Feature Matching-based Approaches to Improve the Robustness of Android Visual GUI Testing

LUCA ARDITO, ANDREA BOTTINO, RICCARDO COPPOLA, FABRIZIO LAMBERTI, FRANCESCO MANIGRASSO, LIA MORRA, and MARCO TORCHIANO, Politecnico di Torino

In automated Visual GUI Testing (VGT) for Android devices, the available tools often suffer from low robustness to mobile fragmentation, leading to incorrect results when running the same tests on different devices.

To soften these issues, we evaluate two feature matching-based approaches for widget detection in VGT scripts, which use, respectively, the complete full-screen snapshot of the application (*Fullscreen*) and the cropped images of its widgets (*Cropped*) as *visual locators* to match on emulated devices.

Our analysis includes validating the portability of different feature-based visual locators over various apps and devices and evaluating their robustness in terms of cross-device portability and correctly executed interactions. We assessed our results through a comparison with two state-of-the-art tools, EyeAutomate and Sikuli.

Despite a limited increase in the computational burden, our Fullscreen approach outperformed state-of-the-art tools in terms of correctly identified locators across a wide range of devices and led to a 30% increase in passing tests.

Our work shows that VGT tools' dependability can be improved by bridging the testing and computer vision communities. This connection enables the design of algorithms targeted to domain-specific needs and thus inherently more usable and robust.

CCS Concepts: • **Software and its engineering** → **Software defect analysis**; **Software verification and validation**; **Software testing and debugging**; • **Computing methodologies** → *Matching;*

Additional Key Words and Phrases: Mobile computing, software testing, visual GUI testing, feature matching

ACM Reference format:
Luca Ardito, Andrea Bottino, Riccardo Coppola, Fabrizio Lamberti, Francesco Manigrasso, Lia Morra, and Marco Torchiano. 2021. Feature Matching-based Approaches to Improve the Robustness of Android Visual GUI Testing. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 21 (November 2021), 32 pages.
https://doi.org/10.1145/3477427

1 INTRODUCTION

Modern Android apps have reached a high complexity level, almost bridging the gap with traditional desktop software in terms of exhibited features. Nowadays, mobile apps are using sophisticated frameworks and modern development processes, with quick delivery cycles. Moreover, there is a very tight competition on the online markets where they are released: the official Google Play

Authors' address: L. Ardito, A. Bottino, R. Coppola, F. Lamberti, F. Manigrasso, L. Morra, and M. Torchiano, Politecnico di Torino, Corso Duca degli Abruzzi 29, Torino, Italy, 10129; emails: {luca.ardito, andrea.bottino, riccardo.coppola, fabrizio.lamberti, francesco.manigrasso, lia.morra, marco.torchiano}@polito.it.

1049-331X/2021/11-ART21 $15.00
https://doi.org/10.1145/3477427

store counts 2.96 million apps available as of June 2020, and 84.3 billion downloads for the whole 2019 [53]. These characteristics should encourage a thorough Verification and Validation phase to gain the necessary confidence that the apps behave correctly and systematically use automated techniques to support and complement the slow and error-prone manual practice.

The vast majority of Android apps are **Graphical User Interface (GUI)** intensive and collect most of the user's interaction through GUI widgets, or *Views*. Thereby, a critical issue is ensuring that the running app correctly renders the widgets. Recent years witnessed the development of many **End-to-End (E2E)** testing tools that allow developers to create repeatable test scripts mimicking a human interaction with the finished app and evaluate their response [12]. Many of these tools identify the GUI widgets through layout properties of the GUI structure that serve as textual *locators* and are hence called *Layout-based* testing tools. Similarly, the execution correctness is validated by verifying the properties of the GUI layout. These tools, however, are unable to test the actual graphical appearance of the **Application Under Test (AUT)** when shown to its final user. Thus, visual failures may slip through undetected.

To address this limitation, researchers proposed to tackle app testing with the **Visual GUI Testing (VGT)** paradigm [2]. With this paradigm, the verification of behavior's correctness involves a visual comparison between the app's current and expected visual appearance. This comparison leverages image recognition algorithms and uses screen snapshots as both *visual locators* (to identify the widgets) and *oracles* (to determine whether the displayed widget is correct). Thus, the main advantage of the VGT approach is that it allows testers to verify that the widgets are displayed in the correct position and appearance. In contrast, layout-based techniques have a limited ability to verify the rendered GUI's appearance and typically only check a widget's instantiation and not whether its appearance is correct.

In addition to the visual verification component, VGT techniques allow emulating user interaction with the GUI in an entirely agnostic way to AUT implementation, **operating system (OS)**, and platform. These features make VGT techniques optimal for testing those applications that need to provide high portability across different platforms. In particular, they represent a valid alternative to the (costly) manual testing of those applications that contain a lot of visual content, whose rendering must be carefully verified. Although the demonstrated benefits of applying VGT tools for desktop software [3], in the industrial practice of mobile app development, the inherent characteristics of the domain have slowed down their adoption. Indeed, visual tests of Android apps are very fragile due to hardware fragmentation [31]. Therefore, since every app must be compatible with many different devices (with varying screen sizes, pixel densities, and rendering specifications), marginal variations in the graphical rendering can invalidate the recognition of visual locators and oracles. As a result, test cases may not be portable to devices different than those on which the captures have been performed. Moreover, graphical changes in the same app's consecutive releases may break the tests, requiring additional maintenance in existing test suites to adapt locators and oracles. Our previous studies have shown that state-of-the-art VGT testing tools are complicated to port to different devices unless leveraging hybrid techniques that regenerate VGT tests for each device from an original layout-based test suite [22]. However, such an approach increases the test suite maintenance costs, since visual locators' regeneration is a time-consuming operation that must be performed at each new application release that includes changes in the GUI appearance.

With this discussion in mind, one possible question is how to improve the effectiveness of the VGT paradigm. The hypothesis underlying our work is that the algorithms currently used by state-of-the-art VGT tools (which usually leverage pixel-to-pixel comparisons) can be made more robust to possible graphical variations across devices, OSs, and versions. In particular, our work aims at analyzing the role of feature matching algorithms, a specific class of **Computer Vision (CV)**

algorithms, as an enabling technology for more robust Android VGT tools [59]. These algorithms rely on the analysis of local textural features invariant to several image variations (like color, shape, scale, and rotation), allowing them to detect one or more targets in cluttered scenes robustly.

The idea of applying feature matching algorithms in the VGT domain is not new [1, 48, 58]. However, to the best of our knowledge, the approach presented in this article represents the first attempt to systematically evaluate the suitability of different feature matching algorithms to support VGT applications and, specifically, their robustness to device fragmentation. In particular, we present the design of two different feature matching-based algorithms to identify widgets' visual locators, referred to in the following as *Fullscreen* and *Cropped*. Our experimental results show how our approach can increase the robustness of visual locator matching, improve over state-of-the-art VGT tools, and achieve higher portability across devices.

To validate the robustness of visual locator matching, we perform a twofold analysis. First, we validate the portability of different locator matching strategies and feature descriptors on different devices over a large set of Android apps by comparing recall, precision, and execution time. Second, we perform an exploratory study on a real test suite, and we compare our feature matching approaches with state-of-the-art VGT tools in terms of resilience to device change.

Additionally, we contribute to state of the art in the field by introducing DatAndroid,[1] a dataset including screenshots and associated metadata (i.e., View Hierarchies) of about 100 apps emulated on 14 different devices, for a total of roughly 1,400 combinations, which could serve as a public benchmark for the portability of VGT techniques. To the best of our knowledge, the only available public large-scale datasets with screenshots are RICO and ERICA [26]. However, they do not provide screenshots of the same app rendered on different devices, with varying screen sizes, pixel densities, and rendering specifications; hence, they are not suitable for investigating the performance, robustness and portability of VGT tools, or their underlying CV techniques.

The present manuscript is structured as follows: Section 2 provides background information about Android GUI Testing techniques, VGT tools, and CV techniques for GUI testing; Section 3 introduces the two matching algorithms that we propose; Section 4 illustrates the experimental procedure to evaluate feature matching algorithms for VGT; Section 5 provides a description and discussion of the collected results; Section 6 summarizes the threats to the validity of the current study and, finally, Section 7 concludes the article and provides directions for future work.

2 BACKGROUND AND RELATED WORK

In this section, we first illustrate the main concepts behind automated GUI testing, then we examine the Android-specific VGT techniques and we highlight their major limitations. We finally provide an overview of recent literature related to image recognition algorithms' application to VGT for Android apps.

2.1 Android GUI Structure and Automated GUI Testing

Android apps (and mobile apps, in general) can be divided into three main categories, according to the way they are built: *native* apps, created using the Android-specific design guidelines and the related development framework; *web-based* apps, i.e., web applications optimized to be loaded on browsers installed on Android devices; *hybrid* apps, combining the principles of native and web-based apps by employing components that are capable of loading dynamic content from the web at run time.

While web-based apps and the content of hybrid apps are designed by following the usual web development principles, native apps are built with a precise set of visual components. Each app

[1]The dataset is available for download on Zenodo or at the link https://frankissimo.github.io/datAndroid/.

screen is managed by an *Activity*. This component defines and builds the GUI shown to the user, which is composed of Views (i.e., widgets) arranged in a tree structure according to a specific layout [52]. The layout can be defined programmatically (in the code of the Activity class) or by defining static XML files that can be loaded (or "inflated") as the Activity's first operation. Screen transitions are obtained by registering callbacks on interactable GUI elements (like buttons and text fields) and processing user inputs (like clicks, text insertions, and swipe operations).

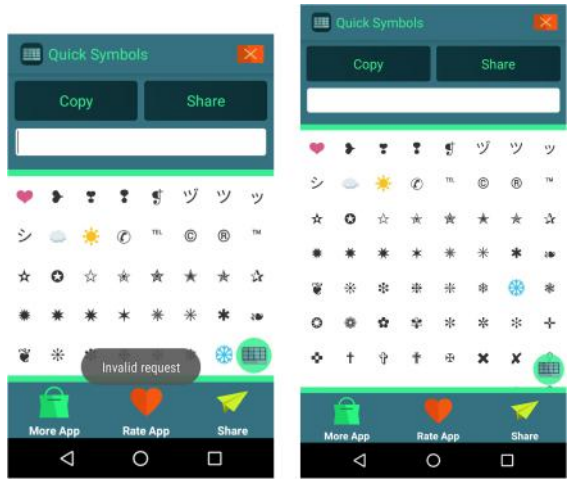
One of the most widely used approaches to automated GUI testing for Android applications is layout-based testing. This approach leverages the possibility to extract, at any given time during the app execution, the current screen hierarchy, where each View is associated with a set of platform-specific parameters, like unique id, textual content, size, and screen position. These properties allow identifying the visible elements used by Layout-based testing tools as both locators (to identify the Views to interact with) and oracles (to verify if a specific View has been shown on the screen with the desired properties).

The relevance of Layout-based testing is witnessed by the fact that most of the research in automated GUI testing for Android apps focused on this approach. Linares-Vasquez et al. [38] provided a classification of over 80 testing tools based on the approach adopted to define the test sequences. According to this classification, *Automation Frameworks and APIs* allow testers to create JUnit-like test scripts to be run against Android GUIs. Examples of this category are the tools embedded with the Android development framework (Espresso [50], UIAutomator [64]), and Robotium [63]). *Record and Replay* tools create sequences of interactions by capturing manual sequences executed by a tester into repeatable scripts. Recent examples are Barista [28] and OBDR [46]. *Automated Test Input Generation Techniques* execute tests without the need for manually defining input sequences; the sequences can be generated randomly, as with SAPIENZ [42], or Monkey [27], the official GUI random exerciser provided by Android. More sophisticated approaches automatically create the test sequences by generating and using GUI models, like the finite state machines exploited by MobiGUITAR [7] and DroidMate [15].

2.2 Tools for Visual GUI Testing

VGT is an alternative approach to GUI testing that uses CV to automate testers' tasks [19]. Test cases developed by the VGT approach show several key features: high readability (since each interaction is described with screen captures of the application and not with code); a more natural development of scripts with respect to the Layout-based approach (which, for the definition of each locator, requires the tester/developer search unambiguous properties in the layouts); complete platform independence (since the GUI tests are agnostic of the specific platform and OS for which the AUT is engineered, and they only require to render or emulate it). Empirical controlled experiments have demonstrated that VGT tools can significantly increase testers' productivity [10].

Several commercial and open-source tools have adopted the VGT approach. Sikuli [62] uses image recognition algorithms to identify GUI components. The tool provides an **integrated development environment (IDE)** for assisted script generation and can output test scripts in Python, Ruby, and JavaScript. Libraries for using Sikuli matching commands inside JUnit test cases are also available. EyeAutomate (evolution of JAutomate [6]) uses a proprietary algorithm to recognize graphical oracles and provides an IDE for manual script definition as well as a Java library to write down JUnit-like test methods. The tool supports the creation of text-based repeatable scripts written in a proprietary syntax. **Layout Bug Hunter (LBH)** validates bugs in the GUI rendering by checking layout errors caused by changing the screen size [32]. Commercial products are also available, like AppliTools, which uses visual inputs combined with machine learning capabilities, and PhantomCSS, based on pixel-by-pixel screen comparisons [60].



