Mobile GUI Testing Fragility: A Study on Open-Source Android Applications

Mobile GUI Testing Fragility: A Study on Open-Source Android Applications

Riccardo Coppola, Maurizio Morisio, Marco Torchiano

***Abstract*—Android applications do not seem to be tested as thoroughly as desktop ones. In particular, GUI testing appears generally limited. Like web-based applications, mobile apps suffer from GUI test fragility, i.e. GUI test classes failing or needing updates due to even minor modifications in the GUI or in the Application Under Test.**

The objective of our study is to estimate the adoption of GUI testing frameworks among Android open-source applications, the quantity of modifications needed to keep test classes up to date, and the amount of them due to GUI test fragility. We introduce a set of 21 metrics to measure the adoption of testing tools, the evolution of test classes and test methods, and to estimate the fragility of test suites.

We computed our metrics for six GUI testing frameworks, none of which achieved a significant adoption among Android projects hosted on GitHub. When present, GUI test methods associated with the considered tools are modified often and a relevant portion (70% on average) of those modifications is induced by GUI-related fragilities. On average for the projects considered, more than 7% of the total modified lines of code between consecutive releases belong to test classes developed with the analysed testing frameworks. The measured percentage was higher on average than the one required by other generic test code, based on the JUnit testing framework.

Fragility of GUI tests constitutes a relevant concern, probably an obstacle for developers to adopt test automation. This first evaluation of the fragility of Android scripted GUI testing can constitute a benchmark for developers and testers leveraging the analysed test tools, and the basis for the definition of a taxonomy of fragility causes and guidelines to mitigate the issue.

***Keywords*—Mobile Computing, Software Engineering, Software Metrics, Software Maintenance, Software Testing.**

I. Introduction

As several market analyses underline, Android has gained a very significant market share with respect to other mobile operating systems, reaching the 86.2% in the second quarter of 2016[1]. Mobile devices, nowadays, offer their users a very wide range of different applications, that have reached a complexity that just a few years ago was exclusive of high-end desktop computers.

One of the points of strenght of the Android operating system is the availability of several marketplaces, that allow developers to easily sell the applications or release them for free. Because of the huge amount of apps available on such platforms, and the resulting competition, it is crucial for developers to make sure that their software works as promised to the users. In fact, applications that crash unexpectedly during their normal execution, or that are hampered by bugs, are likely to be quickly abandoned by their users for competitors [1], and to gather very negative feedback [2]. Mobile applications must also comply to a series of strict non-functional requirements, that are specific to a mobile and context-aware environment [3].

In such a scenario, testing mobile apps becomes a very crucial practice. In particular, it is fundamental to test the Graphical User Interfaces (GUIs) of the apps, since most of the interaction with the final user is performed through them.

There is evidence that relevant players of the industry perform structured testing procedures of their mobile applications, also leveraging the aid of automated tools (for instance, Alegroth et al. documented the long-term adoption of Visual GUI testing practices at Spotify [4]). By contrast, it has been proved by several studies that open-source

The authors are with the Department of Computer Engineering and Automatics, Politecnico di Torino, Torino, Italy. e-mail: first.last@polito.it

Manuscript received April 19, 2005; revised January 11, 2007.

[1]https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

mobile developers rarely adopt automated testing techniques in their projects. Kochar et al. [5] found that, on the set of open-source projects (mined from F-Droid[2]) they examined, just 14% of the set featured any kind of scripted automated test classes; Vásquez et al. [6] found that the majority of an interviewed set of contributors to open-source projects relied just on the execution of manual test cases, even though a variety of automated testing tools (open source or not) is available.

Performing proper testing of Android apps presents a set of domain-specific challenges, principally due to the very fast pace of evolution of the operating system, and to the vast number of possible configurations and features the apps must be compatible with. In addition to that, the development process for Android apps is typically very quick, and the need for making the applications available to the public as soon as possible may be a deterrent for developers to perform complex forms of testing. Muccini et al. [7] stress the differences between traditional software and Android applications when it comes to testing them: the huge quantity of context events, to which apps have to react properly; the diversity of devices, to which the apps must be compatible; the possible lack of resources for some devices.

Similar to what happens for web applications testing, automated GUI testing of Android apps is also hampered by the fragility issue. For our purposes, we define a GUI test case as fragile if it requires interventions when the application evolves (i.e., between subsequent releases) due to modifications applied to the Application Under Test. Being system level tests, GUI test cases are affected by variations in the functionalities of the application (as it happens for lower-level component tests) but also from even small interventions in the appearance, definition and arrangement of the GUI presented to the user.

Fragility is a significant issue for Android application testing, since a failing test may require in-depth investigations to find out what are the causes of the failures, and entire test suites may need modifications due to even minor changes in the GUIs and in their definition. If that happens, developers may decide to not exercise any kind of structured scripted testing. In our previous work [8], we developed a small test suite (made of eleven test classes) for K9-Mail[3] – a popular, large-sized open source Android mail client – and tracked the modifications that were needed by test classes to be executable on different releases. We found out that up to 75% of tests that we developed had to be modified, because of modifications performed on the GUI of the app. If scripted test cases were obtained through the use of Capture&Replay techniques, for some transitions between releases the entirety of the test suite had to be rewritten.

In this work, we aimed at gathering information about existing test suites featured by open-source Android applications. We extended the context of previous similar work (like the one by Kochar et al. [5], who analysed a set of about 600 open-source Android apps collected from F-Droid), considering all the projects hosted on GitHub that contained proper Android applications and that featured a history of releases, for a total of 18,930 projects. We identified six open-source Android GUI testing tools cited in available literature and producing test classes in Java, and we searched for the presence of code written with those tools in the mined Android projects. This way, we subdivided the projects in six subsets, according to the testing tool they featured. Then, change metrics about the evolution of testing code produced with a given testing tool were computed for each project and averaged over the respective sets. In addition to its evolution, we measured the relevance of testing code with respect to the total production code for each project, in terms of quantitative comparisons of the respective amount of lines of code. To estimate the fragility issue, we defined a set of metrics that can be obtained for each project by automated inspection of the source code. Thus, we can give a characterization and a quantification of the average fragility occurrence for each of the testing tools considered.

The results of this paper provide a quantitative analysis of automated test suites developed with a selected set of GUI automation frameworks, for open-source Android applications. Since many GUI automation frameworks have been provided in recent literature for mobile applications, this paper does not provide a comprehensive evaluation of all available alternatives for Android developers to test their application. These metrics can be used as a benchmark by Android developers adopting those

[2]https://f-droid.org/en/

[3]https://github.com/k9mail/k-9

tools or similar ones, to evaluate the maintenance effort needed by their scripted test suites, and decide whether it is convenient to keep leveraging automated testing. Based on this fragility evaluation, it can be possible in the future to define a taxonomy of fragility causes for scripted Android GUI testing, and to give more in-depth actionable guidelines for developers to circumvent some of them. Finally, automated tools can be developed to adapt test methods to the modifications that are performed in the GUI source code and appearance.

The remainder of the manuscript is organized as follows: section II gives background information about Android apps and the components they are made of; a taxonomy of automated GUI testing tools for Android is provided, along with a brief literature review of studies addressing the complexity of Android GUI testing. Section III describes the study we conducted; the research questions are detailed and all the proposed metrics are described; we also give insights about the data extraction procedure we adopted, and about the testing tools we selected for our investigations. Section IV gives insights about the instruments and the procedure we used to conduct our study. Section V collects all the measures obtained for the considered projects, and discusses the findings that can be based upon them. Section VI lists all the possible threats to the validity of the conclusions presented in this paper. Section VII provides a conclusive higher-level discussion about the fragility issue and anticipates possible future work on the topic.

II. Background and Related Work

This section provides an introduction to Android programming and testing, and a survey of existing literature about Android testing and its challenges.

A. Android applications

A definition of *mobile apps* is given by Muccini et al. [7] as mobile software (i.e., applications that run on electronic devices that may move) that in addition to the user's input is also *context-aware*, i.e. it adapts and reacts to the context in which is run (for instance, performing sensing of the physical environment, and context-triggered actions).

A first classification of mobile apps may be made between *native*, *web-based* and *hybrid* applications. As explained by Kirubakaran et al. [9], native apps are the ones designed to be run on a particular mobile platform, following its design patterns and guidelines. Web-based apps are pretty similar in their nature to typical web applications, and are based on web sites that are engineered to be loaded by a browser on a mobile device. Hybrid apps use native code for specific platforms, to provide a client to the user and to access functionalities of the device they run on, but still their core logic is written as a web application, and is loaded dynamically at run-time.

The Android development platform, as described by Amalfitano et al. [10], is an infrastructure composed by four layers: the final applications, at the top of the stack; the Android Application Framework, providing the components with which the apps are built; the Static Library and the Android Runtime (including the Dalvik Virtual Machine, on which the apps are executed); the Linux kernel, in charge of abstracting the underlying hardware.

The Android Application Framework provides four basic components, that are the elements with which applications are constructed. *Activities* build the various components of the GUIs, defining their elements and handling the reponses that need to be triggered by different classes of user inputs (e.g., tactile, or vocal). *Services* manage long-running background operation, that need no interaction by the user, like the management of network connections. *Broadcast Receivers* listen to events that are launched by the Android system (e.g. incoming calls or low battery signal), and manage the way the application responds to them. *Content Providers* manage the data stored by the application, and allow to share data with other apps and access external data. Each component is characterized by a specific life cycle, and each transition between its states should be properly tested, in order to guarantee that no crashes or unexpected behaviours happen. Since the interactions with the GUI and the graphic elements are entirely managed by Activities, GUI testing for Android apps is strictly tied to Activity testing.

B. Testing Android apps

As done by Gao et al. [11], Mobile Testing can be defined as "testing activities for native and Web applications on mobile devices using well-defined software test methods and tools to ensure quality

in functions, behaviours, performance, and quality of service".

There are different levels of testing for Android applications, in addition to the traditional unit testing, integration testing, system testing and regression testing. Scopes that are specific to the mobile scenario must be considered. Kaur et al. [12] list, among the most prominent testing needs for mobile apps, compatibility testing (i.e., to ensure that the application works on different combinations of handheld models and OS versions), performance testing (i.e., to ensure that the application does not exploit any of the resources available, since they can be limited), and security testing (i.e., to ensure that no unauthorized use of data and capabilities of the handheld device is performed by the application). GUI testing is identified as a very prominent testing need for all mobile applications, since GUI malfunctions can hamper significantly the user experience provided by an app.

GUI testing of Android apps can be performed in various ways. The first and most immediate option is to describe and execute manual test cases on the GUI of the applications. Linares-Vásquez et al. [13], who conducted an empirical study in the field of performance testing, identified manual testing as the option preferred by developers, along with the examination of reports and feedbacks given by the users. However, as discussed by Kropp et al. [14], the manual execution of test cases – in addition to requiring significative effort from testers – is rarely exhaustive, error prone and not easily reproducible. On the other hand, automated GUI testing techniques may define sets of scripts to exercise exhaustively – in a quick and repeatable way- - all the main functionalities of a GUI. In addition to that, automated test scripts can be also used to test the presence of regressions, in the transition between two consecutive releases of an application.

Several approaches exist for automated GUI testing of Android applications. Linares-Vásquez et al. [15] give the following classificaton: Fuzzy (or random) testing, systematic exploration-based tesing, Capture & Replay, event-sequence generation, scripted testing. Most of those ways of testing Android apps allow to generate repeatable test scripts, in some cases without having the need of accessing the source code (i.e., only the .apk package of the application is needed). If the source code is available to the tester, it is also possible to perform script-based white box testing, as it is made possible by the Android Testing Framework.

Random and fuzzy testing techniques provide random sequences of inputs to the individual activities of the applications, in order to trigger potential defects and crashes. In their simplest forms – as in Monkey[4], the random tester supported by Android – no additional information about the AUT (i.e., Application Under Test) is required. More complex random testers (like the ones proposed by Machiry et al. [16], and Zhauniarovich et al. [17]) can leverage a model of the GUI, to distribute inputs in a more intelligent way, and create test cases that resemble typical user interactions.

Model-based testing techniques leverage models (typically, finite state machines, state charts or UML diagrams) of the GUI of the apps under test, that can be created manually or extracted automatically through a process named *GUI ripping*. Such models are therefore used to generate systematic test cases traversing the GUI. The tools and studies by Amalfitano et al. [18] [19] and Yang et al. [20] can serve as examples of this approach.

Capture & Replay testing tools (examples are presented in works by Kaasila et al. [21], Gomez et al. [22] and Liu et al. [23]) record the operations performed on the GUI to generate repeatable test scripts. Image recognition techniques [24] can be used to verify visual oracles for black-box test cases. Event-sequence generation tools are based on the construction of test cases as streams of events: examples of this paradigm are provided in works by Choi et al. [25] and Jensen et al [26].

Less coverage (examples are given in works by Kropp et al. [14] and Singh et al. [27]) is present in literature about white-box approaches and scripted GUI Automation Frameworks, which require the developer to manually select the sequences of operations to be performed on the AUT, identifying objects of the GUI through their definition and properties, and write test code accordingly.

C. Challenges for testing Android apps

Several studies (like the ones by Muccini et al. [7], Kirubakaran et al. [9] and Kaur et al. [12]) are focused on the peculiarities of Android apps that make testing them properly a complex

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

challenge: limited energy, memory and bandwidth; rapid changes of context and connectivity type; constant interruptions caused by system and communication events; the necessity to adapt the input interface to a wide set of different devices; very short time to market; very high amount of multitasking and communication with other apps.

Kochar et al. [5] underline the difference between mobile apps and typical desktop applications, and give a summary of the reasons why developers neglect testing: they find that time constraints, compatibility issues, complexity and lack of documentation of available tools are among the most relevant challenges experienced by the interviewed developers.

Vásquez et al., as a result of a set of interviews to contributors to open-source projects [6], found other relevant reasons that make mobile developers prefer manual to automated testing: costs in terms of money and time to maintain automated testing artifacts; time-related issues towards customers and project management decisions; general difficulties related to the usability of tools; size and maturity of open-source apps, that can be not big enough to justify the adoption of automated testing.

D. Fragility measurements

Testing fragility can be a problem for different kinds of software. In general, a test case is said to be fragile when it fails or needs maintenance due to the natural evolution of the AUT, while the specific functionalities it tests have not been altered. Investigations have been made in the field of web application testing, with effort from Leotta et al. aimed at comparing the robustness of capture-replay vs. programmable test cases [28], and of tests written using different locators for the components of the AUT [29]. A list of the possible causes of fragilities specific to GUI testing of mobile applications was proposed in our previous work [8]: identifier and text changes inside the visual hierarchy of activities; deletion or relocation of the elements of the GUI; usage of physical buttons; changes in the layout and graphic appearance, especially if visual recognition tools are used to provide oracles to test cases; adaptation to different hardware and device models; activity flow variations; execution time variability.

