OPIA: A Tool for On-Device Testing of Vulnerabilities in Android Applications

Laura Bello-Jiménez[1], Alejandro Mazuera-Rozo[2,1], Mario Linares-Vásquez[1], Gabriele Bavota[2]
[1]Universidad de los Andes, Bogotá, Colombia
[2]Università della Svizzera italiana, Lugano, Switzerland
ln.bello10@uniandes.edu.co, alejandro.mazuera.rozo@usi.ch, m.linaresv@uniandes.edu.co, gabriele.bavota@usi.ch

Abstract*—Mobile developers constantly have to deal with users pressure for continuous delivery of apps while keeping quality attributes such as confidentiality and data integrity. To better support developers in testing security vulnerabilities during evolution and maintenance of mobile apps, in this demo we present a novel tool, OPIA, for on-device security testing. OPIA allows developers/testers to (i) conduct SQL-injection attacks and collect logs to identify leaks of sensitive information through record-and-replay testing, and (ii) extract data stored in local databases and shared preferences to identify sensitive information that is not properly encrypted, anonymized. OPIA is publicly available at GitHub. *Videos list: https://tinyurl.com/y379oror Website: http://tiny.cc/1pah8y

***Index Terms*—Android; security; testing; confidentiality**

I. Introduction

Mobile apps and devices support humans daily activities, and have even become personal vaults where their private information is stored directly by them or by apps that want to offer offline features. By 2018, 74.45% of smartphone users had Android [1] and each of these users open, on average, 9 apps per day [2]. The official Google App Store counts 2.6M apps [3] with more than 20 Billions of downloads [4].

Testing apps is a challenging task due to aspects such as fragmentation at operating system and device levels. While significant progress has been made in the area of security testing, the available tools suffer of some noteworthy limitations: (i) previous approaches mostly rely on static analysis, which is a time-expensive, resource-expensive and false-positives prone technique; (ii) most approaches are focused only on finding vulnerabilities that are not related to the local storage of private data, for instance, overprivileged access, insecure statements and lack of input validation on inter-component communication; and (iii) some approaches care about availability, avoiding crashes. In summary, few approaches focus on dynamic analysis and in particular on caring about keeping user's data safe and immutable.

Motivated by the aforementioned limitations, the amount of sensitive information that users and apps store in their smartphones, and the growing amount of android apps, we designed and implemented **OPIA**, a practical tool that helps developers to dynamically test and discover security vulnerabilities in Android applications (*e.g.,* exposure of sensitive data and insertion or alteration of data without authorization) directly on their device.

Linares-Vásquez was partially supported by a Google Latin American Research Award 2018.

OPIA replays the behavior of a user (previously recorded) and manipulates user inputs in order to inject malign SQL strings, pulls out the logs printed on the console by the developer, and extracts data stored in database tables and shared preferences (*i.e.,* data that is saved locally in a key-value format) saved on the phone without proper encryption. The SQL injection and on-device data extraction are aimed at testing data integrity and confidentiality, and app availability. The **OPIA** workflow is semi-automated since it only needs the developer to record the execution scenarios and to check whether there is private information leaked on the logs or unencrypted sensitive data in the local storage.

To make **OPIA** an easy to use tool, we developed a distributed architecture composed of an Android-Java mobile app, a Python server and a NoSQL database. With the **OPIA** mobile app, a user can select the app to be tested in order to record and replay user's behavior. **OPIA** includes an Execution Engine which extracts unencrypted information from databases, shared preferences and logs; a front-end module is also provided to display the retrieved data. This architecture is designed to achieve horizontal scalability and flexibility.

II. The Opia testing tool

OPIA overcomes the limitations of existing tools to test Android applications in several ways: (i) **OPIA** is able to detect and test vulnerabilities dynamically; (ii) the approach is not only focused on testing availability but confidentiality and integrity as well; (iii) the tool is app-crash resilient, *i.e.,* if there is a crash on the app under test, **OPIA** restarts it and continues the execution; (iv) to operate, **OPIA** only requires the app to be installed on a device (*i.e.,* no code is required).

OPIA's target base population is app developers who wish to keep user's data safe by finding security leaks before releasing their apps after evolution/maintenance tasks. From a developer's perspective, the **OPIA**'s workflow is simple; because **OPIA** uses accessibility services, then it must be enabled in the device via the Android settings app. Then, the app under test must be selected on the **OPIA** app. Once the application is selected, **OPIA** records actions and enables the options to replay, inject, extract data and display its findings.

