Information and Software Technology 105 (2019) 95–116

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Combining Automated GUI Exploration of Android apps with Capture and Replay through Machine Learning

Domenico Amalfitano, Vincenzo Riccio, Nicola Amatucci, Vincenzo De Simone, Anna Rita Fasolino*

Department of Electrical Engineering and Information Technologies, University of Naples Federico II, Via Claudio 21, Naples 80125, Italy

ARTICLE INFO

Keywords:
Android
Automated GUI Exploration
Dynamic analysis
Automated input generation
Machine Learning
Capture and Replay

ABSTRACT

Context: Automated GUI Exploration Techniques have been widely adopted in the context of mobile apps for supporting critical engineering tasks such as reverse engineering, testing, and network traffic signature generation. Although several techniques have been proposed in the literature, most of them fail to guarantee the exploration of relevant parts of the applications when GUIs require to be exercised with particular and complex input event sequences. We refer to these GUIs as Gate GUIs and to the sequences required to effectively exercise them as Unlocking GUI Input Event Sequences.

Objective: In this paper, we aim at proposing a GUI exploration approach that exploits the human involvement in the automated process to solve the limitations introduced by Gate GUIs, without requiring the preliminary configuration of the technique or the user involvement for the entire duration of the exploration process.

Method: We propose juGULAR, a Hybrid GUI Exploration Technique combining Automated GUI Exploration with Capture and Replay. Our approach is able to automatically detect Gate GUIs during the app exploration by exploiting a Machine Learning approach and to unlock them by leveraging input event sequences provided by the user. We implement juGULAR in a modular software architecture that targets the Android mobile platform. We evaluate the performance of juGULAR by an experiment involving 14 real Android apps.

Results: The experiment shows that the hybridization introduced by juGULAR allows to improve the exploration capabilities in terms of Covered Activities, Covered Lines of Code, and generated Network Traffic Bytes at a reasonable manual intervention cost. The experimental results also prove that juGULAR is able to outperform the state-of-the-practice tool Monkey.

Conclusion: We conclude that the combination of Automated GUI Exploration approaches with Capture and Replay techniques is promising to achieve a thorough app exploration. Machine Learning approaches aid to pragmatically integrate these two techniques.

1. Introduction

Over the last decade the number of users of mobile technology and smartphones has considerably increased. The total number of smartphone users worldwide will surpass 2.5 billion in 2019.[1] This is causing a constant demand for new software applications running on mobile devices, commonly referred as *apps*. At the same time, effective methods, techniques, and tools are being requested to support the developers in the software lifecycle activities [1].

Automation tools can facilitate mobile app lifecycle activities since they save humans from routine, time consuming and error prone manual tasks. In the last years, automated approaches for exploring the behavior of existing apps showed to be extremely useful in several contexts. To list just a few, automated exploration of mobile apps has been exploited in the field of software testing [2], reverse engineering [3], network traffic generation and analysis [4,5], security [6,7], accessibility testing [8], performance and energy consumption analysis [9]. Even some major cloud services providers like Amazon[2] and Google[3] are currently offering testing services to Android mobile apps developers, which exploit automated exploration techniques.

* Corresponding author.
E-mail address: annarita.fasolino@unina.it (A.R. Fasolino).

[1] https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/.
[2] https://aws.amazon.com/it/device-farm/.
[3] https://console.firebase.google.com/.

https://doi.org/10.1016/j.infsof.2018.08.007
Received 23 November 2017; Received in revised form 10 August 2018; Accepted 13 August 2018
Available online 14 August 2018
0950-5849/

Existing techniques for automated analysis of mobile apps behavior implement exploration strategies very similar to the ones used by robots for exploring unknown spaces [10] or by software agents in network traversing and analysis [11]. They usually exploit an iterative approach that is based on sending input events to a running app through its GUI until a termination criterion is satisfied [12]. We refer to these techniques as *Automated GUI Exploration Techniques* (AGETs). AGETs use information that an app exposes at runtime through its GUI to derive the set of input events to be fired; events may be chosen either randomly [13] or according to a more systematic exploration strategy of the GUI [14].

Although these techniques provide a viable approach for automatically exercising mobile apps, they suffer from the intrinsic limitation of not being able to replicate human-like interaction behaviors. In fact, some app features need to be exercised by exploiting app-specific knowledge that only human users can provide. As a consequence, these techniques often fail in exploring relevant parts of the application that can be reached only by firing complex sequences of input events on specific GUIs and by choosing specific input values [15,16].

In this paper, we name *Gate GUIs* the GUIs that need to be solicited by specific user input event sequences to allow the exploration of parts of the app that cannot be reached otherwise. Moreover, we refer to the action of providing the specific input event sequence needed to overcome a Gate GUI as to the activity of *unlocking* the Gate GUI.

There may be several types of Gate GUIs in real apps, such as *Login* Gate GUIs that a user needs to overcome to access reserved functionality offered by the app, *Settings* Gate GUIs that require a user to correctly configure the settings of services he intends to use through the app, or *QR code* Gate GUIs that request a user to scan a valid QR code through the device camera in order to access further features of the app. The challenges posed by Gate GUIs to the app automated exploration processes are well-known not only in the traditional field of software testing, but also in that of app network traffic signature generation [4] where dynamic analysis is used by large network vendors (e.g., Palo Alto Networks, Dell, HP, Sophos, MobileIron) to trigger app network activities.

Although most of the automated GUI exploration techniques proposed in the literature does not explicitly address the issues tied to Gate GUIs, some techniques offer solutions for unlocking Gate GUIs. Part of these solutions leverages on predefined input event generation rules embedded in the technique [6,17,18]. These approaches may not be able to exercise Gate GUIs that need app-specific knowledge. Other solutions require programming skills to understand the app-specific GUI structure and/or configure the AGET to properly manage each distinct Gate GUI [5,14,19–21]. These approaches are indeed labor intensive and may not extend to different applications. Finally, there are solutions that exploit manual user intervention. They suffer from the drawback of requiring an extensive human involvement throughout the entire exploration [13,22]. In fact, a user has to recognize the Gate GUI and intervene in the process to properly exercise it.

To address the limitations of existing automated GUI exploration techniques, in this paper we propose a novel hybrid approach, named *juGULAR (Gate gui UnLocking for AndRoid)*. Unlike other approaches, it does not require programming skills, mobile framework knowledge, and app comprehension for unlocking Gate GUIs. juGULAR automatically detects the occurrence of a Gate GUI and exploits human intervention to unlock it at runtime. However, the human intervention to unlock a specific Gate GUI is limited only to the first time that GUI is encountered. In fact, the human interaction to unlock a Gate GUI is captured by juGULAR and replayed when the same GUI is detected again during the exploration. The result is an hybrid exploration technique that combines automated GUI exploration with Capture and Replay [23].

A key aspect of this approach is the ability to automatically detect the occurrence of a Gate GUI. This can be considered as a GUI classification problem that we decided to solve with Machine Learning (ML). We adopted ML techniques to train classifiers to recognize given classes of Gate GUIs and exploited these classifiers to automatically detect Gate GUIs during the exploration.

We implemented the juGULAR hybrid exploration approach targeting Android apps. The approach was validated by an experiment involving 14 real Android apps. The experiment showed that combining Capture and Replay with automated exploration improves the effectiveness of the exploration. juGULAR covered more source code and Activities and generated more network traffic than the purely automated exploration, thanks to the automatic detection of two classes of Gate GUIs, i.e., Login and Network Settings. The additional time for the manual intervention required by juGULAR was reasonable, being on average lower than 3% of the entire exploration time for all the considered apps. Moreover, the experiment showed that juGULAR outperformed the state-of-the-practice in terms of exploration effectiveness.

The paper improves the literature on automated GUI exploration with the following contributions:

- a novel Hybrid GUI Exploration Technique that combines Capture and Replay with automated exploration, named *juGULAR*;
- a Machine Learning approach for the automatic detection of Gate GUIs;
- an experiment involving real Android apps showing the validity of the proposed hybrid technique.

The remainder of the paper is organized as follows. Section 2 reports related work, whereas Section 3 presents a motivating example. Section 4 illustrates the Machine Learning-based approach we exploit for obtaining the Gate GUI classifiers that we use to automatically detect Gate GUIs. Section 5 describes the juGULAR approach and how we implemented it in a software platform. Section 6 presents the experiment we performed and the results we obtained. Finally, Section 7 reports conclusions and future work.

2. Related work

2.1. Automated GUI Exploration Techniques for Android apps

A widely used automated GUI exploration tool for Android apps is Monkey,[4] that is part of the Android SDK. This tool adopts a quite simple exploration approach by sending pseudo-random events to the app under test. It is mainly used for a quick and repeatable robustness testing of Android apps, revealing crashes, unhandled exceptions and Application Not Responding (ANR) errors. This tool is regarded as the current state-of-practice for automated Android app testing [18,24], being the most widely used tool of this category in industrial settings [15,25].

In recent years several smarter automated GUI exploration techniques for Android apps have been proposed in the literature, especially in the context of online testing [26]. Each technique adopts its own strategy to define input event sequences to explore the app behavior.

Amalfitano et al. [12] analyzed a set of 13 testing techniques implementing AGETs and abstracted, in a general framework, the characteristics of the different GUI exploration approaches.

Choudhary et al. [2] presented a comparative study of the main existing tool-supported test input generation techniques for Android, including 7 tools exploiting an AGET. They concluded that Monkey outperforms the considered tools; however, they highlight that each tool shows perks that can be leveraged and combined in order to achieve significant overall improvements.

Zeng et al. [15] investigated the limitations of the Monkey tool in an industrial setting in which the apps can be far more complex than the

[4] developer.android.com/studio/test/monkey.html.

open-source ones considered by Choudhary et al. [2]. One of the solutions they suggested to enhance the capabilities of Monkey consists in manually constructing and performing sequences of events based on the user knowledge when the app requires the user to login, provide valid address information or scan a valid QR code.

In the following we describe the related work reporting their contribution and providing details about how and to what extent they dealt with exploration limitations related to the ones discussed in this paper. These contributions have been organized in three main groups on the basis of how they generate input event sequences that may be useful to interact with Gate GUIs.

2.2. AGETs that rely on predefined input event generation rules

A first group of AGETs leverages on predefined input event generation rules embedded in the technique such as textual input generation rules or rules to exercise specific GUI object types.

Karami et al. [6] proposed a software inspection framework for the identification of malicious apps. Like our approach, they exploit an AGET to send random sequences of GUI events to the app. It is able to generate significant input data for text fields, by applying rules predefined in the tool based on the detected text field type. The text length can also be tuned by the user before the exploration. However, their automated input data generation strategy fails to unlock Gate GUIs that need app-specific knowledge. Instead, we leverage input event sequences provided by the user to unlock Gate GUIs.

A^{3}E [27] implements a model-based automated GUI exploration strategy for Android apps. Like our approach, it automatically detects Activities related to special responsibilities, such as login, using a rule-based classification approach. Instead, we adopt a Machine Learning approach since it does not require a strong expert involvement to define rules for each specific Gate GUI class. Their approach exercises these Activities with input events predefined in the tool. The authors have also raised the need for complex interactions to reproduce certain apps functionality. They have not addressed this limitation but planned to do it through Record and Replay as future work. Our approach successfully combines automated GUI exploration with Capture and Replay to exercise GUIs that require particular and complex input event sequences.

CrashScope [17] is a tool that explores Android apps using systematic input generation and exploiting several strategies with the aim of triggering crashes. It detects the type of text expected by an app field and automatically generates text input data to exercise it by applying rules predefined in the tool. Unlike our approach, these rules do not aim at exercising Gate GUIs to unlock them. Instead, CrashScope fills textual fields with expected and unexpected data inputs to trigger crashes due to input data not correctly handled in the code.

Sapienz [18] is a multi-objective search-based automated Android app exploratory testing approach; it is based on a preliminary exploration performed by an AGET. Like our approach, Sapienz adopts strategies to explore parts of the apps that can be reached only by exercising specific GUIs with particular and complex input event sequences. Its dynamic exploration technique exploits information retrieved by a static analysis of the app resources to fill the textual fields. Its authors addressed the need for complex interactions by using predefined patterns, referred to as *motif genes*, that capture testers' experience and allow to reach higher coverage when combined with atomic events. Our approach exploits neither static analysis nor predefined patterns to unlock Gate GUIs. Instead, we capture the human knowledge necessary to unlock a Gate GUI by recording input event sequences provided by the user at runtime.

The approaches belonging to this group may suffer from limitations in exercising Gate GUIs that need app-specific knowledge or contextual information that is available only at runtime and thus results hardly predictable before the app exploration.

2.3. Configurable AGETs that exploit input event sequences predefined by the user

The solutions belonging to this group also rely on input event sequences defined before the app exploration. But they allow the user himself to define ad-hoc rules in order to enhance the exploration by app-specific knowledge.

Amalfitano et al. [14], propose a configurable tool that implements both random and systematic GUI exploration strategies and has been exploited also for model-based testing [28]. Their work points out that the ability of the tool to cover the app source code and to discover faults depends on several factors including the timing between consecutive input events and input values provided to the GUI input fields. To this aim, the user can provide an ad-hoc and app-specific configuration of the tool before the exploration. Unlike AndroidRipper, our approach does not require human effort to preliminarily configure the exploration technique. Instead, we detect Gate GUIs during the app exploration by exploiting a Machine Learning approach, without any previous app-specific knowledge. Moreover, we unlock the Gate GUIs by using the input event sequences provided by the user during the exploration.

Choi et al. [19] designed an automated technique, named SwiftHand, that uses active learning to reconstruct a model of the app during testing. The AGET implemented by SwiftHand uses the learned app model in order to select at each iteration the next input event to be executed; it is chosen among the input events enabled at the current state. This technique can detect `EditText` GUI objects and fill them with significant input strings defined by the user before the exploration with the aim to improve the exploration. Also our approach aims at improving the app exploration. However, we do not need to predefine app-specific input and we are not limited to textual inputs. Instead, we exploit the input events provided by the user during the exploration. Moreover, we do not aim at detecting any `EditText` GUI objects, but we automatically detect GUIs that require to be exercised with particular input event sequences by exploiting a Machine Learning approach.

PUMA [20] is a programmable framework that can be exploited to dynamically analyze several app properties, such as correctness, performance and security. It provides a generic AGET that can be extensively configured to guide the app exploration. It can generate a textual input when it is needed according to a policy coded by the user. Moreover, the user can also specify app-specific events to be applied when the exploration reaches a *codepoint*, i.e., a precise point of the app binary. Also our approach uses app-specific events when the exploration reaches a certain state, i.e., a Gate GUI. However, our approach requires neither human effort to code ad-hoc policies nor the knowledge of the app binary to preliminarily configure the exploration technique. Instead, we detect Gate GUIs during the app exploration by exploiting a Machine Learning approach without any previous app-specific knowledge. Moreover, we do not have to code before the exploration the app-specific events needed to exercise the detected Gate GUI. Instead, we leverage input event sequences provided by the user during the exploration.

Hu et al. [21] proposed Appdoctor, a testing tool able to perform a quick exploration, called *approximate execution*. Their exploration strategy is faster than real execution since it exercises an app by invoking directly event handlers. Our technique instead triggers real events because they represent better real user interactions. Appdoctor presents a component for the generation of proper input for text fields, number pickers, lists and seekbars in order to improve code coverage. It detects the type of text expected by a text field and applies input data drawn from dictionaries predefined in the tool. Alternatively, it can exploit rules defined by the user before the exploration to generate the input for a specific GUI object. We also aim at improving the code coverage reached by the app exploration. Unlike Appdoctor, our approach does not need app-specific inputs defined by the user before the

exploration. Instead, we exploit input event sequences provided by the manual user intervention during the exploration.

AndroGenerator [5] generates network traffic through automated exploration of Android apps. Its authors pointed out the limitations of their adopted technique since it is not able to provide right inputs to trigger the app code that generates network traffic. Therefore, their approach exploits also input event sequences defined by the user before the app exploration. Instead, in our approach input event sequences are provided during the exploration by the manual user intervention. Therefore, we do not need any previous app-specific knowledge.

The main limitation of these approaches is that they require programming skills to understand the app-specific GUI structure and/or configure the AGET to properly manage each distinct Gate GUI. These approaches are indeed human-intensive and may not extend to different applications.

