze version 8.37.0.98, and compare AnForA's results against those reported in [6] (for version 6.31.0.709), [36] (for an unspecified version), and [38] (for an unspecified version);
- *Facebook Messenger*: we analyze version 113.0.0.21.70, and compare AnForA's results against those reported in [6] (for version 68.0.0.22.67), [27] (for version 86.0.0.17.70), and [48] (for version 113.0.0.21.70).

All the experiments have been performed on two virtualized mobile devices, namely the Google Pixel 2 smartphone and the Google Pixel C tablet, both running Android 9 for the Intel Atom x86_64 platform.

The results of our validation can be summarized as follows. AnForA has been able to identify and locate all the data reported in the literature for the applications we considered. In most cases, however, these data were stored in different files and/or different paths than those previously reported; we believe this is a direct consequence of the fact that in our experiments we considered later versions of these applications and of Android.

Furthermore, for the Gmail application, AnForA found artifacts that had not been previously reported in the literature. This could also be due to differences in the versions of Gmail considered in these studies with respect to the one we use in our experiments.

The results of our experiments demonstrate the ability of AnForA to achieve artifact coverage to a greater extent than previous studies, as it has been able to find – for each application – all and more data than what reported in previous studies. Furthermore, they also demonstrate the ability to achieve a high degree of precision, since for each application, only the data it generates (either directly or indirectly) have been included by AnForA into its reports.

Finally, our experiments also demonstrate the advantage of using AnForA instead of manual analysis, both in terms of time and efficacy, as it enables to quickly and accurately repeat the analysis of new versions of already-analyzed applications when needed.

In the following, we discuss the comparison of AnForA's results against those obtained in published works for *Gmail* (Sec. 5.1), *Skype* (Sec. 5.2), and *Facebook Messenger* (Sec. 5.3).

5.1. Google Gmail

The results of the analysis of Gmail are reported in Table 2, in which we compare the relevant files identified by AnForA (column `AnForA`) against those reported by [6, 38] (column `Literature`).

Table 2: Location of artifacts for Google Gmail. Unless otherwise specified, all paths are relative to `/data/data/com.google.android.gm`.

Row	Artifact	Location	
		AnForA	Literature
1	*Preferences*	`shared_prefs/*.xml`	`shared_prefs/*.xml`
2	*Account details (Google account)*	`shared_prefs/MailAppProvider.xml`,	`cache/<username>@gmail.com.db`
3	*Messages (Google account)*	`databases/bigTopDataDB.<account-id>`	`databases/mailstore.<username>@gmail.com.db`
4	*Attachment metadata (Google account)*	`databases/metadata.<account-id>`	`cache/<username>@gmail.com/`
5	*Attachment files (Google account)*	`files/downloads/`	`cache/<username>@gmail.com/`
6	*Account details (IMAP account)*	`shared_prefs/MailAppProvider.xml`, `databases/EmailProvider.db`	n/a
7	*Messages (IMAP account)*	`databases/EmailProvider.db`	n/a
8	*Attachments metadata (IMAP account)*	`databases/EmailProvider.db`	n/a
9	*Attachment files (IMAP account)*	`cache/*.attachment`	n/a
10	*Third-party services*	Android's account files, Google Mobile Services' files	n/a

As in the studies already appeared in the literature, AnForA found that the installation folder of

Gmail is `/data/data/com.google.android.gm`. However, it also found several important differences with respect to them, namely:

- There are differences in the location and contents of the data generated when using a Google email account with respect to using an account of another provider via the *IMAP* protocol.
- For Google email accounts, AnForA identified the same data reported in the literature, but located them in different files and folders (rows $2-5$ in Table 2).
- For generic email accounts, AnForA found that Gmail generates data not identified by previous studies (rows $6-9$ in Table 2), and in particular:
  - When a new generic account is registered in Gmail, a new record is added to table `Account` of database `databases/EmailProvider.db` to store account information, including the email address, the display name and the sender name. Furthermore, a file named `Account-<imap server>.xml` is created to store cipher settings for the IMAP server `<imap server>` associated with this account.
  - Messages are stored in table `Message` of database `databases/EmailProvider.db`.
  - Attachment metadata are stored in table `Attachment` of database `databases/EmailProvider.db`.
  - Downloaded attachment contents are saved as files, named as `<name>.attachment`, in the `cache` subdirectory, where `<name>.attachment` encodes the download timestamp (e.g., `2019-02-27-21:57:1577702464221827 52973.attachment`). These attachments are also stored as URIs in field `cachedFile` of table `Attachment` (so that a downloaded attachment can be easily related to its metadata).
- Unlike previous studies, AnForA has been able to identify two third-party services (namely, *Android account service* and *Google Mobile Services*) that are used by Gmail, and that may write – in their folders – data generated on the behalf of Gmail. In particular, when an account is added to Gmail, a related entry is also added to the device's accounts by updating the Android's contacts database `contacts2.db` located in the directory

/data/data/com.android.providers.contacts/databases. Furthermore, for Google accounts, the Google Mobile Services collection is also updated accordingly by updating the corresponding databases located in the directory /data/data/com.google.android.gms/databases.