A. **OPIA** Architecture

OPIA's architecture is outlined in Figure 1. **OPIA** has three main components: (i) User Interface (UI) implemented as a mobile app, (ii) an Execution Engine (EE) that executes ADB commands, and (iii) a NoSQL database hosted at Firebase. The **OPIA**'s architecture has been designed with the following design principles in mind:

- *Data privacy*: neither the UI nor the EE should store any information of the executions or extracted data after completing the workflow;
- *Horizontal scalability*: achieved through a central server able to communicate with many **OPIA** UIs running on different devices;
- *Maintenance*: by keeping functionalities and responsibilities distributed between the UI, EE and NoSQL database.

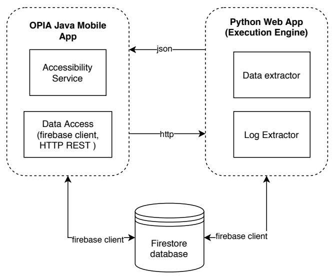


Fig. 1: Architecture of the OPIA tool.

We decided to use a "Call-Return" architectural style to achieve the consumption of centralized resources by the UI from the EE keeping the possibility of horizontal scalability. We used a Python Server as the Execution Engine, which contains a set of features that can be requested from the phone in which the **OPIA** UI is installed. The Call-Return style allows easy addition of new clients which can communicate with the EE. Also, the EE provides a REST-based interface that allows the **OPIA** UI to request the execution of a given feature at the EE.

B. The **OPIA** Mobile Application (UI)

The goal of the UI is to give the users the possibility of: (i) observe and reproduce user's actions in GUI Components, (ii) request the execution of **OPIA** tasks in the EE to extract tables, shared preferences and logs, and (iii) execute SQLi attacks. **OPIA** records three type of user actions: (i) *view text changed*, which recognizes when the content of an EditText changes, such as character addition, deletion, cropping and pasting; (ii) *view clicked*, identifies when a view, such as Button, RadioButton, CheckBox, etc, is touched; (iii) *view scrolled*, which notices when a scrollable view is scrolled. The events are saved in the NoSQL database with information related to the executed GUI components such as package name, type, location, execution time, among other information.

The sequence of recorded events are saved on Firebase as documents in a collection. Then, to replay a previous execution, the **OPIA** UI queries the database for the corresponding sequence of events. For each event it searches the GUI Component based on classname, text and bounds in parent/screen and performs the action specified on the event (text, scroll, click). Once the action is performed on the device, the UI sends a HTTP request to the EE, requesting to extract the log of the application under analysis. Also, the UI communicates with the EE to extract the database tables and shared preferences used by the app, in order to later execute (on demand) SQLi attacks by replacing text events with malicious inputs, such as drop statements to delete tables from the database and malformed queries to bypass login or crash the application under analysis.

C. The **OPIA** Execution Engine (EE)

The Execution Engine is in charge of executing Android Debug Bridge (ADB) commands remotely and processing the outputs. In particular, the EE: (i) extracts a backup of the mobile app to retrieve tables and shared preferences; (ii) retrieves, filters and clears the logcat during the execution of a sequence of events; (iii) saves the extracted information in the NoSQL database; (iv) displays extracted information in a browser that can be accessed directly on the device; and (v) restarts the application under analysis when it crashes.

The EE runs on a local server (outside of the device) that receives requests from the **OPIA** UI. When the request is related to extract tables, the EE gets the name of the package and executes the command **`adb backup -noapk`** ***`package-name`*** to get a backup of the application as a **`backup.ab`** file. It is worth noting that, if the application under analysis has its data encrypted, the EE cannot extract the backup to a local folder. Later, the EE searches for two important folders: (i) /db/ which contains all the databases as .db files; (ii) /sp/ which contains all the Shared Preferences as .xml files. Next, the EE saves all extracted data in the NoSQL database and the developer can see them on a browser. To do this, the EE gets and parses the values to create a HTML file displaying the database tables and shared preferences.

The EE is also in charge of retrieving the Android logcat during the execution of a sequence of events. First, the EE finds the number of the process of the analyzed application and filters the logcat to get only the lines related to the application. Once the EE acquires the log, it looks only for lines with application logs, *i.e.,* the ones a developer prints during development time, such as: `System.out.print`, `Log.d`, `Log.i`, `Log.w`, `Log.v` and `Log.e`. Afterwards, **OPIA** checks if the current activity is an 'Application Error' activity, that means that the application has crashed and the standard Android crash dialog has appeared. If a crash is encountered, the EE replies that an error occurred and executes **`adb shell am force-stop`** ***`package-name`*** to stop the application and restart it in order to continue with the workflow. **OPIA** preserves the log of each execution to display it to the developer as a HTML table, so the developer can check if private/sensitive information is leaked in the logcat.