2.4. AGETs exploiting manual user intervention

These AGETs combine automatically generated input event sequences with manual user intervention. In this way, they obtain human knowledge necessary to achieve a meaningful app exploration at runtime.

Dynodroid, proposed by Machiry et al. [13], is a system that generates relevant input sequences to Android apps. They clearly expressed the need to introduce human intelligence for exercising some app functionality that cannot be exercised otherwise by an AGET. Like us, their technique allows the user to generate arbitrary events directly on the app UI. To this aim, a Dynodroid user must first stop manually the automated exploration. Instead, our approach can automatically stop the automated event generation when a Gate GUI is detected.

NetworkProfiler [22], is a tool that implements a technique for inferring fingerprints of Android apps from the traffic they generate. It allows to perform complex input event sequences by fuzzing and replaying manual user traces captured before the exploration. We also exploit Capture and Replay but, unlike NetworkProfiler, we capture manual user traces during the exploration.

Another work that combines automated GUI exploration with captured user event sequences through machine learning has been proposed by Ermuth and Pradel [16] in the field of Web apps testing. This work defines a macro-based test generation approach for client-side Web applications. It aims at augmenting automated test generation techniques with complex sequences of events that represent realistic user interactions. A *macro event* abstracts a single logical step that users commonly perform to interact with real apps. This approach exploits machine learning techniques to cluster multiple similar event sequences belonging to different recorded usage traces and to infer from them single macro events. Instead, in our work we use machine learning to achieve a different goal, i.e., to train a classifier to detect Gate GUIs by providing it GUIs belonging to real Android apps. Both approaches require recorded usage traces to augment the automated GUI exploration. However, juGULAR captures usage traces only when a Gate GUI is detected for the first time during the app exploration. Instead, the macro-based technique requires adequate sets of traces to be preliminary recorded for each analyzed app.

These approaches are promising since they obtain human knowledge directly from manual intervention without needing programming skills or ad-hoc tool configurations, but they still suffer from some limitations. In fact, a Dynodroid user has to be constantly involved in the automated exploration in order to recognize a Gate GUI and intervene to properly exercise it. The other approaches belonging to this group, instead, require adequate sets of manual traces that exploit app-specific knowledge and need to be recorded before the app exploration.

3. Motivating example

In this section, we present a motivating example to show how the exploration of two real Android apps improves when their Gate GUIs are unlocked. In this work, we focused on two classes of Gate GUIs: Login and Network Settings.

Login Gate GUIs offer the login feature to registered users. These GUIs require the users to provide the credentials they used to register themselves to the app provider in order to be authenticated. The login feature usually allows to gain access to app functionality restricted to registered users only. Only through the insertion of valid and previously registered account credentials the exploration of the remaining parts of the application is allowed. Network Settings Gate GUIs are GUIs exposing Settings features to configure network parameters, such as: URLs, server address, port numbers, channels. This feature is necessary to configure the app access to remote resources.

We selected two publicly available Android mobile apps that expose Gate GUIs, e.g., Twitter[5] and Transistor[6]. Twitter renders the Login Gate GUI shown in Fig. 1(a), whereas Transistor exhibits the Network Settings Gate GUI reported in Fig. 1(b).

The Twitter Login Gate GUI requires valid and registered account credentials, without which the access to the app features that are available only to authenticated users is restricted. A purely automated exploration approach could be able to generate syntactically valid login and password, but it cannot be able to automatically generate text strings actually corresponding to a valid Twitter user account. This information should be necessarily defined by a human tester. The Transistor Network Settings Gate GUI requires the user to specify a valid audio stream URL to be reproduced. This scenario is very difficult to be resolved by a fully automated approach; even if it is able to automatically generate syntactically valid URLs, it may not be able to generate any correct URL actually corresponding to an audio stream.

First, we explored these apps using the current state-of-the-practice AGET, Monkey, in its default configuration. Both explorations lasted one hour and were performed on an Android Virtual Device (AVD).[7] We monitored the explorations of Monkey and noticed that it did not unlock the two Gate GUIs, being unable to provide correct credentials for the Twitter authentication, or a valid streaming URL to configure Transistor.

Then, we manually unlocked the Gate GUIs and ran Monkey again on both apps. Also in this case, Monkey was executed in its default configuration for the duration of an hour. To unlock the Gate GUIs, we interacted manually with the AVD where the apps were installed to provide proper input event sequences able to unlock them.

At the end of each exploration, we evaluated the covered Activities and inferred the Dynamic Activity Transition Graph (DATG) model [27]. It is a graph whose nodes represent the explored Activities and the edges render the transitions triggered at runtime. We enriched this model by adding weights on each edge that indicate the number of times the transition has been traversed.

We inferred the DATG model from the analysis of the system message log dumped by the Android `Logcat`[8] tool. This did not require any app code instrumentation. We parsed the system message log and extracted the sequence of the names of the Activities that were created during the app exploration. Activities having different names have been considered different nodes of the DATG. Each edge of the DATG links two consecutive Activities in the sequence. Android may not create an Activity every time it is encountered during the exploration but it could reuse a previously created instance of the same Activity[9]. Therefore, we had to enable the on-device `"Don't keep activities"` developer

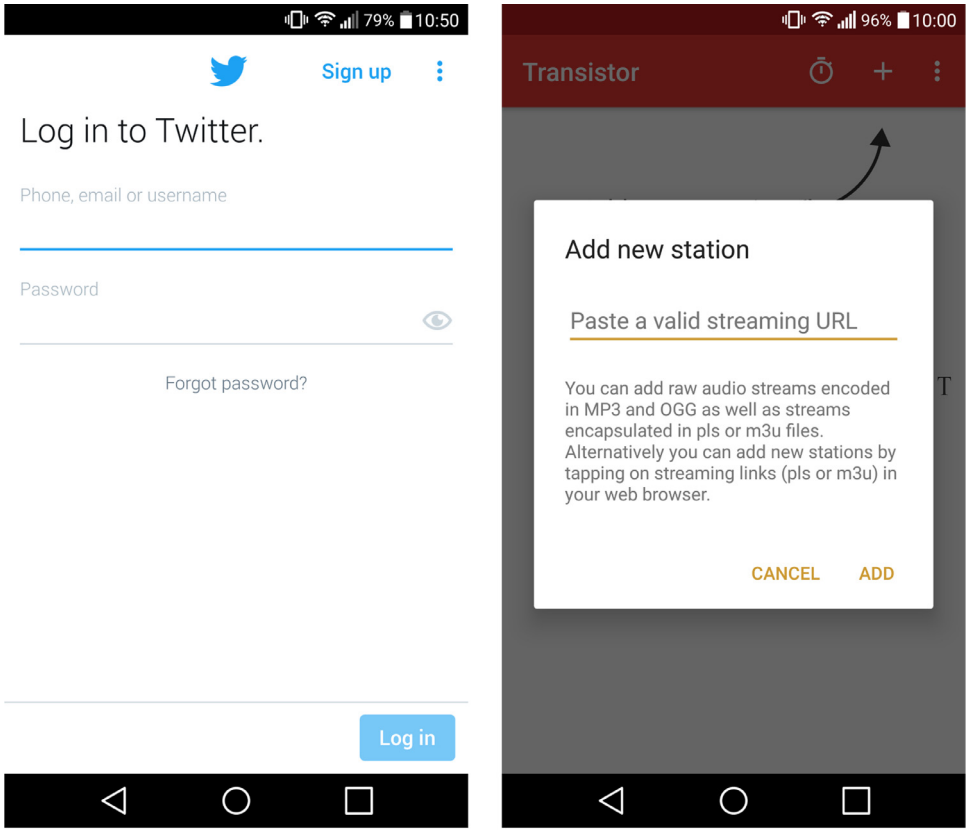
[5] Version 6.40.0 - https://play.google.com/store/apps/details?id=com.twitter.android.

[6] Version 2.2.0 - https://f-droid.org/repository/browse/?fdid=org.y20k.transistor.

[7] https://developer.android.com/studio/run/emulator.html.

[8] https://developer.android.com/studio/command-line/logcat.html.

[9] https://developer.android.com/reference/android/app/Activity.html.



(a) Twitter - Login Gate GUI

(b) Transistor - Network Settings Gate GUI

Fig. 1. Gate GUIs exhibited by the considered Android apps.

option that destroys every Activity as soon as the user leaves it. In this way, we were sure that each explored Activity was created and thus logged. Of course, this option may change the app behavior, introducing additional invocations of Android framework callback methods. However, a similar behavior may be observed also when the app is exposed to other common events, such as the orientation change event [29]. Since we enabled this option in all the Monkey runs, we believe that our comparison is fair and the usage of the "`Don't keep activities`" option is acceptable for our purposes.

Fig. 2 shows the DATGs inferred after the two explorations of Twitter.

As we can notice from the DATG shown in Fig. 2(a), Monkey was able to discover only 4 different app Activities in the first run, where the Login Gate GUI was not unlocked. After the Gate GUI was unlocked, Monkey explored up to 14 previously unreached Activities that are highlighted in white in Fig. 2(b). These new Activities expose functionality exclusively available to authenticated users, such as showing the user timeline, posting a new tweet or sending a private message to a another Twitter user. We evaluated also the network traffic produced by the app. To this aim, we counted the number of bytes transmitted over the network during the app explorations. To obtain this data, we used the `TCPdump`[10] command-line packet analyzer. The unlocking brought improvements also in network traffic generation. Without unlocking, Monkey generated around 1MB of network traffic that increased up to 380MB when valid login credentials were provided.

We were not able to measure the code coverage achieved during the exploration, since the Twitter app provided compiled and obfuscated code.

As for Transistor, we had access to the app source code. Therefore, besides the generated network traffic and the covered Activities, we were also able to measure the source code coverage. To this aim, we had to preliminarily instrument the app source code by exploiting the `JaCoCo`[11] code coverage library.

As Fig. 3(a) shows, the exploration of the Transistor app was limited to 2 Activities, when no valid URL was provided in the Main Activity. Instead, Monkey was able to reach a further Activity when a valid audio stream URL was provided, as shown in Fig. 3(b). This additional Activity offered features for controlling and reproducing the audio stream located at the URL provided to unlock the Gate GUI.

Without unlocking, Monkey was able to cover just 9.51% of app LOCs (Lines of Executable Code) and did not generate network traffic, whereas it executed the 58.25% of LOCs and transmitted more than 27MB over the network when a valid audio stream URL was provided.

We also explored the same two apps using three state-of-the-art automated GUI exploration tools, i.e., Sapienz,[12] AndroidRipper[13] and Dynodroid.[14] Each tool is representative of one the three AGET types reported in Section 2, respectively. We analyzed how they dealt with the considered Gate GUIs during the exploration.

We observed that Sapienz, implementing an AGET relying on predefined input generation rules, did not unlock autonomously the two Gate GUIs, producing unsatisfactory results in terms of covered Activities, LOCs, and generated network traffic.

As for AndroidRipper and Dynodroid, we were able to unlock the

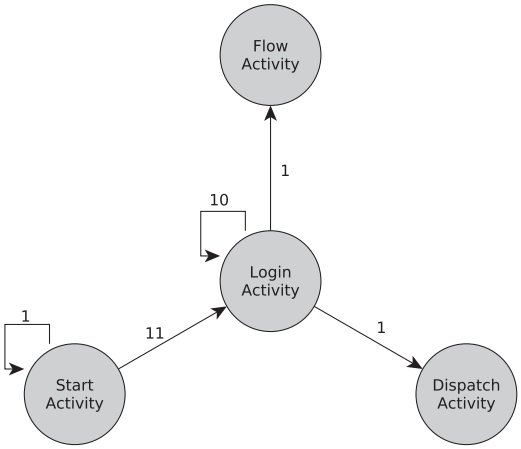
[10] http://www.tcpdump.org/.

[11] http://www.eclemma.org/jacoco/.

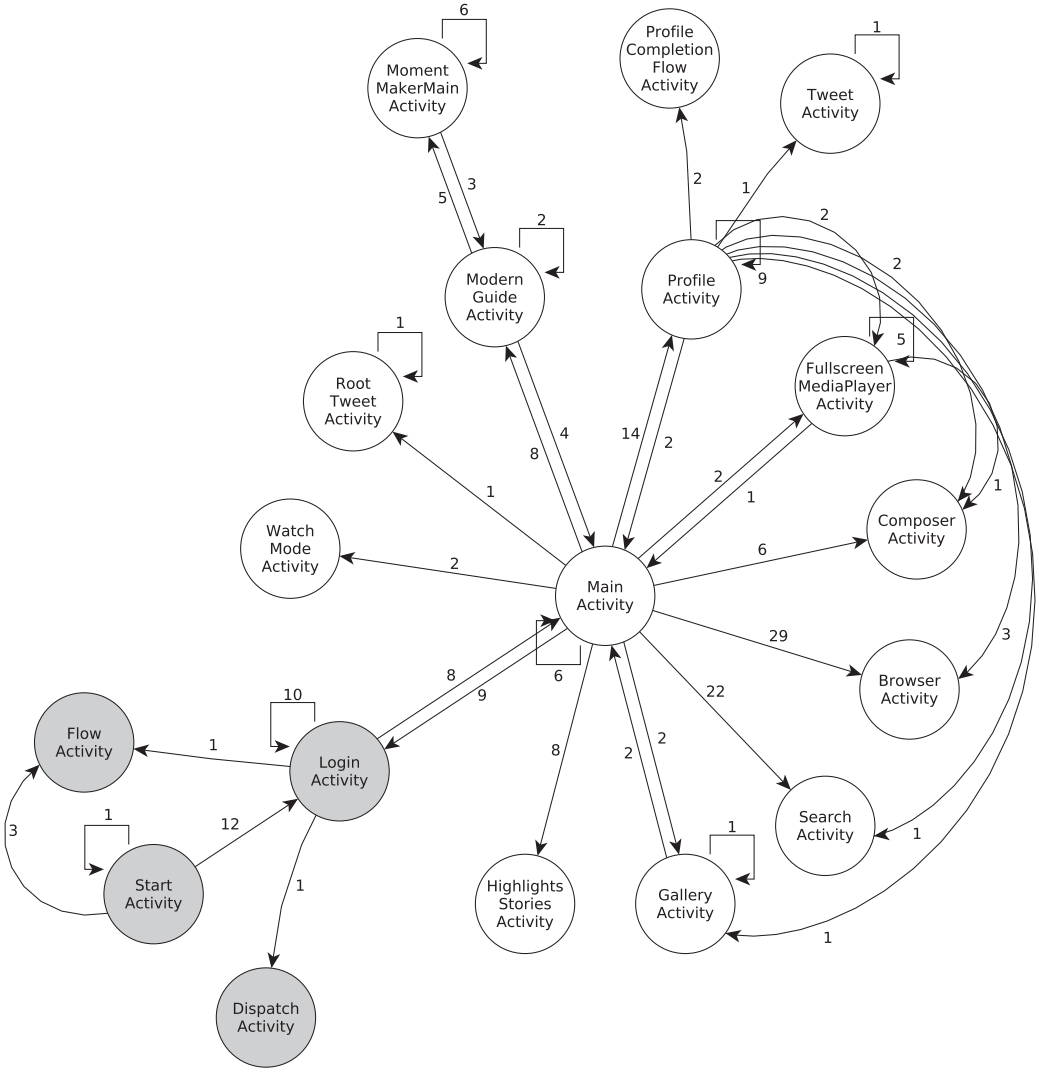
[12] https://github.com/Rhapsod/sapienz.

[13] https://github.com/reverse-unina/AndroidRipper.

[14] http://www.seas.upenn.edu/mhnaik/dynodroid.html.

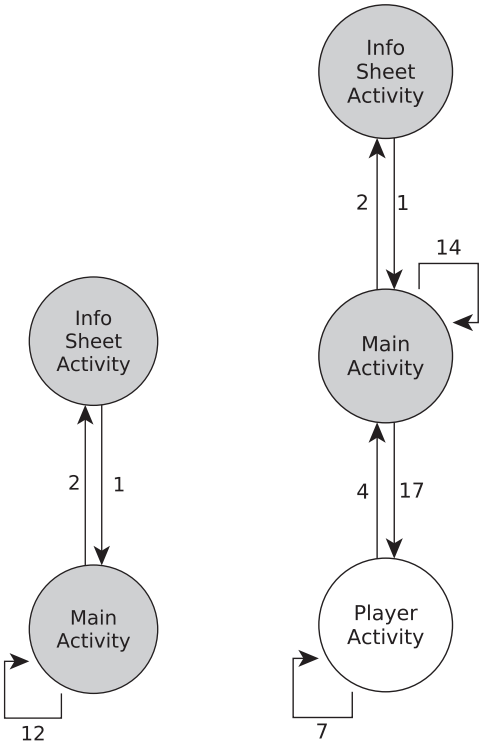


(a) DATG inferred without unlocking



(b) DATG inferred with unlocking

Fig. 2. Twitter app: the DATGs inferred by Monkey explorations without (a) and with (b) Gate GUI unlocking.



