Automated, Dynamic Android App Vulnerability and Privacy Leak Analysis: Design Considerations, Required Components and Available Tools

Kris Heid
Jens Heider
{kris.heid,jens.heider}@sit.fraunhofer.de
Fraunhofer SIT
Darmstadt, Germany

ABSTRACT

Smartphones apps aid humans in plenty of situations. There exists an app for everything. However, without the user's awareness, some apps contain vulnerabilities or leak private data. Static and dynamic app analysis are ways to find these software properties. Especially setting up a dynamic analysis environment is not a trivial task. Several peculiarities of Android have to be considered, existing tools for different aspects have to be evaluated, selected and setup to work together. Existing literature is often outdated and only covers tools for one aspect but doesn't combine them together in a big picture. This paper presents a generic design for an automated dynamic app analysis environment and highlights the required components as well as functionality to reveal security and privacy issues. Available tools are listed, realizing different aspects of the proposed environment design. Tool features are evaluated and tool usability for an automated large scale dynamic app analysis is compared. This document should serve as a reference to all who need to implement dynamic analysis on Android (or some aspects) and require an overview of available and usable solutions.

ACM Reference Format:
Kris Heid and Jens Heider. 2021. Automated, Dynamic Android App Vulnerability and Privacy Leak Analysis: Design Considerations, Required Components and Available Tools. In *European Interdisciplinary Cybersecurity Conference (EICC), November 10–11, 2021, Virtual Event, Romania.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3487405.3487652

1 INTRODUCTION

Google-Play, the largest Apps-Store for Android, currently accommodates ~3 million apps. Apps sometimes leak personal data or unintentionally contain vulnerabilities. Vulnerabilities and privacy violations are mostly invisible to the users and can only be found by well-educated users and the use of additional tools. However, users should be informed about such undesired properties and possibly avoid those apps. Users should instead be able to choose apps without or with fewer privacy and security risks from the plenitude provided by the Apps-Store.

EICC, November 10–11, 2021, Virtual Event, Romania

ACM ISBN 978-1-4503-9049-1/21/11…$15.00
https://doi.org/10.1145/3487405.3487652

Static and dynamic analysis methods are used to detect these software properties and defects. Static methods analyze the source-code or binary-code of an app, while dynamic analysis executes the app and automatically navigates through the app monitoring app behavior in the background to exhibit privacy frauds or vulnerabilities. Both methods have their strengths and weaknesses depending on the scenario. This paper focuses on dynamic analysis part. The main aspects of dynamic analysis are UI automation and behavior monitoring. However, Android app development practices have become more and more complex with background services, system broadcasts (waking up apps on specific events) and callback methods, which must also be considered in a dynamic analysis. Thus, the term *app stimulation* instead of UI automation is used to signal the inclusion of these features.

Currently, there exist (mostly outdated) overview papers and tools for app stimulation as well as for app behavior monitoring. There also exists no paper bringing together app stimulation and behavior monitoring in one paper to form a wholesome dynamic analysis design. Thus, our main motivation to write this paper is to provide an up to date list, also published on Github to remain updated and not become outdated like previous publications. The lists contain required tools for dynamic analysis along with their main features for quick comparison. Additionally, an abstract design is provided showing how the different types of tools can be combined.

2 RELATED WORK

There already exist surveys and overview papers of app stimulation (UI automation) as well as behavior monitoring tools for Android app analysis. However, we realized during our research that these are often outdated and proposed tools are not maintained anymore and are thus not well usable. Other publications describe tools, but the implementation is unavailable, thus not usable.

A good read to start with App stimulation is [26, 28, 29, 32, 38, 45]. Choudary et al.[38] and Sapienz[32] are dated back to 2015/2016 and even though they offer a good introduction into the topic as well as an evaluation of automation tool effectiveness, the information is not up to date. Vasquez et al.[28] deliver in 2017 a slightly more up to date overview paper. It gives a very comprehensive tool overview and mentions pros and cons of the different possible methods. Luo et al.[29] deliver in 2020 an updated overview of app stimulation tools. However, their objective is more functionality test oriented than security and privacy oriented. They focus on the simulation of a realistic sensor environment for testing the app.

Lam et al.[26] also provide an overview of UI automation tools, but only with the background of record and replay functionality. This provides the ability to record test sequences for known apps, but this concept doesn't work in a fully automatic approach. Tramontana et al.[45] provide in 2019 a systematic mapping study on automated functional testing. Due to the nature of such a study, it reveals hot research topics, current trends and popular papers in this area but lacks in giving recommendations on practically usable tools.

A good overview paper about app behavior monitoring is presented by Neuner et al.[35] in 2019. Unfortunately, the paper is no more up to date and also mainly focuses on app sandboxing for malware analysis. In general, most papers in this area are focused on the needs of malware analysis which however is often not applicable for privacy or vulnerability analysis. A more comprehensive and up to date list of tools is provided on Github.

In summary, we've found no overview paper or survey covering app stimulation (UI automation) as well as app behavior monitoring in the context of privacy and security. Thus, we want to propose an infrastructure model along with practically usable, up to date tools for large scale dynamic analysis environment for Android in this paper.