5.2. Microsoft Skype

The results of the analysis of Skype are reported in Table 3, in which we compare the relevant files identified by AnForA (column AnForA) against those reported by [36, 6, 38] (column Literature).

Table 3: Location of artifacts for Skype. Unless otherwise specified, all paths are relative to /data/data/com.skype.raider.

Artifact	Relevant paths	
	As reported by AnForA	Literature
Contacts, conversations, and call logs	databases/s4l-live:<username>.db	files/live#3 <username>/main.db [6], files/<username>/main.db [36, 38]
Shared media metadata	databases/s4l-live:<username>.db	files/<username>/main.db [38]
Shared media files	cache/FileCache/	files/live#3 <username>/media_messaging/media_cache/ [6], cache/ [36]

Unlike the Gmail case, the results obtained by AnForA for Skype agree with those reported in the literature, as far as the data that are generated by this app are concerned, while differences have been found with respect to the location of these data, as shown in Table 3. Hence, we do not discuss the contents of these files here (the interested reader may refer to the literature for this discussion).

5.3. Facebook Messenger

The results of the analysis of Facebook Messenger are reported in Table 4, in which we compare the relevant files identified by AnForA (column AnForA) against those reported by [6, 27, 48] (column Literature). Again, as in the Skype case, the results obtained by AnForA agree with those reported

Table 4: Location of artifacts for Facebook Messenger. Unless otherwise specified, all paths are relative to /data/data/com.facebook.orca.

Artifact	Relevant paths	
	As reported by AnForA	Literature
Contacts	databases/contacts_db2	databases/contacts_db2 [6, 48]
Conversations	databases/threads_db2	databases/threads_db2
Call logs	databases/call_log.sqlite	databases/call_log.sqlite [6]
Shared media files	cache/fb_temp/	cache/fb_temp/ [6], cache/, files/ [27, 48]

in the literature as far as the data that are generated by this app, while differences have been found with respect to the location of these data, as shown in Table 4. Therefore, as for the Skype results, we do not discuss the contents of these files here (the interested reader may refer to the literature for this discussion).

6. Conclusions and future work

In this paper we have presented the design, implementation, and evaluation of AnForA, a software tool that automates most of the activities that need to be carried out to forensically analyze Android applications designed in such a way to yield various properties namely fidelity, artifact coverage, precision, effectiveness, repeatability, and generality. In particular, AnForA relies on the use of virtualized Android devices, on which the application is installed, and on a set of software components that (a) interact with its interface as a human user would do, (b) monitor the changes to the file systems induced by these interactions, (c) extract modified files, and (d) locate these modifications.

We have devised a proof-of-concept implementation of AnForA, and we have used it to assess its ability in achieving its design goals, by analyzing through it several Android applications already studied in the literature, so that we can compare AnForA's results against those reported in these papers. The results of our evaluation confirm that AnForA greatly simplifies the forensic analysis of Android applications, and exhibits all the properties mentioned above, namely fidelity, artifact coverage, precision, effectiveness, repeatability, and generality, to a higher extent than previous studies published in the literature.

As future work, we plan to extend AnForA's capabilities in decoding files along two directions: (a) integrate in it suitable helper applications able to decode files whose encoding is known (e.g., PDF or Microsoft Office files), and (b) use machine learning techniques to enable it to automatically discover the proprietary encoding schemes that many application use for the data they store on the device (e.g., applications that serialize complex data structure containing fields of different types).

Acknowledgments

This research has a financial support of the Università del Piemonte Orientale.

References

[1] Anglano, C., 2014. Forensic Analysis of WhatsApp Messenger on Android Smartphones. Digital Investigation 11, 201–213. doi:10.1016/j.diin.2014.04.003.

[2] Anglano, C., Canonico, M., Guazzone, M., 2016. Forensic Analysis of the ChatSecure Instant Messaging Application on Android Smartphones. Digital Investigation 19, 44–59. doi:10.1016/j.diin.2016.10.001.

[3] Anglano, C., Canonico, M., Guazzone, M., 2017. Forensic Analysis of Telegram Messenger on Android Smartphones. Digital Investigation 23, 31–49. doi:10.1016/j.diin.2017.09.002.

[4] Bodden, E., Havelund, K., 2010. Aspect-oriented race detection in java. IEEE Trans. Softw. Eng. 36, 509–527. doi:10.1109/TSE.2010.25.