(a) Google Nexus S (b) Google Nexus 5

Fig. 1. Main Activity of the ASCII Notes application on devices with different screen sizes. On the smaller Nexus S screen, a small number of components is displayed than on the Nexus 5 screen.

Several studies in the literature proved the applicability of these tools in real-world scenarios. Alegroth et al. documented the feasibility of the VGT technique's long-term usage through an industrial case study [3]. Borgesson et al. performed comparative studies between Sikuli and JAutomate in an industrial setting, finding that VGT is an applicable technology for automated system testing with significant advantages compared to manual system testing and manual scripting and proving that the technique can be successfully used for automated acceptance testing [16].

However, as highlighted by most VGT studies, the main limitation of this approach is the high maintenance cost of visual test suites [4]. Several studies tried to address the maintenance issues by providing translation-based or hybrid approaches that combine visual and Layout-based locators' characteristics by using the latter to automatically re-generate the visual captures [5, 11, 36]. These approaches are however limited to the translation of a limited set of types of locators and interactions, and generally do not achieve a 100% precision in visual oracle generation, thereby requiring manual intervention at times. They also require the layout-based test suite to be rendered against each emulated device for the creation of visual screen captures, therefore they are not indicated for large-sized test suites.

2.3 Issues with VGT of Android Apps

Beyond the limitations introduced in the previous section, Android fragmentation is recognized as a relevant issue to tackle by developers and testers [42]. The fragmentation concept can be divided into *hardware fragmentation*, i.e., inconsistencies among various hardware specifications that can cause differences in GUI renderings, and *software fragmentation*, i.e., inconsistencies due to the different versions of the OS, **application programming interface (API)**, and GUI customizations [51]. Fragmentation mandates additional effort by both developers and testers to ensure that their apps' features are entirely portable to the devices that the app must provide compatibility with. A comprehensive testing procedure shall test the app on multiple handsets, each with its hardware and software characteristics [34].

Fragmentation significantly impacts GUI visual test practices, since it results in the impossibility to guarantee that the image recognition algorithms can correctly find the locators and oracles

captured on a source device. A further issue is that the Android framework allows rendering layouts in different ways (i.e., with a variable number and arrangement of widgets) on devices with different pixel densities and screen sizes. For instance, multiple widgets can be compacted in menus or inserted into scroll views on smaller screen sizes and visualized only after swipe operations (as in Figure 1, which reports an example of the same Activity rendered on two different devices). Thereby, GUI test cases are intrinsically fragile to fragmentation, since even if no faults are present when executing the app on different devices, visual test scripts may require the locators to be adapted to the specific size and arrangement of the widgets on the target device to avoid wrong test executions.

Another major problem documented for VGT, although not exclusive to the Android domain, is the *graphical fragility*, driven by pure graphical changes to widgets or modifications in AUT layout properties (e.g., textual content or hierarchy changes) [24]. These situations require test maintenance, forcing testers to provide new visual locators for each widget whose appearance has changed. Controlled experiments on mobile applications have measured that maintenance effort due to graphical fragilities in VGT test suites can account for up to 30% of total test maintenance time [21].

A final drawback of VGT tools is that they are commonly run in a desktop environment that emulates the AUT on an **Android Virtual Device (AVD)**. This approach adds an additional layer of fragility to visual tests, since the recognition of the locators is influenced by the rescaling operated on the AVD by the host OS [11, 37].

2.4 Computer Vision Techniques for Automated GUI Testing

Several VGT tools like Sikuli [2] and EyeAutomate [1] rely on CV techniques to automate the search of visual locators. The most common approach, which is also our work's goal, is to recognize a visual locator's location in the device screen capture using image matching algorithms.

The literature has documented several failures and issues associated with commercial and open-source VGT tools. One of the main difficulties in analyzing the causes of these problems is that the algorithms' full specifications are often undisclosed. Nonetheless, some data on the matching algorithms' performance (mainly exploiting their implementation in the OpenCV library) is available in the literature [17, 47, 48].

Among the available VGT tools, Sikuli is based on a simple *template matching* technique that searches the visual locator position in the target screenshot. The same approach is also discussed in Reference [47]. This algorithm simply slides the template image over the target screenshot and compares the template and the underlying target patch. This comparison involves a similarity metric such as the squared difference or the normalized correlation of pixel intensities. The sliding approach of template matching potentially requires sampling a large number of points. Thus, VGT tools typically reduce the search space's size by stopping as soon as a suitable match is found. However, this choice may be suboptimal if the same or similar widgets occur within the application. Moreover, template matching is not robust to variations in scale (which occur when the image is rendered on devices with different physical characteristics) and rotation and color (which are relatively frequent during the evolution of the AUT [49]), leading to wrong executions due to image recognition failures.

More recently, matching algorithms based on local feature descriptors emerged as a more effective and robust alternative to pixel-based similarity. Feature matching refers to a set of CV techniques that aim to find corresponding or similar image patches in two images, with numerous applications including stereo matching and image retrieval [25]. A feature matching algorithm comprises three essential steps: *keypoint extraction*, *local feature extraction*, and *keypoint matching*.

The goal of keypoint extraction is to find distinctive locations that can increase the uniqueness of the features extracted, usually promoting high-contrast regions of the image, such as object corners. These keypoints are then associated with a *local feature vector*, i.e., a local texture *descriptor* that provides a strong characterization of the local image patches surrounding an image point, invariant to several image variations (like color, shape, scale, and rotation). Several local feature descriptors have been defined [39, 59]. Popular choices include SIFT [39], SURF [14], and Akaze [29], which differ in terms of computational cost, accuracy, and robustness [54]. Finally, keypoints extracted from the two images are matched using suitable metrics in the feature space (i.e., matched keypoints identify visually similar regions with a low distance in the feature space) to establish a point-to-point correspondence between the two images.

An example of keypoint descriptors' application to VGT is MAuto [58], a tool that generates test sequences for mobile games through the Record and Replay technique. MAuto leverages AKAZE feature descriptors to identify locators in the AUT. However, the excessive number of features in the query images used to identify locators limited the approach's applicability. As a workaround to this issue, the authors restricted the search to a limited region of the query image, which had to be appropriately and heuristically calibrated to provide a suitable number of features for identifying the locators.

Available studies related to feature matching algorithms applied to the VGT tasks suffer from two main limitations. The first is algorithmic in nature, as most of the state-of-the-art tools and approaches take as input only the cropped image of a single input visual locator [1, 2, 58]. This approach is not robust in the presence of repeated or similar widgets, where the locator could be potentially matched to multiple widgets. To address this issue, we propose a Fullscreen matching algorithm that leverages all visual information available in the source and target screenshots, and compare it experimentally with the state-of-the-art approach. The second limitation lies in the limited experimental validation, which is usually based on a few selected apps or case studies [58]. Previous studies therefore do not provide a comprehensive assessment of state-of-the-art CV algorithms. To close the gap, we offer to the community DatAndroid, a dataset of roughly 100 apps rendered on 14 different devices, as well as an experimental methodology to systemically compare the robustness of different algorithms and descriptors in the presence of device fragmentation.

3 FEATURE MATCHING BASED VGT

As mentioned in the introduction, our primary goal was to design and develop an approach capable of (i) matching widgets rendered on Android app screens, (ii) performing interactions on individual widgets, and (iii) running test sequences on them. The main constraint on our method is that it must locate the widgets of interest on various devices where the app may run. In the following, we adopt these definitions:

- *Source device*: the device where the visual locator is captured,
- *Target device*: the device where the locator should be found.

Our purely visual testing approach uses only rendered screens as input and, differently than several state-of-the-art applications of VGT to Android apps, we search for the visual locators directly in the screen captures obtained by **Android Debug Bridge (ADB)** interaction with the Android devices rather than in the renderings of the screen on the host desktop system where the device is emulated. This approach's advantages are twofold: it can be applied to both emulated and real devices, and it does not need to consider the resizing of the virtual device GUI operated by the host OS. We also emphasize the fact that, in principle, the source and target devices may coincide. This condition occurs, for example, when comparing different versions of GUIs that have graphical variations.

Table 1. Input and Output of the Two Algorithms

	Fullscreen algorithm	Cropped algorithm
Input/ Locator	fullscreen capture of the source device screen, bounding box of the widget	Cropped screen capture containing only the widget
Result	x, y coordinates belonging to the widget in the target device screen	

The proposed feature matching-based VGT is based on determining, for each visual locator in the source screenshot, its location in the target image (or lack thereof). This analysis leverages feature matching algorithms, which identify in the target the location of a region of the source image that represents the widget to be searched. Under these conditions, the matching algorithms can be handled in two different ways, referred to in the following as *Fullscreen* and *Cropped* (summarized in Table 1). The main steps of both algorithms are illustrated and compared in Figure 2.

Assuming for simplicity that the test involves a single widget, the visual locator on the source device (e.g., the bounding box of the widget) is identified in both cases by a human tester during the creation of the test suite. The main difference between the two algorithms is that, in Fullscreen, we use for identification all visual information available in the source screenshot, whereas, in Cropped, we use only the information present in the visual locator.

The identification of the position of the visual locator is based on established feature matching techniques. These techniques leverage local texture descriptors (see Section 2.4 for details and Section 4.3.2 for the description of the specific descriptors analyzed in our research). For tasks like image registration, object tracking, and recognition, robust matching algorithms can be applied to pair keypoint descriptors obtained from the source and target images. These algorithms work as follows. First, the best match between descriptors is computed by means of a suitable metric in the feature space (e.g., euclidean distance). Then, since the extracted feature point pairs might suffer from significant correspondence errors or mismatches in the pairing, a common strategy is to postprocess candidate matches with robust data fitting techniques such as **Random Sample Consensus (RANSAC)** [13]. RANSAC starts from an initial estimate of the homography relating the two images (computed from all the matches) and then iteratively removes the outliers, i.e., the matches that are not consistent with the estimated homography, to update the homography with the remaining set of inliers. Figure 3 shows an example of matching a source and target snapshot, where the initial keypoint matches (Figure 3, left) are "cleaned" applying RANSAC as a post-processing step (Figure 3, right). The remaining matches will be the base of the clustering method used to identify the locator position in the target image.

In the following sections, we detail the Fullscreen and Cropped algorithms where, for the sake of clarity and without loss of generalization, we assume again that the test involves a single widget. Implementation details are reported in Section 4.3.2.

3.1 Fullscreen Matching Algorithm

The Fullscreen matching algorithm (Algorithm 1) is summarized in the top part of Figure 2. It receives as input both the source and target fullscreen images, along with the (manually identified) bounding box of the source locator.

With this approach, we first extract the keypoints from both source and target images, and then we post-process their matches with RANSAC. To identify the widget's target location, we use the widget bounding box to prune the set of matches (i.e., by discarding all matches whose source keypoints fall outside the bounding box). Finally, to obtain the most likely interaction coordinate