(a) DATG inferred without Gate GUI unlocking

(b) DATG inferred with Gate GUI unlocking

Fig. 3. Transistor app: the DATGs inferred by the Monkey explorations without (a) and with (b) Gate GUI unlocking.

Gate GUIs and to obtain exploration improvements similar to the ones achieved by Monkey. However, these improvements required a considerable manual effort in both cases.

AndroidRipper provides configuration APIs that enable the tool to fire user-defined event sequences on GUI objects belonging to given app GUIs. We used these APIs to unlock the Gate GUIs of the considered apps. However, the tool configuration required a considerable manual effort. We had to analyze the properties of the specific Gate GUIs and extract the ones needed to detect them at runtime. Moreover, we had to identify the GUI objects to be exercised and define the corresponding events.

Dynodroid implements instead an AGET that can exploit manual intervention at runtime. Therefore, a human was actively involved for the entire duration of the explorations to supervise the Dynodroid execution and to manually unlock the Gate GUIs. He had to monitor the Activities that were encountered by Dynodroid and to promptly stop the exploration each time he recognized a Gate GUI. After the human stopped Dynodroid, he manually unlocked the GUI by providing a valid input event sequence and then restarted the automatic exploration.

These experiences exposed the limitations of currently available techniques for automated app exploration and motivated us to investigate novel and more effective solutions.

4. A Machine Learning-based approach for detecting Gate GUIs

In this paper, we present juGULAR, an AGET that is able to autonomously detect Gate GUIs during an app exploration and to unlock them exploiting human intervention. To define juGULAR, we had to preliminarily find an approach for automatically recognizing whether the GUIs that are encountered during the app exploration belong to a Gate GUI class.

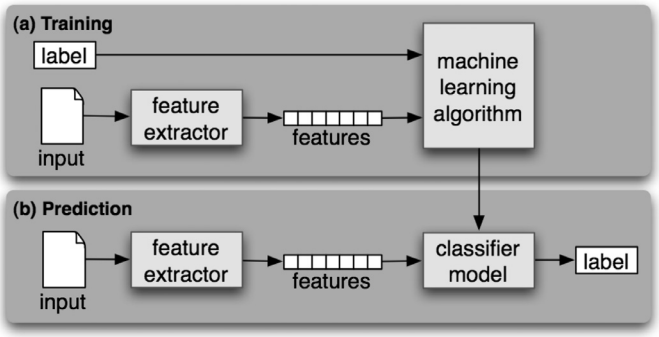


Fig. 4. Supervised classification framework. The upper part of the figure (a) represents the Training phase. The lower part of the figure (b) shows the Prediction phase.

To solve this kind of classification problems, rule-based or machine learning techniques are commonly employed in the literature [30].

Rule-based approaches exploit expert knowledge to make decisions. Rules are obtained from experts who encode their conditional beliefs into heuristics manually crafted for each specific class. These rules are usually elicited by error-prone and time-consuming processes that require a strong expertise about the considered domain [31]. Moreover, these rules work effectively only if all the possible situations under which decisions can be made are known ahead of time.

Instead, Machine Learning approaches aim to learn how to classify automatically through experience [32]. Therefore, they do not require expert involvement and are more effective at deriving general rules for classification problems, finding insights in data that may be underestimated by a human.

In our work, we decided to adopt a Machine Learning (ML) approach that trains a supervised classifier to determine the class a given GUI belongs to. Fig. 4 shows the general framework used for the supervised classification[15]. According to this framework, the supervised classification foresees two main phases: Training and Prediction. During the Training phase, a feature extractor is used to convert each input item instance to an abstract representation. This representation consists in a *Feature Vector* that captures the basic information about each input that should be used to classify it. In this phase, each input item is provided with a label that identifies the class the item belongs to. Pairs of feature vectors and labels are fed into a machine learning algorithm to generate a classifier model. The trained classifier can be used to predict the class of unseen input item instances. During the Prediction phase, the same feature extractor is used to convert unseen inputs to feature vectors. These feature vectors are then fed into the classifier model, which generates predicted labels.

To obtain the feature vector associated to a GUI, our approach relies on a component-based GUI description model that abstracts the GUI in terms of its component objects and their properties [29,33,34]. In particular, our GUI description leverages the XML GUI representation provided by UI Automator.[16] Among all the component object properties, we consider the ones that contain textual information, i.e., id, text, hint, and content description. Fig. 5 shows an example of a GUI and an excerpt of its XML description.

We assume that the descriptions of GUIs belonging to the same Gate GUI class are likely to share common textual information that we refer to as *keywords*. We select as features the presence or absence of such keywords in the values assumed by the considered properties.

In our approach, we did not want to arbitrarily predefine the keywords to be considered in the classification problem, but we wanted to

[15] http://www.nltk.org/book/ch06.html.

[16] https://developer.android.com/training/testing/ui-automator.html.

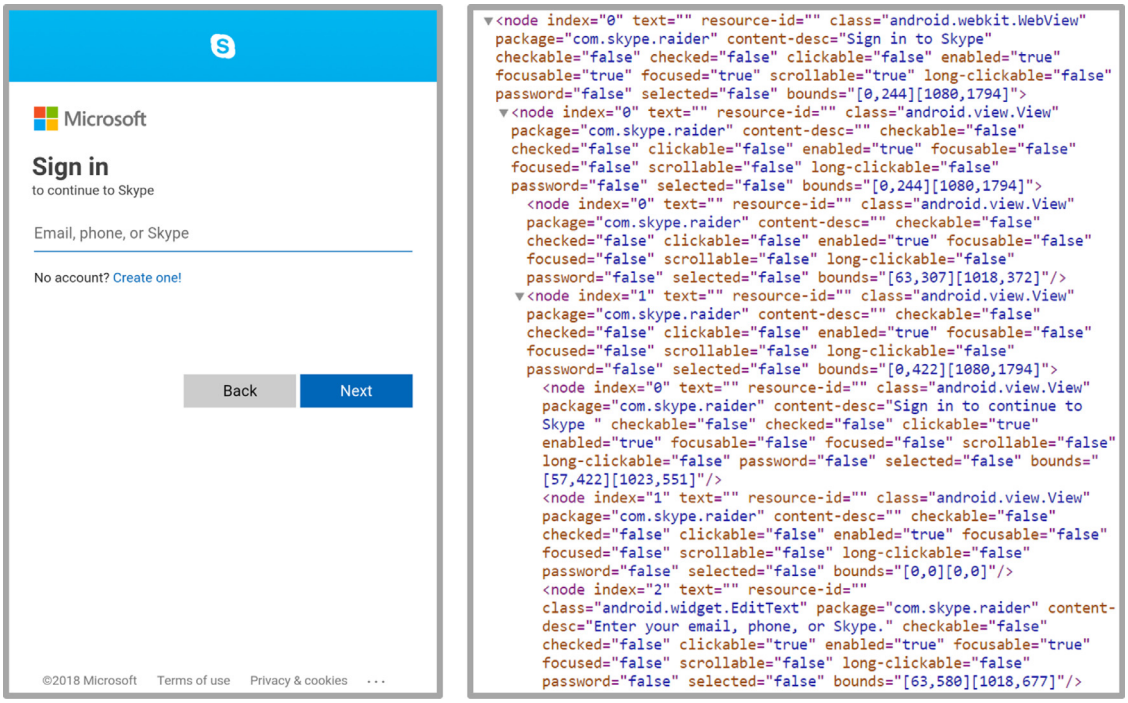
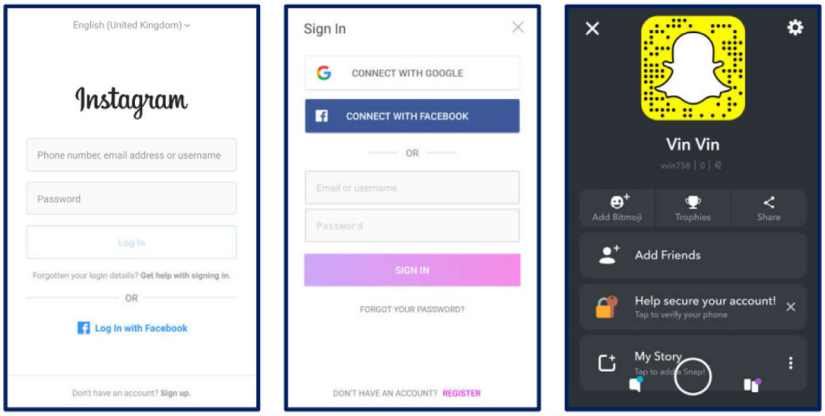


Fig. 5. A GUI (left) and an excerpt of its XML Description (right).



Keyword	(a)	(b)	(c)
forgot	✓	✓	✗
login	✓	✗	✗
password	✓	✓	✗
facebook	✓	✓	✗

Fig. 6. GUI textual information content. The table in the lower part of the figure reports the presence (✓) or absence (✗) of the "forgot", "login", "password", "Facebook" keywords in the description of the Instagram (a), PicsArt (b), and Snapchat (c) GUIs shown in the upper part of the figure.

empirically infer them for each considered Gate GUI class. For this purpose, we chose as keywords the most frequent terms among the GUIs belonging to the same Gate GUI class.

Fig. 6 presents our intuition about how a GUI can be characterized by the presence of a set of keywords. The Figure reports four keywords that should characterize Login Gate GUI descriptions and shows whether they are present in three GUI descriptions belonging to different Android apps. The GUI instances in Fig. 6(a) and in Fig. 6(b) are actually Login screens and their descriptions present at least 3 out of the 4 distinctive keywords. Instead, the GUI instance in Fig. 6(c) is not a Login screen and its description does not contain any of the considered keywords.

In the following, we describe the process we designed to select the keywords and for training the classifiers.

Our process is depicted in Fig. 7 and consists of three main activities: *Dataset Construction, Keyword Extraction* and *Classifier Training*. We implemented it using the features provided by the Natural Language Toolkit 3.2.5[17] platform.

This process is based on Information Retrieval approaches that solve the problem of classifying documents into a set of known categories,

[17] http://www.nltk.org/.

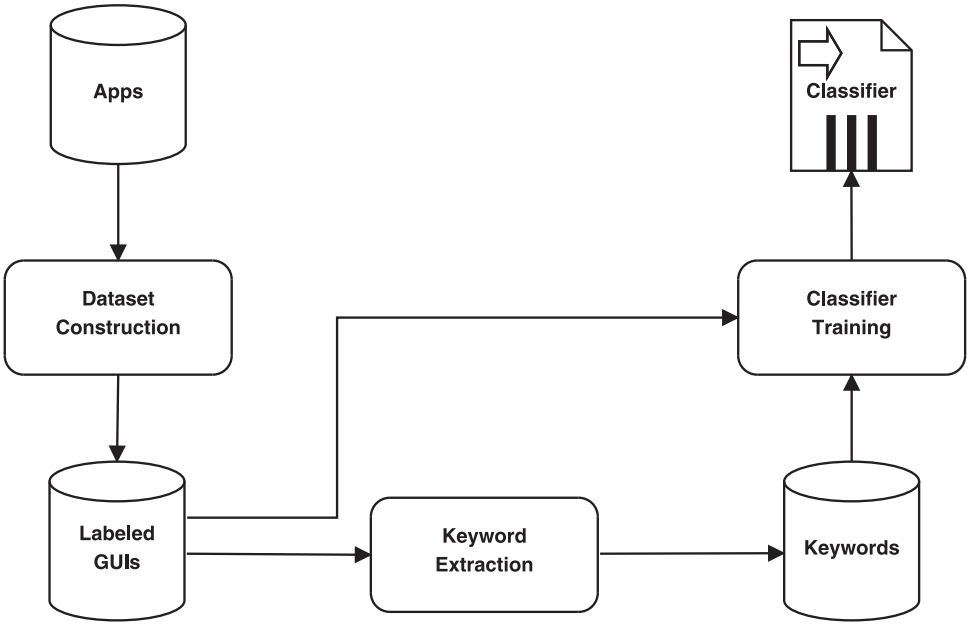


Fig. 7. The Machine Learning based process for selecting the keywords and training Gate GUI classifiers. It is composed of three activities: Dataset Construction, Keyword Extraction and Classifier training.

given a set of documents along with the classes they belong to Manning et al. [32]. More specifically, we adopted semistructured retrieval since we consider the XML representation of GUIs. This kind of approaches is well-known and is used to solve several classification problems, such as detection of spam pages, unwanted content, and sentiments, or email sorting [32] and app reviews' content classification [35]. To the best of our knowledge, we are the first to use these techniques to solve the mobile app GUI classification problem and to improve the app automated exploration.

The process is general and we exploited it for the two specific Gate GUI classes considered in this study: Login and Network Settings. The same process can be reused to build new classifiers for other GUI classes we want to automatically detect.

4.1. Dataset construction

Since there was no existing base of knowledge to be used for our purposes, we built our own dataset consisting of GUI descriptions belonging to real Android apps and labeled them according to our needs.

To this aim, we randomly picked 5000 real Android apps that were distributed by the official Google Play store[18] and recruited 100 Computer Engineering M.Sc. students to obtain a set of labeled descriptions of GUIs belonging to these apps. Each student was asked to manually explore 50 of the selected apps and label their GUI interfaces by assigning them one of three possible labels: *Login Gate GUI, Network Settings Gate GUI, Other* (i.e., a GUI that cannot be classified as one of the two considered Gate GUI classes). The students were provided with a *GUI Labeler* desktop application we developed to support the GUIs labeling task. The tool allows to select a label and assign it to the description of the GUI currently rendered on the device screen connected to the host PC. The tool was developed in Python and relied on the Android Debug Bridge (adb).[19] The tool produces as output an image file of the captured screen in PNG format and the UI Automator GUI hierarchy in XML format with the chosen label. Fig. 8 shows the interface of the GUI Labeler tool.

We provided each student with a device equipped with Android 6, that has been reset to the factory settings to ensure that each capture was executed in the same conditions. Moreover, the system language was set to *English* in order to avoid inconsistencies among the captures. Each student was asked to complete the assigned task within a month and to spend at least 15 min and not more than 30 min for exploring each app.

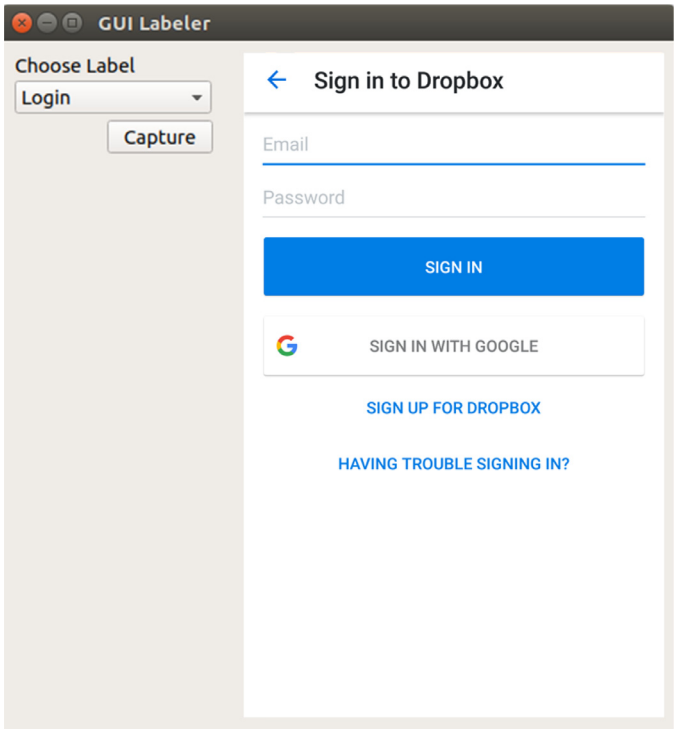


Fig. 8. An example of GUI Labeler tool interface. In this case the user captured a Dropbox GUI and assigned it the "Login" label.