Modifications performed on test cases may be due to different reasons. Yusifoglu et al. [30] classify the modifications of test code under four categories: perfective maintenance, when test code is refactored to enhance its quality (e.g., to increase coverage or to adopt well-known test patterns); adaptive maintenance, to make the test code evolve according to the evolution of the production code; preventive maintenance, to change aspects of the code that may require intervention in future releases; corrective maintenance, to perform bug fixes. According to our definition of GUI testing fragility, we are interested in cases of adaptive maintenance.

To implement the classification of tests as fragile or not in an automated tool, we first assume – as it is commonly done for tests based on JUnit – that each test case is described by an individual test method, and we call a collection of test methods in a single Java file a *test class*. We consider any modified method inside a GUI test class as fragile. When a test class is modified, we consider it as non-fragile if there are no modified methods inside it; for instance, the modifications may involve only import statements and test constructors, or test methods may have been added or removed but not modified. We suppose, in fact, that the addition of a new method should reflect the introduction of new functionalities or new use cases to be tested in the application, and not the modification of existing elements of the already tested activities. On the other hand, if some lines of code inside a single test method had to be changed or added, it is more likely that tests had to be modified due to minor changes in the application and possibly in its GUI (e.g. modifications in the screen hierarchy and in the transitions between activities).

III. Study Design

We can describe the goals of this work as: (i) analyzing the adoption and usage of a set of popular GUI testing tools among open source Android applications; (ii) quantifying the amount of modifications that are performed on production code and on test code during the history of Android open source projects; (iii) giving a characterization of the fragility issue, and an estimation of its occurrence in a typical Android open-source project.

Based on these three goals, we can formulate the following research questions:

RQ1 *Adoption: What is the level of adoption of a given set of automated testing tools,*

among open-source Android projects?

RQ2 *Evolution: How much are GUI test classes associated with the analysed set of tools modified through consecutive releases of an open-source Android project?*

RQ3 *Fragility: How fragile are GUI test classes associated with the analysed set of tools to modifications performed on open-source Android projects and their graphical appearance?*

The first step of our research has been an estimation of the adoption of a set Android GUI testing frameworks among open-source Android projects. To do so, we mined from GitHub a set of Android applications (i.e., we identified projects characterized by the use of the "Android" keyword and by the typical structure of an Android project) featuring a history of releases, and cloned all of them locally. Then, we selected six testing tools used for GUI testing among the ones frequently cited in available literature, and perfomed code searches on the extracted set of projects, in order to detect and quantify their usage. The selected testing tools are described in section III.B, and the metrics used to answer RQ1 are defined in section III.A.1 as "Adoption and size" metrics.

Then, we studied the evolution of applications throughout their release history, both for the production code as a whole, and for the code pertaining to the six testing tools that we selected. We performed this study of modifications by means of automated file-by-file comparisons between consecutive releases. We tracked also the modifications of individual test classes and test methods, to compute a set of change indicators. These metrics, that allow us to answer RQ2 and RQ3, are defined respectively in section III.A.2 as "Test evolution" metrics and in section III.A.3 as "Fragility of test classes and methods" metrics. The procedure adopted to compute them is explained in detail in section IV.B and IV.C.

A. Metrics definition

To count and classify the modifications in test code, some change metrics have already been given in literature. For instance, Tang et al. [31] define a set of eighteen metrics, with the aim of describing bug-fixing change histories in source files. Tang et al. describe three different categories of metrics: size (e.g., added or removed lines of code, number of modified classes, files or methods); atomic (e.g., boolean values indicating whether a class features added methods); semantic (e.g., number of added or removed dependencies inside a file).

With this paper we introduce a set of 21 metrics, to give a characterization of the adoption of individual tools among Android open-source repositories, and to quantify the amount of modifications performed on the test cases featured by the projects. To the set of absolute metrics provided by Tang et al., our ones add the possibility – according to our definitions – of performing investigations about volatility and fragility of test classes and test methods, since they aim at capturing the weight that each individual modification in test classes or methods has if compared to the whole amount of test code of the application. Most of the metrics are normalized, to allow comparisons across projects of different sizes; they are normalized with respect to the size of the test suite, to the lifespan of individual test classes, or to the amount of changes performed to production code. The metrics we defined, as detailed later in the computation procedure, assume test classes and methods written in Java, so that a comparison between test code and production code is possible. Hence, the metrics are not applicable to quantify the amount of code and modifications of testing tools that produce test scripts or descriptions of the AUT to test using different languages.

The metrics we introduced can be defined as compound metrics based on change metrics that already exist in literature. For instance, considering again the metrics and nomenclature given by Tang et al., *Tdiff*, the amount of added, deleted or modified lines of code of files related to a specific testing tool, can be computed as the sum of *LA* (i.e., Lines of code Added) and *LD* (i.e., Lines of code Deleted) for the files that contain scripts written in that given tool; as well, *Pdiff*, the amount of added, deleted or modified lines of code on which we base several of our metrics, can be computed as the sum of *LA*, and *LD* for all the files of the release. Finally, our *MRTL* (i.e., Modified Relative Test LOCs) metric can be computed as the ratio between the two aforementioned sums.

The metrics can be subdivided in three groups, each one answering one of the research questions we formulated. Table I shows all the definitions of

TABLE I: Metrics definition

Group	Name	Explanation	Type	Range
Adoption and size (RQ1)	TA	Tool Adoption	Real	(0, 1)
	NTR	Number of Tagged Releases	Integer	[2, ∞)
	NTC	Number of Tool Classes	Integer	[1, ∞)
	TTL	Total Tool LOCs	Integer	[1, ∞)
Test evolution (RQ2)	TLR	Tool LOCs Ratio	Real	(0, 1]
	MTLR	Modified Tool LOCs Ratio	Real	[0, ∞)
	MRTL	Modified Relative Tool LOCs	Real	[0, 1]
	TMR	Tool Modification Relevance Ratio	Real	[0, ∞)
	MRR	Modified Releases Ratio	Real	[0, 1]
	TCV	Tool Class Volatility	Real	[0, 1]
	TSV	Tool Suite Volatility	Real	[0, 1]
	TJR	Tool Code to JUnit code Ratio	Real	[0, ∞)
	MTJR	Modifications of Tool code to JUnit code Ratio	Real	[0, ∞)
Fragility (RQ3)	MCR	Modified Tool Classes Ratio	Real	[0, 1]
	MMR	Modified Tool Methods Ratio	Real	[0, 1]
	FCR	Fragile Classes Ratio	Real	[0, 1]
	RFCR	Relative Fragile Classes Ratio	Real	[0, 1]
	FRR	Fragile Releases Ratio	Real	[0, 1]
	ADRR	Releases with Added-Deleted Methods Ratio	Real	[0, 1]
	TCFF	Tool Class Fragility Frequence	Real	[0, 1]
	TSF	Tool Suite Fragility	Real	[0, 1]

the metrics, their type and the ranges they belong to. The metrics are explained in much detail in the following.

1) Adoption and size (RQ1): To estimate the adoption of Android automated GUI testing tools among open-source projects and the size of test suites using them, we defined the following metrics:

TA (Tool Adoption) is defined as the percentage, among a set of projects, of those featuring test code written with a given testing tool. In the context of our experiment this ratio, expressed as a percentage, gives the percentage of Android applications whose GUI is automatically tested with the six analysed frameworks.

NTR (Number of Tagged Releases) is the number of tagged releases of a project (i.e., the ones that are listed by using the command *git tag* on the GIT repository).
In the context of our experiment, this metric can give an idea of which kinds of applications (whether small apps developed for immediate release and then abandoned, or long-lived projects) are most likely to have their GUIs tested with the six analysed GUI automation frameworks, and is used to identify the projects provided with a history of releases to be studied with the other metrics.

NTC (Number of Tool Classes) is the number of classes featured by a release of a project, featuring code relative to a specific tool. As discussed in section IV.A, in our experiment classes are associated with a given testing tool if they contain imports or method calls that are specific to the tool.

TTL (Total Tool LOCs) is the number of lines of code belonging to classes that can be attributed to a specific testing tool in a release of a project.
In the context of our experiment this metric, along with the previous one, allows us to quantify, inside Android projects, the absolute quantity of code that can be associated with a set of relevant testing frameworks that can be used for GUI testing.

2) Test evolution (RQ2): The metrics answering RQ2 aim to describe the evolution of open-source projects and of the respective test suites; they have been computed for each release, or for each couple of consecutive tagged releases.

TLR (Tool LOCs Ratio) defined as

$$TLR_i = TTL_i / Plocs_i$$

where $Plocs_i$ is the total amount of production LOCs for release i. This metric, lying in the $[0, 1]$ interval, allows to quantify the relevance of the testing code associated with a specific tool.

MTLR (Modified Tool LOCs Ratio) defined as:

$$MTLR_i = \frac{Tdiff_i}{Tlocs_{i-1}},$$

where $Tdiff_i$ is the amount of added, deleted or modified LOCs in classes that can be associated with a specific tool, between tagged releases $i-1$ and i. This quantifies the amount of changes performed on existing LOCs that can be associated with a given tool, for a specific release of a project. A value higher than 1 of this metric means that more lines are added, modified, or removed in test classes in the transition between two consecutive tagged releases, than the number of lines already featured by them.

MRTL (Modified Relative Tool LOCs) defined as:

$$MRTL_i = \frac{Tdiff_i}{Pdiff_i}$$

where $Tdiff_i$ and $Pdiff_i$ are respectively the amount of added, deleted or modified tool and production LOCs, in the transition between release $i-1$ and i. It is computed only for releases featuring code associated with a given testing tool (i.e., $TRL_i > 0$). This metric lies in the $[0, 1]$ range, and values close to 1 imply that a significant portion of the total code churn during the evolution of the application is needed to keep the test cases written with a specific tool up to date.

TMR (Tool Modification Relevance Ratio) defined as:

$$TMR_i = \frac{MRTL_i}{TLR_{i-i}}$$

This ratio can be used as an indicator of the portion of code churn needed to adapt classes relative to a given testing tool during the evolution of the application. It is computed only when $TLR_{i-1} > 0$. We consider a value greater than 1 of this metric as an index of greater effort needed in modifying the test code than the actual relevance of testing code, with respect to the modification of application code. On the other hand, we consider lower values of this indicator as an evidence of easier adaptability of code associated with a given testing tool to changes in the AUT.

MRR (Modified Releases Ratio), computed as the ratio between the number of tagged releases in which at least a class associated with a specific testing tool has been modified, and the total amount of tagged releases featuring classes associated with that tool. This metric lies in the range $[0, 1]$ and bigger values indicate a minor adaptability of the test suite (i.e., the set of test classes associated with a given testing tool) to changes in the AUT.

TCV (Tool Class Volatility), can be computed for each class associated with a given tool as

$$TCV_j = Mods_j / Lifespan_j,$$

where $Mods_j$ is the amount of releases in which the class j is modified, and $Lifespan_j$ is the number of releases of the application featuring the class j.

TSV (Tool Suite Volatility), is defined for each project as the ratio between the number of classes associated with a given tool that are modified at least once in their lifespan, and the total number of classes associated with that tool in the project history.

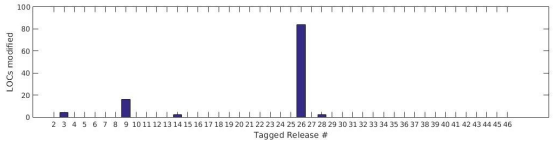
TJR (Tool Code to JUnit code Ratio), can be computed for each release of a project as:

$$TJR_i = \frac{Tlocs_i}{Jlocs_i},$$

where $Tlocs_i$ is the number of lines of code associated with the considered tool in release i, and $Jlocs_i$ is the number of lines of code associated with other JUnit tests in release i. The metric is not defined when $Jlocs_i = 0$, i.e. no JUnit test code is featured by release i. A high value for this metric implies that release i contains more code associated with the given tool than other generic testing code leveraging the JUnit framework.

MTJR (Modifications of Tool code to JUnit code Ratio), can be computed for each release of a project as:

Fig. 1: Modified test LOCs for class NotesScreenTest.java of project nhaarman/Triad



$$MTJR_i = \frac{Tdiff_i}{Jdiff_i},$$

where $Tdiff_i$ is the amount of added, deleted or modified code associated with a given tool between tagged releases $i-1$ and i, and $Jdiff_i$ is the amount of added, deleted or modified lines of code associated with other JUnit tests between tagged releases $i-1$ and i. The metric is not defined when $Jdiff_i = 0$, i.e. no lines of code in classes associated with JUnit are modified in the transition from release $i-1$ and i. A high value for this metric means that between releases $i-1$ and i more interventions were performed on code associated to the given tool than on other test code leveraging the JUnit framework.

As an example of the computation of TCV, figure 1 shows the number of LOCs modified in test class NotesScreenTest.java of the project nhaarman/Triad[5] (featuring the GUI testing tool Espresso), for each tagged release that features it. The test class is introduced in release 0.2.0 (number 2 in the history of the project) and is present until the master release (number 46 in the history). Hence, the lifespan of the class is 45. From the bar plot in Figure 1 it is evident that the test class has been modified five times during its lifespan. Thus, the volatility of the test class can be computed as $TCV = \frac{5}{45} = 0.11$.

The same reasoning can be repeated for all the test classes of the project featuring Espresso, obtaining the values shown in table II (for the sake of simplicity, we did not write the full path names of the classes in the first column). The final TCV value for the project, 0.07, is computed as the average of the volatilities of all the test classes. It is common that different test classes have the same number of modifications throughout their lifespans (and, as a consequence, the same value for the volatility metric): this is due to the fact that the same modifications in the AUT production code may lead to multiple test classes needing modifications in the same release.

[5]https://github.com/nhaarman/Triad

TABLE II: Volatility for individual classes of project Nhaarman/Triad

Class n.	First Release	Last release	# Mods	Lifespan	Volatility
1	1	6	1	6	0.1667
2	1	46	4	46	0.0869
3	2	46	5	45	0.1111
4	2	46	5	45	0.1111
5	2	46	5	45	0.1111
6	2	2	0	1	0
7	3	46	5	44	0.1135
8	28	46	0	19	0
9	28	46	0	19	0
10	28	46	0	19	0
11	28	46	1	19	0.0527

3) Fragility of test classes and methods (RQ3): With an automated inspection of test code, information about modified methods and classes can be obtained. Based on such data, the metrics answering RQ3 aim to give an approximated characterization of the fragility of test suites.