2.1 Contributions of this work

Abstract design: We identify required tools types and aspects of Android apps for a wholesome dynamic security and privacy analysis. We propose an abstract design housing the required components for an analysis environment.

Updated dynamic analysis tool overview: We give an updated combined overview of app stimulation (UI automation) **and** app behavior monitoring tools and maintain updated lists on Github to not become obsolete.

Identify essential tool features: We name necessary features in each area, and give suggestions on fitting tools containing these features and supporting current Android environments. Additionally, tool comparison tables offer a comprehensive, up to date summary of tool features for self evaluation.

3 DYNAMIC ANALYSIS ENVIRONMENT DESIGN

This section proposes an abstract dynamic analysis environment design (Figure 1). The following sections describe the required components, their capabilities and the required communication. The central element in the design is the device or the emulator, with the different software abstraction layers. On the one hand, there is an App Stimulation component, which remotely controls the device. On the other hand there is a Behavior Monitor and a Network Monitor which observe the app's interaction with device. A separation is desirable to simplify network traffic analysis. All performed actions and monitored device interaction are provided to the control logic. It primarily logs the execution details for later analysis after the dynamic analysis run.

3.1 App Stimulation

Basic interactions are tap, swipe or textual input. These interactions are handed to the OS kernel by privileged system apps and are usually triggered through interaction with the device hardware.

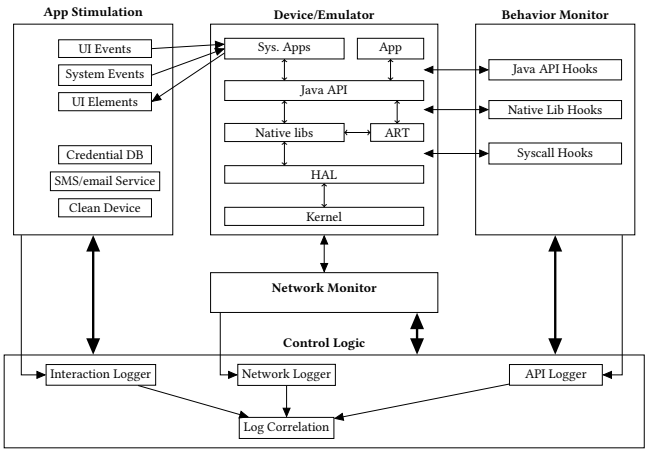

Figure 1: Proposed Tool Composition for a Dynamic Analyzer

To get an idea of necessary interactions, we manually interacted with the top 20 apps on Google Play. Besides basic UI events, such as tap or swipe actions, also more complex actions are required. An important first step for many apps is to overcome the sign in process. This mostly requires either a classic username/email and password login or they "Signin with Google" for authentication through Google. Alternatively, mostly messaging apps require a phone number to send authentication codes to, which need to be transferred to the app (SMS/email Service in Figure 1). Some apps also demand personal information like address, name, age, birthday. These fields need to be recognized and filled with appropriate information. Content hints are often provided on these input fields by the app programmer, they need to be evaluated and matched against predefined values in the credentials/personal info database. This information might later on be subject due to privacy leaks.

Another topic of interest is the settings menu. Menu items sometimes let one choose the used encryption and reveal default settings. The privacy terms and usage terms are also often found in this menu and changes in these terms from version to version are of interest for the user.

There also exist extended UI events such as pinch to zoom or other two finger actions. On the one hand, these multi-touch actions are quite diverse and are not (yet) covered through the ADB protocol. On the other hand, they are besides from games not very common and if used, are often also represented as separate buttons. Thus, such interaction seems necessary to find software crashes but is not required for the described goal due to the often redundant interaction possibilities.

Another category of events is system events, also called broadcast actions. This could for example be "new picture", where each app can sign up for to be notified when a new picture was taken. The signin for around 300 of such broadcasts was formerly required be stated in the `AndroidManifest.xml` of each app. Currently, apps can also dynamically signin for broadcasts during runtime. These broadcasts trigger methods which are usually never reachable during interaction via the app's UI. For comprehensive dynamic app analysis, the different registered broadcasts should be triggered and the app's behavior observed.

Since the intention is to build app stimulation, which repeatedly tests applications, it is desirable that the observed results and therewith the interaction sequence remains constant for the same app version. Additionally, a record and replay functionality would be useful to be able to also manually inspect analysis results (hinted as "Interaction Logger" in Figure 1).

3.2 App Behavior Monitor

We leveraged OWASP Mobile Top 10 (↗) and the Appicaptor Security Index (↗) to identify common vulnerabilities of apps. By far the most common security vulnerabilities are unprotected communication such as http instead of https access, faulty certificate validation in self implemented trust managers used in SSL/TLS communication or insecure encryption usage. For the aspect of privacy it is relevant to check the usage of data stored on the phone such as: contacts, messages, call history, ... but also device properties such as: GPS location, connected WiFi/Bluetooth networks, ... or installed apps. The usage of these properties needs to be tracked during behavior monitoring and in a second stage evaluated.