Upon the completion of the GUI Labeling task by all the recruited students, three Ph.D. students and a Postdoctoral Researcher having knowledge of Android development, reviewed the labeled XML descriptions aided by the correspondent screen captures in order to

[18] https://play.google.com/store/apps.
[19] https://developer.android.com/studio/command-line/adb.html.

validate them.

At the end of this step, we selected 400 XML descriptions for each of the three considered labels and stored the resulting 1200 descriptions in a repository of labeled GUIs.

4.2. Keyword extraction

This activity allows to obtain a set of distinctive keywords for each Gate GUI class starting from GUI descriptions belonging to that class. Therefore, it has to be repeated for each considered Gate GUI class.

To this aim, for each Gate GUI class we partitioned the set of 400 XML descriptions labeled as belonging to the considered class in two subsets of 200 XML descriptions that we hereafter refer to as $G1$ and $G2$, respectively. The $G1$ subset was submitted to a keyword extraction process including linguistic preprocessing steps [32]. The $G2$ subset was instead exploited for training the classifier.

The keyword extraction process consisted of five steps:

1. *XML Nodes Extraction:* in this step, each XML GUI description belonging to $G1$ was filtered to obtain the XML nodes related to its Android `View` objects.[20] These objects represent the elements composing the GUI. We considered the values of the XML node attributes containing textual information, i.e *resource-id, hint, text, content-desc*. These values provided us a set of strings associated with each GUI description.
2. *Text Normalization:* in this step, special symbols and punctuation marks were removed from all the strings and each string was split into its constituent words. If a word was an identifier using the camel-case convention, it was split into the composing words (e.g., "processFile" is split into "process" and "File"). Finally, we converted each resulting word to lowercase.
3. *Stop words Removal:* in this step, we removed English *stop words* (like `and`, `a`, `to`, `do`, `of`) from the normalized strings. These stop words frequently appear in many GUI description and do not help much in differentiating one GUI description from another. We also removed terms specifically related to the Android SDK that are general and are not discriminating to identify Gate GUIs, e.g., `View`, `Toolbar`, `Button`.
4. *Stemming:* in this step, words were transformed to their root forms exploiting the Porter Stemming Algorithm.[21] For example, `localization`, `localized`, `localize`, and `locally` were all simplified to `local`.
5. *Term Frequency Evaluation:* in this step, the words obtained from the XML GUI descriptions belonging to $G1$ were gathered in a single set of terms, named T1. For each term of T1, the term frequency (*tf*) value was calculated producing a rank. The *tf* value of a term is equal to the number of occurrences of the term in the document or a corpus of documents [32]. In our study, we considered as corpus the set of terms $T1$. As an example, if a term *term1* occurs 40 times in $T1$, then its *tf* value will be 40. The terms having a *tf* greater or equal to a given threshold were selected and used to define a keyword set. We used more threshold values to define different keyword sets. We built seven sets of keywords, using threshold values varying from 5 to 35, with a step of 5. Each set of terms obtained at the end of this process provided a candidate set of *keywords*.

In the following, we present a simple example to illustrate the Keyword Extraction process. We consider to submit to the Keyword Extraction process the description of a GUI with a button having "`Click here if you have problems logging in!`" as textual label and "`login_troubleshooting_button`" as identifier.

In the XML Nodes Extraction step, the Button View object is identified and the values of its XML node attributes *resource-id, hint, text, content-desc* are returned. The output of this step is the following set of strings {Click, here, if, you, have, problems, logging, in!, loginTroubleshooting_button}.

In the Text Normalization step, the "!" and "_" special symbols are removed from the set. Moreover, the "loginTroubleshooting" string is split into "login" and "Troubleshooting" strings. Finally, all the words are converted to lowercase. As a result, the following set of strings is obtained {click, here, if, you, have, problems, logging, in, login, troubleshooting, button}.

In the Stop words Removal step, the English *stop words* "here", "if", "you" and "in" along with the generic Android SDK term "button" are removed from the set. The output of this step is the following set of strings {click, problems, logging, login, troubleshooting}.

In the Stemming step, the words in the latter set are transformed in their root form, obtaining the final set of terms {click, problem, log, login, troubleshoot}.

This set of terms will be gathered with the ones obtained from the other XML GUI descriptions belonging to the considered corpus. The resulting set of terms will be submitted to the Term Frequency Evaluation step, in which the *tf* value is calculated for each term and compared against the considered threshold. The terms having a *tf* value greater than the threshold will finally provide the set of keywords.

4.3. GUI classifier training

For each considered Gate GUI class, a distinct Binary classifier [36,37] had to be trained. Binary classification is a type of supervised learning in which a classifier is used to distinguish between a pair of classes. It is trained by using examples of objects belonging to both classes. We train a Login Gate GUI Binary classifier to predict whether a GUI belongs to the Login Gate GUI class or not. Moreover, we train a Network Settings Gate GUI Binary classifier to predict whether a GUI belongs to the Network Settings Gate GUI class or not. To train each classifier, We used two sets, the former consisting of 200 XML GUI descriptions labeled as belonging to each considered Gate GUI class (and different from the ones exploited in the Keyword Extraction step) we previously referred to as $G2$. The latter, that we hereafter refer to as $G3$, made of 200 XML GUI descriptions labeled as not belonging to the considered class.

Each labeled GUI description belonging to $G1 \cup G2$ was automatically processed by executing the XML Nodes Extraction, Text Normalization, Stop Words Removal, and Stemming steps. Then, it was associated to a *Feature Vector* of binary values in which the ith element represents the presence (1) or absence (0) of the ith keyword in the GUI description. This step was repeated seven times, each one considering a different candidate set of features, thus obtaining seven distinct classifiers.

We decided to use Naïve Bayesian (NB) classifiers as Binary classifiers. An NB classifier is a statistical classifier based on the Bayes'theorem that implements a simple, computationally efficient classification algorithm. NB classifiers are widely employed in several areas, including text classification, with comparable results to decision trees and artificial neural networks [38].

We trained and validated each classifier using a 10-*fold cross-validation process. In this process, an original dataset is randomly divided into 10 equal-sized subsamples that are exploited for 10 validation steps. At each validation step, a single subsample is used for validation and the other nine subsamples are utilized for training.

Finally, we compared the accuracies obtained by the seven classifiers and chose the one achieving the best F-measure value [32]. For both Gate GUI classes, the selected classifier was obtained using the set of keywords corresponding to the *tf* threshold of 20.

Table 1 reports for both the considered Gate GUI classes the average values of precision, recall and F-measure the selected classifiers obtained over the 10 validation steps.

[20] https://developer.android.com/reference/android/view/View.html.
[21] https://tartarus.org/martin/PorterStemmer/.

Table 1
Performance in terms of average values of precision, recall and F-measure of the Login Gate GUI and Network Settings Gate GUI classifiers.

	Login Gate GUI	Network Settings Gate GUI
Precision	0.814	0.751
Recall	0.807	0.900
F-measure	0.807	0.813

5. The proposed Hybrid GUI Exploration Technique

In this Section, we present our hybrid GUI exploration technique named juGULAR that targets Android mobile apps. Since Android apps are event-based software systems [39,40], juGULAR explores the analyzed apps by automatically sending events to them. Our technique explores mobile apps regardless of whether they run completely on the Android device, or they belong to more complex and distributed systems. Indeed, juGULAR aims to explore the client-side Android app of such systems and can interact with their remote side by events that are fired on the UI of the Android client app.

Unlike other event-based exploration techniques reported in the literature [12,34], juGULAR implements a novel approach that pragmatically combines fully automated GUI exploration with Capture and Replay, in order to enhance the app exploration and to minimize the human involvement. It is able to automatically detect Gate GUIs during the app exploration by exploiting classifiers that can be obtained through the Machine Learning approach introduced in Section 4. Moreover, it can unlock Gate GUIs by leveraging input event sequences provided by the user through a Capture and Replay technique.

The exploration implemented by juGULAR extends the automated GUI exploration algorithm presented in [12]. The workflow of juGULAR is described by the UML Activity diagram shown in Fig. 9. The Activity states describing the steps of the original algorithm are reported in white, whereas the ones introduced by our approach are in gray.

Each app exploration is started by the *App Launch* step that installs and launches the app on an Android device. In the *Current GUI Description* step, a representation of the GUI state currently exposed by the app is inferred. It includes the (*attribute, value*) pairs assumed by GUI components at runtime. The GUI description is analyzed in the *Gate GUI Detection* step to evaluate whether it is an instance of a Gate GUI.

If the current GUI is not a Gate GUI, the *Input Event Sequence Planning* and *Input Event Sequence Execution* steps are executed. In these steps, an event is chosen among all the ones triggerable on the current GUI and then it is executed. juGULAR considers as triggerable the predefined sets of events that can be fired on GUI objects having the properties `clickable`, `enabled`, and `visible` set to `true` in the current GUI description. The value of the `type` attribute of the GUI object determines the set of possible events that can be triggered on it. As an example, events like `click` and `longclick` can be fired on `Button` and `ImageView` objects, whereas `selectItem` and `scroll` events can be sent to `ListView` objects.

Otherwise, if juGULAR detects a Gate GUI, the *Gate GUI Unlocking* step is executed. In this step, either an input event sequence will be captured for unlocking a Gate GUI or a recorded input event sequence will be replayed.

The *Termination Condition Evaluation* step evaluates whether the termination condition is met and the exploration can be stopped.

The UML Statechart diagram in Fig. 10 provides an overview of how juGULAR combines automated app exploration with Capture and Replay.

In the *App Exploring* state, juGULAR iteratively fires input event sequences to the subject app according to a given input event generation strategy until it detects a Gate GUI, or a predefined termination condition is met. When juGULAR detects a Gate GUI, it evaluates whether it has been previously encountered or not. To this aim, it compares the current GUI description with the ones of the Gate GUIs already encountered during the app exploration. Two GUI descriptions are considered as equivalent if they include the same set of objects and the same values of the objects' attributes [12].

If juGULAR detects a Gate GUI for the first time, it stores its GUI description and transits to the *Unlocking Input Event Sequence Capturing* state, where it captures an unlocking input event sequence that is manually provided by the user.

When the capturing ends, juGULAR returns to the *App Exploring* state. In this state, if juGULAR detects a Gate GUI that was previously encountered, it either will continue the automated app exploration, or will transit into the *Unlocking Input Event Sequence Replaying* state in which the corresponding input event sequence is replayed. The choice of the next state will depend on the value of a `ReplayCondition` random boolean variable that assumes the `true` value with a predefined probability p_{true}. The usage of this variable prevents the exploration process from being biased by the user's choice for unlocking a given Gate GUI.

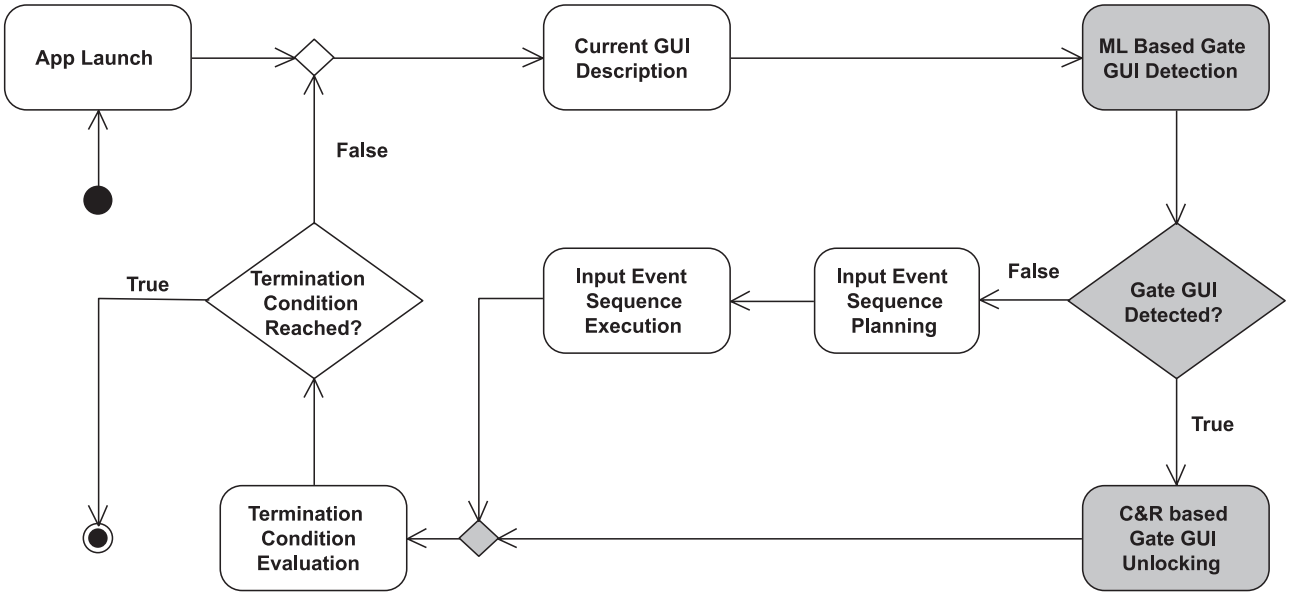


Fig. 9. UML Activity Diagram describing the juGULAR workflow.

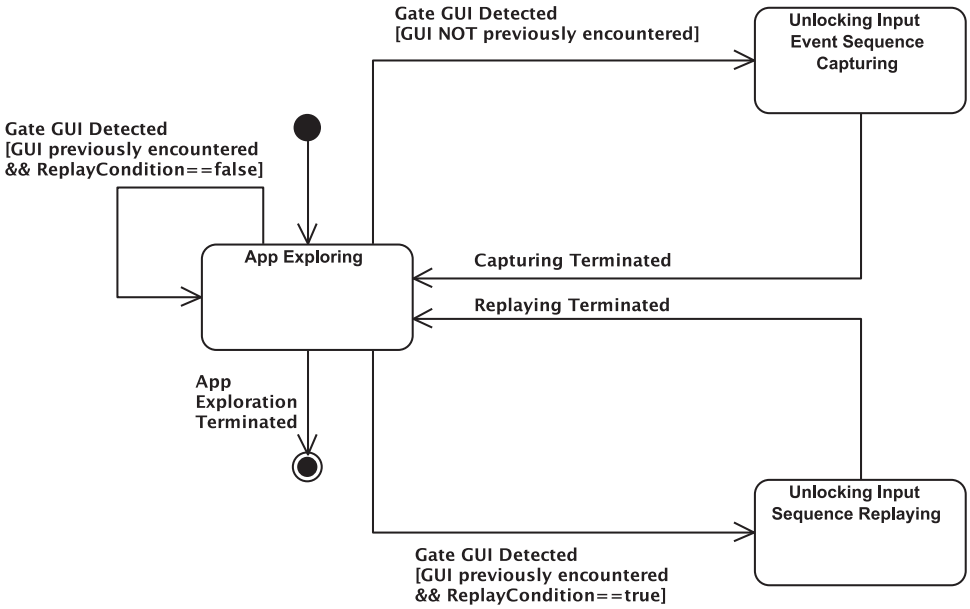


Fig. 10. UML Statechart Diagram describing how juGULAR combines automated app exploration and C&R techniques.

At the end of the replaying, juGULAR returns to the *App Exploring* state.

It is worth pointing out that, in the *Replay* step, non-determinisms of the app may cause the app to expose a GUI that is different from the one exercised in the Capture, or to behave differently from the recorded behavior. In these cases, the recorded event sequence replay does not guarantee that the Gate GUI will be correctly unlocked. This is a known weakness of Capture and Replay approaches [16] and poses a limitation to juGULAR. In fact, juGULAR returns in the *App Exploring* state after each Replay, regardless of whether the recorded event sequence has successfully unlocked a Gate GUI. In case of app non-deterministic behavior, there is the risk that juGULAR indefinitely encounters the same Gate GUI and tries to unlock it with the same recorded event sequence. The `ReplayCondition` random boolean variable mitigates this risk since it allows juGULAR to fire also event sequences different from the recorded Unlocking Input Event Sequence when it re-encounters a Gate GUI.

5.1. The juGULAR platform

We implemented juGULAR in a software platform which targets Android mobile apps. An overview of the platform architecture is reported in Fig. 11.