[5] Bush, W.R., Pincus, J.D., Sielaff, D.J., 2000. A static analyzer for finding dynamic programming errors. Softw. Pract. Exper. 30, 775–802. doi:10.1002/(SICI)1097-024X(200006)30:7<775::AID-SPE309>3.0.CO;2-H.

[6] Cahyani, N.D.W., Rahman, N.H.A., Glisson, W.B., Choo, K.K.R., 2017. The role of mobile forensics in terrorism investigations involving the use of cloud storage service and communication apps. Mobile Networks and Applications 22, 240–254. doi:10.1007/s11036-016-0791-8.

[7] Cheng, C.C.C., Shi, C., Gong, N.Z., Guan, Y., 2018. EviHunter: Identifying Digital Evidence in the Permanent Storage of Android Devices via Static Analysis, in: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, ACM, New York, NY, USA. pp. 1338–1350. doi:10.1145/3243734.3243808.

[8] Darwin, I.F., . The file(1) command and the libmagic(3) library. Available at http://www.darwinsys.com/file/. Accessed: August 2019.

[9] Das, M., Lerner, S., Seigle, M., 2002. Esp: Path-sensitive program verification in polynomial time, in: Proc. of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, ACM, New York, NY, USA. pp. 57–68. doi:10.1145/512529.512538.

[10] De Pauw, W., Jensen, E., Mitchell, N., Sevitsky, G., Vlissides, J., Yang, J., 2002. Visualizing the execution of java programs, in: Diehl, S. (Ed.), Software Visualization, Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 151–162. doi:10.1007/3-540-45875-1_12.

[11] Engler, D., Ashcraft, K., 2003. Racerx: Effective, static detection of race conditions and deadlocks, in: Proc. of the Nineteenth ACM Symposium on Operating Systems Principles, ACM, New York, NY, USA. pp. 237–252. doi:10.1145/945445.945468.

[12] Ernst, M.D., 2004. Invited Talk Static and Dynamic Analysis: Synergy and Duality, in: Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, ACM, New York, NY, USA. pp. 35–35. doi:10.1145/996821.996823.

[13] Evans, D., Larochelle, D., 2002. Improving security using extensible lightweight static analysis. IEEE Software 19, 42–51. doi:10.1109/52.976940.

[14] Flanagan, C., Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R., 2002. Extended static checking for java, in: Proc. of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, ACM, New York, NY, USA. pp. 234–245. doi:10.1145/512529.512558.

[15] Google, a. Android Debug Bridge. Available at https://developer.android.com/studio/command-line/adb.html. Accessed: August 2019.

[16] Google, b. Android SDK Tools. Available at https://developer.android.com/studio/index.html. Accessed: August 2019.

[17] Google, c. dumpsys. Available at https://developer.android.com/studio/command-line/dumpsys. Accessed: August 2019.

[18] Google, d. Run Apps on the Android Emulator. Available at https://developer.android.com/studio/run/emulator.html.

[19] Google, e. UI Automator. Available at https://developer.android.com/training/testing/ui-automator. Accessed: August 2019.

[20] Gosain, A., Sharma, G., 2014. A Survey of Dynamic Program Analysis Techniques and Tools, in: Satapathy, S., Biswal, B., Udgata, S., Mandal, J. (Eds.), Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA), Springer, Cham. pp. 113–122. doi:10.1007/978-3-319-11933-5_13.

[21] Gosain, A., Sharma, G., 2015. Static Analysis: A Survey of Techniques and Tools, in: Mandal, D., Kar, R., Das, S., Panigrahi, B. (Eds.), Intelligent Computing and Applications, Springer, New Delhi. pp. 581–591. doi:10.1007/978-81-322-2268-2_59.

[22] Gregorio, J., Gardel, A., Alarcos, B., 2017. Forensic analysis of telegram messenger for windows phone. Digit. Investig. 22. doi:10.1016/j.diin.2017.07.004.

[23] Hastings, R., Joyce, B., 1991. Purify: Fast detection of memory leaks and access errors, in: Proc. of the Winter 1992 USENIX Conference, pp. 125–138.

[24] Hovemeyer, D., Pugh, W., 2004. Finding bugs is easy. SIGPLAN Not. 39, 92–106. doi:10.1145/1052883.1052895.

[25] Husain, M.I., Sridhar, R., 2010. iForensics: Forensic Analysis of Instant Messaging on Smart Phones, in: Goel, S. (Ed.), Digital Forensics and Cyber Crime. Springer Berlin Heidelberg. volume 31 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*. doi:10.1007/978-3-642-11534-9_2.

[26] Johnson, S.C., 1978. Lint: a C program checker. Technical Report. Bell Laboratories.

[27] Lin, X., 2018. Android Forensics. Springer. pp. 335–371. doi:10.1007/978-3-030-00581-8_15.

[28] Lin, X., Chen, T., Zhu, T., Yang, K., Wei, F., 2018. Automated forensic analysis of mobile applications on android devices. Digital Investigation 26, S59 – S66.