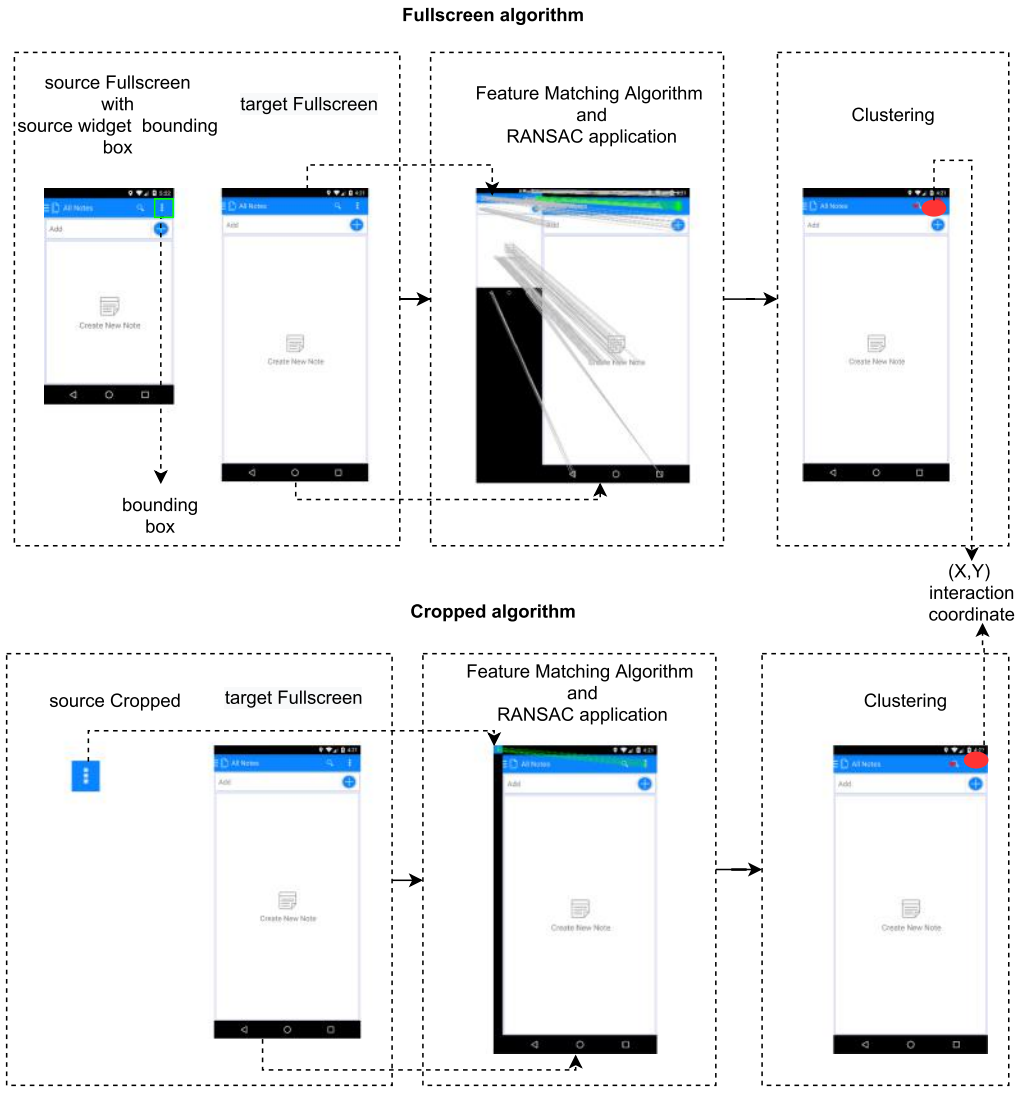


Fig. 2. Schematic representation of the two matching algorithms compared in this work. The example shows the AcsNotes app with Nexus S and Pixel 3a XL as source and target devices, respectively. First row: *Fullscreen*. From left to right: the source and target screenshot (the green region indicates the visual locator); results of the keypoint matching (green lines indicate the matched keypoints associated to the source widget, grey lines the other matched keypoints); the identified interaction coordinates (identified by the larger red circle). Second row: *Cropped*. From left to right: the cropped widget and target screenshot; results of the keypoint matching; the identified interaction coordinates (identified by the larger red circle). Both algorithms match the source widget with two potential target widgets, identified by the clustering step as two clusters of keypoints (highlighted by the small and large red circles); the largest one is selected as interaction coordinates.

pairs (i.e., the widget interaction coordinates on both source and target, which are needed by the test procedure), we apply the MeanShift clustering algorithm [30] to the source and target keypoints. On each device, the largest cluster found identifies the target locator, and its centroid is returned as the final output of the matching algorithm.

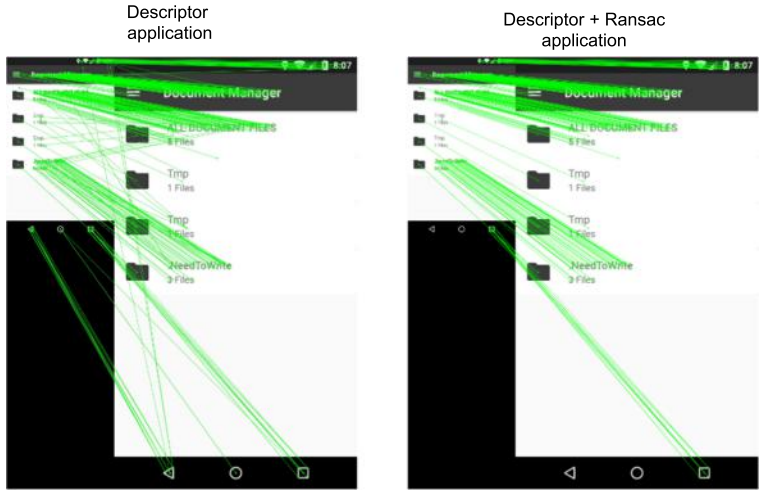


Fig. 3. Example of RANSAC post-processing (in Fullscreen algorithm). Left: Initial keypoint matches. Right: RANSAC-refined matches. In both images, the green lines indicate the matched keypoints.

ALGORITHM 1: Fullscreen Matching Algorithm

```
1: FsSource ← Source image of Full Screenshot
2: FsTarget ← Target image of Full Screenshot
3: Src_bb_target ← Source bounding box
4: function FullScreen(FsSource, FsTarget, Src_bb_target)
5:     src_pts, targ_pts ← feature_matching(FsSource, FsTarget)
6:     src_pts, targ_pts ← RANSAC(src_pts, targ_pts)
7:     src_pts, targ_pts ← select_target_keypoints(src_pts, targ_pts, Src_bb_target)
8:     click_coordinates ← Meanshift(targ_pts)
       return click_coordinates
```

The advantages of the Fullscreen algorithm are twofold. First, it reduces the time needed to process the entire test suite if more than one widget has to be extracted for the test generation. Second, RANSAC relies on the computation of a homography that implicitly favors spatial widget arrangements that are similar between source and target. Therefore, when the same (or similar) widgets are repeated in the GUI, it is possible to match each widget with its most similar counterpart, with RANSAC ensuring that the final matching is also spatially coherent. A typical example that can benefit from these characteristics is a calendar application, where the same numeric symbols represent different days of the month. However, in these situations, RANSAC may occasionally remove a correct match. The main disadvantage of this algorithm, however, is the higher computational cost than Cropped when a single widget has indeed to be identified.

3.2 Cropped Matching Algorithm

The Cropped matching algorithm (Algorithm 2) is summarized in the bottom part of Figure 2. It receives as input the cropped image of the source device widget and the target screenshot (i.e., the same input required by Sikuli and other comparable tools). The source keypoints are extracted only from this cropped image and matched to those extracted from the target screenshot. Again,

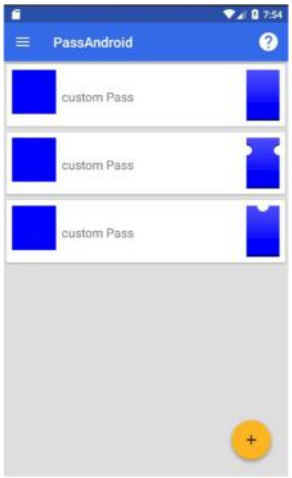


Fig. 4. Sample content for the main Activity of PassAndroid with repeated visual locators in the same screen (e.g., the "custom Pass" text box, and the blue squares on the left).

ALGORITHM 2: Cropped Matching Algorithm

```
1: CsSource ← Source image of Cropped Screenshot
2: FsTarget ← Target image of Full Screenshot
3: function CROPPED(CsSource, FsTarget)
4:     src_pts, targ_pts ← feature_matching(CsSource, FsTarget)
5:     src_pts, targ_pts ← RANSAC(src_pts, dst, targ_pts)
6:     click_coordinates ← Meanshift(targ_pts)
       return click_coordinates
```

we apply RANSAC for outlier removal and MeanShift for the identification of the interaction coordinates.

This algorithm's main advantage is that it is faster than Fullscreen when the test involves a single widget. However, there can be identification errors when the source locator is present multiple times in the target screen (as in the example in Figure 4, which shows a screenshot of the PassAndroid application that contains repeated graphical and textual content in the same Activity). In these cases, the matching algorithm returns at most one locator, which is not necessarily the correct one.

4 EXPERIMENTAL DESIGN

Our experimental assessment, summarized in Figure 5, addresses two different aspects, the feature matching algorithms and the visual testing process.

4.1 Goals

We report the **Goal-Question-Metric (GQM)** [18] template for the study in Table 2.

The first goal aims at assessing the effectiveness of feature matching algorithms in identifying widgets on Android GUIs. We compared different algorithms and descriptors based on standard performance measures for classification and retrieval algorithms (recall, precision) and

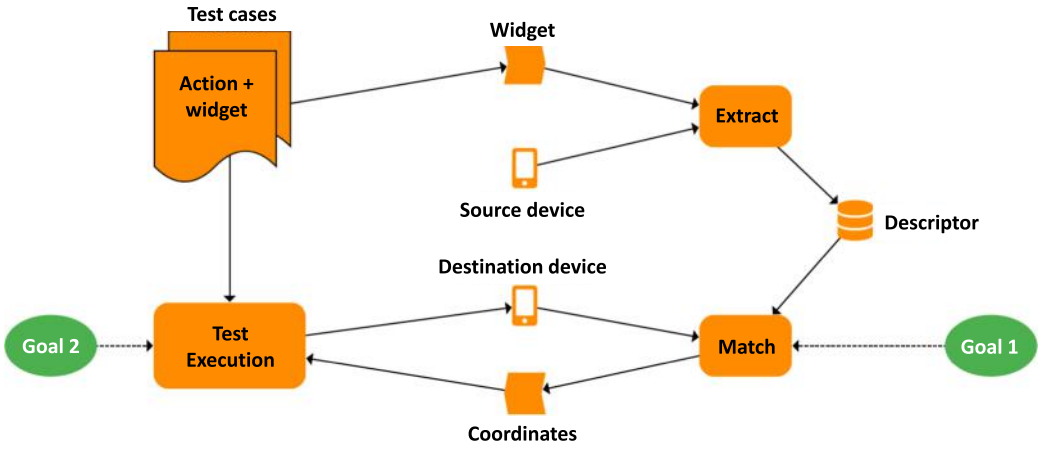


Fig. 5. Visual experimental design summary.

Table 2. Goal-Question-Metric Template for the Study

	Goal 1	Goal 2
Object of Study	Feature matching algorithms	Visual testing
Purpose	Compare different descriptors and matching algorithms	Compare with state-of-the-art VGT tools
Focus	Precision, recall, execution time	Device fragmentation, working tests, portability
Context	Matching of mobile GUI components	Mobile GUI test suites
Stakeholders	Researchers	Researchers, developers, testers

execution time. The results are then interpreted according to researchers' perspective in the CV field, providing evidence about the performance of such techniques in the domain of Android GUIs. The aim is to enable selecting the combination of algorithm and descriptor with the highest performance on a wide range of applications, widgets, and source/target devices.

The second goal concerns feature matching algorithms' applicability for recognizing visual locators and oracles in visual test suites for Android applications. We compared the feature matching technique with state-of-the-art VGT tools by considering the impact of device fragmentation on test scripts. The results pertaining to this goal are then interpreted according to researchers' perspectives in the GUI testing field, testing tool creators, and developers. Although, in theory, we expect that a feature matching technique with higher performance can enhance test portability, several factors may reduce or increase portability in practice. For example, the impact of individual widgets on final performance is unbalanced, because some of them are selected more frequently than others (and thus, their correct recognition is more critical).

4.2 Research Questions and Metrics

4.2.1 RQ1: Feature Matching Performance. To achieve the first goal of the study, we formulated the following research question:

RQ1: How well do feature and template matching algorithms perform when applied to Android GUI widgets?

The research question was divided into the following sub-questions:

RQ1.1: Which widget matching algorithm between Fullscreen and Cropped performs best?
RQ1.2: Which feature descriptor between SIFT, SURF, and AKAZE performs best? How do they compare with template matching approaches?
RQ1.3: How do feature descriptors and matching algorithms compare in terms of execution time?
RQ1.4: Which are the main issues for widget matching using feature descriptors?

To answer RQ1.1 and RQ1.2, we resorted to precision and recall, standard performance measures for retrieval and classification techniques.

We performed an overall performance assessment considering all the widgets extracted from the source and target devices' screen hierarchy. The advantage of this approach is that it enables a comprehensive, large scale and easily automated performance evaluation; however, it does not account for the fact that different types of widgets are selected more or less frequently in test suites and thus have different impacts on the perceived performance of the VGT tool.

From the screen hierarchy of each device, we extracted for each widget the bounding box, along with the `content-desc`, `text`, and `resource-id` properties. This information is used to generate the *reference standard* or *ground truth*: for each pair of devices, two graphic components represent the same widget if they have the same id, text, and description. In addition, for each leaf element in the screen hierarchy, we trace the path to its root: if two paths traverse the same containers, preserving the spatial order, they are assigned to the same widget. This constraint allows us to enforce each widget's uniqueness, preserve the relationships between them, and account for cases in which some properties are left empty by the app developer. Another critical aspect to be considered is the timing of the app execution. In particular, the screenshot and dump grabbing must be synchronized and executed after the application is fully rendered to guarantee that the two are correctly aligned.

Based on this reference standard, we are able to define which locators are correctly and which are incorrectly matched by the visual algorithms.

In particular, given two visual locators, one in the source screen and one in the target screen, we define:

- ***True Positive (TP)***, if the locators correspond to the same widget in the reference standard and are matched by the algorithm;
- ***False Positive (FP)***, if the locators do not correspond to the same widget in the reference standard but are matched by the algorithm;
- ***False Negative (FN)***, if the locators correspond to the same widget in the reference standard, but they are not matched by the algorithm.