The core of this architecture is the juGULAR component that embeds four inner components, namely *App Explorer, Gate GUI Detector, Gate GUI Unlocker*, and *Bridge*.

The *App Explorer* implements the app exploration logic. The *Gate GUI Detector* has the responsibility to automatically infer whether a GUI belongs to a Gate GUI class. The *Gate GUI Unlocker* offers the feature to unlock a Gate GUI. The *Bridge* allows the juGULAR components to interact both with a *User Terminal* and with an *Android Device* where the app being explored is installed and executed.

The *User Terminal* allows a user to launch juGULAR and to receive notifications when a Gate GUI is detected for the first time and should be unlocked. Thanks to this feature, the user does not have to monitor the app exploration waiting for a Gate GUI detection, but juGULAR notifies him when his intervention is needed for unlocking a Gate GUI. When the user has accomplished the capture activity, he resumes the app exploration via the User Terminal.

The juGULAR components and the User Terminal are deployed on a host PC running either Windows or Linux operating system and equipped with the Android SDK.[22] The PC must be connected through the Android Debug Bridge (adb)[23] with an Android virtual device (avd)[24] hosted on the host machine, or a real Android device connected to the host machine via a USB connection.

Our platform can be used to explore an Android app for reaching different goals. Depending on the specific goal, additional tools can be exploited for capturing relevant information about the performed exploration. As an example, in the study that we present in Section 6, we aimed at evaluating the app coverage and the network traffic generated by the exploration. To reach this goal, we instrumented the app source code using the `jaCoCo` library[25] and ran tcpdump[26] on the host machine to get the network packets capture file in `pcap` format.

Additional implementation details about the platform components are reported in the following.

5.1.1. App explorer component

This component can be configured to explore an app using different exploration strategies, such as the Random or Active Learning ones [12]. The strategy determines the next event to be triggered on the app. The *App Explorer* uses the *Trigger Event* and *Get GUI Description* APIs offered by the *Bridge* component to send events to the app and to retrieve the description of the current GUI rendered by the device, respectively.

Moreover, it uses the *Detect* API offered by the *Gate GUI Detector* to assess whether the current GUI can be classified as a Gate GUI. When a Gate GUI is detected, the App Explorer uses the *Unlock* API provided by *Gate GUI Unlocker* component for unlocking it.

5.1.2. Gate GUI Detector component

This component offers the *Detect* API that is exploited by the *AppExplorer* to assess whether a GUI can be classified as a Gate GUI. Its architecture is represented in Fig. 12 by a UML Component diagram.

The *Gate GUI Detector* comprises three components, i.e., the *Gate GUI Detector Manager*, the *Login Gate GUI Identifier* and the *Network*

[22] Available for free download at https://developer.android.com/studio/index.html.

[23] https://developer.android.com/studio/command-line/adb.html.

[24] https://developer.android.com/studio/run/managing-avds.html.

[25] http://www.eclemma.org/jacoco/.

[26] http://www.tcpdump.org/tcpdump_man.html.

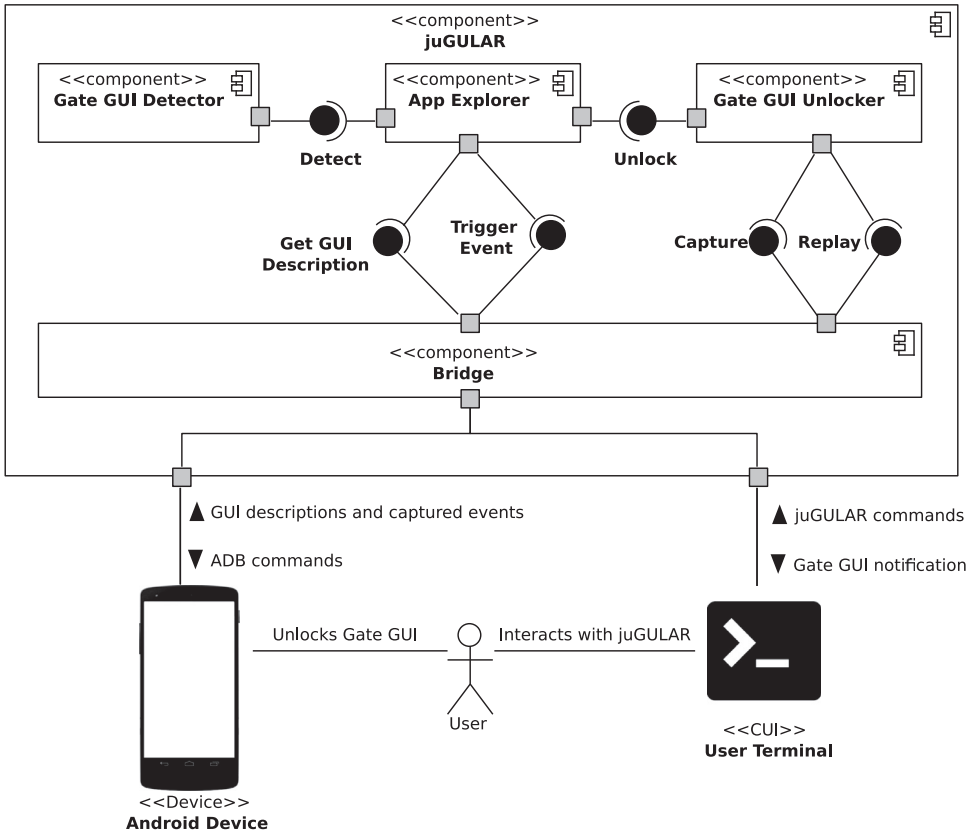


Fig. 11. UML Component diagram describing the juGULAR platform architecture.

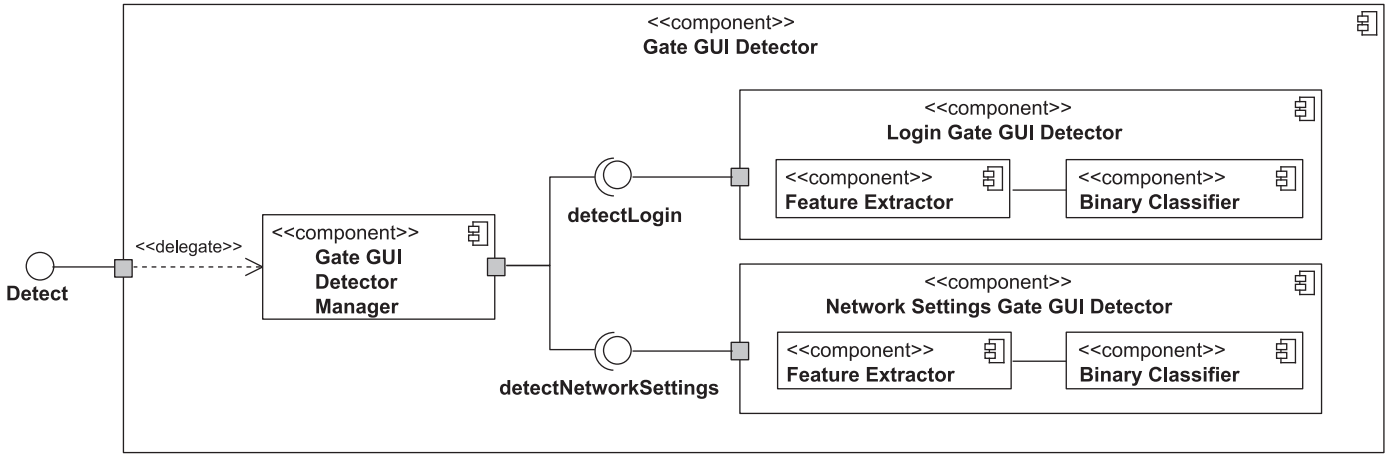


Fig. 12. UML Component diagram describing the Gate GUI Detector architecture.

Settings Gate GUI Identifier. Each identifier is able to detect a different Gate GUI class, i.e., Login and Network Settings.

The *Gate GUI Detector Manager* takes as input a GUI description in XML format, forwards it to the inner *Gate GUI Identifier*s and gathers their outputs. According to these outputs, the *Gate GUI Detector Manager* returns a boolean value indicating whether a Login Gate GUI or a Network Settings Gate GUI have been detected.

The *Login Gate GUI Identifier* and the *Network Settings Gate GUI Identifier exploit the Binary classifiers* and the set of keywords obtained by the approach proposed in Section 4. Each of these components is in turn composed by a *Feature Extractor* and a *Classifier* component. The *Feature Extractor* takes as input the GUI description and represents it as a feature vector. To this aim, it elaborates the GUI description through the preprocessing steps introduced in Section 4, i.e., XML Nodes Extraction, Text Normalization, Stop Words Removal, and Stemming. Then, it associates the GUI description to a vector of binary values, where the *i*th element represents the presence (1) or absence (0) of the *i*th keyword. Finally, it forwards the feature vector to the corresponding *Classifier*.

The *Gate GUI Detector* is a modular ensemble of Binary classifiers and thus it can be easily extended by introducing additional Gate GUI identifiers for other classes of Gate GUIs.

5.1.3. Gate GUI Unlocker component

The *Gate GUI Unlocker* component provides the *Unlock* API that requires as input a GUI description in XML format. It stores in a local repository the descriptions of the Gate GUIs encountered during the app exploration. Moreover, it stores for each GUI description the sequence

of user events that was recorded to unlock the corresponding GUI.

When the *Unlock* API is invoked, the component checks whether the input description matches with one of the descriptions stored in the repository.

If the *Unlocker* does not find any matching GUI description, it requires to capture a sequence of user events using the *Capture* API provided by the *Bridge*. Upon completion of the capturing, the input GUI description along with the captured sequence of user events will be stored in the repository.

Otherwise, the *Unlocker* either will invoke the *Replay* API to replay the related sequence of user events, or will return the control to the *App Explorer*, on the basis of the value assumed by the random boolean variable `ReplayCondition`.

The *Gate GUI Unlocker* can be configured by setting the p_{true} value the `ReplayCondition` relies on. The default value of p_{true} is 0.9.

5.1.4. Bridge component

The *Bridge* component allows juGULAR to interact with the device where the app runs and with the User Command Prompt. It provides the *Trigger Event* and *Get GUI Description* APIs that are used by the *App Explorer* to send events to the app and to retrieve the XML description of the current GUI rendered by the device, respectively. These APIs are realized exploiting the UIAutomator framework.[27]

The Bridge provides also the *Capture* and *Replay* APIs that are used by the *Gate GUI Unlocker* component. The *Capture* API sends a notification on the *User Terminal* to the user about the occurrence of a Gate GUI and records the event sequence the user performs to unlock it. The *Replay* API is used for replaying the recorded user event sequence that unlocks a specific Gate GUI.

Android encodes each user input event (e.g., Tap, Long Tap, Scroll, Hardware Button Press) into a large group of kernel-level events. We will refer to the former ones as *high-level user events* to distinguish them from kernel-level events.

The *Capture* and *Replay* APIs have been developed exploiting the *getevent* and *sendevent* tools, respectively.[28] These tools, shipped within the Android SDK, are able to capture and replay the kernel-level event stream produced by the user interaction.

The *getevent* tool provides a live dump of kernel-level input events. The captured stream contains information about the input events, such as their timestamps, input device names, event types, and screen coordinates. The *sendevent* tool allows developers to send kernel-level events to the device.

As already reported in other works [41], the *sendevent* command does not allow to set the timing between consecutive sent events. Therefore, we had to develop an ad-hoc solution to replay the kernel-level events with a proper timing. This solution is intended both to avoid a too quick replay of high-level events and to faithfully replay them. To this aim, we implemented in the Bridge a pipeline that post-processes the input event stream captured by *getevent* and transforms it into an output stream suitable to be replayed by *sendevent*. The pipeline groups meaningful chunks of kernel-level events representing high-level events, inserts delays between them, and translates the stream in the format supported by *sendevent*.

The stream chunks are isolated using a set of clustering rules that are based on timestamps and event encoding patterns that Android exploits for transforming high-level user events into kernel-level ones. We inferred these patterns by analyzing streams of kernel-level events produced by triggering user events. The pipeline uses these rules to decompose each high-level event into a sequence of lower level events, e.g., Press, Release, Move. For clarity, we refer to these as *low-level events*. Each low-level event is in turn composed by a sequence of kernel-level events having the same timestamp and that is ended by a `SYN` code made of a sequence of zero values.

Fig. 13 shows an example of a kernel-level event stream captured by *getevent* and how the clustering rules abstract from it four types of high-level events, i.e., Tap, LongTap, Scroll, and Key Home Press. The columns in the Figure report the timestamp between square brackets, the input device id followed by a colon, and an hexadecimal string representing the encoding of the kernel-level events. The leftmost brace groups consecutive kernel-level events into low-level events, whereas the rightmost brace groups low-level events into high-level user events. The figure illustrates that a Tap event is composed by the sequence of Press and Release low-level events between which there is a delay less than 500 ms. A Long Tap is composed by a sequence of Press and Release between which there is a delay greater or equal to 500 ms. A Swipe is made by a sequence of a Press and a Release interspersed with one or more Move low-level events. A Key Home Press event is composed by a sequence of a Key Home Down and a Key Home Up low-level events.

Finally, the pipeline processes the captured stream composed by kernel-level events and produces an `unlocking sequence description` file. This file will be provided to the *Gate GUI Unlocker* component and it will be interpreted by the Bridge *Replay* API to unlock the corresponding Gate GUI by sending the kernel-level events with the proper timing. Fig. 14 reports the unlocking sequence description file corresponding to the stream shown in Fig. 13. An unlocking sequence description file includes sequences of kernel-level events supported by *sendevent* along with specific commands, i.e., `#Start`, `#End`, `#SendEvents`, and `#Sleep`. The kernel-level events in this file are obtained from the corresponding ones in the captured event stream by removing the timestamps and the colons and by translating the hexadecimal values in decimal format. The `#Sleep` command is used to introduce delays between events. The Bridge adds delays of 1500 ms between kernel-level event chunks, each one representing a single high-level event. This is intended to mitigate the risk of failures in the replay step due to a time not long enough to complete a requested UI task, e.g., UI updating, Web resource fetching. We found that a delay of 1500 ms is long enough to complete mobile UI tasks in our benchmark apps. Moreover, a delay of 600 ms is introduced between the Press and the Release of Long Tap events to not mistake them for simple Tap events.

The event stream stored in the unlocking sequence description file allows the *Replay* API to execute the events on the same screen coordinates of the corresponding event stream acquired through the *Capture* API. We chose to stick to the same coordinates since the juGULAR architecture has the limitation that the same device is used both for capturing and replaying user event sequences. We are aware that if different devices want to be used in the Capture and Replay steps, our solution should be enhanced.

6. Experiment

In this section, we describe the study we conducted to evaluate the performance of juGULAR. The exploration technique implemented by juGULAR can be exploited in different contexts and for reaching different objectives. Thus, in this study, we considered two usage scenarios of juGULAR: software testing and mobile app network traffic signatures generation.

Our goal was to understand how the hybridization proposed by juGULAR does impact the ability of fully automated GUI exploration techniques in analyzing apps and at what cost. Moreover, we were interested in evaluating how juGULAR compares with other state-of-the-practice AGETs. More precisely, the study aimed at answering the following three research questions:

RQ$_1$: How does the hybridization introduced by juGULAR affect the effectiveness of an automated exploration technique?
RQ$_2$: How does the manual intervention required by juGULAR affect the costs of the hybrid exploration approach?

[27] https://developer.android.com/training/testing/ui-automator.html.
[28] https://source.android.com/devices/input/getevent.