The number of modified classes with modified methods can be different from the total number of modified classes in three different cases (and their combinations): (i) when the modifications performed to the classes involve insignificant portions of code like comments, imports, declarations; (ii) when the modifications performed to the classes involve only additions of test methods; (iii) when the modifications performed to the classes involve only removal of test methods. Additions and removals of test methods are considered the consequence of a new functionality or a new use case of the application, hence they are not considered as an evidence of fragility of test classes. On the other hand, modified test methods may reflect changes in features of the application or in its GUI definition, and hence make the test class that contain them as fragile according to our definition.

MCR (Modified tool Classes Ratio) defined as

$$MCR_i = MC_i / NTC_{i-1},$$

where MC_i is the number of classes associated with a given testing tool that are modified in the transition between release $i-1$ and i, and NTC_{i-1} the number of classes associated with

the tool in release $i-1$ (the metric is not defined when $NTC_{i-1} = 0$). The metric lies in the $[0, 1]$ range: the larger the values of MCR, the less the classes are stable during the evolution of the app.

MMR (Modified tool Methods Ratio) defined as

$$MMR_i = MM_i / TM_{i-1},$$

where MM_i is the number of methods in classes associated with a given tool that are modified between releases $i-1$ and i, and TM_{i-1} is the total number of methods in classes associated with the tool in release $i-1$ (the metric is not defined when $TM_{i-1} = 0$). The metric lies in the $[0, 1]$ range: the larger the values of *MMR*, the less the methods are stable during the evolution of the app they test.

FCR (Fragile Classes Ratio) defined as

$$FCR_i = MCMM_i / NTC_{i-1},$$

where $MCMM_i$ is the number of classes associated with a given testing tool that are modified, and that feature at least one modified method between releases $i-1$ and 1. This metric represents an estimate of the percentage of fragile classes associated with the tool, upon the entire set of test classes featured by a tagged release of the project. The metric is upper-bounded by *MCR*, since by its definition $MCR_i = MC_i / TC_i$, and $MCMM_i \leq MC_i$.

RFCR (Relative Fragile Classes Ratio) defined as

$$RFCR_i = MCMM_i / MC_i,$$

where $MCMM_i$ and MC_i are defined as above.

FRR (Fragile Releases Ratio), computed as the ratio between the number of tagged releases featuring at least a fragile class among the ones associated with a given tool, and the total amount of tagged releases featuring test classes associated with such tool. This metric lies in the range $[0, 1]$ and is upper-bounded by *MRR*.

ADRR (Releases with Added-Deleted Methods Ratio), computed as the ratio between the number of tagged releases in which at least a method has been added or removed in classes associated with a given testing tool, and the total amount of tagged releases featuring test classes associated with the tool. This metric lies in the range $[0, 1]$, and according to our hypothesis higher values imply more frequent changes in application functionalities and defined use cases to be tested.

TCFF (i.e., Tool Class Fragility Frequence) defined as:

$$TCFF_j = FR_j / Lifespan_j,$$

where FR_j is the the amount of releases in which the class j, associated with a given testing tool, contains modified methods, and $Lifespan_j$ is the number of releases of the application featuring the class j. This metric is upper bounded by TCV, since by construction MR_j (the number of releases in which the class is modified) is higher or equal to FR_j.

TSF (Tool Suite Fragility), is defined for each project as the ratio between the number of classes associated with a given tool that feature fragilities at least once in their lifespan, and the total number of test classes associated with the tool in the project history.

For instance, in figure 2 the output of the Git Diff command for the test class TheFullScreenBarcodeActivity.java of the repository ligi/PassAndroid[6], featuring Espresso), between releases 3.2.0 and 3.2.1, is shown. The class is modified, but there are no modifications inside test methods. In fact, the only two lines modified are among import statements. Hence, this class counts as a modified class (thus counting for *MC*) but, since the *MM* metric for the class is equal to 0, it is not considered as a fragile class. Thus, it does not count for *MCMM*.

The sample in figure 3 is the diff for the test class ThePassEditActivity.java. In the class there are four rows modified inside three different test methods. In this case, the class counts for MC and also for $MCMM$, since $MM = 3$.

The third case that has to be considered is when methods are not modified, but instead added or removed. In such case the class still counts for *MC*, but not for *MCMM*.

In the case of the transition between release 3.2.0 and release 3.2.1, just three modified classes out of eight feature modified methods. Thus *RFCR* = *MCMM*/*MC* = 0.375. Comparing the number of

[6]https://github.com/ligi/passandroid

Fig. 2: Diff for test class TheFullScreenBarcodeActivity.java of ligi/passandroid (releases 3.2.0 - 3.2.1)

```
@@ -2,7 +2,7 @@ package org.ligi.passandroid;

 import android.graphics.Bitmap;
 import android.graphics.drawable.BitmapDrawable
     ;
-import
    android.test.sitebuilder.annotation.MediumTest;
+import android.suport.test.filters.MediumTest;
 import android.widget.ImageView;
 import butterknife.ButterKnife;
 import com.squareup.spoon.Spoon;
```

Fig. 3: Diff for test class ThePassEditActivity.java of ligi/passandroid (releases 3.2.0 - 3.2.1)

```
@@ -1,7 +1,7 @@
 package org.ligi.passandroid;

 import android.annotation.TargetApi;
-import android.test.suitebuilder.annotation;
.MediumTest;
+import android.support.test.filters.MediumTest;
 import com.squareup.spoon.Spoon;
 import javax.inject.Inject;
 import org.ligi.passandroid.model.PassStore;

@@ -43,6 +43,7 @@ public class The
    PassEditActivity extends BaseIntegration<
    PassEditActivity> {
 onView(withId(R.id.categoryView)).perform(click
     ());
+onView(wihText(R.string)
.select_category_dialog_title).perform(click());

@@ -53,6 +54,7 @@ public class The
    PassEditActivity extends BaseIntegration<
    PassEditActivity> {
 public void testSetToCouponWorks() {
  onView(withId(R.id.categoryView)).perform(click
      ());
+onView(withText(R.string)
.select_category_dialog_title).perform(click());
  onView(withText(R.string.category_coupon)).
      perform(click());
  assertThat(passStore.getCurrentPass().getType()
      ).isEqualTo(PassType.COUPON);

@@ -73,7 +75,7 @@ public class The
    PassEditActivity extends BaseIntegration<
    PassEditActivity> {
 public void testColorWheelIsThere() {
  onView(withId(R.id.categoryView)).perform(click
      ());
-onView(withText(R.string)
.button_text_change_color).perform(click());
+onView(withText(R.string)
.change_color_dialog_title).perform(click());

  onView(withId(R.id.colorPicker)).check(matches(
      isDisplayed()));
```

classes found as fragile to the total number of classes, we can obtain *FCR* = *MCMM*/*TC* = 3/10 = 0.3. Counting the total amount of methods that have been modified among all classes allow to obtain the *MMR* metric.

B. Selected Testing Tools

Since our objective was to document the evolution of testing code of Android open-source applications, we focused our study on a set of GUI Automation frameworks, that allow to write GUI-level tests through hand-written code. GUI Automation frameworks typically identify the elements of the GUI through their properties or the definition of the screens of the app (e.g., the Layout files in Android programming) and offer to the developers a set of functions that allow to perform actions on GUI components, in addition to assertion statements to verify the current state of the app.

We selected a set of open-source GUI Automation frameworks that have been cited in available literature. A selection criterion for the tools was the ability of producing test scripts in Java, since our metrics considered code comparison with production code of Android apps.

The resulting set of tools can be based on the review of existing Android testing tools provided by Linares-Vasquez et al. [32]: from the variety of GUI Automation Frameworks that are presented in such review, we kept the five ones that complied to our selection criterion, and added Selendroid to the set, since it is the adaptation of Selenium, a very widely used testing framework for web-applications.

The six selected tools cover together the principal peculiarities that can be attributed to GUI Automation frameworks, hence the results and discussion about each of them can be representative of other GUI Automation Frameworks with similar characteristics.

Table III summarizes the features provided by the selected tools. These characteristics partly reflect the ones listed by Linares-Vasquez et al. in their description of GUI testing frameworks [32], to which we added the support to image recognition. In the table, the columns are dedicated respectively to: the nature (black-box, or white-box) of test scripts produced; the ability to exercise non-native apps or hybrid apps in addition to native ones; the possibility of writing test cases spanning multiple

TABLE III: Characteristics of the selected GUI Testing Frameworks

Framework	Black Box	Non-native app testing	Multi-app	C&R	Multi-OS	Level	Image Recognition
Espresso	No	Partial	No	No	No	GUI-Level	No
UIAutomator	Supported	Partial	Supported	No	No	GUI-Level	No
Robolectric	No	No	No	No	No	Unit-Level	No
Robotium	Yes	Supported	No	Supported	No	Unit-Level	No
Selendroid	Yes	Supported	No	No	No	GUI-Level	No
Appium	Yes	Supported	No	Supported	Yes	GUI-Level	Supported

applications or the GUI of the operating system; the support to the creation of test scripts through capture and replay, in addition to full manual scripting; the support to operating systems other than Android; the applicability to unit testing or to GUI testing only; the support to image recognition to identify elements of the interface, as done by novel Visual GUI Testing approaches [24].

The selected frameworks, along with their most relevant aspects and literature proposing evaluations or other tools based on them, are better detailed in the following.

- **Espresso**: an automation framework that allows to test the GUI of a single app, leveraging a grey-box approach (i.e., the developer has to know the internal disposition of elements inside the view tree of the app, to write scripts exercising them). Espresso tests are developed inside the app project, so with full access to the code of the Activities, giving higher control of the tested functionalities with respect to tests only accessing the graphical elements exposed to the final users. The framework is designed specifically to test one activity at a time, and it is not possible to interact to the GUI of the underlying operating system. It could originally test only native applications, but support is given to the test of hybrid applications through the EspressoWeb extension[7]. Espresso has an internal synchronization mechanism that manages the GUI, thus allowing tests to be written without the need for polling or waiting mechanisms [33]. It is part of the official Android Instrumentation Framework, as the suggested way to test the GUI of an app in isolation from the GUI of the operating system[8]. Several tools proposed in literature, like RacerDriver [34] or Barista [35] leverage the Espresso framework for GUI automation.
- **UIAutomator**: it is available only since Android API 16, and adds to Espresso the possibility to check the device status and performance, to perform testing on multiple applications at the same time, to perform operations on the system GUI and on the device (e.g., turning on WiFi or changing display settings). It can be used to test native and hybrid apps through a support for WebView testing. As opposed to Espresso, tests written in UIAutomator are not based on the application implementation but on the GUI objects that are exposed by the app, hence they enable black-box testing of app GUIs. UIAutomator is part of the official Android Instrumentation Framework, as the suggested way to test the GUI of multiple apps[9]. Some tools proposed in literature, like T+ [15] or Fusion [36], use the APIs exposed by UIAutomator as a basis to develop more sophisticated approaches to Android GUI testing.
- **Selendroid**: a testing framework based on Selenium, that allows to test the GUI of native, hybrid and web-based applications [37]; the tool allows to retrieve elements of the application and to inspect the current state of the app's GUI without having access to its source code, hence enabling the execution of black box tests, and to execute test cases on multiple devices at the same time. It is based on the Android Instrumentation Framework

[7]https://developer.android.com/training/testing/espresso/web.html

[8]https://developer.android.com/training/testing/ui-testing/espresso-testing.html

[9]https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html

to instrument the application to test. Having full integration with the Selenium WebDriver [10] framework, it can be used also to test web apps. Testing multiple application at the same time is not possible, since each application to test requires a running instance of the Selendroid server. Segen [38] is an example of testing tool leveraging Selendroid for test creation.

- **Appium**: it leverages WebDriver and Selendroid for the creation of black-box test methods that can be run on multiple platforms (e.g., Android and iOS) [39]. Test cases can be created either through manual scripting or via an inspector that enables basic functions of recording and playback. Integrations with image recognition libraries (e.g., SikuliX[11]) are possible to perform Visual GUI testing through image recognition. It can be used to test native apps, hybrid apps and web-based applications accessed through a mobile browser. Test scripts can be data-driven and can be written in multiple scripting languages (e.g., Python or C#) in addition to Java. Several studies on Android testing are performed using this tool, like it is done by Shah et al. [40] and Singh et al. [27].
- **Robolectric**: cited, among others, by Amalfitano et al. [41], Milano et al. [42], and Mirzaei et al. [43], it is an example of unit testing tool for Android, that can be used to perform testing directly on the Java Virtual Machine, without the use of a real device or an emulator. Robolectric does not render the GUI, and all assertions are at code-level, so it can be used only for white-box tests that are not testing the actual appearance of the app, as it is shown to the user. Emulation is mandatory if the functionalities to be tested pertain the interaction of the application with the full Android environment. Robolectric can be considered an enabler of Test-Driven Development for Android applications, since the instrumentation of Android emulators is significantly slower than the direct execution of test cases on JVM. Several studies on Android testing are performed using Robolectric, like it is done by Sadeh et al. [44] and Allevato et al. [45].
- **Robotium**: an extension of JUnit for the testing of Android apps, that has been one of the prominent testing tools since the inception of Android programming [46][47]. It can be used to write black-box test scripts or function tests (if the source code is available) of both native and web-based apps. Record and playback functionalities are made available by the Robotium Recorder extension. Being the test code coupled to the package of the application under test – as with Espresso – Robotium allows to test a single application at a time. Robotium is used as an automation frameworks by Amalfitano et al. for A2T2 [48] and AndroidRipper [41], and by Liu et al. [23].

Several alternative closed-source and/or commercial GUI Instrumentation Frameworks are available, with characteristics that can be considered similar to the ones exposed by the discussed six tools. Ranorex[12] is a cross-platform test automation framework for black-box testing of desktop, web and mobile applications; similar to Appium, it leverages an inspector that allows to create object-based test cases, creating test scripts in C# or VB.NET. Quantum Automation Framework[13] extends the capabilities of the tools of the Android Instrumentation Framework with the possibility of a cloud-based execution of test cases on multiple devices at the same time. At a similar level of abstraction of the tools of the Android Instrumentation Framework, Calabash[14] focuses on the definition of test scripts for Android and iOS apps through the simulation of interactions with emulated devices and the evaluation of assertions about the appearance of the application and its contents; test scripts can be written using natural language with Cucumber, or in Ruby.

C. Instruments

This section describes the tools and scripts that have been used to extract the set of projects on which we conducted our study, and to collect statistics about them.

[10] http://www.seleniumhq.org/projects/webdriver/
[11] http://sikulix.com/
[12] https://www.ranorex.com/
[13] http://projectquantum.io/
[14] http://calaba.sh/

1) Data extraction from GitHub: We decided to perform a scripted search on the GitHub database, leveraging the GitHub APIs.