Precision and recall are then calculated as

$$precision = \frac{\sum TP}{\sum TP + \sum FP},$$

$$recall = \frac{\sum TP}{\sum TP + \sum FN}.$$

To answer RQ1.3, we measured the total time needed for the feature matching algorithms to be applied to the subject images, including feature extraction and matching. In the Fullscreen algorithm, we estimated the processing time as the time to process the entire source and the target screens, plus the additional processing time due to the clustering phase. In the Cropped algorithm, we estimated the time to recover the coordinates for a widget as the sum of the descriptor calculation time and the clustering phase. All calculations prudentially refer to the worst-case scenario

in which a single widget is matched on each target screenshot; in the case of multiple widgets to be matched simultaneously, the Fullscreen algorithm's processing time should be divided by the number of widgets to be matched.

4.2.2 RQ2: Application to GUI Testing. To achieve the second goal of the study, we formulated the following research question:

RQ2: How can feature matching algorithms enhance the portability of GUI test cases in the Android domain?

The research question can be divided into the following sub-questions:

RQ2.1: How do feature description algorithms compare with state-of-the-art visual testing tools in terms of portability of test scripts to different devices?
RQ2.2: How do feature description algorithms compare with state-of-the-art visual testing tools in terms of performance?

To answer RQ2.1, we measured the following metrics on the executions of visual test cases on different devices:

- the number of test cases that are executed correctly on different devices,
- the number of correctly identified unique locators,
- the number of correctly performed interactions.

To answer RQ2.2, we measured the total execution time of the scripts, as well as the average time obtained by dividing the former time by the total number of interactions.

4.3 Experimental Subjects and Instruments

4.3.1 Selected Applications. For the first research question, we mined applications from the UpToDown .apk store,[2] a marketplace providing more than 50k free Android apps already used as a source for experimental studies [44, 45]. We limited our work to a specific category of Android apps to avoid considering applications with very different graphical appearances, e.g., games or apps with prominent multimedia content. We selected the *Writing and Notes* category, containing mostly utilities to manage and organize text-based content and item lists. We mined with a Python script the entire population of 195 apps belonging to the selected category as of January 2020. We also verified whether the apps were compatible with the set of emulated devices that we selected for our analysis. After this verification phase, we came up with a population of 95 valid apps.

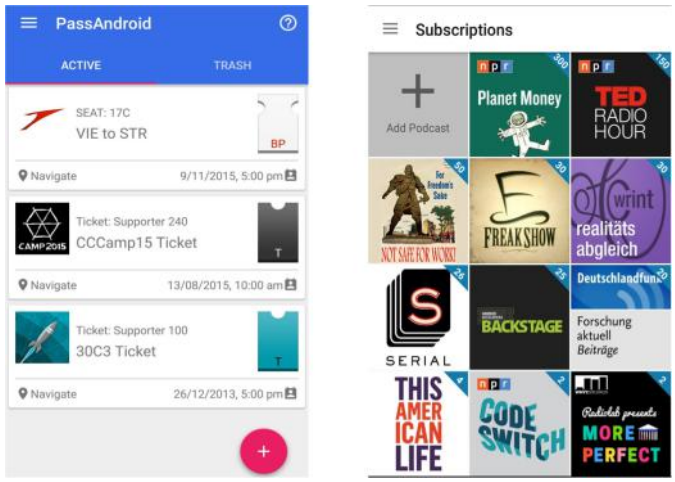
For the second research question, we selected two different experimental subjects: PassAndroid [8, 22, 43, 61], a tool for storing and managing different types of tickets through QR codes belonging to the Writing/Notes category of Android apps; AntennaPod [33, 35, 40, 50, 56], an application for listening and managing podcast subscriptions, belonging to the Multimedia/Video category of Android apps. The two apps are popular and long-lived open-source projects on GitHub and are available on the PlayStore and FDroid. We report details about the two apps in Table 3 and sample screen captures in Figure 6.

The test suites were developed by one of the authors of this study with the Appium automation framework. The PassAndroid test suite was partly based on an existing test suite used for a previous study [23]. The author was given no indication about the purpose of the test suite and the designed experiment to avoid possible biases in creating the test scripts. We generated the screen

[2] https://en.uptodown.com/.

Table 3. Details About the Experimental Subjects Selected for RQ2

	PassAndroid	AntennaPod
GitHub commits	1,697	7,281
GitHub releases	104	98
GitHub stars	542	3.4k
GitHub forks	120	928
PlayStore downloads	1M+	500k+
PlayStore rating	4.3*	4,7*



(a) PassAndroid - Pass Activity

(b) AntennaPod - Subscriptions Activity

Fig. 6. Sample screen captures of the experimental subjects for RQ2.

Table 4. Details about Interactions and Locators of the Test Suite Developed for the Second Experimental Goal

	PassAndroid			AntennaPod		
Property	Total	Avg.	Median	Total	Avg.	Median
Number of interactions	190	6.3	6	306	10.2	11
Interactions requiring visual locators	170	5.6	5	234	7.8	8
Number of interacted widgets	46	5.3	3	77	7.4	8
Number of different locators	52	5.3	5	77	7.4	8

captures and visual locators for each interaction with the GUI by tracing the Layout-based test suite's execution with proper callbacks.

The characteristics of the test suite are reported in Table 4. The suite consists of 30 different test cases with a varying number of interactions. Not all the interactions of the test suite require a visual locator (e.g., *PressBack* or *GoHome* are directly executed by the tool through calls to APIs of the ADB and not by interacting with the emulated GUI). The actual number of interacted widgets is markedly lower than the number of interactions performed in the tests, meaning that there are multiple interactions with the same widgets in different tests or even in the same test. It is also

worth noting that, in the case of PassAndroid, the number of different visual locators is higher than the number of unique interacted widgets. This fact means that in different test cases or different execution moments, the same widget may have different graphical appearances.

4.3.2 Feature Matching Implementation Details. We used the latest releases (at the time of the experiment, in June 2020) of the Python version of OpenCV and RANSAC libraries (3.4.2.17).

In this work, we compare three scale, rotation and translation invariant feature matching techniques: SIFT [39], SURF [14], and AKAZE [29, 58]. They represent classes of descriptors with different trade-offs in terms of accuracy, memory occupation, and execution time [55]. AKAZE was already successfully used in VGT tools [58]. It belongs to the class of binary descriptors (like ORB and BRISK) and offers a reduction of the computational burden. To perform the actual matching, we use brute force matching, i.e., each keypoint is associated with the closest one in the feature space using as a metric the Euclidean distance for SIFT and SURF and the Hamming distance for AKAZE.

As for the RANSAC, we used a recent variant named Graph-Cut RANSAC [13], which is faster and more accurate than standard RANSAC. Graph-Cut RANSAC is applied with the following parameters: the smallest number of data points to evaluate model parameters is set to 10, the maximum number of iterations to 1,000, and the threshold value (to determine which data points are fit by the model) is set to 100.

4.3.3 State-of-the-art VGT Tools. As state-of-the-art tools to be included in the comparison, we selected EyeAutomate [6], release 2.2, and SikuliX [62], release 1.1.2, since they are the most cited in empirical studies on visual testing. We have used the tools by leveraging the provided APIs in Java.

To achieve a more systematic comparison with state-of-the-art tools for RQ1, we re-implemented the matching algorithm employed by the open-source software SikuliX using the same library (OpenCV) and settings employed in the tool. This choice made it possible to extract the raw matching performance of SikuliX on a widget-by-widget basis, exploiting the experimental setup and script developed for RQ1. This approach could not be used for EyeAutomate, since the latter is not open source and leverages a proprietary algorithm on which it was not possible to apply reverse engineering.

4.3.4 Android Virtual Devices. As our emulated Android devices set, we leveraged the 14 default devices in the Android AVD Manager. The properties of the considered devices are reported in Table 5. All the devices used Android API 25 (version 7.11) and mounted x86 system images. The emulated devices were not hardware-accelerated, had device frame and keyboard inputs enabled, while all the animations were disabled to avoid errors in transitions between Activities. In the table, we reported in bold the devices that we used as sources for the experiments. As it can be seen, we selected three devices with very different resolutions and pixel densities.

4.3.5 Experimental Setup. All the experiments were performed on a desktop PC with an Intel i7-4770 running at 3.40 GHz clock, with 8 GB RAM and Ubuntu 20.04 LTS 64-bit as OS.

4.4 Procedure

4.4.1 RQ 1. We designed and implemented an automatic procedure for performance assessment across our app's database; this procedure allows extensive validation with minimal human effort.

We evaluated the feature matching algorithms on all the widgets shown in the main Activities for each pair of source and target devices. We only used these Activities to avoid the need to navigate the different screens of the apps. We leveraged the Android emulator's debugging capabilities

Table 5. List of Emulated Devices Considered for the Research

Device model	Screen size (pixels)	Resolution
Nexus 5	**1,080 × 1,920**	**xxhdpi**
Pixel 3	1,080 × 2,160	440dpi
Pixel 2	1,080 × 1,920	420dpi
Nexus 5X	1,080 × 1,920	420dpi
Nexus 6P	1,440 × 2,560	560dpi
Nexus 6	1,440 × 2,560	560dpi
Nexus S	**480 × 800**	**hdpi**
Pixel 3a	1,080 × 2,220	440 dpi
Pixel 3a XL	1,080 × 2,160	402 dpi
Pixel	1,080 × 1,920	420 dpi
Nexus 4	768 × 1,280	xhdpi
Pixel 3 XL	**1,440 × 2,960**	**560 dpi**
Pixel 2 XL	1,440 × 2,880	560 dpi
Pixel XL	1,440 × 2,560	560 dpi

to retrieve from all the devices detailed information about these Activities; to this aim, we used the *dump files* containing all properties of the current visualized widgets.

The dump files' information can emulate the cropping and bounding box drawing operations that the tester would perform to prepare the visual test suite. It should be stressed that the information contained in the dump files is not used by the matching algorithm (which relies purely on the visual content) but is merely exploited to automate the assessment procedure.

In our experiments, we extracted all possible widgets from the source device screen and attempted to find the corresponding visual locators in the target device screen with different algorithms and descriptors. We repeated the procedure on 3×13 pairs of devices, using one of the three selected devices as source and the remaining 13 devices as the potential target. Recall and precision were separately computed for each app, then their distribution was calculated over the entire database grouping by the statistical factors of interest.

Like in a real test case, we assume to perform matching based only on the visible components. Due to fragmentation, some components may be rendered on the source device but not on the target device (Figure 1). Since our approach is purely visual, such locators are not included in the ground truth, and any accidental matching would be counted as an FP. Finally, it is worth underlining that a VGT tool would fail if the missing component is included in a test suite. This entails that even a perfect recall does not guarantee, in principle, perfect test portability. This aspect is taken into account in RQ2 and the analysis of the generated dataset.

4.4.2 RQ 2. We collected the screen captures and the cropped widget locators for all interactions in the test suite on the three devices we used as sources. Then, we executed the test cases on all 14 devices of the set and measured the number of correctly identified locators, correctly performed interactions, and completely executable test cases. A test case is considered completely executable if a given feature matching algorithm correctly identifies all locators used in it. We also considered the situation in which the source and target devices coincide, since the matching algorithms may yield wrong coordinate pairs even if the test script is applied on the same device where the locators were captured (e.g., in the case of multiple widgets with the same appearance). The application of a test script on the original device is a common scenario, since VGT techniques can be used for regression testing on new releases of the application.

We ran all test cases by embedding the feature matching algorithms (which provide as output the coordinate pairs of the identified widgets) in scripts that executed the found coordinates' interactions by using the Appium library. We verified the correctness of the resulting coordinates by checking the dump files obtained after each interaction.

We then used EyeAutomate and Sikuli with the cropped screen captures to replicate the test executions. This time, we used only the Cropped captures, since the pixel-per-pixel comparisons used by the tools would mostly lead to failures in recognizing fullscreen captures even in the presence of minor changes in device screen sizes.

Visual test scripts were always executed on a solid black background to minimize other visual elements' interference. No other computationally intensive program was run concurrently to avoid external influences on execution times.

4.4.3 Statistical Analysis. A multi-way factorial **Permutational Analysis of Variance (PERM-ANOVA)** [9] was conducted to examine the main effects of matching algorithm, descriptor, target, and source device, as well as the interaction effects between target and source devices on each metric defined for research questions RQ1 and RQ2. PERM-ANOVA is a non-parametric multivariate statistical test. The null hypothesis tested by PERM-ANOVA is that the centroids of the partitions or groups defined by a similarity metric (e.g., the Euclidean distance) are equivalent for all groups. Exact p-values are obtained by calculating the test statistics' value for all (or a large random subset) of permutations of the observations across different groups. In particular, the proportion of the values of the statistics under different permutations (i.e., random re-allocation of individual samples to different groups) that are equal to or higher than the observed value. If the null hypothesis was true, then any observed differences would be similar to those obtained under permutation. PERM-ANOVA only requires exchangeability and does not make any assumption on the sample distribution, accommodating severely non-normal variables, contain a large number of zeros, or are ordinal or qualitative in nature [9].