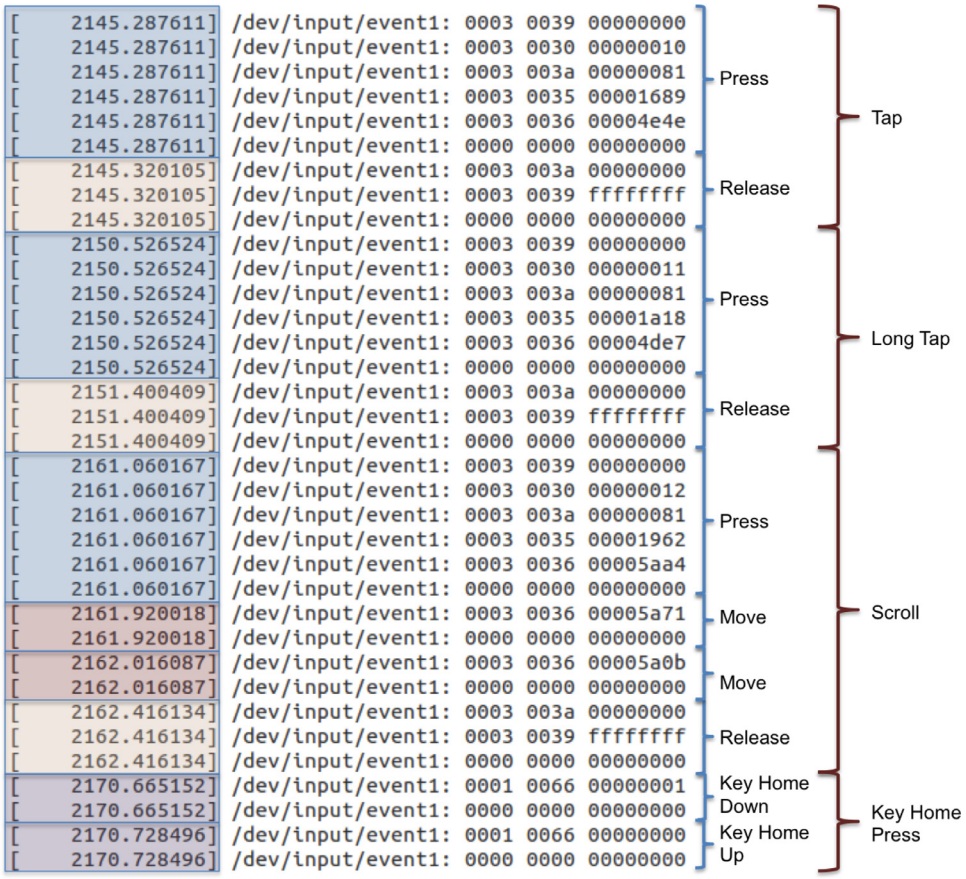


Fig. 13. A sequence of kernel-level events captured by getevent. Each line reports a kernel-level event characterized by its timestamp, input device id, and the corresponding hexadecimal code. The leftmost brace groups consecutive kernel-level events into low-level events, whereas the rightmost brace groups low-level events into high-level user events.

RQ$_3$: How does the exploration effectiveness of juGULAR compare to the effectiveness of the AGET implemented by the state-of-the-practice Monkey tool?

6.1. Objects selection

The presence of Gate GUIs in the object Android apps was a requirement for carrying out this study. Therefore, we needed to select apps that exposed at least one GUI belonging to the considered Gate GUI classes. To this aim, we chose a subset of apps from the official Google Play store whose GUIs belong to the dataset we built in the process described in Section 4 and that were not used in the Keyword Extraction and Classifier Training activities. In addition, since we wanted to evaluate also the code coverage reached due to the app exploration, we required that the selected apps belonged also to F-Droid.[29] F-Droid is a well-known repository of Free and Open Source Software (FOSS) applications for the Android platform that has been widely used in many other studies on Android proposed in the literature [2]. Among the apps that satisfied these criteria, we randomly chose a sample made of 14 apps. Table 2 reports for each app its ID, the app name, the name of the Android app package, the considered app version, and a brief description of the app functionality. Table 3 instead shows for each app the app ID, the total number of Activities, the number of LOCs, the number of classes, the number of methods, and the presence of GUIs belonging to the considered Gate GUI classes.

We considered only the Java classes that contain the app code, i.e., we took into account neither the Java classes belonging to third party libraries nor the code written in native C/C++ used to develop the app.

As it emerges from the data shown in the tables, the selected apps are sufficiently diverse since they offer different functionality and have a variable size both in terms of Activity number and LOCs.

6.2. Subjects selection

Since the juGULAR approach requires manual intervention of an end user to overcome the encountered Gate GUIs, we recruited 14 M.Sc. Software Engineering students. The selected subjects had to be involved in the study for providing the Unlocking Input Event Sequences, when needed, during the automated exploration of the object apps. They had a background on software testing and on network traffic analysis matured during their studies. They were selected by an interview. They had no prior in-depth knowledge about the selected apps nor a thorough knowledge about the underlying concepts of the Android Framework.

6.3. Metrics definition

In this section we describe the metrics we chose to answer the proposed research questions.

6.3.1. Effectiveness metrics

Since in the study we focused on software testing and network signatures generation scenarios, we decided to evaluate the effectiveness of juGULAR as the ability to: (1) cover app Activities, (2) cover app Lines of Code (LOC), and (3) generate network traffic. To this aim we defined the following set of metrics:

- The *Covered Activities* percentage (*CA*%) reports the percentage of the Activities covered during the exploration on the total number of App Activities; it can be measured according to the following formula:

[29] https://f-droid.org/.

```
#Start
        #SendEvents
                /dev/input/event1 3 57 0
                /dev/input/event1 3 48 16
                /dev/input/event1 3 58 129
                /dev/input/event1 3 53 5769
                /dev/input/event1 3 54 20046
                /dev/input/event1 0 0 0
                /dev/input/event1 3 58 0
                /dev/input/event1 3 57 4294967295
                /dev/input/event1 0 0 0                          Tap
        #Sleep(1500)
        #SendEvents
                /dev/input/event1 3 57 0
                /dev/input/event1 3 48 17
                /dev/input/event1 3 58 129
                /dev/input/event1 3 53 6680
                /dev/input/event1 3 54 19943
                /dev/input/event1 0 0 0                          Long Tap
        #Sleep(600)
        #SendEvents
                /dev/input/event1 3 58 0
                /dev/input/event1 3 57 4294967295
                /dev/input/event1 0 0 0
        #Sleep(1500)
        #SendEvents
                /dev/input/event1 3 57 0
                /dev/input/event1 3 48 18
                /dev/input/event1 3 58 129
                /dev/input/event1 3 53 6498
                /dev/input/event1 3 54 23204
                /dev/input/event1 0 0 0
                /dev/input/event1 3 54 23153
                /dev/input/event1 0 0 0                          Scroll
                /dev/input/event1 3 54 23051
                /dev/input/event1 0 0 0
                /dev/input/event1 3 58 0
                /dev/input/event1 3 57 4294967295
                /dev/input/event1 0 0 0
        #Sleep(1500)
        #SendEvents
                /dev/input/event1 1 102 1
                /dev/input/event1 0 0 0                          Key Home
                /dev/input/event1 1 102 0                        Press
                /dev/input/event1 0 0 0
        #Sleep(1500)
#End
```

Fig. 14. An Unlocking Sequence Description File. The sequence is contained within `#Start` and `#End` commands and includes five `#SendEvents` commands, followed each by a sequence of kernel-level events to be provided to the *sendevent* tool. The `#Sleep` commands are used to introduce delays between consecutive events.

Table 3
Characteristics of the Android apps involved in the study.

App ID	# App Activities	# App LOC	# App Classes	# App Methods	Presence of Gate GUIs	
					Login	Network Settings
A1	8	4487	195	762		✗
A2	20	23,548	634	3675	✗	
A3	10	4498	284	850	✗	✗
A4	3	2313	135	424		✗
A5	27	29,829	919	5249	✗	✗
A6	10	4394	232	921	✗	✗
A7	10	3671	258	1035	✗	
A8	16	9638	495	2716	✗	✗
A9	22	18,840	481	2973	✗	✗
A10	5	1272	97	321		✗
A11	9	8436	247	1436		✗
A12	12	7256	236	1198	✗	✗
A13	4	3904	210	681		✗
A14	13	7256	200	1079		✗

App ID: unique identifier of the app.
 # App Activities: number of Activity classes of the app.
 # App LOC: overall number of executable Lines Of Code of the app.
 # App Classes: overall number of classes of the app.
#App Methods: overall number of methods exposed by the classes of the app.
Presence of Gate GUIs: the ✗ marker indicates the type of Gate GUI exposed by the app.

$$CA\% = \frac{\#\,Covered\ Activities}{\#\,App\ Activities} * 100$$

- the *Covered Lines of Code* percentage (*CLOC*%) defines the percentage of the app LOC exercised during the automated exploration on the total number of the App LOC. It is expressed by:

$$CLOC\% = \frac{\#\,Covered\ LOCs}{\#\,App\ LOCs} * 100$$

- the *Network Traffic Bytes* (*NTB*) metric responds to the need to evaluate the ability of the technique to trigger the generation of network traffic; it measures the number of bytes received or sent on

Table 2
The Android apps involved in the study.

App ID	App name	Package name	Version	App description
A1	Flym News Reader	net.fred.feedex	1.9.0	Simple, modern and totally free RSS reader.
A2	Conversations	eu.siacs.conversations	1.19.5	Jabber/XMPP client for Android.
A3	DAVdroid	at.bitfire.davdroid	1.5.0.3-ose	Calendar synchronization app.
A4	Transistor Radio	org.y20k.transistor	2.2.0	App for listening to radio over internet.
A5	k9-Mail	com.fsck.k9	5.206	Email client supporting multiple accounts.
A6	mGit	com.manichord.mgit	1.5.0	Git client and text editor.
A7	Muspy	com.danielme.muspyforandroid	3.4.48	Client for Muspy.com.
A8	OpenRedmine	jp.redmine.redmineclient	3.20	Android Redmine client.
A9	OwnCloud	com.owncloud.android	2.3.0	Android client for private ownCloud Server.
A10	PortKnocker	com.xargsgrep.portknocker	1.0.11	App that pings a specific TCP/UDP port.
A11	LibreTorrent	org.proninyaroslav.libretorrent	1.4	Original Free torrent client.
A12	Connectbot	org.connectbot	1.9.2-oss	Powerful open-source Secure Shell (SSH) client.
A13	PodListen	com.einmalfel.podlisten	1.3.6	Free Podcast app.
A14	ServeStream	net.sourceforge.servestream	0.7.3	Open source HTTP streaming media player and media server browser.

App ID: unique identifier of the app.
App Name: name of the app.
Package Name: name of the application package.
Version: version of the app.
App Description: description of the main functionality offered by the app.

the network by the app during the exploration and it is expressed by the following formula:

$$NTB = \#App\ Sent\ Bytes\ +\ \#App\ Received\ Bytes$$

6.3.2. Manual intervention cost metric

To evaluate the cost of the manual intervention required by juGULAR, we considered for each app exploration process, the time spent in each capture activity, i.e., $CaptureTime_i$, and the Total Exploration Time of the technique. Therefore, we introduced the:

- *Manual Intervention Time Percentage* (*MIT*%) that defines the percentage of time spent in the human interventions required for unlocking the encountered Gate GUIs on the total exploration time. It is expressed by:

$$MIT\% = \frac{\sum_i CaptureTime_i}{TotalExplorationTime} * 100$$

6.4. Experimental procedure

The experimental procedure we designed for carrying out the study consisted of three sequential steps: *Training, Apps Exploration, Data Collection and Analysis*.

In the *Training* step, two researchers involved in the definition of juGULAR explained its approach to the selected subjects. The training was carried out through examples, where the researchers illustrated how the technique works, the type of Gate GUIs it is able to detect and the features it provides for unlocking them. In order to verify that all the subjects had correctly understood the approach and how to provide Unlocking Input Sequences when needed, in the last part of the Training, the subjects were asked to perform an exploration process on a sample Android app we appositely developed. The sample app exposed both a Login and a Network Settings Gate GUIs. Each subject was asked to provide Unlocking Input Sequences when needed. We did not instruct the subjects on the Unlocking Input Sequence to provide, they were free to insert the sequence they considered the most appropriate. At the end of this process, we analyzed the results of the exploration sessions. All the subjects were able to correctly adopt juGULAR and to provide the Unlocking Input Event Sequences. The entire Training step lasted 8 h.

In the *Apps Exploration* step, we executed three different exploration processes. In the first process we used juGULAR, in the second one we exploited juGULAR with the Hybridization Disabled, and in the third we employed Monkey. JHD is an ad hoc juGULAR configuration that performs the automated app exploration without exploiting the hybrid features, i.e., detection and capture and replay. In the following, we name JHE the actual implementation of juGULAR. Since Monkey implements a random app exploration and we wanted to implement a fair comparison among the considered tools, we configured also JHE and JHD to explore the apps using a random exploration strategy. Regarding the process involving JHE, we decided to repeat each app exploration with different subjects, in order to mitigate the dependence of the exploration effectiveness on a specific subject's judgment. We divided the selected subjects in 2 groups made of seven students, namely G_1 and G_2. We gave the subjects of each group the task of exploring 7 of the object apps, that were randomly assigned to each group. Table 4 reports the object apps assigned to the groups. To carry out the exploration tasks with JHE, we provided each student with a PC equipped with the tool. The students had to launch the app explorations and to intervene in the process only when a Gate GUI was encountered for the first time. For each app, since the explorations were random, three runs lasting 60 min had to be executed by each subject. We configured the `ReplayCondition` to assume the `true` value with a probability $p_{true} = 0.8$ so that the recorded Unlocking Input Event Sequences were not the only ones to be executed when a Gate GUI was detected. At the end of the Exploration step, we obtained 21 exploration runs for each app. A researcher controlled that subject performed the experiment according to the instructions provided in the Training step. The experiment was conducted under "exam conditions", i.e., subjects were not allowed to communicate with others for not biasing the experimental findings. As to the processes involving JHD and Monkey, we launched 21 exploration runs lasting 60 min for each app. In this way we obtained the same number of explorations as JHE. All the explorations were carried out on desktop PCs having an Intel(R) Core(TM) i7 4790@3.60 GHz processor and 8GB of RAM, running a standard Nexus 5 Android Virtual Device (AVD)[30] with Android API 19; the host PC was equipped with the Ubuntu OS, version 16.04. Each experiment was executed on a newly created AVD.

In the *Data Collection and Analysis* step we analyzed the reports produced by JHE, JHD, and Monkey to obtain the number of Activities and LOC covered during the explorations, as well as the generated network traffic bytes for the three exploration processes. As for the explorations with JHE, we also evaluated the capture times spent by each subject during the three exploration runs performed on each app.

Table 4
The Android apps assigned to each group of students.

GROUP	APP IDs
G_1	A3, A4, A7, A9, A11, A12, A13
G_2	A1, A2, A5, A6, A8, A10, A14

6.5. Experimental results

Table 5 reports the average effectiveness values we measured for the explorations carried out with JHE, JHD and Monkey, respectively. The average values have been obtained with respect to the 21 exploration runs for each app. The Table also reports the average values of all the metrics calculated considering all the selected apps.

The same results are shown by the histograms in Fig. 15 that provide a graphical visualization and allow the comparison among the average values of CA%, CLOC% and NTB obtained with JHD, JHE and Monkey.

As emerged from the analysis of the reports produced by the JHE explorations, juGULAR successfully detected all the Gate GUIs we identified in the object apps. These data also showed that each subject spent different amounts of capture time to unlock each Gate GUI.

In order to answer RQ_1, we compared the average effectiveness values obtained using JHE and JHD.

As regards the Activities exploration capability, the CA% data values reported in Table 5 show that JHE covered a greater percentage of Activities than JHD in 13 out of the 14 object apps. Only for *A*10 (PortKnocker), juGULAR achieved the same results covering 3 out of 5 Activities either with or without the hybridization. However, the two unexplored Activities of this app could not have been reached otherwise, since one of them is rendered only for older Android versions and the other one is accessible only from the app external widget. The average *CA*% increment with JHE was of 23%, while the minimum and maximum increment values were of 6.25% in *A*8 and up to 50% in *A*6, respectively.

In the case of *A*8 (OpenRedmine), the reduced increment in Activities coverage was essentially due to the choice of the input event sequences values provided by the subjects to unlock the Login Gate GUI. We observed that all the app features regarding the project management could not be exercised even after the unlocking. This

[30] https://developer.android.com/studio/run/managing-avds.html.

Table 5
Effectiveness results of the app explorations performed by JHD, JHE, and Monkey.