To know how many Android projects are available on the GitHub database, a repository search can be made using the GitHub Repository Search API: it allows to extract all projects containing a certain word (that would be "Android" in our case, case-insensitive) in their names, readme files or descriptions. It is also possible (by setting the "language" parameter) to limit the search of the keyword inside files of a certain type. To use the GitHub API we have used the bash cURL function, inside a bash script. The output, that is obtained in the form of a Json file, has been examined using the jsawk tool[15].

The "created" parameter allows to filter out the results of the GitHub Repository Search, selecting only the projects that have been created in a certain time interval. This parameter comes in handy because the query returns the details of up to 1000 projects. Therefore, date ranges have to be provided in order to limit, under 1000 units, the results of every single search performed.

2) Git Code Search: The GitHub Code Search API allows to search for particular keywords inside a given project. The search can be parametrized using the "filename" parameter, which constrains the search only in files named as indicated (if the parameter is omitted, the keyword is searched in all the files of the repository). The "filename" parameter can also be leveraged to search for the presence of files named in a specific way inside a repository, regardless of the code they contain. The "repo" parameter is used to specify the repository in which the search has to be performed.

Some limitations apply to the GitHub Code Search API, as explained in the Git Documentation: (i) only the default branch (in most cases the master branch) is considered for the code search; thus, if tests are present in older releases but are removed in the master branch, the project will not be extracted; (ii) only files smaller than 384kb are searchable; (iii) only repositories with fewer than 500,000 files are searchable. The second and third issues may be not very relevant in our context, since the size of projects and files considered is typically not so big (with the exception of projects containing whole firmwares, or clones of the Android Operating System).

3) Count of lines of code: We used the open-source cloc[16] tool to count the total lines of code inside a repository (or, in general, a set of files).

To compute the number of modifications performed to files of a GitHub project, the *git diff* command is used, to obtain all the modified, added and removed lines of code between the two releases considered. By default, the diff command shows the modifications performed to the whole repository; as an alternative, it is possible to specify the full paths of a file for both the releases, to obtain the modifications that were performed only on it.

The -M parameter allows to identify files that have been renamed or moved in the transition between the releases, without considering such operation as the combination of a deletion and an addition of a file. The git diff command takes into account also blank lines and rows of comments inside files.

4) Java class parser: We developed an automated Java class examiner, in order to track the modifications not only in whole files, but also in individual test methods. We based our examiner on JavaParser, an open source tool (available on GitHub[17]) that can be used to explore the structure of any Java application. We developed a console tool that, given two releases of the same test class as parameters: (i) extracts all methods declared in both the releases; (ii) identifies all methods that have been removed in the transition between releases; (iii) identifies all methods that have been added in the transition between releases; (iv) inspects the diff file computed for the release transition, and for each modified line checks whether it belongs to an existing method, to state -according to the heuristic explained in section II.D- if the class is fragile or not. The examiner cycles on all the tagged releases of any Git Repository. The list of tagged releases can be obtained using the *git tag* command, inside the Git folder in which the project has been cloned.

A final integrated script has been developed, using the instruments detailed in the previous paragraph, in order to: (i) cycle over all the projects of the context, and over all the tagged releases for each project; (ii) compute the modifications

[15]https://github.com/micha/jsawk

[16]http://cloc.sourceforge.net/

[17]https://github.com/javaparser/javaparser

and fragility metrics for each release; (iii) obtain averaged results for each project and for the entire sets.

IV. Adopted Procedure

The following paragraphs describe the operations that have been performed, with the aid of the instruments described in the previous section, to obtain the results presented in this paper.

A. Test Code Search (RQ1)

The first operation performed to conduct our study was a definition of our context, i.e. the set of projects that were used for the subsequent investigations. We performed three different steps to extract the set of projects used as our context, the first one being a search for the word "Android" in descriptions, readme files and names of Java projects hosted on GitHub. GitHub Search API has been leveraged to this purpose, using the following search string:

```
curl -x, -u $USER:$PASSWORD -H 'Accept:
    application/vnd.github.v3.text-match+
    json' 'https://api.github.com/search/
    repositories?q=android+language:java+
    created:"'$CURR_DATE_RANGE'"&sort=
    stars&order=desc&page='$CURRENT_PAGE''
```

This way, we gathered a total of 280,447 GitHub repositories.

We then applied a filter to cut out from the context all the projects that have no tagged releases. This is done because the aim of the study is to track the evolution of the considered projects, and – as it is detailed later – differences between tagged releases are computed. Considering that, projects without at least one tagged release (which allows for a single comparison, made between itself and the master release) are not of interest. To find how many tagged releases are featured by a project, the *git tag* command is used. This way, we obtained a set of 20,638 Android projects with a history of tagged releases.

Looking only for the keyword "Android" would have included in the results also libraries, utilities and applications for other systems that are engineered to interact with Android counterparts. Therefore, a method was needed to filter out those spurious results from the selected context. As it is done by Das et al. [49], we considered the presence (or absence) of Manifest files as a discriminant between true Android projects and false positives of the search procedure. As it is explained in the official Android developer's guide[18], in fact, it is mandatory for any Android app to have an AndroidManifest.xml file in the root directory of each of its builds. The Manifest file provides essential information about the app to the Android system, and the system needs it before running any of the app's code. Projects that do not contain any Manifest file are cut out from our context, since they are not likely to contain Android apps. The presence of multiple Manifest files in a single GitHub project may imply the fact that the project contains actually a set of Android apps and/or that multiple alternative builds are provided for a single Android app. Since the metrics are defined on GitHub projects and not strictly on individual apps, these cases are still considered as single projects in the following.

To search for Manifest files we leveraged the GitHub Code Search API on the projects that were still part of the context, providing "AndroidManifest.xml" as "filename" parameter, as in the following search string:

```
curl -x, -u $USER:$PASSWORD -H 'Accept:
    application/vnd.github.v3.text-match+
    json' 'https://api.github.com/search/
    code?q=manifest+filename:
    AndroidManifest.xml+repo:ligi/
    passandroid' | jsawk 'return this.
    items' | jsawk 'return this.path'
```

This way, we obtained a final set of 18,930 Android projects with tagged releases and Manifest files. The mentioned limitations of the GitHub code search API about the maximum total number of files (500.000) and the maximum size of a searchable file (384 kilobytes) proved to be not a real concern for our study. We counted the number of total files and the size of all Java classes in the mined Android projects, and found – as it was expected, being projects for Android apps typically not very large-sized – that no project had more than 500.000 files. Only 31 projects featured Java classes whose size exceeded the size threshold for the code search API. The projects of the studied contetx featured on average 419 files, and Java classes had an average size of 10.13 kilobytes.

[18] https://developer.android.com/guide/topics/manifest/manifest-intro.html

To search for any of the testing tools considered, a *GIT Code Search* has been performed on the repositories that are part of the context. The names of the tools themselves are evidence of their usage, since they are part of include statements that are needed to make them work. An exception has been made for Espresso, for which the presence of "test.espresso" among include statements has been searched for (being the term "espresso" a common word found in a number of applications having no connection at all with app testing). Projects were then divided into six sets, based on to the tools they featured. For each of the tools, its adoption has been estimated computing the *TA* (Tool Adoption) metric. Sets of projects featuring different tools are not necessarily disjoint: it is possible that a repository features more than just one scripted GUI testing tool.

Any Java class featuring a keyword related to a given testing tool in its code is considered as a class associated with the tool (for instance, a class featuring the statement "import static android.support.test.espresso.Espresso.onView" is considered as a class featuring Espresso). If a Java class contains code related to more than just a single testing tool, it is considered as associated with each of them; the projects is then inserted in all the sets pertaining the testing tools with which the class is associated. For each test class the lines of code are counted with the use of the *cloc* Bash tool, and contribute to the computation of the Size metrics defined to answer RQ1. *TTL* (Total Tool LOCs) and *NTC* (Number of Tool Classes) have been computed for each project, on the master release. As discussed earlier, the use of the *git tag* command also allows to obtain the *NTR* (Number of Tagged Releases) metric for any of the considered projects.

A search was also made on projects for the presence of JUnit test code. JUnit is a test automation framework that, most typically, is used to perform unit testing of Java applications. However, it can be used also as an engine for various testing tools and testing levels, ranging from integration to system level testing (for instance, the testing frameworks of the Android Instrumentation Framework are themselves based on the JUnit engine). Being JUnit among the most widespread testing tools for Java applications, the presence of generic JUnit test classes can serve as a first comparison for the adoption and code churn exposed by the studied testing frameworks.

Recent releases of the Android Studio IDE add by default a file called "ExampleUnitTest.java" to new projects, so as a consequence the amount of projects containing JUnit classes increased remakably in newer projects even though they did not contain actual test code written by the developers. Since those projects are clearly not significant to our purposes – the presence of that particular class is not an evidence of a testing process – the context has been pruned of all the projects featuring only that single test class. Even though some of the chosen tools are based on JUnit, the searches for the individual keywords and the "junit" keyword have been conducted independently. Obviously, if a tool is based on JUnit, the set of projects featuring JUnit will be a superset of the set of projects featuring that specific tool.

For further investigations, subsets of the complete set of projects were extracted. In particular, we focused on four different subsets of the whole context: apps with more than a thousand LOCs, apps that have more than five tagged releases and feature at least 10% of testing code code, apps that are released on Play Store, and apps that have been modified during last year. The first two sets can be obtained by simply examining the values extracted for any project during the computation of basic metrics. To find which apps were actually released on the Play Store, a manual search was performed for the names of the applications on the Play Store search[19].

Finally, to see which of the considered projects were modified recently, the *GitHub Stats API* has been used. In particular, the request "GET /repos/:owner/:repo/stats/commit_activity" returns the commit activity of last year, giving the number of total commits per week; summing the values over the year gives as result the total number of commits performed during last year (different time intervals could also be considered by taking into account only the values for a number of most recent weeks). Thus, projects with a value different from 0 were tagged as part of the subsets of "alive" projects.

The data extraction procedure from GitHub has been completed between September and December 2016.

[19]https://play.google.com/store/search

B. Test LOCs analysis (RQ2)

In the exploration of the history of Android repositories, the versions that have been considered for tracking the evolution of test classes are the tagged points of release histories. In addition to those, that can be extracted using the *git tag* command, the current master branches of projects have been considered, as the last updates of the repositories with which the last code comparisons are performed.

To answer RQ2, for each pair of consecutive versions of the selected projects, the *git diff* command has been executed on the whole repository to obtain the total amount of Java LOCs changed with respect to the previous release. Then, the *git diff* command has been used again to obtain the number of LOCs added, removed or modified for each .java file previously associated with a testing framework. The values extracted this way allowed us to compute TLR (Tool LOCs Ratio), $MTLR$ (Modified Tool LOCs Ratio), $MRTL$ (Modified Relative Tool LOCs) and TMR (Tool Modification Relevance Ratio) for each test tagged release of any project.

Then, global average values have been computed on the whole lifespans of the projects, using the formulas $\overline{TLR} = Avg_i\{TLR_i\}$, $\overline{MTLR} = Avg_i\{MTLR_i\}$, $\overline{MRTL} = Avg_i\{MRTL_i\}$, $\overline{TMR} = Avg_i\{TMR_i\}$ with $i \in [1, NTR]$ being NTR the number of tagged releases featured by the project.

To compute the amount of LOCs and modifications of other test code based on JUnit inside each project, the files featuring the "junit" keyword only (and not the ones containing keywords relative to the current testing tool under inspection, to avoid considering the same classes if the tool is based on the JUnit framework) have been considered as JUnit test files, and the *git diff* command has been used to track also their evolution. This way, the TJR (Tool Code to JUnit Code Ratio) and $MTJR$ (Modifications of Tool code to JUnit code Ratio) metrics could be computed for each tagged release. Finally, averaged values have been computed for them as $\overline{TJR} = Avg_i\{TJR_i\}$ and $\overline{MTJR} = Avg_i\{MTJR_i\}$.

For each project, the lifespan and volatility (TCV_j) have been computed for each test class, and an overall average has been computed as $\overline{TCV} = Avg_j\{TCV_j\}$ with $j \in [1, NTC]$, being NTC the number of test classes of the project. Throughout all our study, we have considered moved or renamed files as different test classes. The analysis of the test classes for whom TCV is not equal to zero allows to compute TSV (Test Suite Volatility) for any project.

C. Test classes history tracking, Fragility (RQ3)

We have finally tracked the evolution of single test classes and methods, taking into account the tagged releases in which each test class and test method has been added, modified or deleted.

Then, for each tagged release we have obtained the number of modified classes and methods, i.e. MCR (Modified Classes Ratio) and MMR (Modified Methods Ratio), and the derived metrics $RFCR$ (Relative Fragile Classes Ratio) and FCR (Fragile Classes Ratio). Also in this case, at the end of the exploration averages have been computed as $\overline{MCR} = Avg_i\{MCR_i\}$, $\overline{MMR} = Avg_i\{MMR_i\}$, $\overline{FCR} = Avg_i\{FCR_i\}$, with $i \in [1, NTR]$.

Since $RFCR$ makes sense only when modifications are actually present, $\overline{RFCR}$ has been computed as an average of $RFCR$ only for release transitions in which test classes have been modified (i.e., $MCR \neq 0$).

At the end of the exploration of the tagged releases of each project, ***FRR*** (Fragile Releases Ratio) and ***ADRR*** (Releases with Added-Deleted Methods Ratio) have been computed to quantify the percentage of them featuring, respectively, fragile and non-fragile modifications.

Based on the recognition of classes affected by fragilities, the ***TSF*** (Test Suite Fragility) overall value has been computed for each project.

Finally, a manual inspection of a set of modified test classes with modified methods has been conducted, in order to identify the reasons behind fragilities in test classes and methods, quantified by MMR and FCR, and link them to modifications in the GUI appearance and/or definition.

V. Results and Discussion

In the following paragraphs, we report the results we obtained by applying the described procedure. Each of the following subsections concerns one of the three research questions we defined. The results measured for the metrics defined in section III.A

TABLE IV: Number of projects and *TA* per testing tool.

Tool	Total	W. releases	W. Manifest(s)	TA
Android	280,447	20,638	18,930	-
Espresso	2,617	426	423	2.23%
UI Automator	846	154	107	0.57%
Selendroid	56	9	6	0.03%
Robotium	1,643	163	150	0.79%
Robolectric	3,767	875	842	4.44%
Appium	276	29	18	0.09%
JUnit	22,939	4,253	3,669	19.38%

TABLE V: *NTR*, *NTC*, *TTL*, *TLR* per testing tool: average and median (in parentheses) values for master release.