For RQ1, the matching algorithm and feature descriptors were modeled as separate fixed factors, along with their interaction. For RQ2, we considered the overall technique as a fixed factor, which could be either one of the combinations of the descriptor and matching algorithm (e.g., SIFT–Fullscreen) or one of the state-of-the-art VGT tools. Each app was modeled as a different subject with repeated measures. *Post hoc* comparison between the different combinations of matching algorithms and feature descriptors was performed using distribution-free pairwise two-sample permutation tests [41] applying Bonferroni correction. We also measured the effect size for any pair of groups of observations by using Cliff's delta formula. Statistical analysis was performed in R, and the *effsize* package was adopted for effect size computation [57].

The raw data and markdown scripts have been made available on a GitHub repository.[3] Null hypotheses, detailed results for the *post hoc* comparison and values of effect size are reported in the supplementary material of the present manuscript.[4]

5 RESULTS

5.1 Dataset Characteristics

In this section, we describe the characteristics of the subjects collected for the RQ1 study. In the population of 95 apps, a total of 787, 850, and 900 unique widgets were identified for the Nexus S, Nexus 5, and Pixel 3 XL devices, respectively.

[3] https://github.com/SoftengPoliTo/image_matching_study.
[4] https://figshare.com/articles/online_resource/Feature_Matching-based_Approaches_to_Improve_the_Robustness_of_Android_Visual_GUI_Testing_Supplementary_material/14912292.

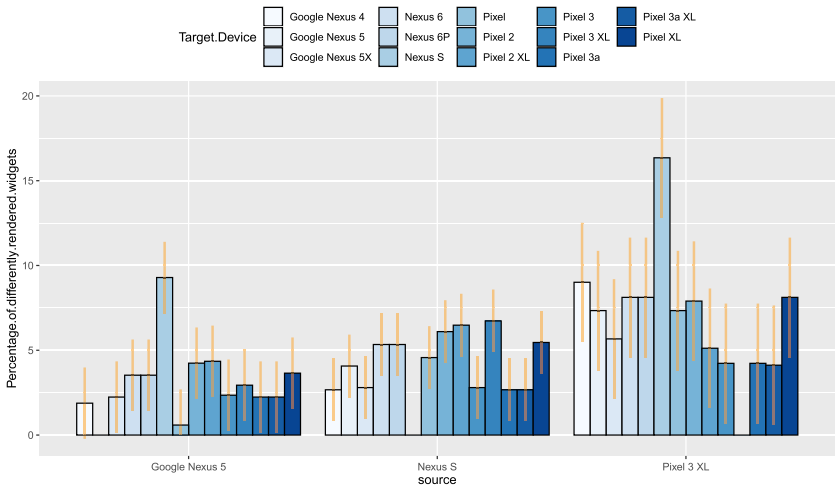


Fig. 7. Percentage of widgets that are present in the source device but not in the target device, based on the comparison of the XML trees in the dump files. The percentage is calculated separately for each source and target device pair and is higher for device pairs with very different screen sizes. Yellow lines report the intervals of confidence calculated on a binomial distribution.

As mentioned in Section 2.3, some widgets could not be localized in both the source and target devices. A number of possible causes were identified for this phenomenon. Most commonly, Android performs rescaling and resizing operations to optimize the User Experience, adapting the interface layout to the screen size, which in turn changes the position, dimension, and resolution of the widgets. Some widgets may be grouped together to reduce visual clutter on small devices. In fact, the average number of individual widgets per app is proportional to the screen size of the emulated device, ranging from 8.11 (Nexus S) to 9.28 widgets/app (Pixel 3). Less frequently, discrepancies between the source and target devices arise due to banners incorporated in the app's layout, which are selected randomly and remain on screen with an automatic refresh time.

The above-mentioned differences may constitute an intrinsic limitation to the performance of visual matching algorithms, at least in the current Record and Replay scenario, which assumes that the interactions recorded on the source device, and their visual locators, can be found and replicated on the target device. To estimate the order of magnitude and potential impact of this issue, we calculated for each device pair the percentage of widgets (mean and confidence interval) that were present in the ground truth of the source device but not in the ground truth of the target device, and reported the distribution in Figure 7. It should be noticed that this percentage is independent of the specific algorithm or visual testing tool and depends solely on the combination of source and target devices. The percentage of widgets that cannot be correctly located is in most cases well below 10%, except for the two devices with the largest difference in resolution and screen size (Nexus S and Pixel 3 XL).

5.2 RQ1: Feature Matching Performance

Table 6 shows the average and standard deviation values for the metrics selected for RQ1, namely, recall, precision, and execution time.

The distribution of the recall and precision is reported in Figures 8 and 9. Recall and precision are calculated for each app and for each target device and then grouped by source device and technique. Overall, there is a statistically significant difference across all groups for both precision

Table 6. Average Values (and Std. Deviation) of Precision, Recall, and Execution Time, Grouped by Technique

	Recall	Precision	Execution time (s)
SIKULI–Cropped	0.38 (0.33)	0.46 (0.37)	**0.41 (0.49)**
AKAZE–Cropped	0.71 (0.31)	0.84 (0.26)	0.69 (0.71)
SIFT–Cropped	**0.95 (0.12)**	0.91 (0.17)	1.0 (0.51)
SURF–Cropped	0.91 (0.18)	0.86 (0.21)	0.99 (1)
AKAZE–Fullscreen	0.87 (0.20)	0.92 (0.19)	1.28 (0.7)
SIFT–Fullscreen	0.88 (0.19)	**0.93 (0.18)**	1.48 (0.63)
SURF–Fullscreen	0.89 (0.18)	0.92 (0.18)	1.31 (1.06)

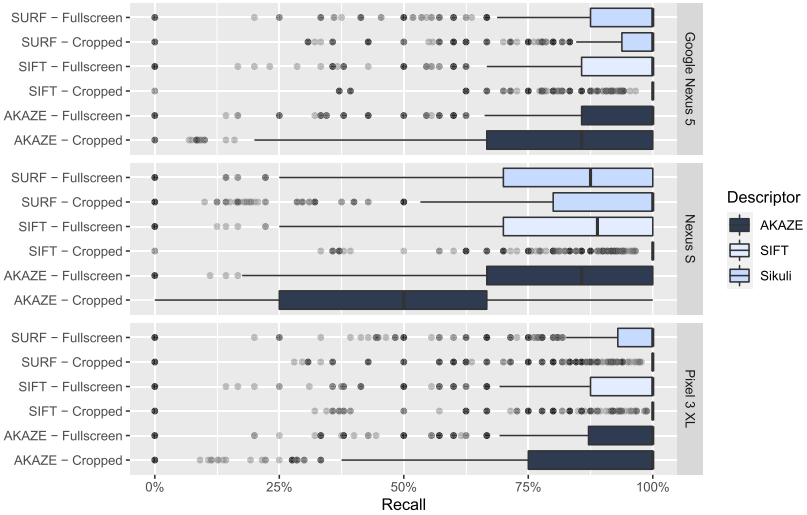


Fig. 8. Distribution of the recall over the set of apps and target devices, per source device and technique. Recall is reported for each combination of descriptor and matching algorithm, Fullscreen (blue) vs. Cropped (light blue).

(p-values < 2e−16) and recall (p-values < 2e−16). In our results, precision and recall exhibit similar behavior: Since a given source widget on the source device can be associated only to a single target widget, a correct matching would increase both precision and recall, whereas an incorrect matching would affect both. We thus report for brevity a detailed analysis only for recall.

We observed a significant effect of matching algorithm, descriptor, and their interaction on both precision and recall (p < 1e−10). At *post hoc* analysis, the Fullscreen technique achieved higher recall than the Cropped technique for the AKAZE (p < 1e−10), SIFT (p = 2e−07), and SURF descriptors (p < 1e−10). For the Cropped technique, SIFT and SURF outperformed both the AKAZE descriptor (p < 1e−10) and Sikuli template matching (p = 0.0); SIFT also achieved higher recall than SURF (p = 0.0). For the Fullscreen technique, SURF slightly outperformed both AKAZE (p = 3.5e−05) and SIFT (p = 2e−13), whereas differences between SIFT and SURF were not statistically significant (p = 0.09). Sikuli can only operate in the Cropped modality, hence *post hoc* comparison with the Fullscreen algorithms was not attempted. The worst performing technique was Sikuli, followed by the Cropped algorithm with the AKAZE descriptor. In general, the Fullscreen algorithm appeared more robust to the choice of the descriptor.

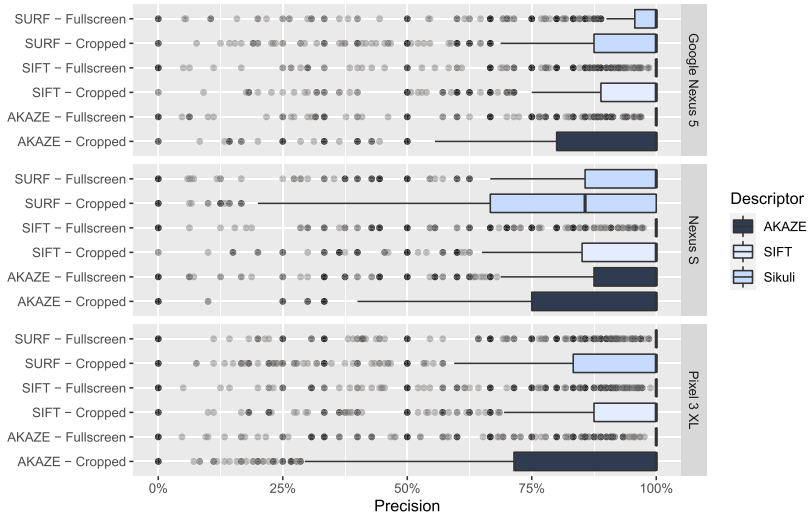


Fig. 9. Distribution of the precision over the set of apps and target devices, per source device and technique. Recall is reported for each combination of descriptor and matching algorithm, Fullscreen (dark blue) vs. Cropped (light blue).

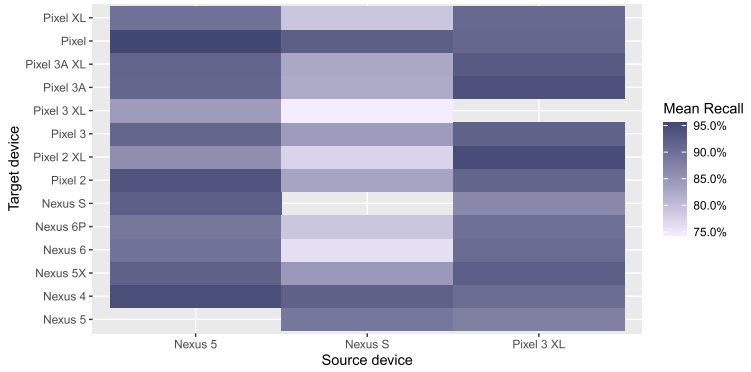


Fig. 10. Average recall of the SIFT–Fullscreen combination, with varying source and target devices.

The source and target devices had a significant effect on the recall ($p < 1e{-}10$) and, as expected, a significant interaction ($p < 1{-}10$). The best performance was obtained when using the Pixel 3 XL as a source (the device with a larger screen size and resolution), whereas starting from the smallest device (Nexus S) as the source device, generally poorer results were observed. This is further illustrated by the color map in Figure 10, reporting the recall for source and target device pairs for the best performing technique (i.e., SIFT–Fullscreen).

Both algorithms were, on average, quite successful in locating widgets. Across all tested source-target device pairs and apps, roughly 60% of the cases achieved a perfect recall of 100% (6,431/10,997 for the Fullscreen and 6,879/10,997 for the Cropped algorithm), meaning that all source widget locators could be correctly matched on the target device, whereas roughly 75% of them (8,208/10,997 and 7,919/10,997) achieved a recall higher than 80%. Therefore, the distribution is highly skewed on the left side, which explains a large number of outliers in the boxplots reported in Figures 8 and 9.

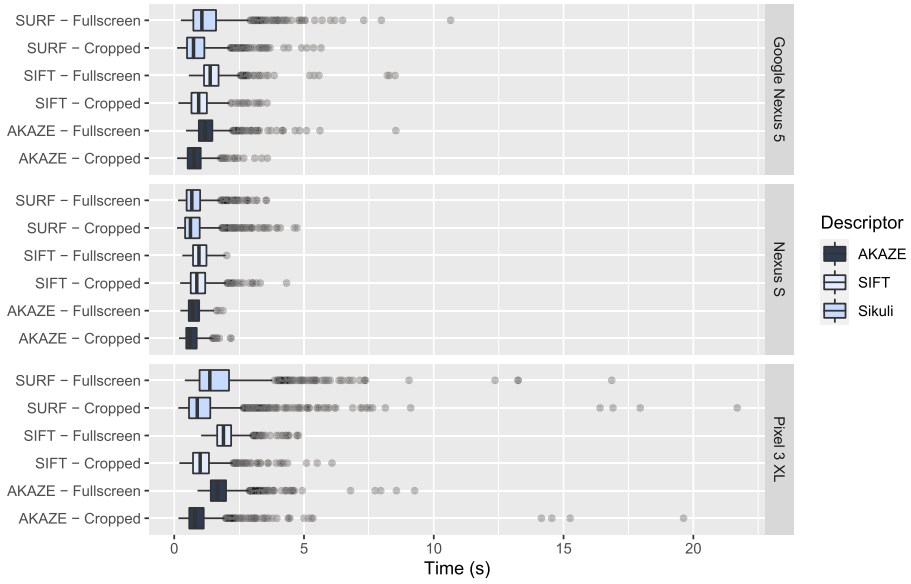


Fig. 11. Distribution of the execution time (in seconds) over the set of apps and target devices, per source device and technique. Recall is reported for each combination of descriptor and matching algorithm, Fullscreen (blue) vs. Cropped (light blue).