During UI stimulation, one needs measures to observe what the app actually does in the background. Therefor, different points of action exists. Like shown in Figure 1 the Android system has a layered architecture and in-between each layer there is the possibility to intercept the function calls. The higher the layer, the more abstract the function calls become. The highest layer is the Java API. For example the current GPS position can be requested via the API function `locationManager.getLastKnownLocation(LocationManager.GPS_PROVIDER)`. Thus, intercepting this function would exhibit GPS usage and one could for example even read the GPS position and see if this location is sent over the network later on. The data identification can be accomplished since the provided data like address book entries or provided GPS location are set or known during the automated device interaction. Since data can internally be converted into other representations, like base64, hex and different hashing functions, the intercepted data would also need to be checked against these alternative representations.

The native libraries reside below the Java API. These libraries are compiled to the target CPU architecture and give a slightly abstracted interface to kernel functionalities as libraries. This also allows C/C++ app-libraries to run on Android through the Java Native Interface (JNI). However, using sensors through JNI takes a lot of boilerplate code to initialize the sensors compared to the Java interface and some sensors (GPS) are not covered through JNI↗. In conclusion, intercepting native library calls is necessary to observe functionality of apps partly written in C/C++ but it can not replace Java API interception.

On the bottom layer is the hardware abstraction layer (HAL) and the kernel consuming so called system-calls. These system-calls handle for example threads, user rights management, file and file system interaction and sockets. Reading sensor data comes down to reading a specific file and configuring the sensor to write data to a file. Thus, calls to the Java API or native libraries are reflected in several system-calls. It is not possible to filter or log all system calls due to the immense call frequency. Thus, we don't recommend intercepting system-calls or make it very selectively and carefully.

3.3 Network Monitor

Even though network traffic could also be observed through API calls in the operating system, it might not be the most convenient way. Using for example a proxy server as network monitor lets one easily observe called addresses, the content of the transmission and if the requests are protected. It is also fairly easy to check for the usage of faulty trust managers, since certificates can easily be exchanged. As described in Section 3.2, it is also necessary to decode data in different data formats.

3.4 Control Logic

The control logic is thought of as the central element controlling the others. The control logic should on the one hand trigger the app stimulation and on the other hand set up appropriate hooks in the API and Network Monitor to observe the behavior triggered by the app stimulation. At the end of the interaction, the different logged information about app stimulation and API calls have to be correlated and security vulnerabilities or privacy frauds detected.

4 APP STIMULATION TOOL EVALUATION

This section evaluates different features provided by app stimulation tools (Table 1). We explain the different options that are not self-explanatory and list pros and cons of each method. The column **UI Events** indicates if a tool is able to perform basic UI events such as tap and swipe actions. **Context aware input** indicates the capability to fill out login forms or address fields. **System events** are capabilities to trigger broadcast messages if a photo was taken or the device screen went on/off. The **Instrumentation** column tells how the UI interaction is executed. This can either be a modified operating system (OS), an instrumented test app or no instrumentation at all using build in functionality. Modified operating systems are powerful but time intensive to maintain. Modifications have to be at least yearly renewed for new Android versions. Instrumenting the application has the advantage, that activities can directly be triggered. A disadvantage is that this requires repacking the app, which is detectable by the app. Some tools do not require instrumentation and thus rely on Androids inbuilt ADB. The disadvantage of this method is that the feature set is limited. Different **exploration** strategies exist to interact with UI elements: Random (rnd), Model-based (model) and Systematic (sys). Random approaches randomly trigger UI interactions and usually fire them at high rates. It is common to define seed, to make interaction consistent and reproducible. The advantage of this approach is that it often achieves high UI exploration coverage, mainly due to the high event firing rate. A disadvantage is, that it gets stuck at login screens without further context awareness. High event firing rates often reveal app crashes, which is good for finding bugs, but undesirable for a security and privacy analysis. Model-based exploration often generates a finite-state-machine to model the different app activities as states and the event input combinations to move from one state to another. Advantage of this approach is that different activities can be explored more directly and the tool can easily come back to certain activities in the model to fully explore all available activities. A disadvantage is that it is rather slow, since dumping the UI elements and assigning states requires much computation [22]. Thus, random approaches often achieve the same or higher UI coverage than