III. **OPIA** IN ACTION

After installing **OPIA** in a device and starting the EE on a server, a user can (i) select an app to analyze, (ii) record actions executed on an application to be analyzed, (iii) see a list of previous records, (iv) replay previous executions, (v) see the log of previous replays, (vi) extract and visualize tables and shared preferences (SP) or (vii) execute a SQLi attack with an existing execution record.

The workflow starts with the record and replay (R&R) feature. Thus, the first step to use **OPIA** is to record a sequence of actions (executed on the app by the developer) through the 'Record' button, which starts the application under analysis. Next, the accessibility service detects the exercised GUI component events (*e.g.,* click-event, text entry, scroll, keyevent) and saves them on the NoSQL Database (*i.e.,* Firebase). To stop the recording, the developer presses a button on the top to return to **OPIA**'s UI. For each application there is a list of previous records with the possibility of replay a sequence of recorded events and show the corresponding log (Figure 2 - middle). For example, the list of previous records in Figure 2 shows that the user has recorded four execution scenarios for the Gmail app.

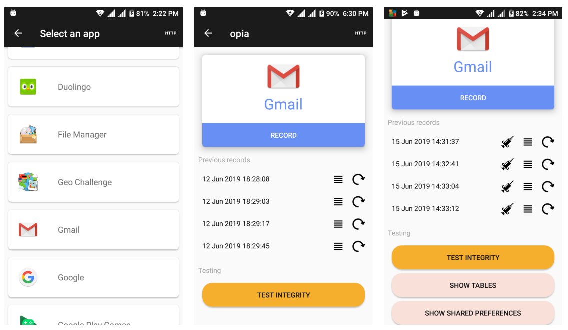
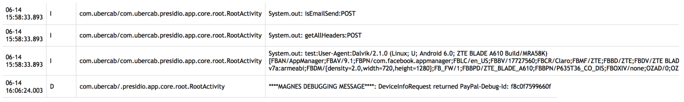


Fig. 2: Screenshots of the **OPIA** UI: List of apps (left), records of a given app (center), and options to check local data and SQLi (right).

To replay previous records, the **OPIA** UI gets from Firebase the sequence of events and parses them to find the GUI components involved in the events. Then, the accessibility service performs the action on the component, gets the log and saves it on the NoSQL database. When the execution finishes, the UI is opened again to give the developer the possibility to perform another action on the app under analysis.

R&R is the most important feature of **OPIA** because it is the foundation for extracting information and executing SQLi attacks dynamically. R&R allows to execute scenarios without dealing with automatic detection of GUIs that require login and automatic generation of authentication data. An example of the R&R feature can be seen in the following video: https://youtu.be/ygoFKBxLITM.

The workflow continues with the Information Exposure feature. The **OPIA** UI also shows (as a HTML table) the log printed during the execution of a scenario (Figure 3). For example, when checking the log of Gmail, there was no extra data printed in console. However, the browser in Figure 3 displays the log after replaying a record of Uber: It is possible to see that the developer is leaking some information about Payment methods and the device in which the app is running. Leaving information printed in console is dangerous because some of it could be sensitive and attackers could steal it.

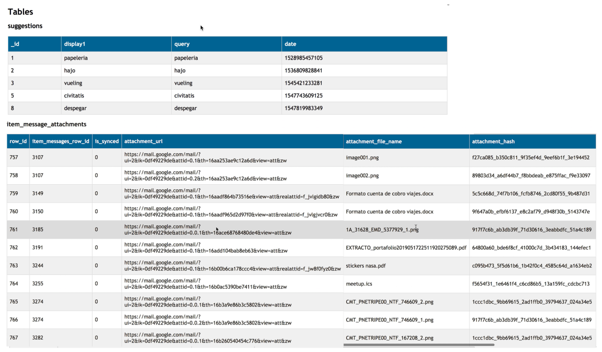


```
06-14         I   com.ubercab/com.ubercab.presidio.app.core.root.RootActivity   System.out: isEmailSend:POST
15:58:33.893
06-14         I   com.ubercab/com.ubercab.presidio.app.core.root.RootActivity   System.out: getAllHeaders:POST
15:58:33.893
06-14         I   com.ubercab/com.ubercab.presidio.app.core.root.RootActivity   System.out: test:User-Agent:Dalvik/2.1.0 (Linux; U; Android 6.0; ZTE BLADE A610 Build/MRA58K)
15:58:33.893                                                                    [FBAN/AppManager;FBAV/9.1;FBPN/com.facebook.appmanager;FBLC/en_US;FBBV/17727560;FBCR/Claro;FBMF/ZTE;FBBD/ZTE;FBDV/ZTE BLAD
                                                                                v7a:armeabi;FBDM/{density=2.0,width=720,height=1280};FB_FW/1;FBBPD/ZTE_BLADE_A610;FBBPN/P635T36_CO_DIS;FBOXIV/none;OZAD/0;OZ
06-14         D   com.ubercab/.presidio.app.core.root.RootActivity              ****MAGNES DEBUGGING MESSAGE****: DeviceInfoRequest returned PayPal-Debug-Id: f8c0f7599660f
16:06:24.003
```