Nonetheless, in a small number of cases, 1% (185/10,997) and 4% (480/10,997) for the Fullscreen and Cropped algorithms, respectively, the recall was below 25%, i.e., most of the visual locators could not be identified on the target device.

At visual inspection, for the Fullscreen algorithm, such outliers appear to be mostly associated with widgets (including buttons) that display long text strings. Widgets that include text are processed as any other widget and, usually provide many robust keypoints, thus facilitating many-to-many feature matching. However, since the latter process is entirely visual and does not take into account the text semantics, the same letters can be matched in different words, leading in a few cases to a high number of FPs. RANSAC becomes less effective in eliminating outliers as the number of FPs increases.

The Cropped algorithm may fail when the keypoints in the source visual locator are matched with multiple target widgets. This can be due to repeated or similar widgets (see the example in Figure 4) or, like for the Fullscreen algorithm, to the presence of long text strings. In these settings, RANSAC does not have enough information to select the correct matching based on spatial consistency, which would require considering all the widgets in the source and target devices simultaneously. Thus, the results of the final clustering may be wrong.

Finally, the execution time is reported in Figure 11. The average processing time is higher for the Fullscreen algorithm (due to the need to calculate all the keypoints in the source image) and increases with the size of the device screen. A few outliers can be observed mostly due to apps with long text strings, which increase the number of matches. As expected from the literature, AKAZE is faster than SURF, whereas SIFT is the slowest descriptor.

5.3 RQ2: Application to GUI Testing

Table 7 reports the average and standard deviation values for the metrics measured to answer RQ2. The average values include the measurements on both the applications considered in the

Table 7. Average Values (and Std. Deviation) of the Percentage of Executed Interactions, Found Locators, Passing Tests, and Time Per Interaction Over All Executions of the Test Suites, Grouped by Technique

	Proportion of correctly executed interactions	Proportion of found locators	Proportion of correctly executed tests	Time per interaction
EyeAutomate	0.59 (0.25)	0.60 (0.23)	0.17 (0.22)	548 (313)
Sikuli	0.43 (0.24)	0.31 (0.26)	0.11 (0.27)	**373 (134)**
AKAZE–Cropped	0.72 (0.18)	0.67 (0.22)	0.24 (0.29)	1238 (458)
SIFT–Cropped	0.89 (0.09)	0.92 (0.08)	0.50 (0.37)	2041 (904)
SURF–Cropped	0.82 (0.16)	0.81 (0.15)	0.43 (0.32)	1925 (1057)
AKAZE–Fullscreen	0.94 (0.06)	0.94 (0.06)	0.76 (0.24)	1927 (1261)
SIFT–Fullscreen	**0.95 (0.06)**	**0.96 (0.04)**	**0.77 (0.25)**	2712 (1836)
SURF–Fullscreen	**0.95 (0.06)**	0.95 (0.04)	0.75 (0.22)	2737 (2044)

Table 8. Average Values (and Std. Deviation) of the Percentage of Executed Interactions, Found Locators, Passing Tests, and Time Per Interaction Over All Executions of the Test Suites, for the PassAndroid SUT

	Proportion of correctly executed interactions	Proportion of found locators	Proportion of correctly executed tests	Time per interaction
EyeAutomate	0.64 (0.24)	0.59 (0.19)	0.26 (0.24)	711 (399)
Sikuli	0.48 (0.24)	0.37 (0.26)	0.15 (0.29)	**418 (139)**
AKAZE–Cropped	0.83 (0.12)	0.78 (0.16)	0.37 (0.34)	1,268 (487)
SIFT–Cropped	0.93 (0.06)	0.95 (0.03)	0.73 (0.24)	1,476 (586)
SURF–Cropped	0.91 (0.06)	0.84 (0.11)	0.62 (0.22)	1,160 (440)
AKAZE–Fullscreen	0.94 (0.07)	0.93 (0.06)	0.78 (0.21)	1,144 (388)
SIFT–Fullscreen	0.93 (0.07)	**0.96 (0.04)**	0.74 (0.30)	1,287 (431)
SURF–Fullscreen	**0.96 (0.03)**	0.94 (0.04)	**0.85 (0.12)**	1,011 (313)

Table 9. Average Values (and Std. Deviation) of the Percentage of Executed Interactions, Found Locators, Passing Tests, and Time Per Interaction Over All Executions of the Test Suites, for the AntennaPod SUT

	Proportion of correctly executed interactions	Proportion of found locators	Proportion of correctly executed tests	Time per interaction
EyeAutomate	0.53 (0.25)	0.61 (0.26)	0.90 (0.17)	385 (169)
Sikuli	0.38 (0.23)	0.26 (0.26)	0.08 (0.26)	**328 (114)**
AKAZE–Cropped	0.63 (0.21)	0.56 (0.21)	0.11 (0.14)	1,225 (445)
SIFT–Cropped	0.84 (0.09)	0.89 (0.10)	0.28 (0.32)	2,610 (816)
SURF–Cropped	0.74 (0.18)	0.77 (0.17)	0.25 (0.29)	2,695 (943)
AKAZE–Fullscreen	0.95 (0.06)	0.95 (0.05)	0.73 (0.26)	2,711 (1,345)
SIFT–Fullscreen	**0.97 (0.04)**	**0.96 (0.04)**	**0.80 (0.19)**	4,137 (1,573)
SURF–Fullscreen	0.93 (0.07)	0.95 (0.04)	0.65 (0.26)	4,399 (1,490)

experimental procedure. Feature matching-based algorithms outperformed the state-of-the-art VGT tools in all the aspects related to RQ2.1. EyeAutomate and Sikuli showed lower portability for the three evaluated metrics with all the source devices. Tables 8 and 9 report the average and standard deviation values for the individual SUTs considered for the experiment.

Regarding the executed interactions, feature matching algorithms guaranteed a percentage of correctly executed interactions that ranged from 72% (for AKAZE–Cropped) to 95% (for SURF–Cropped), significantly higher than state-of-the-art tools, between 43% (Sikuli) and 59% (EyeAutomate). Hence, for the considered combinations of apps and emulated devices, our methodology increased the percentage of correctly executed instructions by at least 30% compared to the best performing state-of-the-art VGT tool (EyeAutomate). We observed a significant effect on the percentage of found locators of the matching algorithm, descriptor, source, and target device, and the combination of source and target devices (with p-values < 10e−16). At *post hoc* analysis, we verified

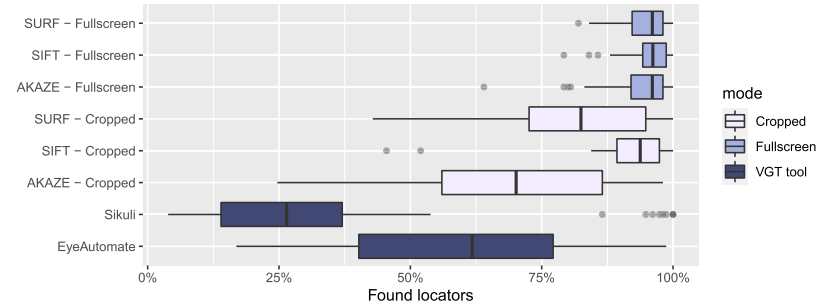


Fig. 12. Distribution of the percentage of found locators for each source and target device pair, grouped by technique. Available tools (dark blue), Fullscreen (blue), Cropped (light blue).

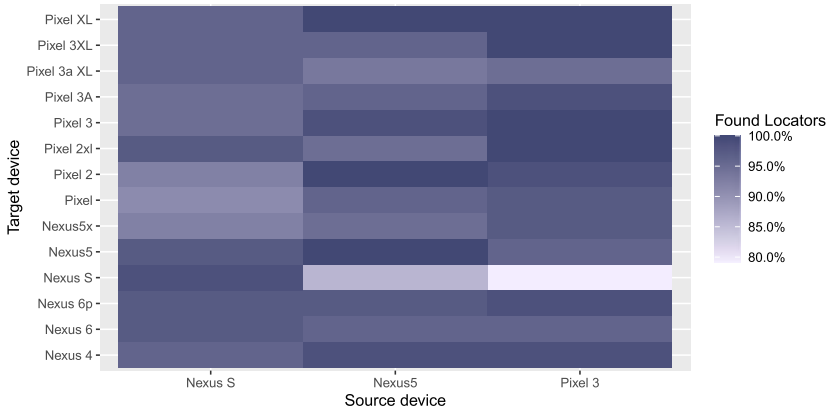


Fig. 13. Percentage of found locators during the test suite execution with the SIFT–Fullscreen technique, with varying source and target devices.

that the SURF–Fullscreen technique outperformed in a statistically significant way all the Cropped matching algorithms and the VGT tools in terms of correctly executed interactions (with p-values ranging from 4.64e−07 for the comparison with SIFT–Cropped to 2.89e−26 for the comparison with Sikuli).

The boxplot in Figure 12 reports the percentage of found locators for each source and target pair, grouped by technique. Sikuli was the worst option (31% of found unique locators), whereas SIFT–Fullscreen proved to be the best one (96%), with 36% more locators found than the best performing state-of-the-art VGT tool analyzed (EyeAutomate). We observed a significant effect on the percentage of found locators of the matching algorithm, descriptor, source and target device, and the combination of source and device (with p-values $<$ 10e−16). When using the Fullscreen algorithm, the percentage of found locators did not differ significantly for each descriptor. The heatmap in Figure 13 reports the percentage of found locators per source and target device pair for the best performing technique (i.e., SIFT–Fullscreen).

The plot shows lower percentages of found locators when the Nexus S was selected as either the source or target device. This outcome was likely due to the small size of the device screen (480 $\times$ 800 pixels). The average percentage of passing test cases was lower than that of executed instructions and found locators. This result was mainly due to the fact that the same unique locator can be used multiple times in different tests: e.g., in PassAndroid, most of the test cases involve a

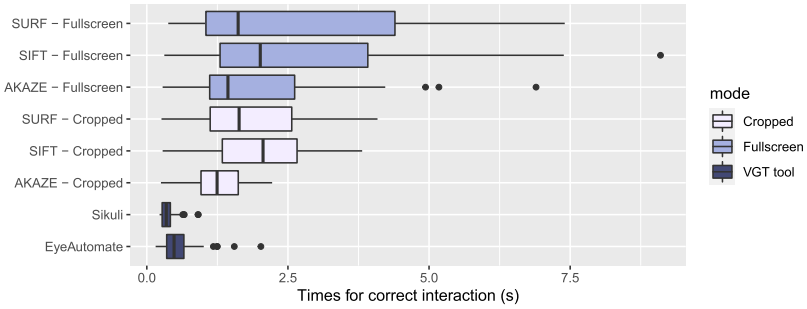


Fig. 14. Distribution of the average time per correct interaction for each source and target device pair, grouped by technique.

click on the *floating action button*, which systematically leads to a FP when the AKAZE–Cropped technique is used; in AntennaPod, many test cases involve the execution of a click on the Android menu button, which in most cases is not recognized by EyeAutomate. However, the percentages of passing tests were very high for the Fullscreen algorithm, regardless of the descriptor used, with 77% for SIFT–Fullscreen. Only 17% and 11% of the test cases were successfully executed for state-of-the-art VGT tools EyeAutomate and Sikuli, respectively. We observed a significant effect of matching algorithm, descriptor, source, and target device on the percentage of passing tests, as well as of the combination of source and target (with p-values < 10e−16). At *post hoc* analysis, the SURF–Fullscreen configuration outperformed all other techniques in a statistically significant way, except for AKAZE–Fullscreen ($p = 0.06$). For feature matching algorithms, with post-hoc tests we measured statistically significant differences except for AKAZE–Fullscreen, SIFT–Fullscreen, and SURF–Fullscreen.

Figure 14 reports the distribution of the average time needed by the VGT tools to perform a correct interaction (i.e., time to identify a widget plus time to execute interaction), grouped by technique. VGT tools exhibited a lower execution time per interaction, with an average time of 373 ms for Sikuli and 548 ms for EyeAutomate. The fastest feature matching technique was AKAZE–Cropped (1.24 s per interaction), whereas SIFT–Fullscreen was the slowest (2.74 s per interaction). We observed a significant effect on the percentage of passing tests of matching algorithm, descriptor, source, and target device (with p-values < 10e−16), whereas no significant interaction was found between the time to find a locator and the combination of source and target devices ($p = 0.99$).