Tool	*NTR*		*NTC*		*TTL*		*TLR*	
Espresso	15	(6)	5	(2)	588	(190)	8.8%	(4.1%)
UIAutomator	60	(25)	12	(3)	3,155	(1,134)	8.6%	(0.6%)
Selendroid	46	(17)	76	(1)	8,627	(126)	19.4%	(0.2%)
Robotium	44	(7)	5	(1)	873	(227)	8.7%	(3.3%)
Robolectric	22	(6)	11	(3)	1,448	(399)	16.4%	(11.4%)
Appium	27	(15)	38	(4)	4,469	(1,096)	37.3%	(6.0%)
Average	25		9		1,338		13.3%	

TABLE VI: Number of released projects, projects with more than 1000 LOCs, and projects active during last year

	Espresso	UIAutomator	Selendroid	Robotium	Robolectric	Appium
Apps on Play Store	116 (27.42%)	16 (14.95%)	3 (50.00%)	32 (23.19%)	128 (15.20%)	8 (44.44%)
Apps with 1000+ LOCs	368 (89.10%)	105 (98.13%)	6 (100.00%)	138 (92.00%)	720 (85.51%)	15 (98.13%)
Apps modified last year	290 (68.56%)	55 (51.40%)	5 (83.33%)	55 (36.42%)	490 (58.19%)	12 (66.67%)

are detailed, along with the conclusions we can base on them.

The detailed measurements extracted for all the examined projects have been published as a dataset hosted on FigShare[20]. For each testing tool, we have created two different .csv files, one pertaining all releases of each project (containing their amount of production code, test code and modified lines, classes and methods) and one pertaining all classes of each project, and their evolution throughout the release history. The average values that are computed – as explained in the Procedure section – are based on this raw data.

TABLE VII: Metrics pertaining RQ1

Name	Explanation
TA	Tool Adoption
NTR	Number of Tagged Releases
NTC	Number of Tool Classes
TTL	Total Tool LOCs

A. RQ1: Adoption

We initially gathered a total of 280,447 GitHub repositories featuring the term *Android* in their names, descriptions or readme files. Then, a significant amount of projects were pruned because of the lack of tagged releases (so they had no history to be investigated), and because of the lack of Manifest files. A final set of 18,930 Android projects was

[20]https://figshare.com/articles/Testing_Fragility_data/4595362

obtained.

In tables IV and V the measures answering the metrics pertaining RQ1 are shown. A summary of the definitions of the metrics is given in table VII. The columns of table IV show, respectively: the total number of projects featuring each of the six tools considered; the number of projects featuring at least one tagged release; the projects featuring at least a Manifest file; finally, the Tool Adoption (i.e., TA) metric, computed for each of the selected GUI testing frameworks as the ratio between the amount of projects featuring it and provided with manifest file(s), and the total amount of projects of the context provided with manifest file(s). Table V shows the average and median values for Number of Tagged Releases (NTR), Number of Tool Classes (NTC), Total Tool LOCs (TTL) and Tool LOCs Ratio (TLR), computed on the sets of projects featuring each testing tool, for their master release. The last row in table V shows average values for all the considered projects, weighted by the number of projects for each set.

Considering the overestimation due to possible overlaps (since a single project can feature multiple testing tools, hence the sets for the individual tools are not necessarily disjoint) about 8.5% of the projects feature tests belonging to one of the six selected tools. None of the testing frameworks reached by itself an important level of adoption in the considered set of Android open-source projects. In particular, the absolute number of projects featuring Selendroid and Appium test cases, respectively 6 and 18, is practically irrelevant. A higher number (the 4.44% of the total) of projects featuring Robolectric has been found, but the tool has been available for a longer time with respect to other ones (especially Espresso and UI Automator) and is often used also for Unit Testing.

As a comparison for the adoption of the selected GUI testing frameworks, we counted also the number of projects featuring the JUnit testing framework, which can be used for unit/component testing as well as an enabling engine for other forms of testing. We counted 3,669 projects (with tagged releases and manifest files) featuring JUnit, among the total set of open-source Android projects we extracted (the 19.38%). Even though this percentage is higher than the ones obtained for the individual testing frameworks, it shows that also the JUnit test automation framework has a limited adoption among Android open source projects hosted on GitHub, being featured by about one fifth of the set.

Even though the total number of Android projects extracted can take into account some projects that are not likely to feature test classes (e.g. experiments, duplicates, exercises, prototypes, projects that are abandoned at very early stages) the measures computed for the metric TA give evidence of the lack of an extensive usage of scripted automated GUI testing on open-source Android projects. Anyhow, it is worth highlighting that the study we performed is limited to the testing tools we considered, i.e. it is possible that different scripted testing tools are used by some other projects of the context.

The average and median number of test classes in the sets of projects can be quite small (e.g., just 5 and 2, respectively, in the case of Espresso) due to the typical coding patterns for Android applications, in which – usually – one GUI test class is written specifically for each Activity featured by the application. Most applications – especially in the case of small and even experimental open-source projects – do not feature many screens to be shown to their users, and therefore they do not feature many activities to be tested. The projects of the mined context featured an average of 19 Activities defined in their Manifest files, with averages on the individual sets ranging from 10 (for the set of projects featuring Robolectric) to 116 (for the set of projects featuring UIAutomator).

Average TTL and TLR values are very large for the sets of projects featuring both Selendroid and Appium; however, such result is heavily influenced by the small size of the sets (respectively, 6 and 18 projects) and by the presence of the full Selendroid framework for Android[21], counting 47,436 tool LOCs, and of a very big set of Appium API demos[22], counting 48,868 tool LOCs. The influence of those individual projects on the average values is confirmed by the largely smaller corresponding median values.

The fact that the set of projects featuring Espresso has the lowest average TTL can be explained with the following reasons: (i) using a white-box testing technique allows to exercise the functionalities of the application with little coding

[21] https://github.com/selendroid/selendroid

[22] https://github.com/appium/android-apidemos

effort; (ii) the framework is quite accessible even to non-experienced developers, and its usage is encouraged by Android, leading it to be used also in very small projects, in tryouts, and even for experimental and partial coverage of applications use cases. On the other hand, the mean TTL for projects featuring UI Automator is very high, and also significantly higher with respect to the sets featuring Robotium, Robolectric and Espresso. This may reflect an higher complexity in scripting with the UIAutomator testing framework, but can also be due to the cross-application features of UI Automator, which make it suitable for the testing of whole firmwares and application bases, which are typically very big projects requiring a higher number of test cases. An evidence of the bigger size of projects featuring UIAutomator can be deduced from the selected data in terms of the average amount of Production LOCs per project: the set of projects with code associated with UIAutomator feature an average of 489,768 total LOCs, a number nearly ten times as big as the average computed on the master releases of all the projects of the context, amounting at 61,050.

The different size of the projects in which Espresso and UIAutomator are typically used is confirmed by the close average TLR values the sets of projects have, while the respective average TTL values are very different (with the TTL computed for UIAutomator nearly six times as big as the one computed for Espresso). Slightly bigger test suites, with respect to those developed with Espresso, are developed on average with Robotium and Robolectric. The projects featuring Robolectric have an average TLR value that is almost double than the one relative to the projects featuring Espresso: this can be due to an intrinsic additional complexity of the testing tool, or to the fact that it is not exclusively used for GUI testing.

The considered GUI testing frameworks reach, individually, a level of adoption that is always lower than 4.5%. Projects that have their GUI tested with the studied testing frameworks feature on average 9 test classes, with an average total number of of 1,338 LOCs associated with the considered testing frameworks (13.3% of the whole project code).

We also extracted some subsets of the complete context of open-source Android projects hosted on GitHub: we counted how many of them were uploaded on the Play Store to be purchased (or downloaded for free) by the final users, how many featured a relevant (more than 1000) amount of lines of code, how many underwent modifications during last year (the research has been conducted at April 2017). The numbers of projects satisfying those requirements are shown in table VI.

The number of projects that have been published on the Play Store gives an indication about how much the considered context of open-source GitHub projects is representative of actually released Android apps. Without considering the sets of projects featuring Appium and Selendroid (since they are particularly small), the sets of projects featuring Espresso and Robotium are the ones with most appearances on the Play Store. This can be justified by the nature of the Espresso and Robotium scripted testing frameworks, that are specifically designed for testing single activities and actual screens of individual applications; hence, they are more suitable for testing real, finalized apps that are ready for being released to the users. The percentage is lower for the projects featuring UIAutomator and Robolectric. For the former ones, the fact can be justified by the cross-application testing features of UIAutomator: most of the projects featuring UIAutomator are whole firmwares or sets of applications, and hence they are not likely to be found on the Play Store; for what concerns Robolectric, the instrumentation characteristics of the tool make it particularly suitable for testing low level SDKs, graphic libraries and projects providing widgets to be used in other applications.

However, it must be considered that there are cases of full and non-trivial native applications that are uploaded to alternative markets to the official Play Store (e.g., F-Droid), specifically designed to host open-source applications only. Some other applications (e.g., the GitHub repository medic/medic-gateway) are full and working applications, but are not released on any market and are made available through APK download only.

For what concerns the study of applications having a significant amount of lines of code, it can be seen that most of the projects featured more than 1000 LOCs. Only the sets featuring Espresso and Robolectric have less than 90% of

the applications featuring more than 1000 LOCs (this is another confirmation of the smaller size of projects typically tested with those two tools, with respect, for instance, to UIAutomator).

In general, the majority of applications have been modified in the last 12 months. The only exception is given by the set of projects featuring Robotium: it can be justified by the switch of the community to newer testing tools, like Espresso and UIAutomator, that are part of the official Android Testing Framework.

B. RQ2 - Evolution

Table VIII shows the statistics collected about the average evolution of test code, for the six selected testing frameworks. A summary of the definitions of the metrics is given in table IX. For every set, $\overline{TLR}$, $\overline{MTLR}$, $\overline{MRTL}$, $\overline{TMR}$, MMR, TCV and TSV have been averaged on all the projects. The values in last row are obtained as averages of the six values above, weighted by the size of the six sets.

The values reported for average Tool LOCs Ratio ($\overline{TLR}$) show that – when present – the amount of testing code associated with the selected testing frameworks can be an important portion of the project during its lifecycle, if compared to the number of LOCs of production code. The boxplots in Figure 4 show the distribution of $\overline{TLR}$ values for the six sets of projects. The average values range from about 7.3% (for the set of Espresso projects) to 31.9% (for the set of Appium projects). For the biggest set of projects considered (those featuring Robolectric) the mean $\overline{TLR}$ is 13.4%. The $\overline{TLR}$ averaged over the releases of applications is typically smaller than the TLR computed for master releases (see table V): this may be attributable to the graduality of the construction of test suites, which may be very small or absent in intial releases.

Average Modified Tool LOCs Ratio ($\overline{MTLR}$) measures show that typically around 2.8% of test code is modified between consecutive releases of the projects featuring the six analysed GUI automation frameworks. Very small $\overline{MTLR}$ values were obtained for the projects featuring UIAutomator. In general, this should be considered as a consequence of bigger test suites, in terms of absolute LOCs, with respect to the ones written with other testing frameworks. Hence, the influence of a similar

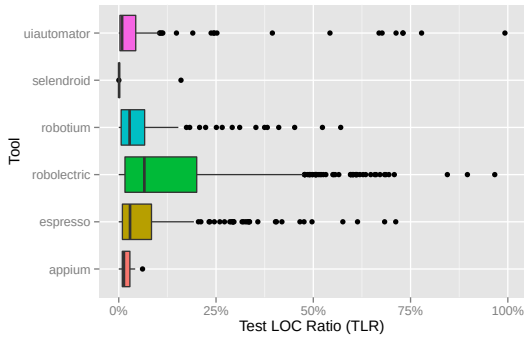


Fig. 4: Distribution of $\overline{TLR}$

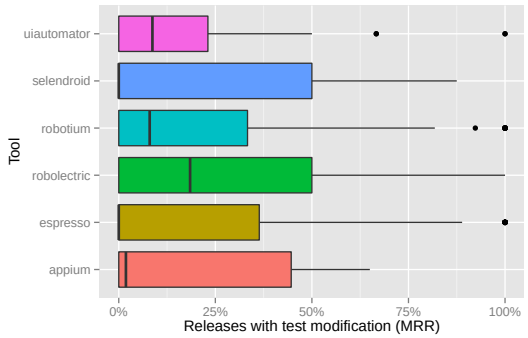


Fig. 5: Distribution of MRR

amount of absolute modified LOCs would result in a lower $\overline{MTLR}$ value. The highest value was found for the set of projects featuring Selendroid: this can be explained with the very high percentage of total LOCs in classes associated with Selendroid for these repositories. However, the set of projects featuring Appium, which also was characterized by a high average TLR, did not exhibit the same trend, having a lower $\overline{MTLR}$: this should mean that, even though the important ratio of testing code above project code, few modifications (in both production and test code) were made between subsequent releases on test code associated with Appium.

The measures about Modified Relative Tool LOCs ($\overline{MRTL}$) show that, on average, when the six selected testing frameworks are used, the 7.4% of the modified LOCs belong to classes containing code associated with those frameworks. With this metric, however, we are still unable to discriminate

TABLE VIII: Measures of RQ2 - evolution of test code (averages on the sets of repositories)

Tool	$\overline{TLR}$	$\overline{MTLR}$	$\overline{MRTL}$	$\overline{TMR}$	MRR	$\overline{TCV}$	TSV	TJR	$MTJR$
Espresso	7.3%	2.6%	4.7%	0.68	22.2%	8.6%	28.6%	2.75	1.25
UI Automator	9.9%	1.4%	3.5%	1.17	16.5%	6.4%	35.9%	1.55	0.78
Selendroid	19.4%	4.3%	11.5%	0.15	39.6%	11.5%	33.7%	0.56	0.24
Robotium	7.8%	3.8%	5.3%	0.56	22.1%	9.9%	36.3%	2.29	3.76
Robolectric	13.4%	2.9%	9.5%	0.79	28.2%	8.6%	30.4%	6.28	5.31
Appium	31.9%	1.8%	16.6%	0.27	27.3%	10.3%	36.2%	2.65	1.17
Average	11.2%	2.8%	7,4%	0.76	25.2%	8.6%	30.9%	4.53	3.67

TABLE IX: Metrics pertaining RQ2

Name	Explanation
TLR	Tool LOCs Ratio
MTLR	Modified Tool LOCs Ratio
MRTL	Modified Relative Tool LOCs
TMR	Tool Modification Relevance Ratio
MRR	Modified Releases Ratio
TCV	Tool Class Volatility
TSV	Tool Suite Volatility
TJR	Tool Code to JUnit code Ratio
MTJR	Modifications of Tool code to JUnit code Ratio

what is the reason behind the modifications to be performed on test classes. The higher $\overline{MRTL}$ values for the sets of projects featuring Appium and Selendroid can be justified by the small size of the two sets, and by the nature of the projects examined. For instance, the Selendroid framework, on GitHub as selendroid/selendroid, is subject to heavy modifications, but in this particular case classes that are actually the code of the testing tool should be mistakenly recognised as test code. While the three sets of projects featuring code associated with Espresso, Robotium and Robolectric exhibit close average $\overline{MTLR}$ values, the average $\overline{MRTL}$ on the set of projects featuring Robolectric is way bigger than the other two. In general, a higher $\overline{MRTL}$ should mean a minor adaptability of a testing tool to modifications performed on the production code, with more changes needed by test code as a consequence of changes in the production code.