Fig. 3: **OPIA**'s log for the Uber app.

Besides, after an execution is recorded, a user can check data integrity by extracting database and SP information from the device. The **OPIA** UI displays two buttons ('Show tables' and 'Show shared preferences') in which the developer can list unencrypted information stored on the tables and SP. Once those values are saved in Firebase, the developer can see them on a browser. To do this, the EE gets the values, parses them and creates a HTML file displaying the tables and SP as HTML tables. Getting database/SP data allows the user to first check if there is unencrypted data in both database and SP. This is due to the fact unencrypted data stored on devices can be easily exposed by using Android SDK utilities, which attacks confidentiality and privacy. Thus, it is a developer's duty to examine which data must be stored in a secure way and handle it to maintain privacy as much as possible. With **OPIA**, a developer can easily identify unencrypted data.

Figure 4 depicts the results of a real execution of **OPIA** for the Gmail app, showing some tables that were stored locally without proper encryption. For example, *suggestions*, which displays information about recent searches and *item-message-attachments*, which stores paths to files attached and sent. This information could be used by an attacker to learn users behavior and even stole attached files using the leaked paths.



Tables

suggestions

_id	display1	query	date
1	papeleria	papeleria	1528985457105
2	hajo	hajo	1536809828841
3	vueling	vueling	1545421233281
5	civitatis	civitatis	1547743609125
8	despegar	despegar	1547819983349