	JHD - juGULAR Hybridization Disabled			JHE - juGULAR Hybridization Enabled			Monkey		
App ID	CA%	CLOC%	NTB	CA%	CLOC%	NTB	CA%	CLOC%	NTB
A1	50	20.25	11,128	75	49.19	3,170,055	20	24.49	0
A2	30	8.33	0	50	18.64	352,196,871	30	5.65	0
A3	40	10	2,256,788	60	28.63	5,450,397	40	12.09	2,055,974
A4	66.7	12.21	0	100	62.73	38,894,905	66.6	10.18	0
A5	7.4	4.22	0	25.93	37.14	93,987,389	7.4	4.9	0
A6	30	14.43	16,890	80	46.61	3,259,397	40	19.18	15,404
A7	10	15.41	33,722	50	49.09	6,295,654	10	18.63	30,462
A8	25	4.96	0	31.25	16.09	74,088	25	6.01	0
A9	4.5	8.11	854,033	22.73	23.90	6,376,599	4.5	9.43	787,737
A10	60	53.62	0	60	60.53	2275	60	44.65	0
A11	12.63	22.22	2,834,611	44.4	39.4	670,369,445	33.3	25.83	4,947,046
A12	33.3	14.25	103,550	58.3	41.21	15,987,272	33.3	17.24	0
A13	91.6	41.8	6758	100	43.34	241,584	75	38.4	9978
A14	30.76	9.61	11,243,538	53.9	30.65	30,496,185	30.7	11.28	1,679,895
Avg	35.13	17.10	1,240,072	57.96	39.08	87,628,722	33.98	17.71	680,464

Avg: average values of the metrics calculated considering all the apps.
App ID: unique app identifier.
CA%: average percentage of covered Activities.
CLOC%: average percentage of covered executable lines of code.
NTB: average number of Bytes sent and received on the network by the app.

happened since all the credentials they provided were associated to Redmine repository accounts having no associated projects. On the contrary, as to the *A*6 (mGit) app, we obtained a considerable Activity coverage increment with JHE, because at least one subject provided Login credentials associated with accounts having non-empty project repositories.

As for the Source Code exploration capability, the CLOC% data values reported in Table 5 show that JHE always covered a wider percentage of source code than JHD, with an increment of 22%, on average. The minimum increment of code coverage percentage was of 1.54%, and was observed in *A*13 (PodListen). The Network Settings Gate GUI exposed by the app was unlocked even without hybridization since PodListen allows the user to subscribe to a podcast not only by adding a podcast URL but also selecting a predefined podcast provided by the internal podcast database. The maximum increment of code coverage percentage was instead of 50.52%, and was observed in *A*4 (Transistor). This app exposed a Network Settings Gate GUI that should be unlocked to execute the code that implements the features for controlling and reproducing audio streams. The only way to unlock this Gate GUI was to provide a valid audio stream URL as had been done by the subjects.

Regarding the ability of generating network traffic, the hybridization allowed juGULAR to obtain impressive results. JHD was not able to produce any network traffic in 5 out of the 14 object apps. Considering all the apps, JHD was able to produce 1MB of network traffic, on average. Instead, JHE produced more traffic than JHD in all the object apps, with about 88MB of generated traffic, on average.

As for *A*10, we had the minimum increment of Network Traffic Bytes of 2275 Bytes. This happened since the app exposed a Network Settings Gate GUI that required a valid and reachable IP address along with a valid Port number that were never provided without hybridization. However, even the amount of network traffic the app generated by unlocking this Gate GUI was still small since it consisted only in a few TCP or UDP network packets used to ping the specified ports.

In the *A*11 app, i.e., LibreTorrent, we noticed the maximum NTB increment that consisted of over 667MB. This was obtained since the subjects unlocked the Network Settings Gate GUI providing valid Torrent URL that pointed to large files.

On the basis of these results we were able to answer the first research question RQ_1 and conclude that:

> *The hybridization introduced by juGULAR had a positive impact on the exploration effectiveness in both the considered scenarios. It allowed to obtain better results in terms of Covered Activities, Covered LOC and Generated Network Traffic.*

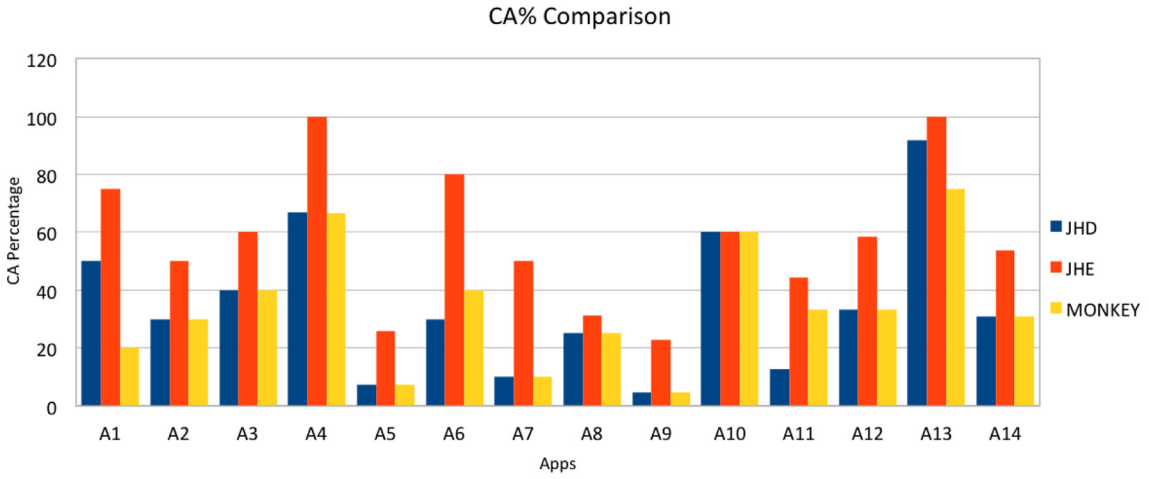
In order to address RQ_2, we considered the cost of the manual interventions required by the hybridization introduced by juGULAR. Fig. 16(a) reports the average MIT% value for each app. The histogram and the table reported in Fig. 16(b) show, for each app, the time spent for the manual intervention and for the automated exploration during the 180 min session time, averaged on all the subjects.

As Figure shows, the time for the manual intervention required by JHE was on average lower than 3% of the entire exploration time for all the considered apps. All the subjects were able to define the Unlocking Input Event Sequence in less than about 5 min on average. For the *A*10 app, it took the subjects less than 40 s, on average, to define the Unlocking Input Event Sequence since the app exposed a simple form in which the subjects mostly inserted well-known IP addresses, e.g., the `localhost` or the Google DNS addresses.

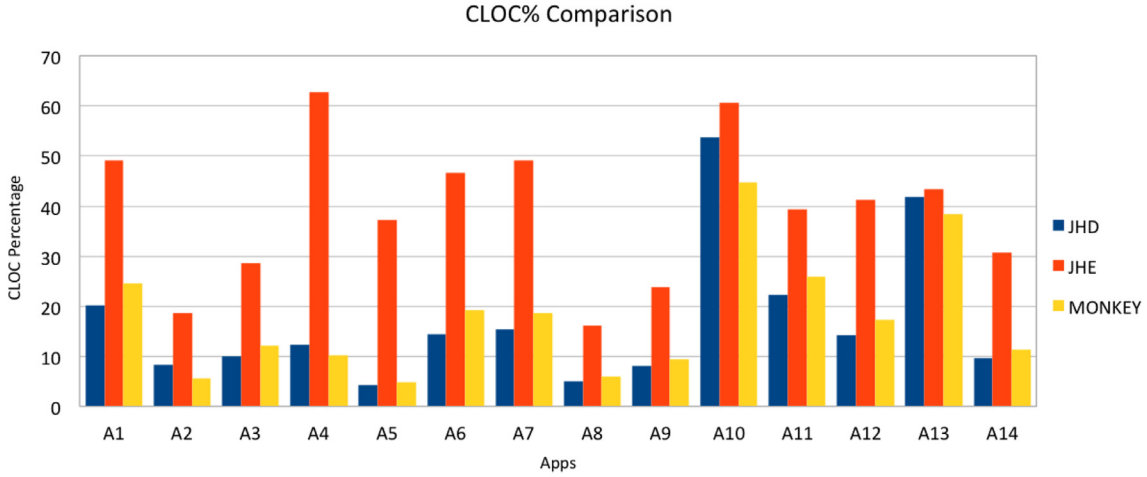
As for *A*2 (Conversations), the subjects had to unlock a Login Gate GUI in which the user had to provide valid credentials of an account registered to an existing Jabber/XMPP service. Since most of the subjects did not own such an account, they spent time to create it before unlocking the Gate GUI.

According to these results we could answer the second research question RQ_2 concluding that:

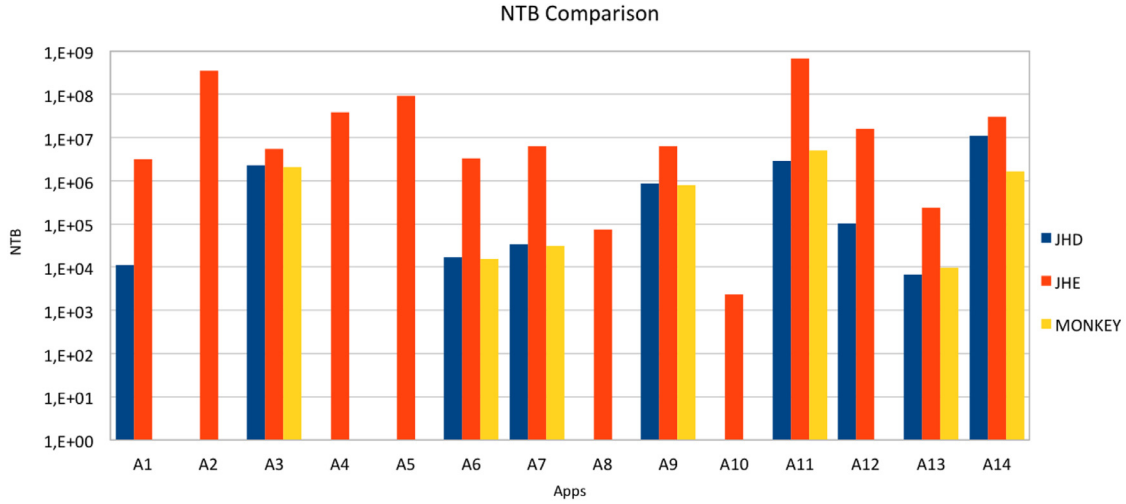
> *The manual intervention required by juGULAR has a limited impact on the cost of the exploration technique, being always lower than 3%.*



(a) Average CA% values obtained by JHD, JHE, and Monkey

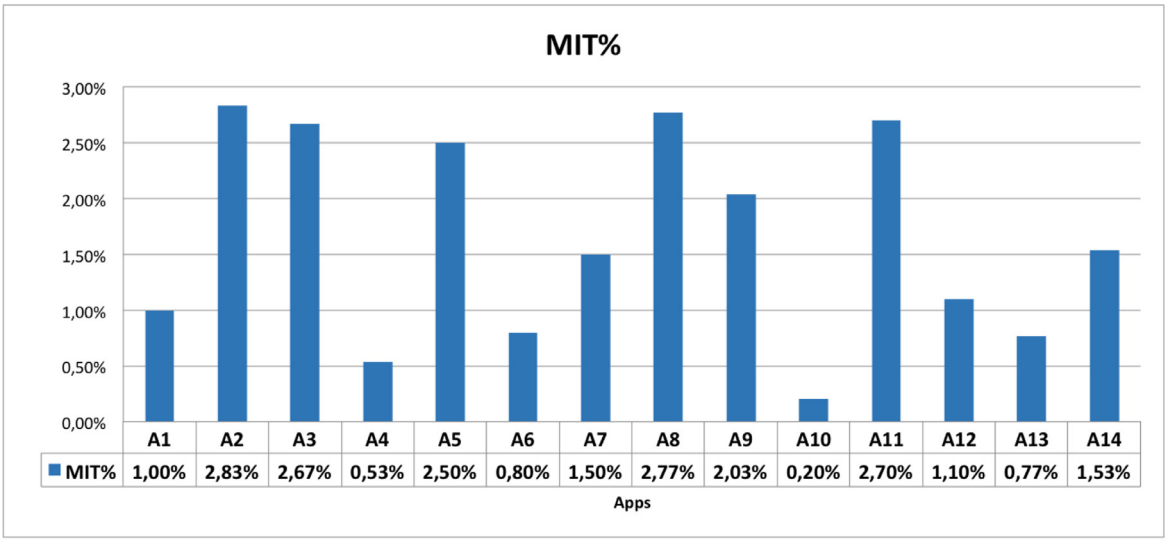


(b) Average CLOC% values obtained by JHD, JHE, and Monkey

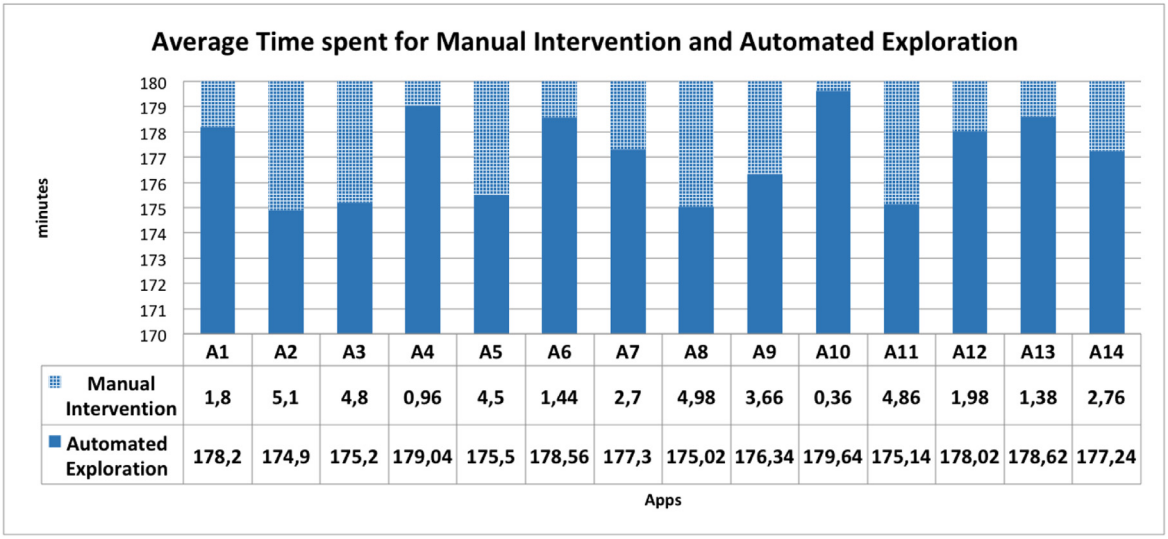


(c) Average NTB values obtained by JHD, JHE, and Monkey

Fig. 15. Average effectiveness results of the explorations executed by JHD, JHE, and Monkey.



(a) MIT%: Average percentage of time spent in the human interventions for unlocking the encountered Gate GUIs on the total exploration time, with the JHE approach in the 180 minutes sessions, averaged on all the subjects.



(b) Average Time (in minutes) spent for the Manual Intervention and for the Automated Exploration with the JHE approach in the 180 minutes sessions, averaged on all the subjects.

Fig. 16. Costs of the manual interventions required by the hybridization introduced by juGULAR.

In order to answer the RQ_3, we compared the effectiveness of JHE and Monkey.

The data in Table 5 shows that JHE was always more effective than the Monkey tool. On average, JHE was able to cover 24% more Activities, 21% more LOCs and to generate 86 network traffic MBytes more than Monkey.

The reported data allowed us to answer the third research question RQ_3:

> *jUGULAR is more effective than the state-of-the-practice tool Monkey in exploring real Android apps.*

6.6. Study conclusion

The experimental results showed that the hybridization of the automated exploration approach proposed by juGULAR produced in average a considerable improvement of the exploration effectiveness. They confirmed us the usefulness of our approach that allows the user to provide knowledge at runtime rather than using pre-configured and generic input event sequences. This is consistent with other work that point out the complementarity between automated machine-generated tests and human tests [42,43].

The fact that juGULAR was always able to outperform the other tools in terms of generated network traffic suggests that this approach may be especially useful in scenarios that leverage on realistic

generated network traffic, such as ground-truth generation of mobile app traffic [44] or mobile app network traffic signatures generation [4].

6.7. Threats to validity

The following threats affect the validity of our experimental study [45].