The mean values of Tool Modification Relevance Ratio ($\overline{TMR}$) stayed in the range between 0.56 and 1.17 for big-sized sets of projects, with lower values for the sets featuring Selendroid and Appium. In general, those values imply that the amount of churn needed for the code associated with a specific testing framework is not linear with the relative amount (with respect to total production LOCs) of such code inside the application: in our case, on average, the ratio between the intervention on test code and the intervention on all production code is about 3/4 of the ratio between test and all production code. The higher $\overline{TMR}$ value for UIAutomator is due to some projects (e.g. Lanchon/android-platform-tools-base) in which TLR is rather small, and where in some releases all modified LOCs belong to test classes (thus leading to $MRTL$ values very close to 1).

The Modified Releases Ratio (MRR) metric gives an indication about how often the developers had to modify any of the classes associated with the considered testing frameworks when they published new releases of their projects. Boxplots in figure 5 show the distribution of MRR for the projects of the considered context. On average, 25.2% of releases needed modifications in any of the test classes (with a maximum of 39.6% for the set of projects featuring Selendroid, and a minimum of 16.5% for the set of projects featuring UIAutomator). Since releases may be frequent and numerous for GitHub projects, this result explains that the need for updating test classes is frequent for Android developers that are leveraging the analysed testing frameworks.

The 8.6% average value for the Tool Class Volatility ($\overline{TCV}$) metric, which characterizes the phenomenon of volatility from the point of view of the individual classes associated with a given testing framework, highlights the fact that each test class has to be modified, on average, every ten tagged releases in which it appears. The computed averages are similar for all the considered sets of projects, with the lower value – also in this case – obtained for UIAutomator, meaning a major stability of individual test classes.

TABLE X: Espresso - Evolution metrics for different subsets of projects

	$\overline{TLR}$	$\overline{MTLR}$	$\overline{MRTL}$	$\overline{TMR}$	MRR	$\overline{TCV}$	TSV	TJR	$MTJR$
Full set	7.3%	2.6%	4.7%	0.25	22.2%	8.6%	28.6%	2.75	1.25
Projects on Play Store	3.9%	2.7%	3.7%	0.72	21.2%	8.3%	32.1%	2.96	2.02
Projects with 1000+ LOCs	5.7%	2.6%	4.7%	0.69	22.7%	8.8%	30.0%	2.71	1.27

The average 30.6% value for the Test Suite Volatility (TSV) metric, which characterizes the phenomenon of volatility from the point of view of whole collections of classes containing code associated with a given testing framework, indicates that about one third of test classes require at least one modification during their lifespan.

The results for the *Tool Code to JUnit code Ratio* (TJR) metric show that typically, when both are present, the code associated with the six selected testing tool weighs more, in terms of LOCs, than other code associated with the JUnit testing framework (which can be related to unit testing, or to other levels of testing leveraging JUnit as a testing engine). Only the code associated with Selendroid was, on average, less prominent than the code associated with JUnit; on the contrary, the TJR value is particularly high for the set of projects featuring Robolectric. This may be considered as an evidence of a major complexity in writing test classes using such tool, and in maintaining them. If the hypothesis that JUnit is mostly used for unit testing of Java code applies, this finding is in accordance with our expectations: GUI testing code, typically traversing multiple widgets of the GUI to recreate a full user scenario, is likely to be more complex than unit testing code exercising small components of the application code in isolation. Those results are also in accordance with the other evolution metrics, which see the projects featuring Robolectric as the ones (without considering the small sets of projects featuring Selendroid and Appium) in need of more interventions in test classes throughout their history of releases (e.g., the average MRTL is nearly two times as big as the one computed for Espresso).

The average results gathered for the *Modifications of Tool code to JUnit code ratio* ($MTJR$) go from 0.24 (for the set of projects featuring Selendroid) to 5.31 (for the set of projects featuring Robolectric), meaning that the amount of modifications to perform on test classes associated with the considered frameworks used for GUI-level testing can be five times bigger than the one needed to perform on classes associated with the JUnit testing framework. With the exception of Selendroid and UIAutomator, the considered frameworks feature a mean $MTJR$ higher than 1, confirming that on average the code associated with them is more expensive to keep up to date with the application than code based on the JUnit framework. The result is in accordance with our expectations, since – being at a higher level than unit tests – GUI-related test code is likely to be influenced by modifications in every layer of the application, from individual low-level components to the presentation of the GUI itself. Hence, as confirmed by the findings, more maintenance – in the form of measurable code churn – has to be put on GUI-level test code during the evolution of a project featuring different levels of testing based on JUnit (not necessarily limited to the unit level).

It is however worth underlining that those two comparison metrics cannot serve as a precise estimation of the relative importance and needed effort for different typologies of testing: as discussed in more detail in the Threats to Validity section, an exact comparison between two testing frameworks is never possible even if they are based on the same coding language, and this especially applies for JUnit which can be used at several levels of testing abstraction.

On average, near 3% of the code associated with the selected testing frameworks is modified between consecutive tagged releases. 7.4% of the overall project LOCs modified between consecutive tagged releases belong to code in classes associated with the frameworks. On average, one fourth of tagged releases feature modifications in the set of classes associated with the frameworks, and one third of those classes needs modifications during the project history.

We computed a set of evolution metrics on different subsets of the full context of applications (the ones reported in table VI). In table X we report, as an example, the detailed results that we obtained for the projects featuring Espresso.

Projects with at least 1000 LOCs of code associated with Espresso tend to have a smaller $\overline{TLR}$ value with respect to the full set of projects. This can be an evidence of the fact that there is no linear link between the total amount of production and test code, meaning that test suites tend to be smaller, if compared to production code, for larger projects. This trend is confirmed by all the other five sets of projects. Furthermore, low TLR values may suggest that the testing code in the selected projects provides only partial coverage of the production code: in such case, it is reasonable that there is not an exact mapping between the amount of production code and test code, with a divergency that becomes bigger with the size of the application. No relevant differences are found for the other evolution metrics (with the obvious exception of $\overline{TMR}$, which depends on the TLR value).

Also for the set of projects released on the Play Store there is a difference in the $\overline{TLR}$ metric, arguably for the same reason of the minor $\overline{TLR}$ obtainable from applications with more than 1000 LOCs - i.e., small and/or experimental applications, that are less likely to be released on the Play Store, may have bigger test suites with respect to the total amount of production code. In general, we suppose that every testing framework brings a constant overhead of LOCs, that makes the TLR metric bigger for small projects.

While there is no relevant difference in the measured $\overline{MTJR}$ for the projects with 1000+ test LOCs (with respect to the complete set) those values are almost doubled for projects released on the Play Store. Two reasons may justify this finding. First, it is reasonable that when the applications are eventually released to a public more emphasis and attention is paid in exercising user scenarios and performing system testing traversing the GUIs, hence it is likely that testers perform adjustments in the automated GUI testing code more frequently. Second, it can also be supposed that the applications that are released to a public feature more complex GUIs than open-source projects not exposed on the Play Store: such GUIs may be subject to more frequent modifications, with a resulting higher amount of interventions needed in GUI test code to keep it aligned with the evolution of the application.

This can serve as a confirmation of our hypothesis: for applications that are eventually released to a public, test code associated with GUI-related testing frameworks becomes more crucial than unit testing or other forms of testing based on the JUnit framework, and thus requires a bigger (in this case, double) amount of modifications to be aligned to the evolution of the app.

C. RQ3 - Fragility

As possible estimations of fragility, we have considered (as detailed in the metrics section): the average percentage of modified classes and methods containing code associated with a given testing framework (respectively, MCR and MMR); the ratio of fragile classes upon the total number of classes associated with a given tool, and with respect to the total number of modified test classes (respectively, FCR and $RFCR$); the percentage of releases featuring fragile classes, and of releases featuring added or removed methods inside classes associated with a tool (respectively, FRR and $ADRR$); the average possibility for a class to be fragile during its lidespan, and the percentage of classes in each test suite experiencing fragilities (respectively, $TCFF$ and TSF). A summary of the definitions of the metrics is given in table XII. Averages on the whole sets of projects are shown in table XI. The values in last row are obtained as averages of the six values above, weighted by the size of the six sets.

The first column about the Modified Classes Ratio ($\overline{MCR}$) metric shows that, on average, 14.8% of test classes associated with the selected testing frameworks are modified between consecutive tagged releases in the mined set of Android open-source projects. The only value significantly different from the average is the one obtained for the set of projects featuring UIAutomator (9%), but it can be justified with the bigger amount of test classes that they feature on average (see table V).

The 3.6% average value found for the Modified Methods Ratio ($\overline{MMR}$) metric highlights that the percentage of methods associated with the selected tools that are modified is -as expected- smaller than the percentage of modified classes: this is

TABLE XI: Measures for RQ3 - estimations of fragility (averages on the sets of repositories)

Tool	$\overline{MCR}$	$\overline{MMR}$	$\overline{FCR}$	$\overline{RFCR}$	FRR	$ADRR$	$\overline{TCFF}$	TSF
Espresso	15.2%	3.5%	8.3%	59.7%	14.4%	17.7%	4.6%	18.8%
UI Automator	9.0%	1.8%	4.6%	54.4%	10.2%	8.2%	3.1%	16.6%
Selendroid	16.5%	2.7%	4.9%	42.2%	28.2%	23.2%	3.4%	11.9%
Robotium	16.4%	3.5%	9.3%	53.1%	15.2%	21.2%	5.7%	22.8%
Robolectric	15.1%	3.8%	8.5%	60.7%	20.6%	25.8%	4.9%	19.4%
Appium	15.2%	4.6%	7.7%	48.2%	17.1%	23.5%	5.1%	19.6%
Average	14.8%	3.6%	8.2%	59.0%	17.6%	21.9%	4.8%	19.3%

TABLE XII: Metrics pertaining RQ3

Name	Explanation
MCR	Modified Tool Classes Ratio
MMR	Modified Tool Methods Ratio
FCR	Fragile Classes Ratio
RFCR	Relative Fragile Classes Ratio
FRR	Fragile Releases Ratio
ADRR	Releases with Added-Deleted Methods Ratio
TCFF	Tool Class Fragility Frequence
TSF	Tool Suite Fragility

obviously due to the fact that typically multiple test methods are contained in single test classes.

Not all modified test classes associated with a given testing framework could be defined as fragile classes. The Relative Fragile Classes Ratio ($\overline{RFCR}$) metric gives a statistic about the possibility of a modified class to contain modified methods. The results collected show that more than half of the classes having modified lines featured modifications inside the code of test methods as well, hence they could be defined as fragile according to the heuristic definition given in section 2.4. The Fragile Classes Ratio ($\overline{FCR}$) metric gives the ratio between the classes that we define fragile upon all the classes associated with a given testing framework contained by each project. On average, 8.2% of the classes were fragile in the transition between consecutive releases of the same project.

The Fragile Releases Ratio (FRR) metric gives an indication of how many releases of the considered project contain test classes associated with a given testing framework that we identify as fragile. The value is upper-bounded by MRR, which is the frequence of releases featuring any kind of modification in those classes. The average value for FRR is 17.7%, meaning that about one every five releases features fragile test methods associated with a given testing framework. The Releases with

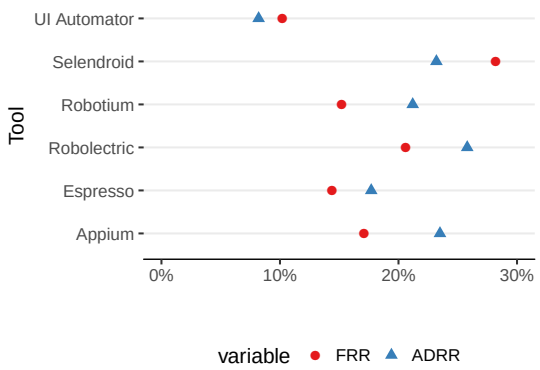


Fig. 6: FRR and ADRR average values for different tools.

Added-Deleted Methods Ratio ($ADRR$) metric quantifies the probability that there is the need – between two subsequent releases – to add or delete test methods inside existing test classes. In general (with the only exception of the set of projects featuring UIAutomator) $ADRR$ is higher than FRR. This result is in accordance with the findings by Pinto et al. [50], who observed – in the context of traditional desktop applications – that the sum of test deletions and additions is higher, on average, than the number of test modifications. Figure 6 reports the average FRR and $ADRR$ measures for the six sets of projects. We can observe that the two values are generally close to each other: during the evolution of the classes associated with a given testing framework the need for modifications in already existing test methods (FRR) occurs roughly as often as the definition of new test methods ($ADRR$). According to our interpretation of the addition of test methods – that we consider as the symptom of new features or new

TABLE XIII: Percentage of projects without modifications in test suites, classes and methods

Tool	Unmodified suites	Unmodified classes	Unmodified methods
Espresso	24.6%	57.0%	65.8%
UIAutomator	16.0%	40.0%	55.0%
Selendroid	60.0%	60.0%	80.0%
Robotium	16.6%	44.1%	60.0%
Robolectric	15.8%	45.3%	53.3%
Appium	27.3%	54.5%	72.7%

use cases to be tested – these values testify that the introduction of new elements that need testing is quite a frequent event during the lifecycle of an open-source Android project featuring the analysed testing frameworks, but just slightly more frequent than the modification of already existing methods.

Upper-bounded by $\overline{TCV}$ (the overall volatility for test classes), the average Test Class Fragility Frequence ($\overline{TCFF}$) provides information about the frequency of modifications that test classes must undergo because of fragile methods. The average value of 4.8% tells that an average test class developed with the analysed testing frameworks must have some modification in its methods every 20 releases in which it appears.

Upper-bounded by TSV (the overall volatility for test suites), the average value for Test Suite Fragility (TSF) provides information about the amount of test classes, in each project, that contain fragile methods. The average value of 20.2% tells that one fifth of the classes in test suites face at least a fragility during its entire lifespan.

In general, 14.8% of test classes and 3.6% of test methods associated with the analysed testing tools are modified between consecutive releases. About 20% of the test classes inside test suites contain fragile methods at least once in their lifespan.
Overall the changes induced by fragility require an amount of code churn comparable to the definition of new tests: 18% releases undergo fragility induced changes, and 22% of releases have test cases added or removed from the test suite.