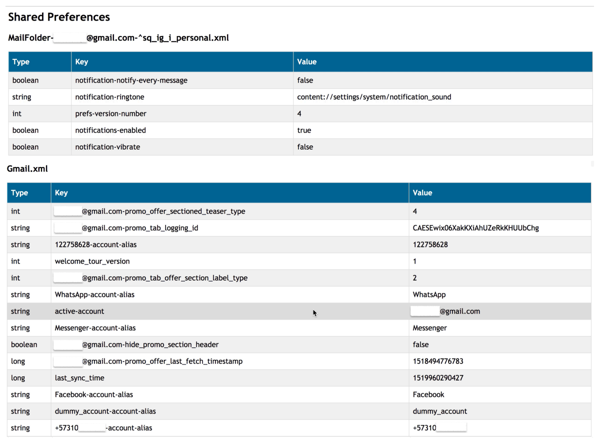
item_message_attachments

row_id	item_messages_row_id	is_synced	attachment_url	attachment_file_name	attachment_hash
757	3107	0	https://mail.google.com/mail/?ui=2&ik=0df49229de&attid=0.1&th=16aa253ae9c12a6d&view=att&zw	image001.png	f27ca085_b350c811_9f35ef4d_9eef6b1f_3e194452
758	3107	0	https://mail.google.com/mail/?ui=2&ik=0df49229de&attid=0.2&th=16aa253ae9c12a6d&view=att&zw	image002.png	89803d34_a6df44b7_f8bbdeab_e875ffac_f9e33097
759	3149	0	https://mail.google.com/mail/?ui=2&ik=0df49229de&attid=0.1&th=16aadf864b73516e&view=att&realattid=f_jvlgidb80&zw	Formato cuenta de cobro viajes.docx	5c5c668d_74f7b106_fcfb8746_2cd80f55_9b487d31
760	3150	0	https://mail.google.com/mail/?ui=2&ik=0df49229de&attid=0.1&th=16aadf965d2d97f0&view=att&realattid=f_jvlgjvcr0&zw	Formato cuenta de cobro viajes.docx	9f647a0b_efbf6137_e8c2af79_d948f30b_5143747e
761	3185	0	https://mail.google.com/mail/?ui=2&ik=0df49229de&attid=0.0.1&th=16ace68768480de4&view=att&zw	1A_31628_EMD_5377929_1.png	917f7c6b_ab3db39f_71d30616_3eabbdfc_51a4c189
762	3191	0	https://mail.google.com/mail/?ui=2&ik=0df49229de&attid=0.1&th=16add104bab8eb63&view=att&zw	EXTRACTO_portafolio2019051722511920275089.pdf	64800a60_bde6f8cf_41000c7d_3b434183_144efec1
763	3244	0	https://mail.google.com/mail/?ui=2&ik=0df49229de&attid=0.1&th=16b00b6ca178ccc4&view=att&realattid=f_jw8f0fyz0&zw	stickers nasa.pdf	c095b473_5f5d61b6_1b42f0c4_4585c64d_a1634eb2
764	3255	0	https://mail.google.com/mail/?ui=2&ik=0df49229de&attid=0.1&th=16b0ac5390be7411&view=att&zw	meetup.ics	f5654f31_1e6461f4_c6cd86b5_13a159fc_cdcbc713
765	3274	0	https://mail.google.com/mail/?ui=2&ik=0df49229de&attid=0.0.1&th=16b3a9e86b3c5802&view=att&zw	CMT_PNETRIPE00_NTF_746609_2.png	1ccc1dbc_9bb69615_2ad1ffb0_39794637_024a34e5
766	3274	0	https://mail.google.com/mail/?ui=2&ik=0df49229de&attid=0.0.2&th=16b3a9e86b3c5802&view=att&zw	CMT_PNETRIPE00_NTF_746609_1.png	917f7c6b_ab3db39f_71d30616_3eabbdfc_51a4c189
767	3282	0	https://mail.google.com/mail/?ui=2&ik=0df49229de&attid=0.0.1&th=16b260540454c776&view=att&zw	CMT_PNETRIPE00_NTF_167208_2.png	1ccc1dbc_9bb69615_2ad1ffb0_39794637_024a34e5

Fig. 4: Browser of **OPIA**'s tables for Gmail app.

Figure 5 depicts the results of requesting the Shared Preferences for the Gmail app, showing some of the configuration settings and user information (*e.g.,* telephone numbers).

Another example of app that does not encrypt its data is Narrate, which stores notes in plain text and saves latitude and longitude where the app was used (see https://youtu.be/bh8nb3u_RAY). However, there are some applications that



Shared Preferences

MailFolder-[redacted]@gmail.com-^sq_ig_i_personal.xml

Type	Key	Value
boolean	notification-notify-every-message	false
string	notification-ringtone	content://settings/system/notification_sound
int	prefs-version-number	4
boolean	notifications-enabled	true
boolean	notification-vibrate	false

Gmail.xml

Type	Key	Value
int	[redacted]@gmail.com-promo_offer_sectioned_teaser_type	4
string	[redacted]@gmail.com-promo_tab_logging_id	CAESEwix06XakKXiAhUZeRkKHUUbChg
string	122758628-account-alias	122758628
int	welcome_tour_version	1
int	[redacted]@gmail.com-promo_tab_offer_section_label_type	2
string	WhatsApp-account-alias	WhatsApp
string	active-account	[redacted]@gmail.com
string	Messenger-account-alias	Messenger
boolean	[redacted]@gmail.com-hide_promo_section_header	false
long	[redacted]@gmail.com-promo_offer_last_fetch_timestamp	1518494776783
long	last_sync_time	1519960290427
string	Facebook-account-alias	Facebook
string	dummy_account-account-alias	dummy_account
string	+57310[redacted]-account-alias	+57310[redacted]

Fig. 5: Browser of **OPIA**'s shared preferences for Gmail app.

encrypt their backup, thus, **OPIA** can not collect the data, *e.g.,* K-9 Mail (https://youtu.be/h0GM-GvoRck).

The workflow finishes by executing SQLi attacks using previous records and the tables' name extracted by **OPIA**. To this, the user should select a previous record to inject and the accessibility service systematically executes three times the sequence of events changing user's text inputs with injection strings generated randomly but using the schema information (*i.e.,* attributes and tables). During the execution, after each event, the EE checks the current activity and, if there is a crash, the EE notifies the **OPIA** UI and restarts the application to continue with the execution. The rationale for including the injection feature is because SQLi attacks the integrity of an app dynamically. If developers do not check properly the inputs or use concatenation instead of prepared statements, the information stored in the databases will be lost. Also, the app could crash and the app availability will be impacted (see https://youtu.be/jkB5EUIwbWQ). Results of using **OPIA** with different apps are in the online appendix: http://tiny.cc/1k1y9y

IV. Related Work

When evolving and maintaining software, security testing is a critical activity to identify vulnerabilities. This process is accomplished by stressing programs concerning their security features. However, this task is highly expensive given the modern systems. Therefore, the research community has focused its efforts on automating security testing by engineering testing tools on mobile apps. For example, there are several proposals [5]–[8] for automatically detecting security flaws by analyzing code or intermediate representations statically.