As a final analysis, we observed the effect of the selected app (in our case, PassAndroid vs. AntennaPod) on the controlled variables. From the average and median values reported in Tables 8 and 9, we can see that the SIFT–Fullscreen algorithm performed best for all metrics with the AntennaPod SUT. Conversely, the SURF–Fullscreen was the best algorithm for the PassAndroid SUT regarding the percentage of executed interactions and the proportion of correctly executed tests, whereas SIFT–Fullscreen was still the best algorithm in terms of found locators. Sikuli was the best performing algorithm in terms of time per interaction for both SUTs. The impact on the results was more marked when the Cropped algorithm, rather than Fullscreen algorithm, was used. This result can be reasonably justified with the assumption that the performance of the Cropped algorithm strictly depends more on the nature of the individual SUT, since the results can be strongly impacted by the different arrangement of widgets in the layouts.

We observed a statistically significant effect on the percentage of found interactions, correct tests, and average time per interaction (p-value < 2.2e−16). The selected app had no significant effect on the percentage of correctly found locators (p-value = 0.094). This result strongly suggests

that the ability to find individual locators of the feature matching algorithms is unrelated to the AUT. The percentage of passing tests and executed interactions, on the contrary, strongly depends on the way the individual test suite has been defined (i.e., the number of locators used in each test case and the repetition of locators in different test cases). The time to execute the feature matching algorithms is also expected to be correlated to the AUT, since it depends on the total number of objects on the screen that have to be examined.

6 DISCUSSION

The performance of VGT tools strongly depends on the performance and quality of the underlying image analysis technique. By systematically exploring different combinations of feature matching algorithms and feature descriptors, we observed that the portability of test suites across mobile devices could be substantially improved over state-of-the-art tools.

For RQ1, we compared different algorithms from a purely visual matching perspective, investigating how often, on average, is it possible to correctly match any given widget from a source device through its relative visual locator on a target device.

Overall, the Fullscreen algorithm proved more robust and precise in locating widgets. By determining the optimal matching for all widgets simultaneously, it can solve complex cases (e.g., repeated widgets) where the Cropped algorithm is likely to fail. The difference is striking for the AKAZE descriptor, which also achieves the lowest overall performance. This result is consistent with the literature indicating that AKAZE is more computationally efficient but less robust to downscaling than SIFT and SURF [55].

For RQ2, we compared how well the different algorithms performed, in terms of robustness of the test cases based on them.

The number of correctly found locators in RQ2 is consistent with the recall in RQ1, and the same trend emerges in both the analyses with respect to both matching algorithm and descriptor. Among the configurations of feature descriptors and algorithms, SURF–Fullscreen is the best solution in terms of the percentage of correct interactions and of test suites executed. SIFT–Fullscreen is the best combination in terms of percentage of found locators. Based on the combined results of RQ1 and RQ2, both SIFT and SURF emerge as viable options, with SIFT achieving slightly higher performance and SURF being slightly faster.

The procedure followed in RQ1 can be easily replicated by researchers willing to evaluate other matching algorithms that can be employed in VGT of Android applications. The methodology systematically and automatically compares the algorithms across applications and devices, bypassing the need to define test cases manually. The procedure, however, does not consider the varying relevance of different types of widgets in real-world test suites, in which the performance of the matching algorithms with individual widgets can influence the result of the execution of several test cases. In our second experiment, we measured an average percentage of passed tests that were much lower than found locators (with a difference between −20% and −30%), primarily due to a single mismatched locator. This result suggests that further research work is needed to assess and improve the performance selectively on the most relevant widgets for real test cases.

When the source is equal to the target device, the EyeAutomate and Sikuli tools showcased very high percentages of found locators (and therefore of correct interactions and passed tests). However, they showed low portability across devices, with EyeAutomate consistently outperforming Sikuli (in accordance with previous studies [21]). The proposed feature matching techniques have higher overall portability, increasing the percentage of found locators and passed tests by at least 30% with respect to state-of-the-art VGT tools.

We postulate that the performance gap arises from the combination of two factors: the matching strategy and the feature extraction. State-of-the-art VGT tools like EyeAutomate and Sikuli are

all based on the less performing Cropped algorithm, whereas our results highlight that the task is best tackled as a many-to-many matching problem where all widgets are simultaneously optimized. Second, SIFT and SURF provide more robust features than pixel-level template matching. In particular, their invariance to scale is of paramount importance when covering a wide range of screen sizes.

These results suggest that state-of-the-art VGT test suites may provide sufficient robustness when used only on a single device, e.g., for regression testing purposes. Conversely, when the portability to different devices is important, our algorithms based on feature description algorithms are preferable. A further advantage of the proposed matching algorithms is that they do not require scaling of the screen captures to the size of the AVD as exactly rendered on the host screen, which on the contrary, is needed when using Sikuli and EyeAutomate.

In terms of execution time, the difference between the Fullscreen and Cropped algorithms is almost negligible. However, Sikuli and EyeAutomate are faster than the feature matching algorithms, with Sikuli being the fastest (the decrease of average time per interaction ranges from 70% with respect to AKAZE–Cropped, to 85% with respect to SURF–Fullscreen). We speculate that the difference is due to the more effective but more computationally expensive descriptors and to the search strategy. EyeAutomate starts searching from the last location (or the upper left corner, by default) [1], whereas our proposed techniques take into account (and compute the features for) all possible locations within the image. This choice has obvious advantages in the case of multiple similar widgets, increasing the accuracy, but it is paid for in terms of execution time. We did not specifically attempt to optimize the feature matching algorithms and their implementation for execution time, leaving this aspect to future work.

The additional time needed can be a limitation when the locators have to be found in highly dynamic GUIs, where the widgets and images can change very rapidly (e.g., in games). However, the feature matching algorithm we propose is commonly used for photos, so it may provide higher precision when used for complex GUIs than Sikuli and EyeAutomate. Additional comparisons with graphically-intensive apps should be performed as future work to investigate this aspect.

By performing a large-scale, automatic validation across multiple applications and devices, we found some limitations to matching algorithms based solely on visual features. The layout of Android applications is optimized based on the screen size by, e.g., resizing, rescaling, and grouping widgets, exploiting vertical and horizontal scrolling, and so on. In a small but not negligible percentage of cases, widgets present in the source device cannot be located in the target device, especially if the difference in screen size is large. In addition, some types of widgets and components (e.g., those with long text strings) are more prone to matching errors. These issues may affect a small number of test suites and hence may remain undetected following the assessment procedure in RQ2. Many of these issues could be tackled more effectively by integrating feature matching algorithms with semantic image interpretation capabilities, identifying the type, content, and function of each widget, and, if needed, modifying the matching strategy or even inserting additional operations (e.g., scrolling) in the test suite.

The proposed algorithms do not distinguish between text-based and image-based widgets and do not explicitly seek to identify or interpret the text. For feature-based matching, this operation is not necessary as the text provides many robust keypoints, which is generally beneficial in terms of performance but may occasionally create robustness issues for a minority of apps with very long text strings. In the case of widget detection, a mixed approach was found beneficial over a purely visual approach [20]. Similar strategies may be integrated in the future by slightly modifying the feature extraction.

Regardless of the algorithm or tool employed, the selection of the source and target devices had a statistically significant impact on all performance measures. Very small devices, like Nexus S,

yield significantly worse performance when used as either source or target devices. This fact has practical implications for developers and testers, who need to consider the range of devices they wish to support carefully. When the range of target devices is wide, selecting a source device with medium to large screen size will increase portability and decrease the effort needed to maintain the test suite.

7 THREATS TO VALIDITY

External validity threats. For the first experiment, we considered a single category of apps mined from a single database. Although this decision sets a realistic context for a high number of Android apps, it does not ensure that the results that we described are applicable to any type of Android app. Further studies are needed in games and other highly dynamic GUIs, which require both highly accurate and fast visual matching algorithms.

To better characterize the properties and generalizability of our results, we have studied the distribution of different types of widgets in our dataset (additional details are provided in the Supplementary Material). We further compared this distribution with that of the RICO dataset [26], which is based on a larger sample of roughly 9,300 apps. Roughly 50% of the widgets (40% in the RICO population) are TextView components. The most frequent widgets include Button (13%), ImageView (11%), View (6%), and ImageButton (6%). All other categories constitute overall 11% of the total number of widgets. The distribution is qualitatively comparable between our dataset and the entire RICO population, with a higher prevalence of TextView, EditText, and Button components in our dataset and a higher prevalence of View and ImageView in the RICO population. From this analysis, we conclude that any approach designed to solve the VGT issues, especially in the mobile domain, must include solid image recognition capabilities. Moreover, although slightly biased toward text components, our dataset can still be considered representative of a larger app population.

Furthermore, for the analysis of the precision and recall of feature matching algorithms, we only used the widgets in the main Activity of the considered apps; this set may not include widget types shown after navigation in the screens of the app. For the second experiment, we considered a single AUT (PassAndroid). Hence, the measured metrics may vary if the approach is applied to different apps.

To answer RQ2, we had to limit our experimentation to just two experimentation subjects for execution time reasons. To avoid cherry-picking, we considered AUTs that are commonly used in the software testing literature. To provide additional external validity to our findings, we compared our test suites with existing GUI test suites in open-source projects, leveraging a repository of GUI tests mined from GitHub. On a total of 3,226 correctly identified widgets interacted in Espresso test cases, we found out that 40% of interactions were on Text-based widgets, followed by Toolbars (10.6%), different types of Buttons (8.1%), NavigationViews (4%), and ListViews (3.5%). These percentages are compatible with the test suites that we developed for our two experimental subjects.

Finally, we only considered 14 emulated Android devices that are embedded in Android Studio; it is not ensured that the measured metrics can be applied to other devices, even if with equal resolution and screen size. Other context factors like, e.g., the OS version, may also have an influence on the metrics. All the considered devices also are part of the Google ecosystem and come equipped with an uncustomized Android version. Extending the experiment to other families of devices would require the use of real hardware devices or the use of third-party emulators. The first solution would require changes in the algorithms used by the VGT tools to search for the widgets in the screen of the connected device instead of the desktop environment screen; the second

would require alternatives to ADB commands, to control the download of screen hierarchies and captures.

Internal validity threats. The metrics that we measured, especially those regarding the execution time of the feature matching algorithms, strongly depend on their specific implementations. In particular, it is possible that better-optimized implementations could lower execution times, especially for the feature matching techniques for which only a Python prototype was available, albeit based on well-established and widely adopted libraries such as OpenCV.

We have considered only one version for the same app across multiple devices. In settings like regression testing, the actual performance may be lower than that observed due to changes in the app layout, widget, and functionalities. The internal validity of our experiments, however, is preserved, since all algorithms and tools were compared on equal grounds.

Construct validity. It is not ensured that the metrics used in this work (i.e., precision and recall for RQ1 and percentage of correct locators, interactions, and tests for RQ2) are the best possible proxies to observe the portability and the robustness to device fragmentation issues for Android VGT. We cannot ensure, for instance, that the measured metrics can correctly evaluate the fault-finding ability of test cases on different devices in a real testing scenario.

Researcher Bias could be introduced by creating a test suite for the PassAndroid app that was made by one of the authors of the article. However, the author was not instructed to insert specific widgets or visual locators in the test suite, neither was inclined to demonstrate a specific result.

8 CONCLUSIONS AND FUTURE WORK

The performance and, ultimately, the applicability of the VGT paradigm strongly depends on the robustness of the underlying image recognition algorithms. Our results show that state-of-the-art tools have limited portability across devices. A holistic approach in which the matching is simultaneously optimized for all widgets, combined with robust local feature descriptors, allowed our configurations to outperform state-of-the-art VGT tools by at least 30% in terms of correctly executed test suites.

Still, much remains to be done to ensure the full portability of test suites across devices and app versions. To support future research in this domain, we release the DatAndroid dataset,[5] which includes close to 100 apps rendered on multiple devices and specifically targets the issue of portability.

As our future work, we envision to embed our matching approaches in a full-fledged VGT tool, with capabilities of test creation, test execution/replication, and test repair in case of device fragmentation fragility. We plan on exploiting deep learning techniques to perform a semantic interpretation of Android GUIs screenshots, complementing and extending matching algorithms based solely on the computation of visual similarities. Finally, we plan to validate our methodology on different types of apps and different virtual devices, to improve the generalizability of our results.

REFERENCES

[1] Synteda AB. 2018. EyeAutomate Documentation. Retrieved from https://eyeautomate.com/wp-content/themes/EyeAutomateTheme/resources/EyeAutomateCertifiedTesterCourse.pdf.

[2] Emil Alégroth. 2015. *Visual GUI Testing: Automating High-level Software Testing in Industrial Practice.* Chalmers University of Technology, Göteborg.

[3] Emil Alégroth and Robert Feldt. 2017. On the long-term use of visual GUI testing in industrial practice: A case study. *Empir. Softw. Eng.* 22, 6 (2017), 2937–2971.

[4] Emil Alégroth, Robert Feldt, and Pirjo Kolström. 2016. Maintenance of automated test suites in industry: An empirical study on visual GUI testing. *Info. Softw. Technol.* 73 (2016), 66–80.