It must also be considered that the averages reported above are heavily lowered by those projects in which classes and methods associated with the

Fig. 7: Descending MRR Measure for the whole context of Android open-source projects

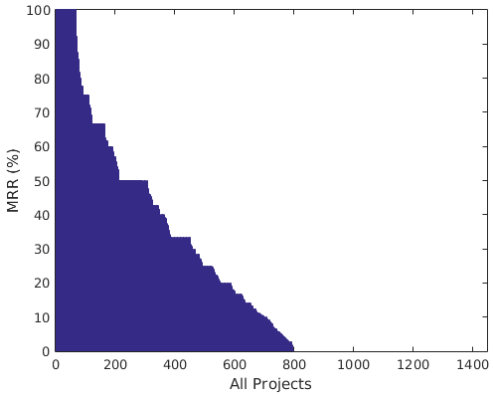
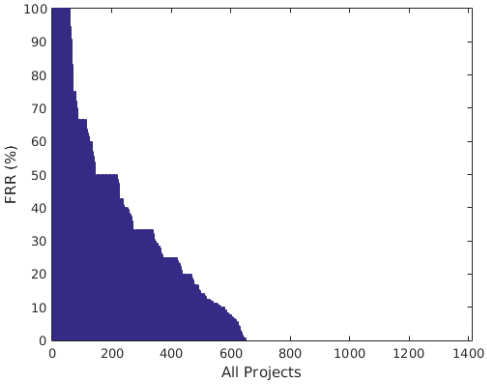


Fig. 8: Descending FRR Measure for the whole context of Android open-source projects



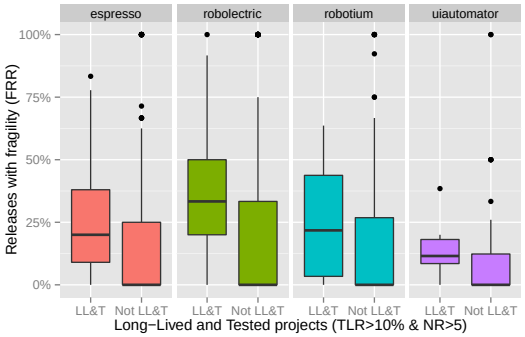


Fig. 9: Distribution of FRR metric for long-lived and highly tested projects

TABLE XIV: Measures for fragility for long-lived and tested projects

Tool	$\overline{MCR}$	$\overline{MMR}$	$\overline{FCR}$	$\overline{RFCR}$	FRR	$ADRR$	$\overline{TCFF}$
Espresso	21.2%	4.5%	12.5%	57.8%	26.6%	30.6%	9.0%
UI Automator	10.0%	0.9%	4.9%	55.9%	11.7%	8.8%	8.1%
Robotium	25.7%	5.2%	13.8%	46.0%	29.0%	36.1%	18.3%
Robolectric	30.0%	5.3%	12.5%	63.5%	36.0%	41.5%	14.7%

TABLE XV: Espresso - Fragility metrics for different subsets of projects

	$\overline{MCR}$	$\overline{MMR}$	$\overline{FCR}$	$\overline{RFCR}$	FRR	$ADRR$	$\overline{TCFF}$	TSF
Full set	15.2%	3.5%	8.3%	59.7%	14.4%	17.7%	4.6%	18.8%
Projects on Play Store	13.6%	3.1%	7.7%	61.7%	14.1%	13.9%	4.5%	20.3%
Projects with 1000+ LOCs	15.0%	3.2%	8.2%	60.0%	14.8%	17.7%	4.6%	19.7%

analysed testing frameworks are inserted – at the beginning or at some point in their history – but are never modified later. In table XIII we show: the percentage of projects whose suites of classes associated with a given testing framework are never modified; the percentage of projects with no modifications in classes associated with a given testing framework (i.e., only additions and modifications of test classes are performed); the percentage of projects with no modifications in methods associated with a given testing framework (i.e., only additions and modifications of methods are performed, and no fragility is detected). For instance, in the case of the set of projects featuring Espresso, 181 out of the 423 projects (the 42.8% of the total) have modified test classes between consecutive tagged releases. Only 34% of the projects have modified test methods between consecutive tagged releases. Graphs in fig. 7 and 8 show, respectively, the descending MRR and FRR measures for all the projects of the context (regardless of the specific testing tools they feature). It is evident from the graph that almost half of the test suites are never modified during the lifespan of the project they belong to; this supports our assumptions that test classes are often not utilized or abandoned. These results may suggest that the test code associated with the studied testing frameworks is subject to a certain level of aging [51]. With the static analysis of code that we have performed in this work, however, we cannot discriminate between test classes and methods that are not modified because they do not need to, and those that are not modified because they are no longer utilised by the developers, but are not removed from the project.

In table XIV we show the results that have been gathered only for those projects that feature a relevant percentage of code associated with the considered testing frameworks among product code (more than 10%) and more than five tagged releases. We show our results for the four biggest sets of projects considered, being the ones featuring Selendroid and Appium not relevant for such a comparison because they are significantly smaller. Our hypothesis, in fact, is that projects that have a longer history and bigger test suites, are more inclined to have test classes and methods modified than shorter-lived projects in which the amount of testing is negligible. The metrics we computed confirm our supposition: with the only exception of the set of projects featuring UIautomator, average FRR is nearly doubled for all sets (comparative boxplots are shown in figure 9). Also the average $\overline{TCFF}$ is largely increased if only applications with relevant test code percentage and release history are considered. On the other hand, no modifications in the average $\overline{RFCR}$ are found: the probability that classes with modified LOCs are also fragile seems to be not related to the size of projects and to the amount of testing inside them. We aim at investigating further such differences and the relative causes in future works.

As we have done for the evolution metrics, we computed the fragility metrics on other subsets of the whole context of Android applications. The results reported in table XV show – limited to the Espresso testing framework, as an example – the metrics computed for the whole set of projects, for

TABLE XVI: Samples selected for the evaluating fragility causes

#	Project Name	Class	Versions	Testing Tool
1	andrew-boiley/classy-apps	MainActivityTest.java	classy_apps.1.12 - classy_apps-1.2	Espresso
2	appium/unlock_apk	Unlock.java	v0.0.1 - v0.1.0	Appium
3	EyeSeeTea/malariapp	SurveyScoresEspressoTest.java	v0.9 - v1.0	Espresso
4	FabriceMK/android-mycv	NavigationViewTest.java	v1.04 - v1.05	Espresso
5	jivimberg/PracticePronunciation	EspressoTest.java	v1.2 - v1.3	Espresso
6	karumbi/Android-Ci-Demo	CharacterDetails.java	v1.1.0 - v2.0.0	Espresso
7	pylapp/SmoothClicker	StandaloneActivity.java	v1.10.0_pre4 - v1.10.0_pre5	Espresso
8	alosdev/amu-roboguice	RoboguiceAndRobotium.java	startPoint - withRoboguice	Robotium
9	arnastofnun/Ordabanki_for_android	ResultScreenActivity.java	v1.1.0 - v1.2.0	Robotium
10	blockchain/My-Wallet-V3-Android	ApplicationTest.java	6.1.20 - 6.1.21	Robotium
11	leapcode/bitmask_android	testDashboard.java	0.5.3 - 0.5.4	Robotium
12	mamewotoko/podplayer	PodcastListPreferences.java	v0.87 - 0.90	Robotium
13	univmobile/unm-android	Scenarios001.java	v0.0.4 - v0.0.5	Appium
14	raatiniemi/worker	NewProjectPresenter.java	v0.8.0 - v0.9.0	Robolectric
15	kidozen/kido-android	KidoApplicationTest.java	v1.2.0 - v1.2.1	Robolectric
16	hello/anime-android-go-99	MultiAnimatorTests.java	v0.4.1 - v1.2.0	Robolectric
17	dan-zx/tedroid	GameboardView.java	v0.2 - v1.2.0	Robolectric
18	bergvandenp/Sabber	ContextHelperTest.java	v1.1.2 - v1.1.3	Robolectric
19	passy/absshadow-sample	ShadowSherlockActivity.java	v0.9 - v1.2.0	Robolectric
20	tomohiro-ihara/android-flowlayout	FlowLayoutOrientationTests.java	v1.10 - master	Robolectric
21	blood1093/RxAndroid_ReactiveX	AppObservableTest.java	v0.24 - v0.25	Robolectric
22	st1hy/Count-Them-Calories	OverviewActivityTest.java	v1.0.2 - v1.1.0	Espresso
23	AoDevBlue/DotDash	IOActivityTest.java	1.0 - master	Espresso
24	zsolt/paperwork	MainActivityTest.java	v1.2.4 - v1.2.7	Espresso
25	rasmussaks/aken-ajalukku	ApplicationTest.java	iteration-3 - iteration-4	UIAutomator
26	AdamFresko/brickator	ExampleBrickatorTest.java	v1.0.1 - master	UIAutomator
27	ngageoint/anti-piracy-android-app	PhoneTest.java	1.0.0-RC1 - 1.0.0-RC2	UIAutomator
28	arquillian/arquillian-droidium	InstrumentationPerformer.java	1.0.0.Alpha1 - 1.0.0.Alpha2	Selendroid
29	androidannotations/androidannotations	ViewsInjecteActivityTest.java	2.1.2 - 2.2	Robolectric
30	tipsi/tipsi-dropdown-android	DropdownTests.java	v0.2 - v0.3	Espresso

the projects released on the Play Store, and for the projects featuring at least 1000 LOCs.

We notice that fragility metrics are very close for all the subsets, meaning that there is no evident correlation between the size of the GitHub project or the fact that the app is released on the Play Store, and the amount of fragility that it experiences during its lifespan. The finding corresponds to what is measured for $MTRL$ and $MRTL$ metrics, regarding the general evolution of test code. Smaller values for MCR are shown by released applications, meaning that on real published applications the Activities undergo less modifications than the average of the total set of Android projects. It can also be seen that, while FRR is very close to the value for the whole context of Android projects, $ADRR$ is lower for released apps: this can be an index of minor probability of variations in the main functionalities of released apps, and hence in a minor need for the addition or deletion of use cases to be tested by new methods.

To give a preliminary estimation of the amount

TABLE XVII: Reasons of the modifications in test methods

Class n.	#m v1	#m v2	#mm	Bug/RF	S/F	NG	G
1	4	4	2	0	0	0	2
2	1	2	1	0	0	0	1
3	16	16	1	0	0	0	1
4	4	4	3	3	0	0	0
5	10	14	4	0	0	0	4
6	10	10	2	0	1	0	1
7	9	9	2	0	0	2	0
8	2	2	1	1	0	0	0
9	13	13	1	0	0	1	0
10	3	3	1	1	0	0	0
11	6	10	1	0	0	0	1
12	13	17	7	1	0	0	6
13	4	4	1	0	0	1	0
14	3	8	3	0	0	0	3
15	4	3	2	0	0	1	0
16	9	14	1	0	0	0	1
17	2	4	2	0	0	2	0
18	11	11	7	0	0	0	7
19	2	2	1	0	0	0	1
20	4	4	4	0	0	0	4
21	9	8	2	2	0	0	0
22	15	16	2	0	0	0	2
23	7	7	1	0	0	0	1
24	10	11	3	0	0	0	3
25	7	7	4	1	0	0	3
26	2	2	1	0	0	0	1
27	9	9	1	0	0	0	1
28	4	4	2	0	0	2	0
29	5	6	1	0	0	0	1
30	6	6	1	0	0	0	1

of fragilities induced in code associated with GUI testing frameworks by modifications in the GUI itself, we selected a sample set of 30 classes identified as fragile from the studied projects. Details about the selected samples (i.e., the names of the repositories, individual classes selected and pairs of versions between which the modifications in the classes had place, along with the testing tool to which the classes are associated) are given in table XVI.

In the 30 selected fragile classes, 64 individual methods are modified and thus considered fragile according to our definition.

Using a similar categorization to the one proposed by Yusifoglu et al. [30], we have considered different classes of modifications for the test methods:

- bug fixing and refactoring (Bug/RF);
- syntactical correction and formatting (S/F);
- adaptation to product code not related to GUI (NG);
- adaptation to product code related to GUI (G).

For each test class selected, we subdivided the contained modified methods in the categories defined above, after manual inspections of the git diff files that allowed us to identify the reasons behind the changes in test code. For instance, changed lines in the diff files containing details of the GUI arrangement (e.g., IDs of the widgets, type of widgets interacted, text contained by textboxes shown to the users) led us to define the respective test methods as hampered by GUI related fragility. Changed lines in diff files related to internal functionalities of the application (e.g., declaration of Intents for launching other Activities, or registrations of BroadcastReceivers) led us to define the respective test methods as hampered by not-GUI related fragility. More insights about the kinds of modifications performed are shown in table XVII, whose columns show, in order: the number of methods before the release transition; the number of methods after the release transition; the number of modified methods in the release transition; finally, the number of methods that are marked as fragile due to modifications that are not related to GUI, and the number of methods that are marked as fragile due to modifications related to GUI.

We found that about 70% of the modifications in test methods are induced by modifications in the arrangement, definition or appearance of the application GUI. Hence, we can consider that modifications in the GUI of the AUT are involved in the majority of the modifications performed on test methods associated with the six considered testing frameworks. This result was indeed expected, since tests created with the use of GUI automation frameworks aim to exercise the overall features of the application from the user perspective. Hence, they are supposed to be impacted mostly by changes of the user interface only. However, being system-level tests, they are in any case impacted by modifications in lower level of abstractions of the application code, e.g. production code refactorings or changes in the data definition. The impact of those categories of modifications to the apps proved to be quite low on the set of test cases considered in our manual inspection.

VI. Threats to Validity

Threats to internal validity. We have identified the following threats to the validity of our conclusions:

- The test class identification process is based on some keywords specific to each testing tool: any file containing one of such keywords is considered as a test file without further inspection. This procedure may miss some test classes, or consider a file as a test file mistakenly.
- The number of tagged releases is used as a criterion to identify a project as worth to be considered for our investigations; it is not assured that this check is the most dependable one for pruning negligible projects.
- The metrics we defined have not been tested outside the scope of this study, hence we cannot ensure the correctness of the assumptions we based on them.
- Our evaluations are based only on files that contain pure Java code. Hence, code in other languages, that may be part of test suites as well as of production code of Android applications, does not contribute to the computations we performed. This may add biases to the presented results.
- Java files containing keywords pertaining to each tool were entirely associated with the tool, and all their lines were counted for

the defined metrics. In addition to that, no discrimination has been made about the use that was made of the individual tools, while some of the considered testing frameworks can be used to perform not only GUI testing. Both threats may add biases to the results, if multiple different testing frameworks are used in the same Java classes, and if the testing tool to which the code is associated is not used to perform GUI testing. Similarly, it is not assured that the Java files automatically associated with JUnit were used for unit testing only or instead as a basis for other – even GUI – testing tools. The latter issue may invalidate part of the reasoning based on the TJR and MTJR metrics.