6.7.1. Internal validity

In this study, a possible threat to the internal validity could have been the assignment of the subjects to the objects. To mitigate the possible bias, we randomly assigned the objects to the subjects. A possible factor that could have influenced the outcome was the subject experience. To mitigate this threat, we explored the same app multiple times with different subjects. The outcome could be also influenced by the Gate GUI classes we considered. A further experimentation considering a wider sample of Gate GUI classes should be carried out to mitigate this threat.

Another possible threat to the internal validity is that the effectiveness improvements that we observed in the experiment were not actually due to the hybridization, but rather to other factors, such as the randomness of the explorations. We tried to mitigate this threat by executing 21 random explorations and involving 7 different subjects for each app and by performing the validation task of the Data Collection and Analysis step.

6.7.2. External validity

We are aware that the choice of object apps is a possible threat to the external validity. The diversity of the selected apps can mitigate this threat. However, to extend the validity of our results, a wider sample of real Android apps including also industrial-strength apps from Google Play store should be considered. Also choosing students as subjects of the study may have affected its external validity. However, they present characteristics that make them representative of possible future users of automated exploration techniques. To further mitigate this threat, case studies in real industrial settings should be carried out to assess the validity of the approach on the field.

In the study we measured the manual intervention costs required by juGULAR by the MIT% metric. Since this metric depends on the total exploration time, the measured manual intervention costs are influenced by the choice of the app run length and our conclusions may not generalize beyond the considered experimental settings. We tried to mitigate this threat using in our experiments the value of the app run length that is adopted in state of the art works in Android app automated exploration. Indeed we set one hour as exploration time for each app run, following the experimental setup used in the previous thorough benchmark assessment study by Choudhary et al. [2] and in the experiment performed by Mao et al. [18].

We cannot claim that our results generalize to other Gate GUI classes. To further extend the validity of our study, an experiment involving a wider set of Gate GUI classifiers trained and integrated in the juGULAR platform should be carried out.

7. Conclusions and future work

Automated GUI exploration techniques are becoming widespread in mobile app development processes, due to their capability to execute time-consuming tasks. However, one of their critical issues is the limited capability of exploring the behavior of apps that require meaningful sequences of input events on specific GUIs, i.e., Gate GUIs, in order to exercise some of their functionality. In this paper, we addressed this issue by proposing juGULAR, a hybrid automated GUI exploration technique that pragmatically combines fully automated GUI exploration with Capture and Replay in order to improve the app exploration and minimize the human intervention. Our technique leverages Machine Learning to train classifiers that we exploit to automatically detect the occurrence of Gate GUI instances during the exploration.

We implemented this technique in a software platform and validated it with an experiment involving 14 real Android apps. In this work, we focused on two specific classes of Gate GUIs: *Login* and *Network Settings*. The experiment showed that the app exploration can improve thanks to the hybridization in terms of Covered Activities, Covered LOC and Generated Network Traffic. The manual intervention required by the technique had a limited impact on the entire exploration costs. The experiment also showed that juGULAR was more effective in app exploration than the state-of-the practice automated Android GUI exploration tool.

We are aware that juGULAR may suffer from the limitation introduced by app non-determinism. As future work, we intend to address the issues of non-deterministic Gate GUI, such as those exposed by Games or containing CAPTCHAs, by investigating effective solutions to handle them. We plan to extend juGULAR by considering more Gate GUI classes besides the ones we have dealt with.

Finally, we plan to extend the validity of our experimental results by carrying out an industrial case study involving real practitioners and a wider set of Android apps. In addition, we would like to consider further performance indicators, such as the diversity of generated network traffic. This aspect is critical for assessing how realistic such traffic is and it can be exploited in several areas, e.g., mobile app traffic ground-truth generation and network traffic signatures generation. To this aim, we intend to investigate suitable measurement approaches and metrics for evaluating such diversity, since it is still an open issue in the literature.

Acknowledgments

The authors thank the anonymous reviewers for their valuable feedback.

References

[1] H. Muccini, A.d. Francesco, P. Esposito, Software testing of mobile applications: challenges and future research directions, Automation of Software Test (AST), 2012 7th International Workshop on, IEEE, Zurich, Switzerland, 2012, pp. 29–35, https://doi.org/10.1109/IWAST.2012.6228987. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6228987 .

[2] S.R. Choudhary, A. Gorla, A. Orso, Automated test input generation for android: Are we there yet? (E), Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), ASE '15, IEEE Computer Society, Washington, DC, USA, 2015, pp. 429–440, https://doi.org/10.1109/ASE.2015.89. https://doi.org/10.1109/ASE.2015.89 .

[3] A. Memon, I. Banerjee, B.N. Nguyen, B. Robbins, The first decade of GUI ripping:extensions, applications, and broader impacts, 2013 20th Working Conference on Reverse Engineering (WCRE), (2013), pp. 11–20, https://doi.org/10.1109/WCRE.2013.6671275.

[4] Y. Chen, W. You, Y. Lee, K. Chen, X. Wang, W. Zou, Mass discovery of android traffic imprints through instantiated partial execution, Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17, ACM, New York, NY, USA, 2017, pp. 815–828, https://doi.org/10.1145/3133956.3134009. http://doi.acm.org/10.1145/3133956.3134009 .

[5] X. Su, D. Zhang, W. Li, X. Wang, AndroGenerator: an automated and configurable android app network traffic generation system, Secur. Commun. Netw. 8 (18) (2015) 4273–4288, https://doi.org/10.1002/sec.1341. sec.1341, https://doi.org/10.1002/sec.1341 .

[6] M. Karami, M. Elsabagh, P. Najafiborazjani, A. Stavrou, Behavioral analysis of android applications using automated instrumentation, 2013 IEEE Seventh International Conference on Software Security and Reliability Companion, (2013), pp. 182–187, https://doi.org/10.1109/SERE-C.2013.35.

[7] S.N. Dutia, T.H. Oh, Y.H. Oh, Developing automated input generator for android mobile device to evaluate malware behavior, Proceedings of the 4th Annual ACM Conference on Research in Information Technology, RIIT '15, ACM, New York, NY, USA, 2015, p. 43, https://doi.org/10.1145/2808062.2808065. http://doi.acm.org/10.1145/2808062.2808065 .

[8] M.M. Eler, J.M. Rojas, Y. Ge, G. Fraser, Automated accessibility testing of mobile apps, 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), Vol. 00, (2018), pp. 116–126, https://doi.org/10.1109/ICST.2018.00021. doi.ieeecomputersociety.org/10.1109/ICST.2018.00021 .

[9] A. Banerjee, L.K. Chong, S. Chattopadhyay, A. Roychoudhury, Detecting energy bugs and hotspots in mobile apps, Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, ACM,

New York, NY, USA, 2014, pp. 588–598, https://doi.org/10.1145/2635868.2635871. http://doi.acm.org/10.1145/2635868.2635871 .
[10] X. Deng, T. Kameda, C. Papadimitriou, How to learn an unknown environment. I: the rectilinear case, J. ACM 45 (2) (1998) 215–245, https://doi.org/10.1145/274787.274788. http://doi.acm.org/10.1145/274787.274788 .
[11] P. Fraigniaud, D. Ilcinkas, G. Peer, A. Pelc, D. Peleg, Graph exploration by a finite automaton, Theor. Comput. Sci. 345 (2) (2005) 331–344, https://doi.org/10.1016/j.tcs.2005.07.014. http://www.sciencedirect.com/science/article/pii/S0304397505003993 Mathematical Foundations of Computer Science 2004.
[12] D. Amalfitano, N. Amatucci, A.M. Memon, P. Tramontana, A.R. Fasolino, A general framework for comparing automatic testing techniques of android mobile apps, J. Syst. Softw. 125 (2017) 322–343, https://doi.org/10.1016/j.jss.2016.12.017. https://doi.org/10.1016/j.jss.2016.12.017 .
[13] A. Machiry, R. Tahiliani, M. Naik, Dynodroid: an input generation system for android apps, Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, ACM, New York, NY, USA, 2013, pp. 224–234, https://doi.org/10.1145/2491411.2491450. http://doi.acm.org/10.1145/2491411.2491450 .
[14] D. Amalfitano, A.R. Fasolino, P. Tramontana, S. De Carmine, A.M. Memon, Using GUI ripping for automated testing of android applications, Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, ACM, New York, NY, USA, 2012, pp. 258–261, https://doi.org/10.1145/2351676.2351717. http://doi.acm.org/10.1145/2351676.2351717 .
[15] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, T. Xie, Automated test input generation for android: are we really there yet in an industrial case? Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, ACM, New York, NY, USA, 2016, pp. 987–992, https://doi.org/10.1145/2950290.2983958. http://doi.acm.org/10.1145/2950290.2983958 .
[16] M. Ermuth, M. Pradel, Monkey see, monkey do: effective generation of GUItests with inferred macro events, Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, ACM, New York, NY, USA, 2016, pp. 82–93, https://doi.org/10.1145/2931037.2931053. http://doi.acm.org/10.1145/2931037.2931053 .
[17] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, D. Poshyvanyk, Automatically discovering, reporting and reproducing android application crashes, 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), (2016), pp. 33–44, https://doi.org/10.1109/ICST.2016.34.
[18] K. Mao, M. Harman, Y. Jia, Sapienz: multi-objective automated testing for android applications, Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, ACM, New York, NY, USA, 2016, pp. 94–105, https://doi.org/10.1145/2931037.2931054. http://doi.acm.org/10.1145/2931037.2931054 .
[19] W. Choi, G. Necula, K. Sen, Guided GUI testing of android apps with minimal restart and approximate learning, Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '13, ACM, New York, NY, USA, 2013, pp. 623–640, https://doi.org/10.1145/2509136.2509552. http://doi.acm.org/10.1145/2509136.2509552 .
[20] S. Hao, B. Liu, S. Nath, W.G. Halfond, R. Govindan, PUMA: programmable ui-automation for large-scale dynamic analysis of mobile apps, Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '14, ACM, New York, NY, USA, 2014, pp. 204–217, https://doi.org/10.1145/2594368.2594390. http://doi.acm.org/10.1145/2594368.2594390 .
[21] G. Hu, X. Yuan, Y. Tang, J. Yang, Efficiently, effectively detecting mobile app bugs with appdoctor, Proceedings of the Ninth European Conference on Computer Systems, EuroSys '14, ACM, New York, NY, USA, 2014, pp. 18:1–18:15, https://doi.org/10.1145/2592798.2592813. http://doi.acm.org/10.1145/2592798.2592813 .
[22] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, D. Song, NetworkProfiler: towards automatic fingerprinting of android apps, 2013 Proceedings IEEE INFOCOM, (2013), pp. 809–817, https://doi.org/10.1109/INFCOM.2013.6566868.
[23] L. Gomez, I. Neamtiu, T. Azim, T. Millstein, RERAN: timing- and touch-sensitive record and replay for android, Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 72–81. http://dl.acm.org/citation.cfm?id=2486788.2486799 .
[24] R. Mahmood, N. Mirzaei, S. Malek, EvoDroid: segmented evolutionary testing of android apps, Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, ACM, New York, NY, USA, 2014, pp. 599–609, https://doi.org/10.1145/2635868.2635896. http://doi.acm.org/10.1145/2635868.2635896 .
[25] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, L. Zeng, Automatic text input generation for mobile testing, 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), (2017), pp. 643–653, https://doi.org/10.1109/ICSE.2017.65.
[26] M. Utting, A. Pretschner, B. Legeard, A taxonomy of model-based testing approaches, Softw. Test. Verif. Reliab. 22 (5) (2012) 297–312, https://doi.org/10.1002/stvr.456. https://doi.org/10.1002/stvr.456 .
[27] T. Azim, I. Neamtiu, Targeted and depth-first exploration for systematic testing of android apps, SIGPLAN Not. 48 (10) (2013) 641–660, https://doi.org/10.1145/2544173.2509549. http://doi.acm.org/10.1145/2544173.2509549 .
[28] D. Amalfitano, A.R. Fasolino, P. Tramontana, B.D. Ta, A.M. Memon, Mobiguitar: automated model-based testing of mobile apps, IEEE Softw. 32 (5) (2015) 53–59, https://doi.org/10.1109/MS.2014.55.
[29] D. Amalfitano, V. Riccio, A.C.R. Paiva, A.R. Fasolino, Why does the orientation change mess up my android application? From GUI failures to code faults, Softw. Test. Verif. Reliab. (2017), https://doi.org/10.1002/stvr.1654. e1654n/aE1654 stvr.1654.
[30] K. Seng, L.M. Ang, C. Ooi, A combined rule-based and machine learning audio-visual emotion recognition approach, IEEE Trans. Affect Comput. PP (99) (2016) 1, https://doi.org/10.1109/TAFFC.2016.2588488.
[31] G.A. Di Lucca, A.R. Fasolino, P. Tramontana, Web pages classification using concept analysis, 2007 IEEE International Conference on Software Maintenance, (2007), pp. 385–394, https://doi.org/10.1109/ICSM.2007.4362651.
[32] C.D. Manning, P. Raghavan, H. Schütze, Introduction to Information Retrieval, Cambridge University Press, New York, NY, USA, 2008.
[33] B.N. Nguyen, B. Robbins, I. Banerjee, A. Memon, GUITAR: an innovative tool for automated testing of GUI-driven software, Autom. Softw. Eng. 21 (1) (2014) 65–105, https://doi.org/10.1007/s10515-013-0128-9. https://doi.org/10.1007/s10515-013-0128-9 .
[34] I. Banerjee, B. Nguyen, V. Garousi, A. Memon, Graphical user interface (GUI) testing: systematic mapping and repository, Inf. Softw. Technol. 55 (10) (2013) 1679–1694, https://doi.org/10.1016/j.infsof.2013.03.004. http://www.sciencedirect.com/science/article/pii/S0950584913000669 .
[35] S. Panichella, A.D. Sorbo, E. Guzman, C.A. Visaggio, G. Canfora, H.C. Gall, ARdoc: app reviews development oriented classifier, Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, ACM, New York, NY, USA, 2016, pp. 1023–1027, https://doi.org/10.1145/2950290.2983938. http://doi.acm.org/10.1145/2950290.2983938 .
[36] M. Galar, A. Fernandez, E. Barrenechea, H. Bustince, F. Herrera, An overview of ensemble methods for binary classifiers in multi-class problems: experimental study on one-*vs*-one and one-*vs*-all schemes, Pattern Recognit. 44 (8) (2011) 1761–1776, https://doi.org/10.1016/j.patcog.2011.01.017. http://www.sciencedirect.com/science/article/pii/S0031320311000458 .
[37] N. Garcia-Pedrajas, D. Ortiz-Boyer, An empirical study of binary classifier fusion methods for multiclass classification, Inf. Fusion 12 (2) (2011) 111–130, https://doi.org/10.1016/j.inffus.2010.06.010. http://www.sciencedirect.com/science/article/pii/S1566253510000734 .
[38] J. Han, M. Kamber, J. Pei, Data mining, Concepts and Techniques, 3rd Ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.
[39] F. Belli, M. Beyazit, A. Memon, Testing is an event-centric activity, Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on, (2012), pp. 198–206.
[40] D. Adamo, D. Nurmuradov, S. Piparia, R. Bryce, Combinatorial-based event sequence testing of android applications, Inf. Softw. Technol. 99 (2018) 98–117, https://doi.org/10.1016/j.infsof.2018.03.007. http://www.sciencedirect.com/science/article/pii/S0950584918300429 .
[41] L. Gomez, I. Neamtiu, T. Azim, T. Millstein, RERAN: timing- and touch-sensitive record and replay for android, 2013 35th International Conference on Software Engineering (ICSE), (2013), pp. 72–81, https://doi.org/10.1109/ICSE.2013.6606553.
[42] K. Mao, M. Harman, Y. Jia, Crowd intelligence enhances automated mobile testing, Proceedings of the 2017 32th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, (2017), pp. 16–26.
[43] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, D. Poshyvanyk, Mining android app usages for generating actionable GUI-based execution scenarios, 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, (2015), pp. 111–122, https://doi.org/10.1109/MSR.2015.18.
[44] V.F. Taylor, R. Spolaor, M. Conti, I. Martinovic, AppScanner: automatic fingerprinting of smartphone apps from encrypted network traffic, 2016 IEEE European Symposium on Security and Privacy (EuroS P), (2016), pp. 439–454, https://doi.org/10.1109/EuroSP.2016.40.
[45] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, Experimentation in Software Engineering, Springer, 2012, https://doi.org/10.1007/978-3-642-29044-2. https://doi.org/10.1007/978-3-642-29044-2 .