a model based approaches. Another challenge is to model states without a state explosion[21, 41]. The last option is a systematic approach. This approach defines certain rules to interact with the UI. For example start clicking buttons from left to right or first fill in text fields before pressing buttons. This approach also allows login handling. The biggest disadvantage of this approach is that the systematic interaction rules have to be carefully designed reach all activities and not get stuck. To overcome this flaw, random exploration is often additionally used. In the context of vulnerability and privacy analysis, a systematic or model-based exploration strategy seems favorable to overcome the login and context aware inputs which are not provided by fully random approaches. A **static app analysis** can be used to get insights of the app to be explored. For example how many activities are defined and from which activities one can switch to other activities [6]. This item is rated neutral due to the additional effort and few advantages in UI exploration coverage [38]. The column **Device Type** indicates if the tool runs only on a real hardware (dev) or just uses an emulator (emu) or runs on both. A tool supporting devices and emulators is preferable, since both have their advantages. Emulators scale very well in parallel on powerful servers. However, some apps refuse to run in emulators. Real devices are more costly and harder to maintain but are capable to run all apps. Thus, a mix of emulators and some devices seems perfect. To test apps, there are different **test type** methods: white-, gray- and black-box tests. Since the intention is to analyze stranger apps without source-code, only gray and black-box tests are feasible. The last feature is the ability to **record and replay** a test. This is a very important feature, as a detailed reproducible proof of judgment as good or bad app. This also gives the possibility to replay a test by hand verify a vulnerability or privacy finding. According to a survey of [45] in 2018, the most relevant publications are Dynodroid[30], A3E[6], Sapienz[32] and AndroidRipper[4]. Dynodroid lacks a license and seems to be not maintained since 2017. A3E is not maintained since 2016 and lacks in contextual sensitive input as well as generation of system events. Sapienz contains all desired features but is also not maintained since 2017. Android ripper lacks in generation of system events, a record and replay functionality and is also unmaintained since 2017. More recent tools with updates in 2019 to 2021 are: app-check, DroidMate, Google RoboTest, Monkey, REAPER and Stoat. DroidMate lacks in the generation of system events and additionally, DroidMate is unable to handle contextual input. Google RoboTest is an online service for dynamic app testing. Thus, it is good for finding bugs, but one can't perform custom tests. The Monkey tool lacks in contextual input generation and would get stuck at every login screen. REAPER lacks in system event generation and record and replay functionality. Stoat only supports white-box testing which is unusable for testing closed-source apps.

From the current situation the only considerable tool covering all desired features is Sapienz. Since the tool is not maintained since 2017, we tried to run it on Ubuntu 18.04, even though the docs state only support for Ubuntu 14.04. We were facing several problems, since Sapienz is written in Python2, which is already end-of-life. Some required packages were no longer available in Ubuntu 18.04 and could not be easily added. However, the supported Ubuntu 14.04 has severe problems with recent Android SDKs. We decided to stop the hassle for an unmaintained tool without an active community. An alternative would be using the currently maintained, but not broadly used tool: app-check. However, app-check lacks in system event generation which would have to be added manually. Alternatively, there also exist a few frameworks (see Section 4.1) which can easily be customized.