- Structure, provided coverage and quality of the developed test cases have not been controlled and taken into account by the automated procedure for computing the metrics. Hence, the effects that low-quality tests have on maintenance effort are not taken into consideration in the discussion we provide.
- The performed study was purely static, i.e. test methods were not executed to understand whether they were actually working even though they were not subject to modifications. This threat may add biases to the amount of fragile test classes and methods that we provided as results, since they do not consider test code which should be modified because of changes in the AUT or its GUI (and hence is fragile by our definition) but is abandoned by developers.

Threats to external validity. We identify the following threats to the generalizability of our work:

- Testing tools and techniques adopted by relevant industrial practitioners may vary significantly from the ones discussed in this work, and by the related ones discussed in earlier sections. It is not assured that our findings, based on a very large repository of open-source projects, can be applicable to the development of commercial projects.
- Our findings are based only on the GitHub open-source project repository. Even though it is a very large repository, it is not assured that such findings can be generalized to closed-source Android applications, neither to those taken from different repositories.
- We have collected measures for just six scripted GUI automated testing tools. It is not certain that such selection of tools is representative of other categories of testing tools or even different tools of the same category, which may exhibit different trends and fragilities throughout the history of their AUT.
- The metrics we defined apply only to testing tools who produce scripts in Java. Other tools producing test scripts in other languages cannot be evaluated using the provided metrics, neither the results of the application of similar metrics on them can be compared to the results we provide in the paper.

VII. Conclusions and Future Work

In this paper we aimed at taking a snapshot of the usage of automated GUI testing frameworks among Android open-source projects. We quantified the use of a set of six tools that can be used for GUI testing and that are cited in available literature – Espresso, UI Automator, Selendroid, Robotium, Robolectric, and Appium – in the projects hosted by the GitHub portal.

We found that the level of adoption of the considered GUI testing frameworks among Android projects hosted on GitHub is very scarce, even for those tools that are part of the Android Instrumentation Framework. The whole adoption of all the six testing tools considered is about 8% of all projects that have a release history. This value can be compared to the 20% adoption of the JUnit framework. For what concerns individual projects, on average, when present, the testing code related to the six selected framework represent 13% of all the production code. This result is slightly lower than those obtained by Kochhar et al. [5] regarding the projects hosted on the F-Droid repository, where only 14% apps contained executable test classes.

Concerning the evolution of test code, on average near 10% of the total changed lines, between consecutive releases of the same project, belong to the code associated with the selected test frameworks. Such percentage is quite low if it is considered that code churn is inevitable during the evolution of an application, and tests must adapt to changing requirements or any kind of change in the AUT. However, the average amount

of changed lines may also reflect a relatively small coverage provided by test suites developed with the studied tools, as it may be suggested by the average relevance of testing code among total production code. Albeit a linear correlation between code churn and man hours in updating code is difficult to be proven, this ratio can be considered as a preliminary indication of the amount of effort that developers must spend to keep their test classes up to date with modifications that are performed on production code.

In addition to that, when test code associated with both the popular JUnit testing framework and the considered six testing frameworks is present, most of the times more code churn is needed in keeping up to date the latter (with a ratio of even 5 to 1 for some of the considered tools). A higher maintenance cost for code related to GUI testing was expected: GUI testing frameworks allow to perform system level testing interacting with the AUT from the level of abstraction of the GUI presented to the user, and are affected by modifications performed on any level of abstraction of the application functionalities. On the other hand, other levels of testing (including unit testing, for which JUnit is commonly used in addition to serving as an enabling engine for other tools and levels of testing) are not influenced by any modification in the GUI arrangement or definition, and in the graphical appearance of the AUT. Hence, it is reasonable that GUI tests require more maintenance during the normal evolution of a mobile application, which is – by nature – typically subject to rapid evolution of its GUI. Future work may aim at refining the implemented test code parser, to discriminate between the actual usage made of the JUnit testing framework (e.g., unit, integration or system testing) and better delimit the comparisons performed to evaluate the amount and churn of GUI testing code.

These results, however, confirm that – as it is deduced by existing surveys among open-source developers [32] – maintaining a GUI test suite is a rather complex and time-consuming task, that can make open-source developers neglect GUI testing at all, or abandon test code – without making it evolve with the application – after it has been written.

Further empirical studies in this field may directly observe open-source as well as industry practitioners, in order to quantitatively measure their effort in keeping test code aligned with the evolution of the apps and their GUIs, e.g. in terms of man-hours per release. Future work may also seek for correlation between modified LOCs or classes (e.g., the MRTL, MMR or MCR metrics proposed in this manuscript) and actual developer effort, or take into account better predictors as suggested from existing literature [52]. Furthermore, dynamic evaluations can be performed to quantify the amount of non-working test code kept by developers in their project without performing maintainance on it, and to evaluate the way developers cope with aging test code.

The fragility of the tests can be estimated with two metrics based on the raw count of classes and methods modified. Overall we can estimate the fragility of the analyzed test classes around 8% (meaning that there is such probability that a test class associated with one of the studied frameworks may include a modified test method, between two consecutive releases). Throughout all the lifespan of the projects, almost one test class every five experiences any kind of fragility-induced modifications. On average, around 15% of releases need intervention in test classes to keep them aligned with the production code. On a sample set of modified classes, that were manually examined to understand the causes underlying the fragility issue, it has been found that around 70% of the fragilities in GUI test methods were related to modifications performed in the definition and arrangement of the GUI itself, and not to other layers of the AUT. These results show that developers contributing to open-source Android projects need rather frequently to adapt their GUI scripted testing suites to the evolution of the application, and suggest that state of the art tools may profit of additional features reducing the amount of effort needed by testers to keep their scripts up and running. The results can also be used as a benchmark by practitioners and developers themselves, to understand whether the amount of fragility happening to their testing code is in line with the typical one for the tools they are using.

Based on these evaluations, we plan in the immediate future to provide a definition of a taxonomy of causes of fragilities through the application of the grounded theory approach to Git diff files, guidelines to help developers to avoid them, and

finally the development of automated tools capable of adapting the test cases to modificatons made in the GUIs, or at least of signaling the developers of the possible occurrence of fragility for specific test classes. We plan to add a dynamic evaluation of test cases to our static analysis, i.e. we aim at executing modified and unmodified test cases to understand whether they were actually made unusable by fragilities, they were still working even though they featured modifications, or they still presented flakiness [53] regardless of adjustments performed. An extension of the study to other databases of open-source projects, to different testing frameworks or types, to other sotware platforms (like iOS), and to commercial closed-source applications is also planned.

ACKNOWLEDGMENT

This work was supported by a fellowship from TIM.

REFERENCES

[1] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing android application crashes," in *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*. IEEE, 2016, pp. 33–44.

[2] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What do mobile app users complain about?" *IEEE Software*, vol. 32, no. 3, pp. 70–77, 2015.

[3] A. I. Wasserman, "Software engineering issues for mobile application development," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 2010, pp. 397–400.

[4] E. Alégroth and R. Feldt, "On the long-term use of visual gui testing in industrial practice: a case study," *Empirical Software Engineering*, vol. 22, no. 6, pp. 2937–2971, 2017.

[5] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, "Understanding the test automation culture of app developers," in *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*. IEEE, 2015, pp. 1–10.

[6] M. Linares-Vásquez, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "How do developers test android applications?" in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 613–622.

[7] H. Muccini, A. Di Francesco, and P. Esposito, "Software testing of mobile applications: Challenges and future research directions," in *Proceedings of the 7th International Workshop on Automation of Software Test*. IEEE Press, 2012, pp. 29–35.

[8] R. Coppola, E. Raffero, and M. Torchiano, "Automated mobile ui test fragility: an exploratory assessment study on android," in *Proceedings of the 2nd International Workshop on User Interface Test Automation*. ACM, 2016, pp. 11–20.

[9] B. Kirubakaran and V. Karthikeyani, "Mobile application testingchallenges and solution approach through automation," in *Pattern Recognition, Informatics and Mobile Engineering (PRIME), 2013 International Conference on*. IEEE, 2013, pp. 79–84.

[10] D. Amalfitano, A. R. Fasolino, P. Tramontana, and B. Robbins, "Testing android mobile applications: Challenges, strategies, and approaches," in *Advances in Computers*. Elsevier, 2013, vol. 89, pp. 1–52.

[11] J. Gao, X. Bai, W.-T. Tsai, and T. Uehara, "Mobile application testing: a tutorial," *Computer*, vol. 47, no. 2, pp. 46–55, 2014.

[12] A. Kaur, "Review of mobile applications testing with automated techniques," *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 4, no. 10, pp. 503–507, 2015.

[13] M. Linares-Vasquez, C. Vendome, Q. Luo, and D. Poshyvanyk, "How developers detect and fix performance bottlenecks in android apps," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE, 2015, pp. 352–361.

[14] M. Kropp and P. Morales, "Automated gui testing on the android platform," *Testing Software and Systems*, p. 67, 2010.

[15] M. Linares-Vásquez, "Enabling testing of android apps," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 2. IEEE, 2015, pp. 763–765.

[16] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 224–234.

[17] Y. Zhauniarovich, A. Philippov, O. Gadyatskaya, B. Crispo, and F. Massacci, "Towards black box testing of android apps," in *Availability, Reliability and Security (ARES), 2015 10th International Conference on*. IEEE, 2015, pp. 501–510.

[18] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "Mobiguitar: Automated model-based testing of mobile apps," *IEEE software*, vol. 32, no. 5, pp. 53–59, 2015.

[19] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 258–261.

[20] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2013, pp. 250–265.

[21] J. Kaasila, D. Ferreira, V. Kostakos, and T. Ojala, "Testdroid: automated remote ui testing on android," in *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia*. ACM, 2012, p. 28.

[22] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing-and touch-sensitive record and replay for android," in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 72–81.

[23] C.-H. Liu, C.-Y. Lu, S.-J. Cheng, K.-Y. Chang, Y.-C. Hsiao, and W.-M. Chu, "Capture-replay testing for android applications," in *Computer, Consumer and Control (IS3C), 2014 International Symposium on*. IEEE, 2014, pp. 1129–1132.

[24] E. Alégroth, R. Feldt, and L. Ryrholm, "Visual gui testing in practice: challenges, problemsand limitations," *Empirical Software Engineering*, vol. 20, no. 3, pp. 694–744, 2015.

[25] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in *Acm Sigplan Notices*, vol. 48, no. 10. ACM, 2013, pp. 623–640.

[26] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 2013, pp. 67–77.

[27] S. Singh, R. Gadgil, and A. Chudgor, "Automated testing of mobile applications using scripting technique: A study on appium," *International Journal of Current Engineering and Technology (IJCET)*, vol. 4, no. 5, pp. 3627–3630, 2014.

[28] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Capture-replay vs. programmable web testing: An empirical assessment during test case evolution," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE, 2013, pp. 272–281.

[29] ——, "Visual vs. dom-based web locators: An empirical study," in *International Conference on Web Engineering*. Springer, 2014, pp. 322–340.

[30] V. G. Yusifoğlu, Y. Amannejad, and A. B. Can, "Software test-code engineering: A systematic mapping," *Information and Software Technology*, vol. 58, pp. 123–147, 2015.

[31] X. Tang, S. Wang, and K. Mao, "Will this bug-fixing change break regression testing?" in *Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on*. IEEE, 2015, pp. 1–10.

[32] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk, "Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing," in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 399–410.

[33] T. W. Knych and A. Baliga, "Android application development and testability," in *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems*. ACM, 2014, pp. 37–40.

[34] H. Tang, G. Wu, J. Wei, and H. Zhong, "Generating test cases to expose concurrency bugs in android applications," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 648–653.

[35] M. Fazzini, E. N. D. A. Freitas, S. R. Choudhary, and A. Orso, "Barista: A technique for recording, encoding, and running platform independent android tests," in *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*. IEEE, 2017, pp. 149–160.

[36] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk, "Fusion: A tool for facilitating and augmenting android bug reporting," in *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on*. IEEE, 2016, pp. 609–612.

[37] M. TAN and P. CHENG, "Research and implementation of automated testing framework based on android," *Information Technology*, vol. 5, p. 035, 2016.

[38] N. M. L. Neto, P. Vilain, and R. d. S. Mello, "Segen: generation of test cases for selenium and selendroid," in *Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services*. ACM, 2016, pp. 433–442.

[39] M. Hans, *Appium Essentials*. Packt Publishing Ltd, 2015.

[40] G. Shah, P. Shah, and R. Muchhala, "Software testing automation using appium," *International Journal of Current Engi neering and Technology*, vol. 4, no. 5, pp. 3528–3531, 2014.

[41] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and G. Imparato, "A toolset for gui testing of android applications," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 650–653.

[42] D. T. Milano, *Android application testing guide*. Packt Publishing Ltd, 2011.

[43] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing android apps through symbolic execution," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, pp. 1–5, 2012.

[44] B. Sadeh, K. Ørbekk, M. M. Eide, N. C. Gjerde, T. A. Tønnesland, and S. Gopalakrishnan, "Towards unit testing of user interface code for android mobile applications," in *International Conference on Software Engineering and Computer Systems*. Springer, 2011, pp. 163–175.

[45] A. Allevato and S. H. Edwards, "Robolift: engaging cs2 students with testable, automatically evaluated android applications," in *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. ACM, 2012, pp. 547–552.

[46] H. Zadgaonkar, *Robotium Automated Testing for Android*. Packt Publishing Ltd, 2013.

[47] R. Grgurina, G. Brestovac, and T. G. Grbac, "Development environment for android application development: An experience report," in *MIPRO, 2011 Proceedings of the 34th International Convention*. IEEE, 2011, pp. 1693–1698.

[48] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "A gui crawling-based technique for android mobile application testing," in *Software testing, verification and validation workshops (icstw), 2011 ieee fourth international conference on*. IEEE, 2011, pp. 252–261.

[49] T. Das, M. Di Penta, and I. Malavolta, "A quantitative and qualitative investigation of performance-related commits in android apps," in *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*. IEEE, 2016, pp. 443–447.

[50] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 33.

[51] R. Feldt, "Do system test cases grow old?" in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 343–352.

[52] E. Shihab, Y. Kamei, B. Adams, and A. E. Hassan, "Is lines of code a good measure of effort in effort-aware models?" *Information and Software Technology*, vol. 55, no. 11, pp. 1981–1993, 2013.

[53] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," 2018.