Facebook Infer [5] is a well-known static code analysis tool which checks (in Android and Java code) for several vulnerabilities such as null pointer exceptions and resource leaks, among others. *Stowaway* [8] analyzes an Android app and determines the maximum set of permissions it may require, thus identifying the prevalence of over privilege issues.

Other tools are focused on testing, as in the case of *Monkey* [9], which is in fact the current industry standard tool that generates purely random inputs to stress-test applications.

It is worth noting that focusing only on source code can lead to unrealistic test cases, thus there are several approaches [9]–[12] that rely on combining static and dynamic techniques. For instance, *LetterBomb* [10] identifies and exploits vulnerabilities, relying on a combined path-sensitive symbolic execution-based static analysis and the use of instrumentation tests. Another tool is *ODBR* [13]; this is the closest tool to **OPIA** in terms of on-device data collection. ODBR supports app testers and developers in the process of On-Device Bug Reporting, being capable of collecting fine-grained user inputs and GUI information with the aim of creating a report conveying actionable and functional bugs.

Most of the aforementioned tools only are prone to false positives or rely on resource and time expensive techniques. Our tool differs from them since **OPIA** is focused on dynamically detecting and testing vulnerabilities, specifically those concerning SQLi and Information exposure.

V. Demo Remarks and Future Work

We presented **OPIA**, an on-device tool for detecting and testing security vulnerabilities in Android apps such as information exposure and lack of SQLi filters. **OPIA** can be used by developers to assure that new releases of an Android app do not suffer of those security issues before being uploaded to a market. Future work will be devoted to include other types of malicious attacks such as boundary values, activity injection, and data leaking/injection via intents. **OPIA** code is publicly available (http://tiny.cc/wn1y9y and http://tiny.cc/uo1y9y).

References

[1] Mobile Operating System Market Share Worldwide. http://gs.statcounter.com/os-market-share/mobile/worldwide. [Online; accessed 10-February-2019].

[2] Report: Smartphone owners are using 9 apps per day, 30 per month. https://techcrunch.com/2017/05/04/report-smartphone-owners-are-using-9-apps-per-day-30-per-month/. [Online; accessed 9-February-2019].

[3] Google Play Store: Number of apps 2018. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. [Online; accessed 9-February-2019].

[4] Global App Downloads Grew 15Ago — App Annie Blog. https://www.appannie.com/en/insights/market-data/global-app-downloads-grew-15-and-consumer-spend-20-in-q2-2018-versus-a-year-ago/. [Online; accessed 9-February-2019].

[5] Facebook Infer. https://fbinfer.com/. [Online; accessed 6-June-2019].

[6] J. Garcia H. Bagheri, A. Sadeghi and S. Malek. Covert: Compositional analysis of android inter-app permission leakage. *IEEE TSE*, 2015.

[7] N. Mirzaei R. Mahmood and S. Malek. Evodroid: Segmented evolutionary testing of android apps. In *ACM SIGSOFT FSE*, 2014.

[8] S. Hanna D. Song A. P. Felt, E. Chin and D. Wagner. Android permissions demystified. In *ACM CCS*, 2011.

[9] Monkey. https://developer.android.com/studio/test/monkey. [Online; accessed 6-June-2019].

[10] N. Ghorbani J. Garcia, M. Hammad and S. Malek. Automatic generation of inter-component communication exploits for android applications. In *ESEC/FSE*, 2017.

[11] N. Ghorbani H. Bagheri A. Sadeghi, R. Jabbarvand and S. Malek. A temporal permission analysis and enforcement framework for android. In *ICSE*, 2018.

[12] P. Liu C. Cao, N. Gao and J. Xiang. Towards analyzing the input validation vulnerabilities associated with android system services. In *ACSAC*, 2015.

[13] C. Bernal-Cardenas B. Otten D. Park K. Moran, R. Bonett and D. Poshyvanyk. On-device bug reporting for android applications. In *MOBILESoft*, 2017.