Table 1: App Stimulation Tools and Frameworks

Tool	Available	Last Active	License	UI Events	Context Input (Text/Email)	System Events	Instrumentation	Root Required	Exploration	Static Analysis	Device Type	Test Type	Record & Replay
A3E Depth-First [6]	✓(os)	16	BSD	✓	✗	✗	app	✗	model	✗	both	black	✗
A3E Targeted [6]	✗	16	✗	✓	✗	✗	app	✗	sys	✓	both	gray	✗
ACTEve [5]	✓(os)	13	BSD	✓	✗	✓	app,os	✗	sys	✓	both	white	✗
AGRippin [2]	✓(cs)	15	✗	✓	✓	✗	app	✗	model	✗	both	black	✓
AndroidRipper [4]	✓(os)	17	AGPL	✓	✓	✗	app	✗	model	✗	both	black	✗
app-check	✓(os)	20	AGPL	✓	✓	✗	✗	✗	rnd,sys	✗	both	black	✓
AppDoctor [23]	✗	14	✗	✓	✓	✓	app	✗	various	✓	both	gray	✗
CrawlDroid [11]	✓(os)	17	✗	✓	✗	✗	app	✗	model	✗	both	black	✗
CrashScope [34]	✗	16	✗	✓	✓	✓	✗	✗	sys	✗	both	black	✓
CuriousDroid [12]	✓(os)	16	GPL3	✓	✓	✗	app	✓	model	✓	both	gray	✗
DroidBot [54]	✓(os)	16	MIT	✓	progr.	✓	✗	✗	model	✗	both	black	✓
DroidFuzzer [53]	✓(os)	16	GPL3	✗	intents	✗	✗	✗	rnd	✓	both	gray	✓
DroidMate [10]	✓(os)	20	GPL3	✓	✗	✗	✗	✗	model	✓	both	gray	✗
Dynodroid [30]	✓(os)	17	✗	✓	login	✓	os	✗	rnd	✗	emu	black	✓
EvoDroid [31]	✗	14	✗	✓	✗	✗	✗	✗	sys	✗	both	white	✗
FragDroid [13]	✗	18	✗	✓	✗	✗	app	✗	model	✓	both	gray	✗
FSMdroid	renamed to Stoat												
RoboTest	✓(cs)	20	✗	✓	✓	✗	✗	✗	sys	✗	both	black	✓
GUIRipper	renamed to MobiGuitar												
Haoyin [22]	✗	17	✗	✓	✓	✓	✗	✗	rnd	✓	both	black?	✗
IntentFuzzer [39] -	✗	14	✗	✗	intents	✗	✗	✗	rnd	✓	both	white	✓
JPF-Android [46]	✓(os)	16	Apache	✓	✗	✗	✗	✗	sys	✓	both	white	✓
LAND [50]	✗	16	✗	✓	✓	✓	app	✗	model	✗	both	black	✗
MAMBA [24]	✗	16	✗	✓	✓	✓	app	✗	sys	✓	both	gray	✓
MobiGuitar [3]	✓(os)	17	✗	✓	✓	✗	app	✗	model	✗	both	black	✗
Monkey	✓(cs)	20	✗	✓	✗	✓	✗	✗	rnd	✗	both	black	✓
MonkeyLab [27]	✗	15	✗	✓	✓	✗	✗	✓	model	✗	both	black	✓
ORBIT [52]	✗	13	✗	✓	✗	✗	✗	✗	model	✓	both	white	✗
PUMA [21]	✓(os)	14	Apache	✓	progr.	✗	app	✗	model	✗	both	black	✗
QBE [25] -	✗	18	✗	✓	✓	✓	✗	✗	model	✗	both	black	✓
Quantum [55] -	✗	14	✗	✓	✗	✓	app	✓	model	✗	both	black	✓
REAPER [17]	✓(os)	19	GPL3	✓	login	✗	✗	✓	sys	✗	both	black	✗
Sapienz [32]	✓(os)	17	BSD	✓	✓	✓	app	✗	model	✓	both	gray	✓
Sig-droid [33]	✗	15	✗	✓	✓	✗	✗	✗	symb. exec.	✓	both	white	✓
Stoat [41]	✓(os)	19	✗	✓	✓	✓	✗	✗	model	✓	both	white	✓
Swifthand [14]	✓(os)	15	BSD	✓	✓	✗	app	✗	model	✓	both	black	✗
Thor [1]	✓(os)	17	✗	✗	✗	✓	app	✗	manual	✓	emu	white	✗
Frameworks													
UI Automator	✓(os)	20	Apache	✓	✓	✗	✗	✗	-	-	both	black	✗
Espresso	✓(os)	20	Apache	✓	✓	✗	app	✗	-	-	both	white	✗
Appium	✓(os)	20	Apache	✓	✓	✓	✗	✗	-	-	both	black	✓
Robotium	✓(os)	16	Apache	✓	✓	✗	app	✗	-	-	both	black	✓
Roboelectric	✓(os)	20	MIT	✓	✓	✗	app	✗	-	-	✗	white	✗
Ranorex	✓(cs)	20	✗	✓	✓	✗	✗	✗	-	-	both	black	✓
Calabash	✓(os)	20	EPL	✓	✓	✗	✗	✗	-	-	both	white	✗
Quantum	✓(cs)	20	✗	✓	✓	✓	app	✗	-	-	both	white	✗
Qmetry	✓(os)	20	MIT	✓	✓	✓	app	✗	-	-	both	black	✓
Selendroid	✓(os)	18	Apache	✓	✓	✗	✗	✗	-	-	both	black	✗

* os=open-source, cs=closed-source, app=instrumented app, os=instrumented operating system, emu=emulator, both=emulator&device, dev=device, white/gray/black=white/gray/back-box testing

4.1 App Stimulation Framework Evaluation

App stimulation frameworks are presented in the second part of Table 1. The property explanations match the ones in the previous section. Currently, only Appium, Quantum and Qmetry support the desired feature of generating system event, even though some events are only supported in the emulator. Quantum is a proprietary tool, only supporting white box tests and no record and replay. Qmetry integrates Appium, Selenuim/Webdriver and Quantum. Thus, it is rather a common interface for test automation. The focus of the framework lies on a test-driven development rather than generic app interaction, which shifts Qmetry out of the focus.