[5] Available at https://frankissimo.github.io/datAndroid/.

[5] Emil Alégroth, Zebao Gao, Rafael Oliveira, and Atif Memon. 2015. Conceptualization and evaluation of component-based testing unified with visual GUI testing: An empirical study. In *Proceedings of the IEEE 8th International Conference on Software Testing, Verification and Validation (ICST'15)*. IEEE, 1–10.

[6] Emil Alegroth, Michel Nass, and Helena H. Olsson. 2013. JAutomate: A tool for system-and acceptance-test automation. In *Proceedings of the IEEE 6th International Conference on Software Testing, Verification and Validation*. IEEE, 439–446.

[7] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M Memon. 2015. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Softw.* 32, 5 (2015), 53–59.

[8] Domenico Amalfitano, Vincenzo Riccio, Ana C. R. Paiva, and Anna Rita Fasolino. 2018. Why does the orientation change mess up my Android application? From GUI failures to code faults. *Softw. Test. Verificat. Reliabil.* 28, 1 (2018), e1654.

[9] Marti J. Anderson. 2014. Permutational multivariate analysis of variance (PERMANOVA). *Wiley Statsref: Stat. Ref. Online* 1 (2014), 1–15.

[10] Luca Ardito, Riccardo Coppola, Maurizio Morisio, and Marco Torchiano. 2019. Espresso vs. EyeAutomate: An experiment for the comparison of two generations of android GUI testing. In *Proceedings of the Evaluation and Assessment on Software Engineering*. ACM, 13–22.

[11] Luca Ardito, Riccardo Coppola, Marco Torchiano, and Emil Alégroth. 2018. Towards automated translation between generations of GUI-based tests for mobile devices. In *Companion Proceedings for the ISSTA/ECOOP Workshops*. ACM, 46–53.

[12] Ishan Banerjee, Bao Nguyen, Vahid Garousi, and Atif Memon. 2013. Graphical user interface (GUI) testing: Systematic mapping and repository. *Info. Softw. Technol.* 55, 10 (2013), 1679–1694.

[13] Daniel Barath and Jiri Matas. 2017. Graph-Cut RANSAC. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE, 6733–6741.

[14] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. 2008. Speeded-up robust features (SURF). *Comput. Vis. Image Underst.* 110, 3 (June 2008), 346–359. https://doi.org/10.1016/j.cviu.2007.09.014

[15] Nataniel P. Borges, Maria Gómez, and Andreas Zeller. 2018. Guiding app testing with mined interaction models. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft'18)*. Association for Computing Machinery, New York, NY, 133–143. https://doi.org/10.1145/3197231.3197243

[16] Emil Borjesson and Robert Feldt. 2012. Automated system testing using visual GUI testing tools: A comparative study in industry. In *Proceedings of the IEEE 5th International Conference on Software Testing, Verification and Validation*. IEEE, 350–359.

[17] G. Bradski. 2000. The OpenCV library. *Dr. Dobb's J. Softw. Tools* 1 (2000).

[18] Victor R. Basili-Gianluigi Caldiera and H. Dieter Rombach. 1994. Goal question metric paradigm. *Encyclopedia Softw. Eng.* 1 (1994), 528–532.

[19] Tsung-Hsiang Chang, Tom Yeh, and Robert C. Miller. 2010. GUI testing using computer vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, New York, NY, 1535–1544.

[20] Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, and Guoqiang Li. 2020. Object detection for graphical user interface: Old fashioned or deep learning or a combination? In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, 1202–1214.

[21] Riccardo Coppola, Luca Ardito, and Marco Torchiano. 2019. Fragility of layout-based and visual GUI test scripts: An assessment study on a hybrid mobile application. In *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. Association for Computing Machinery, New York, NY, 28–34.

[22] Riccardo Coppola, Luca Ardito, Marco Torchiano, and Emil Alégroth. 2020. Translation from visual to layout-based android test cases: A proof of concept. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW'20)*. IEEE, Washington, DC, 74–83.

[23] Riccardo Coppola, Luca Ardito, Marco Torchiano, and Emil Alégroth. 2021. Translation from layout-based to visual android test scripts: An empirical evaluation. *J. Syst. Softw.* 171 (2021), 110845. https://doi.org/10.1016/j.jss.2020.110845

[24] Riccardo Coppola, Maurizio Morisio, Marco Torchiano, and Luca Ardito. 2019. Scripted GUI testing of Android open-source apps: Evolution of test code and fragility causes. *Empir. Softw. Eng.* 24 (2019), 1–44.

[25] Ritendra Datta, Dhiraj Joshi, Jia Li, and James Z. Wang. 2008. Image retrieval: Ideas, influences, and trends of the new age. *ACM Comput. Surv.* 40, 2, Article 5 (May 2008), 60 pages. https://doi.org/10.1145/1348246.1348248

[26] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. Association for Computing Machinery, New York, NY, 845–854.

[27] A. Developers. 2012. Ui/application exerciser monkey.
[28] Mattia Fazzini, Eduardo Noronha de A. Freitas, Shauvik Roy Choudhary, and Alessandro Orso. 2017. Barista: A technique for recording, encoding, and running platform independent android tests. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST'17)*. IEEE, Washington, DC, 149–160.
[29] Pablo Fernández Alcantarilla. 2013. Fast explicit diffusion for accelerated features in nonlinear scale spaces. In *Proceedings of the British Machine Vision Conference*. BMVA Press, Durham, UK. https://doi.org/10.5244/C.27.13
[30] K. Fukunaga and L. Hostetler. 1975. The estimation of the gradient of a density function, with applications in pattern recognition. *IEEE Trans. Info. Theory* 21, 1 (1975), 32–40.
[31] Dan Han, Chenlei Zhang, Xiaochao Fan, Abram Hindle, Kenny Wong, and Eleni Stroulia. 2012. Understanding android fragmentation with topic analysis of vendor-specific bugs. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE'12)*. IEEE, Washington, DC, 83–92.
[32] Kristian Fjeld Hasselknippe and Jingyue Li. 2017. A novel tool for automatic GUI layout testing. In *Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC'17)*. IEEE, Washington, DC, 695–700.
[33] Jiajun Hu, Lili Wei, Yepang Liu, Shing-Chi Cheung, and Huaxun Huang. 2018. A tale of two cities: How webview induces bugs to android applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. Association for Computing Machinery, New York, NY, 702–713.
[34] Jouko Kaasila, Denzil Ferreira, Vassilis Kostakos, and Timo Ojala. 2012. Testdroid: Automated remote UI testing on Android. In *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia*. Association for Computing Machinery, New York, NY, 1–4.
[35] Emily Kowalczyk, Myra B. Cohen, and Atif M. Memon. 2018. Configurations in Android testing: They matter. In *Proceedings of the 1st International Workshop on Advances in Mobile App Analysis*. Association for Computing Machinery, New York, NY, 1–6.
[36] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2018. Pesto: Automated migration of DOM-based Web tests towards the visual approach. *Softw. Test. Verificat. Reliabil.* 28, 4 (2018), e1665.
[37] Ying-Dar Lin, Jose F. Rojas, Edward T.-H. Chu, and Yuan-Cheng Lai. 2014. On the accuracy, efficiency, and reusability of automated test oracles for android devices. *IEEE Trans. Softw. Eng.* 40, 10 (2014), 957–970.
[38] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. 2017. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *Proceedings of the IEEE International Conference on*. IEEE, Washington, DC, 399–410.
[39] David G. Lowe. 2004. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vis.* 60, 2 (2004), 91–110.
[40] Eduardo Luna and Omar El Ariss. 2018. Edroid: A mutation tool for android apps. In *Proceedings of the 6th International Conference in Software Engineering Research and Innovation (CONISOFT'18)*. IEEE, Washington, DC, 99–108.
[41] S. S. Mangiafico. 2015. *An R Companion for the Handbook of Biological Statistics*, Version 1.09c, 274 pp.
[42] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. Association for Computing Machinery, New York, NY, 94–105.
[43] Matias Martinez and Bruno Gois Mateus. 2020. How and Why did developers migrate Android Applications from Java to Kotlin? A study based on code analysis and interviews with developers. arXiv:2003.12730v1 [cs.SE].
[44] Alessio Merlo and Gabriel Claudiu Georgiu. 2017. Riskindroid: Machine learning-based risk analysis on android. In *Proceedings of the IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, Cham, 538–552.
[45] Salvador Morales-Ortega, Ponciano Jorge Escamilla-Ambrosio, Abraham Rodriguez-Mota, and Lilian D. Coronado-De-Alba. 2016. Native malware detection in smartphones with Android OS using static analysis, feature selection and ensemble classifiers. In *Proceedings of the 11th International Conference on Malicious and Unwanted Software (MALWARE'16)*. IEEE, Washington, DC, 1–8.
[46] Kevin Moran, Richard Bonett, Carlos Bernal-Cárdenas, Brendan Otten, Daniel Park, and Denys Poshyvanyk. 2017. On-device bug reporting for android applications. In *Proceedings of the IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft'17)*. IEEE, Washington, DC, 215–216.
[47] Maxim Mozgovoy and Evgeny Pyshkin. 2017. Using image recognition for testing hand-drawn graphic user interfaces. In *Proceedings of the the 11th International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*. IARIA, Barcelona, 25–28.
[48] Maxim Mozgovoy and Evgeny Pyshkin. 2018. Pragmatic approach to automated testing of mobile applications with non-native graphic user interface. In *International Journal on Advances in Software*. IARIA, Wilmington, UK, 239–246.
[49] Maxim Mozgovoy and Evgeny Pyshkin. 2018. *Unity Application Testing Automation with Appium and Image Recognition*. Springer, Cham, 139–150. https://doi.org/10.1007/978-3-319-71734-0_12

[50] Stas Negara, Naeem Esfahani, and Raymond P. L. Buse. 2019. Practical android test recording with espresso test recorder. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*. IEEE, Washington, DC, 193–202.

[51] Je-Ho Park, Young Bom Park, and Hyung Kil Ham. 2013. Fragmentation problem in Android. In *Proceedings of the International Conference on Information Science and Applications (ICISA'13)*. IEEE, Washington, DC, 1–2.

[52] Alireza Sahami Shirazi, Niels Henze, Albrecht Schmidt, Robin Goldberg, Benjamin Schmidt, and Hansjörg Schmauder. 2013. Insights into layout patterns of mobile user interfaces by an automatic analysis of android apps. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. Association for Computing Machinery, New York, NY, 275–284.

[53] Statista. 2021. Global Google Play app downloads 2016-2019. Retrieved from https://www.statista.com/statistics/734332/google-play-app-installs-per-year/.

[54] S. A. K. Tareen and Z. Saleem. 2018. A comparative analysis of SIFT, SURF, KAZE, AKAZE, ORB, and BRISK. In *Proceedings of the International Conference on Computing, Mathematics and Engineering Technologies (iCoMET'18)*. IEEE, Washington, DC, 1–10.

[55] Shaharyar Ahmed Khan Tareen and Zahra Saleem. 2018. A comparative analysis of sift, surf, kaze, akaze, orb, and brisk. In *Proceedings of the International conference on Computing, Mathematics and Engineering Technologies (iCoMET'18)*. IEEE, Washington, DC, 1–10.

[56] Swapna Thorve, Chandani Sreshtha, and Na Meng. 2018. An empirical study of flaky tests in android apps. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'18)*. IEEE, Washington, DC, 534–538.

[57] Marco Torchiano. 2020. *effsize: Efficient Effect Size Computation*. Politecnico di Torino. https://doi.org/10.5281/zenodo.1480624

[58] J. Tuovenen, Mourad Oussalah, and Panos Kostakos. 2019. MAuto: Automatic mobile game testing tool using image-matching based approach. *Comput. Games J.* 8 (Oct. 2019), 215–239. https://doi.org/10.1007/s40869-019-00087-z

[59] Tinne Tuytelaars and Krystian Mikolajczyk. 2008. *Local Invariant Feature Detectors: A Survey*. Now Publishers, Boston, MA.

[60] Mikko Vesikkala. 2014-05-05. *Visual Regression Testing for Web Applications; Selainpohjaisten ohjelmistojen visuaalinen regressiotestaus*. G2 Pro gradu, diplomityö; masterThesis. Aalto University. Retrieved from http://urn.fi/URN:NBN:fi:aalto-201405131809.

[61] Jiwei Yan, Hao Liu, Linjie Pan, Jun Yan, Jian Zhang, and Bin Liang. 2020. Multiple-entry testing of android applications by constructing activity launching contexts. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering (ICSE'20)*. IEEE, Washington, DC, 457–468.

[62] Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. 2009. Sikuli: Using GUI screenshots for search and automation. In *Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology*. Association for Computing Machinery, New York, NY, 183–192.

[63] Hrushikesh Zadgaonkar. 2013. *Robotium Automated Testing for Android*. Packt Publishing, Birmingham, UK.

[64] Denys Zelenchuk. 2019. Espresso and UI automator: The perfect tandem. In *Android Espresso Revealed*. Springer, Cham, 165–189.

Received September 2020; revised July 2021; accepted July 2021