[29] Mehrotra, T., Mehtre, B.M., 2013. Forensic analysis of Wickr application on android devices, in: 2013 IEEE International Conference on Computational Intelligence and Computing Research, pp. 1–6.

[30] Nethercote, N., Seward, J., 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. 42, 89–100. doi:`10.1145/1273442.1250746`.

[31] Ovens, K.M., Morison, G., 2016. Forensic analysis of Kik messenger on iOS devices. Digital Investigation 17. doi:`10.1016/j.diin.2016.04.001`.

[32] Pauck, F., Bodden, E., Wehrheim, H., 2018. Do Android Taint Analysis Tools Keep Their Promises?, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, New York, NY, USA. pp. 331–341.

[33] Pearce, D.J., Webster, M., Berry, R., Kelly, P.H.J., 2007. Profiling with aspectj. Softw. Pract. Exper. 37, 747–777. doi:`10.1002/spe.v37:7`.

[34] Qiu, L., Wang, Y., Rubin, J., 2018. Analyzing the Analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe, in: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ACM, New York, NY, USA. pp. 176–186. doi:`10.1145/3213846.3213873`.

[35] Reaves, B., Bowers, J., Gorski III, S.A., Anise, O., Bobhate, R., Cho, R., Das, H., Hussain, S., Karachiwala, H., Scaife, N., Wright, B., Butler, K., Enck, W., Traynor, P., 2016. &ast;droid: Assessment and evaluation of android application analysis tools. ACM Comput. Surv. 49, 55:1–55:30. URL: `http://doi.acm.org/10.1145/2996358`, doi:`10.1145/2996358`.

[36] Sgaras, C., Kechadi, M.T., Le-Khac, N.A., 2015. Forensics acquisition and analysis of instant messaging and voip applications, in: Garain, U., Shafait, F. (Eds.), Computational Forensics, Springer. pp. 188–199. doi:`10.1007/978-3-319-20125-2\_16`.

[37] Skaletsky, A., Devor, T., Chachmon, N., Cohn, R., Hazelwood, K., Vladimirov, V., Bach, M., 2010. Dynamic program analysis of microsoft windows applications, in: 2010 IEEE International Symposium on Performance Analysis of Systems Software, pp. 2–12. doi:`10.1109/ISPASS.2010.5452079`.

[38] Tamma, R., Skulkin, O., Mahalik, H., Bommisetty, S., 2018. Practical Mobile Forensics. 3rd ed., Packt Publishing.

[39] The Linux Organization, a. Linux programmer's manual: inotify - monitoring filesystem events. Available at `http://man7.org/linux/man-pages/man7/inotify.7.html`. Accessed: August 2019.

[40] The Linux Organization, b. Linux programmer's manual: sched - overview of cpu scheduling. Available at `http://man7.org/linux/man-pages/man7/sched.7.html`. Accessed: August 2019.

[41] The Open Group, . diff - compare two files. Available at `https://pubs.opengroup.org/onlinepubs/9699919799/utilities/diff.html`. Accessed: August 2019.

[42] The Qt Company, . Qt: Cross-platform software development for embedded & desktop. Available at `https://www.qt.io`. Accessed: August 2019.

[43] The SQLite Consortium, . Sqldiff: Database difference utility. Available at `https://www.sqlite.org/sqldiff.html`. Accessed: August 2019.

[44] Tso, Y.C., Wang, S.J., Huang, C.T., Wang, W.J., 2012. iPhone Social Networking for Evidence Investigations Using iTunes Forensics, in: Proc. of the 6th International Conference on Ubiquitous Information Management and Communication, ACM, New York, NY, USA. pp. 1–7. doi:10.1145/2184751.2184827.

[45] Walnycky, D., Baggili, I., A.Marrington, Moore, J., Breitinger, F., 2015. Network and device forensic analysis of Android social-messaging applications. Digital Investigation 14, Supplement 1, S77–S84. doi:10.1016/j.diin.2015.05.009. proc. of 15th Annual DFRWS Conference.

[46] Wu, S., Zhang, Y., Wang, X., Xiong, X., Du, L., 2017. Forensic analysis of WeChat on Android smartphones. Digital Investigation doi:10.1016/j.diin.2016.11.002.

[47] Xu, Z., Shi, C., Cheng, C.C., Gong, N.Z., Guan, Y., 2018. A Dynamic Taint Analysis Tool for Android App Forensics, in: Proc. of the 2018 IEEE Security and Privacy Workshops (SPW).

[48] Zhang, H., Chen, L., Liu, Q., 2018. Digital forensic analysis of instant messaging applications on android smartphones, in: 2018 International Conference on Computing, Networking and Communications (ICNC), pp. 647–651. doi:10.1109/ICCNC.2018.8390330.