Table 2: Behavior Monitor Tools

Tool		Available	Last Active	License	APK Repacking	Root Req.	Sandbox	Device Type	Dyn. Hooking	Hooking Scope	Instrumentation
ANANAS	[18]	✗	13	✗	✗	✗	✗	emu	✗	both	pass
APIMonitor		✓(os)	15	GPL2	✓	✗	✗	emu	✗	both	pass
AppCage	[59]	✗	15	✗	✗	✗	✓	both	✗	both	act
Aurasium	[48]	✓(os)	15	GPL3	✓	✗	✗	both	✗	both	act
AppsPlayground	[37]	✓(os)	14	✗	✗	✗	✗	emu	✗	both	pass
ARTDroid	[15]	✓(os)	15	GPL2	✗	✓	✗	both	✗	both	act
ARTist	[8]	✓(os)	19	Apache	✓	✓	✗	both	✗	both	act
Aurasium	[48]	✓(os)	15	GPL3	✓	✗	✗	both	✗	sys	act
Boxify	[7]	✗	16	✗	✗	✗	✓	both	✓	both	act
CopperDroid	[42]	✗	18	✗	✗	✗	✗	emu	✗	sys	pass
CydiaSubstrate		✗	14	✗	✗	✓	✗	both	✓	usr	act
DroidBox		automation script for TaintDroid & APIMonitor							✗		
DroidScope	[51]	✓(os)	18	GPL3	✗	✗	✗	emu	✗	sys	pass
Frida		✓(os)	20	wxWin	✓	✓	✗	both	✓	both	act
Introspy		GUI for CydiaSubstrate									
Malton	[49]	✗	17	✗	✗	✗	✓	both	✗	sys	pass
MobileSandbox	[40]	DroidBox extension									
Ninja	[36]	✗	17	✗	✗	✗	✗	dev	✓	sys	pass
Njas	[9]	✗	15	✗	✗	✗	✓	both	✗	sys	act
ptrace/strace		✓(os)	20	GPL2	✗	✓	✗	both	✓	sys	act
Retroskeleton	[16]	✗	13	✗	✓	✗	✗	both	✗	both	act
Riru		✓(os)	20	✗	✗	✓	✗	both	✓	usr	act
Ronin	[44]	✓(os)	17	MIT	✓	✗	✗	both	✓	both	act
SandHook		✓(os)	20	anti996	✗	✓	✗	both	✓	usr	act
SIF	[20]	✗	13	✗	✓	✗	✗	both	✗	sys	act
SmartDroid	[58]	✓(os)	13	✗	✗	✗	✗	emu	✗	both	act
StaDynA	[57]	✗	15	✗	✗	✗	✗	modOS	✗	sys	pass
TaintDroid	[19]	✓(os)	13	Apache	✗	✗	✗	modOS	✗	sys	pass
TraceDroid	[47]	✗	13	✗	✗	✗	✗	modOS	✗	sys	pass
UpDroid	[43]	✗	18	✗	✗	✗	✗	both	✗	events	pass
VetDroid	[56]	✓(os)	16	✗	✗	✗	✓	modOS	✗	sys	pass
VirtualHook		✓(os)	20	✗	✗	✗	✓	both	✗	sys	act
Xposed		✓(os)	18	✗	✗	✓	✗	both	✓	sys	act
YAHFA		✓(os)	20	GPL3	✗	✓	✓	both	✓	both	act

* os=open-source, cs=closed-source, emu=emulator, both=emulator&device, dev=device, modOS=modifed Android OS, sys/usr=user/system-space hooks

Appium provides all required features, it has a broad and active user base. Thus, it is a recommendation if one wants to use a framework and implement own UI interaction strategies.

5 BEHAVIOR MONITOR TOOL EVALUATION

Table 2 gives an overview of behavior monitoring tools which can be taken to intercept different API levels. We explain the different options that are not self-explanatory and list pros and cons of each method. The column: **Sandbox** denotes if an app is executed inside a sandbox, similar to a virtual machine. The advantage of sandboxing is that the app can still run on a real device. This technique also offers great opportunities for Java-API call interception. A disadvantage that the performance is mostly very poor. The modified Android (modOS) key is new and describes a modified Android platform. This method offers excellent insights into the application at all inspection levels. The price to pay is that modifications have to be adapted yearly with each new Android release, which is a massive burden. **Dynamic Hooking** indicates the possibility to add and remove hooks i.e. API inspection points during runtime. It is desirable especially for low level APIs to be able to observe only current areas of interest and deactivate other inspection points dynamically. The **Hooking Scope** column tells at which level hooks can be applied. This can be either only in the user (usr) space of the app or even at system level (sys). Only the system level provides a sufficient inspection depth. **Instrumentation** indicates if the hooking tool is able to either only log the different API-calls (passive) or also manipulate return values of function calls (active). It is desirable to have active instrumentation, which makes it easily possible to restrict an app's access to resources or deliver fake data which could later on be identified and observe the app's behavior. Tools fulfilling the described demands are: Boxify, Frida, ptrace/strace, Ronin, Xposed and YAHFA. Boxify and Ronin have not been maintained for four years and thus they can not be recommended. Also, Xposed was once a big player, but support was dropped in 2018 and no license is provided, making it risky to use or continue development. YAHFA is currently under active development, but most of the documentation is only available in chinese. It seems like there is no broad community behind this project which makes us hesitate to give a recommendation. ptrace and strace are very powerful tools for debugging and manipulation, but since they work on system level, they require root access on Android. Additionally, they don't have the most intuitive user interface. Frida internally also uses ptrace mechanism, but is alternatively also able to repack the app. Frida also offers a more convenient and quite well documented interface as well as a broad and active community. Hooks are possible on all Android API levels. For a lot of scenarios, there are scripts publicly available. Objection is a frontend for Frida shipping with some build-in Frida scripts, a beginner-friendly CLI, a HTTP JSON RPC API, and the possibility to extend functionality with own plugins. Thus, Objection looks very promising, especially due to the RPC API.

6 NETWORK MONITOR

For monitoring the network, features such as http(s) packet content inspection, SSL/TLS certificate exchange and packet modification are required. An API to control and export logs would be essential to properly interface the Network Monitor from the Control Logic. There exist plenty of network inspection tools like the well known Wireshark. However, these tools tackle a too low inspection level. Charles and HTTP Toolkit are easy-to-use proxies with a nice UI, but lack an external API, which makes it hard to control in an automated environment. Fiddler is a closed-source tool with an almost two decade long history and provides a plenitude of features. However, it lacks an external control API and automation can only be achieved through JScript.NET extensions/scripts, a mix of .NET and JavaScript. ZAproxy and Burp Suite is a proxy mainly designed for website security testing. ZAproxy offers a great API interaction interface for a variety of languages. However, both tools focus on web page pentests and omit required features such as packet and certificate modification. mitmproxy is a well maintained open-source tool that offers a python API interface for external control and supports all required features. As a result, mitmproxy can be recommended as network monitor tool.

7 CONCLUSION & FUTURE WORK

In this paper, we proposed an abstract design for dynamic Android app vulnerability and privacy analysis and discussed the necessary components of such an infrastructure. An extensive overview of available tools for each component is given and relevant tool properties are displayed in tables for easy comparison, updating outdated related overview papers. Different tools in each category are evaluated based on these aspects and promising ones are promoted. For app stimulation we recommended Appium and app-check, for behavior monitoring a suggestion to Objection and Frida is given in combination with mitmproxy as network monitor. To not share the fate of past overview papers, we propose continuously updated Github tables for everyone to edit. In summary, the available tools today are mostly not usable as plug-and-play. Much implementation effort is still required to achieve a dynamic app analysis environment. As future work, we intent to fill this gap and provide a plug-and-play solution in the context of dynamic Android app analysis. With this publication, we hope to give developers having the goal to generate a dynamic analysis environment a good introduction to the topic as well as a list of available tools at hand.

ACKNOWLEDGMENTS

The project underlying this report was funded by the German Federal Ministry of Education and Research under grant number 16SV8520. The author is responsible for the content of this publication.

REFERENCES

[1] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic Execution of Android Test Suites in Adverse Conditions *(ISSTA)*.
[2] D. Amalfitano, N. Amatucci, A. R. Fasolino, et al. 2015. AGRippin: A Novel Search Based Testing Technique for Android Applications *(DeMobile)*.
[3] D. Amalfitano, A. R. Fasolino, P. Tramontana, et al. 2015. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Software* (2015).
[4] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. 2012. Using GUI ripping for automated testing of Android applications *(ASE)*.
[5] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated Concolic Testing of Smartphone Apps *(FSE)*.
[6] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and Depth-First Exploration for Systematic Testing of Android Apps. *SIGPLAN* (2013).
[7] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. Von Styp-Rekowsky. 2015. Boxify: Full-Fledged App Sandboxing for Stock Android *(USENIX Security)*.
[8] M. Backes, S. Bugiel, O. Schranz, P. von Styp-Rekowsky, and S. Weisgerber. 2017. ARTist: The Android runtime instrumentation and security toolkit *(EuroS&P)*.
[9] A. Bianchi, Y. Fratantonio, C. Kruegel, et al. 2015. NJAS: Sandboxing Unmodified Applications in Non-Rooted Devices Running Stock Android *(CCS/SPSM)*.
[10] Nataniel P. Borges Jr. 2017. Data Flow Oriented UI Testing: Exploiting Data Flows and UI Elements to Test Android Applications *(ISSTA)*.
[11] Yuzhong Cao, Guoquan Wu, Wei Chen, and Jun Wei. 2018. CrawlDroid: Effective Model-Based GUI Testing of Android Apps *(Internetware)*.
[12] P. Carter, C. Mulliner, M. Lindorfer, et al. 2017. CuriousDroid: Automated User Interface Interaction for Android Application Analysis Sandboxes *(FC)*.
[13] J. Chen, G. Han, S. Guo, and W. Diao. 2018. FragDroid: Automated User Interface Interaction with Activity and Fragment Analysis in Android Applications *(DSN)*.
[14] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning *(OOPSLA)*.
[15] Valerio Costamagna and Cong Zheng. 2016. ARTDroid: A Virtual-Method Hooking Framework on Android ART Runtime *(IMPS@ESSoS)*.
[16] B. Davis and H. Chen. 2013. RetroSkeleton: Retrofitting Android Apps *(MobiSys)*.
[17] M. Diamantaris, E. P. Papadopoulos, E. P. Markatos, et al. 2019. REAPER: Real-Time App Analysis for Augmenting the Android Permission System *(CODASPY)*.
[18] T. Eder, M. Rodler, D. Vymazal, and M. Zeilinger. 2013. ANANAS - A Framework for Analyzing Android Applications *(ARES)*.
[19] W. Enck, P. Gilbert, et al. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones *(USENIX OSDI)*.
[20] Shuai Hao, Ding Li, William G.J. Halfond, and Ramesh Govindan. 2013. SIF: A Selective Instrumentation Framework for Mobile Applications *(MobiSys)*.
[21] Shuai Hao, Bin Liu, Suman Nath, et al. 2014. PUMA: Programmable UI-Automation for Large-Scale Dynamic Analysis of Mobile Apps *(MobiSys)*.
[22] L. V. Haoyin. 2017. Automatic android application GUI testing—A random walk approach *(WiSPNET)*.
[23] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. 2014. Efficiently, Effectively Detecting Mobile App Bugs with AppDoctor *(EuroSys)*.
[24] J. C. J. Keng, L. Jiang, T. K. Wee, and R. K. Balan. 2016. Graph-Aided Directed Testing of Android Applications for Checking Runtime Privacy Behaviours *(AST)*.
[25] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez. 2018. QBE: QLearning-Based Exploration of Android Applications *(ICST)*.
[26] Wing Lam, Zhengkai Wu, Dengfeng Li, et al. 2017. Record and Replay for Android: Are We There yet in Industrial Cases? *(ESEC/FSE)*.
[27] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, et al. 2015. Mining Android App Usages for Generating Actionable GUI-Based Execution Scenarios *(MSR)*.
[28] M. Linares-Vásquez, K. Moran, et al. 2017. Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing *(ICSME)*.
[29] C. Luo, J. Goncalves, E. Velloso, et al. 2020. A Survey of Context Simulation for Testing Mobile Context-Aware Applications. *ACM Comput. Surv.* (2020).
[30] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps *(ESEC/FSE)*.
[31] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: Segmented Evolutionary Testing of Android Apps *(FSE)*.
[32] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-Objective Automated Testing for Android Applications *(ISSTA)*.
[33] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek. 2015. SIG-Droid: Automated system input generation for Android applications *(ISSRE)*.
[34] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, et al. 2016. Automatically Discovering, Reporting and Reproducing Android Application Crashes *(ICST)*.
[35] Sebastian Neuner, Victor van der Veen, Martina Lindorfer, Markus Huber, Georg Merzdovnik, Martin Mulazzani, and Edgar R. Weippl. 2014. Enter Sandbox: Android Sandbox Comparison. *CoRR* (2014).
[36] Zhenyu Ning and Fengwei Zhang. 2017. Ninja: Towards Transparent Tracing and Debugging on ARM *(USENIX Security)*.
[37] Vaibhav Rastogi, Yan Chen, and William Enck. 2013. AppsPlayground: Automatic Security Analysis of Smartphone Applications *(CODASPY)*.
[38] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? *(ASE)*.
[39] Raimondas Sasnauskas and John Regehr. 2014. Intent Fuzzer: Crafting Intents of Death *(WODA+PERTEA)*.
[40] M. Spreitzenbarth, F. Freiling, F. Echtler, et al. 2013. Mobile-Sandbox: Having a Deeper Look into Android Applications *(SAC)*.
[41] Ting Su, Guozhu Meng, Yuting Chen, et al. 2017. Guided, Stochastic Model-Based GUI Testing of Android Apps *(ESEC/FSE)*.
[42] Kimberly Tam, Salahuddin Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors *(NDSS)*.
[43] Xiaoxiao Tang, Yan Lin, Daoyuan Wu, and Debin Gao. 2018. Towards Dynamically Monitoring Android Applications on Non-Rooted Devices in the Wild *(WiSec)*.
[44] Nikolaos Totosis and Constantinos Patsakis. 2018. Android Hooking Revisited.
[45] P. Tramontana, D. Amalfitano, et al. 2019. Automated functional testing of mobile applications: a systematic mapping study. *Software Quality Journal* (2019).
[46] Heila van der Merwe, Brink van der Merwe, and Willem Visser. 2012. Verifying Android Applications Using Java PathFinder. *SIGSOFT Softw. Eng. Notes* (2012).
[47] V. van der Veen. 2013. *Dynamic Analysis of Android Malware*. Ph.D. Dissertation.
[48] Rubin Xu, Hassen Saïdi, and Ross Anderson. 2012. Aurasium: Practical Policy Enforcement for Android Applications *(USENIX Security)*.
[49] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. 2017. Malton: Towards on-Device Non-Invasive Mobile Malware Analysis for ART *(USENIX Security)*.
[50] J. Yan, T. Wu, J. Yan, and J. Zhang. 2017. Widget-Sensitive and Back-Stack-Aware GUI Exploration for Testing Android Apps *(QRS)*.
[51] Lok Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis *(USENIX Security)*.
[52] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications *(FASE)*.
[53] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag *(MoMM)*.
[54] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: a lightweight UI-Guided test input generator for android *(ICSE-C)*.
[55] R. N. Zaeem, M. R. Prasad, and S. Khurshid. 2014. Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps *(ICST)*.
[56] Yuan Zhang, Min Yang, Bingquan Xu, et al. 2013. Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis *(CCS)*.
[57] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, et al. 2015. StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications. *CODASPY '15*.
[58] C. Zheng, S. Zhu, S. Dai, et al. 2012. SmartDroid: An Automatic System for Revealing UI-Based Trigger Conditions in Android Applications *(SPSM)*.
[59] Yajin Zhou, Kunal Patel, Lei Wu, Zhi Wang, and Xuxian Jiang. 2015. Hybrid User-Level Sandboxing of Third-Party Android Apps *(ASIA CCS)*.