AVGUST: Automating Usage-Based Test Generation from Videos of App Executions

Yixue Zhao*
yixuezhao@cs.umass.edu
University of Massachusetts Amherst
USA

Saghar Talebipour*
talebipo@usc.edu
University of Southern California
USA

Kesina Baral
kbaral4@gmu.edu
George Mason University
USA

Hyojae Park
hyoj.p20@gmail.com
Sharon High School
USA

Leon Yee
leon.yee000@gmail.com
Valley Christian High School
USA

Safwat Ali Khan
skhan89@gmu.edu
George Mason University
USA

Yuriy Brun
brun@cs.umass.edu
University of Massachusetts Amherst
USA

Nenad Medvidović
neno@usc.edu
University of Southern California
USA

Kevin Moran
kpmoran@gmu.edu
George Mason University
USA

ABSTRACT

Writing and maintaining UI tests for mobile apps is a time-consuming and tedious task. While decades of research have produced automated approaches for UI test generation, these approaches typically focus on testing for crashes or maximizing code coverage. By contrast, recent research has shown that developers prefer *usage-based tests*, which center around specific uses of app features, to help support activities such as regression testing. Very few existing techniques support the generation of such tests, as doing so requires automating the difficult task of understanding the semantics of UI screens and user inputs. In this paper, we introduce AVGUST, which automates key steps of generating usage-based tests. AVGUST uses neural models for image understanding to process video recordings of app uses to synthesize an app-agnostic state-machine encoding of those uses. Then, AVGUST uses this encoding to synthesize test cases for a new target app. We evaluate AVGUST on 374 videos of common uses of 18 popular apps and show that 69% of the tests AVGUST generates successfully execute the desired usage, and that AVGUST's classifiers outperform the state of the art.

CCS CONCEPTS

• **Software and its engineering → Software notations and tools**.

KEYWORDS

Test Generation, UI Understanding, AI/ML, Mobile Application

*Both authors contributed equally to the paper

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9413-0/22/11...$15.00
https://doi.org/10.1145/3540250.3549134

ACM Reference Format:
Yixue Zhao, Saghar Talebipour, Kesina Baral, Hyojae Park, Leon Yee, Safwat Ali Khan, Yuriy Brun, Nenad Medvidović, and Kevin Moran. 2022. AVGUST: Automating Usage-Based Test Generation from Videos of App Executions. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549134

1 INTRODUCTION

Writing UI tests is time-consuming and tedious. The research community has contributed a large body of work that aims to automatically generate UI tests [15, 20, 30, 34, 39, 42, 46, 50, 59, 74, 80]. Such testing techniques generate a test's inputs, and use a pre-defined criterion as the test's oracle. A significant portion of recent work on UI test generation has focused on *mobile platforms* and has predominantly aimed to discover crashes or maximize code coverage [30, 34, 59, 65, 74, 80]. However, studies have repeatedly found that existing testing techniques in this domain fall short in addressing developers' needs in practice [43, 52] or present challenges for practical adoption [35, 52].

Specifically, mobile developers have a strong preference for test cases that are closely coupled to app use cases or features [52]. In line with recent work [85], we refer to this type of preferred test case as *usage-based* UI test. A usage-based UI test consists of a sequence of UI events that mimic *realistic* user behaviors in exercising a specific feature of a given app, such as "adding an item to the shopping cart." The developer preference for usage-based tests is due to the fact that such test cases support specific testing goals in practice, such as regression or performance testing, which in turn require orientation to common app use cases [52].

Automating such testing activities is critical for mobile developers who face unique challenges related to rapidly evolving platforms [18, 51], pressure for frequent releases [38, 41], and a deluge of feature requests and bug reports from user reviews [25, 29, 67, 68]. Despite the importance of these usage-based tests for developers, current automated testing approaches typically do not consider

app usages as a goal or test adequacy criteria, and as such cannot generate these tests [52, 85].

A growing body of research on the topic of *UI test reuse* (also sometimes called test migration, test transfer, or test adaption [19, 20, 50, 62, 71]) has begun to explore the possibility of automating the transfer and adaptation of existing usage-based tests from a *source app* to a behaviorally similar *target app* that contains shared features [19, 20, 39, 50, 61, 62, 71, 85]. However, these test-reuse techniques have three notable limitations that pose challenges for developers to adopt in practice. ❶ To generate tests for a target app, UI test reuse requires *pre-existing, manually-written tests* for a corresponding source app. In practice, creating these source tests is time-consuming and error-prone, leading many mobile developers to forgo writing them [43, 52]. ❷ Test-reuse techniques have typically been designed for, and tasked with, transferring tests between behaviorally similar applications from similar domains (e.g., between two finance or two shopping apps). However, there are many use cases common across apps from varying domains (e.g., logging in or changing the theme), which current test techniques would struggle to effectively transfer. ❸ Many existing techniques rely on expensive and difficult to use program analyses (e.g., bytecode decompilers, Soot [9, 78], Gator [2, 83]) that often require access to an app's source code. The ease of use and scalability limitations of such underlying utilities have hindered the adoption of test case transfer tools in practice.

To help better align automation related to usage-based testing with developers needs, we propose Avgust, a technique for app-video-based generation of usage tests. Avgust is a novel developer-in-the-loop test generation technique that directly addresses the three limitations mentioned above. ❶ Instead of requiring pre-existing source tests written by domain experts, Avgust allows for easy creation of source test scenarios through *screen recordings of app usages*, which are becoming increasingly common software artifacts for mobile apps [26] and can be easily obtained via crowd workers with no testing expertise. After video collection and processing, Avgust operates according to two main phases. In the first phase, neural computer vision (CV) and natural language processing (NLP) techniques are employed to guide developers through a lightweight screen and GUI widget annotation process for video frames that were automatically identified to contain a touch action. Using this information, Avgust is able to generate an app-independent intermediate-representation model (*IR Model*), which represents abstract states and transitions of a usage that can be mapped to multiple apps. This procedure is a one-time effort for developers, and once the IR Model is generated, it can be used to generate tests for multiple target apps. ❷ The generality of the IR Model allows Avgust to synthesize test scenarios across domains, effectively overcoming the second limitation of existing test-case transfer techniques. Avgust's second phase automates the synthesis of new UI test scenarios by guiding a developer with suggestions, made by using predictions from Avgust's CV and NLP techniques, of which GUI elements must be manipulated to exercise a given app feature or usage. ❸ To bolster the applicability and practicality of Avgust from a developer's perspective, Avgust operates *purely on visual information* encoded into screenshots and video frames from UI-screen recordings. As such, it does not require access to an app's source code, instrumentation, or expensive program analyses. Note that solely relying on app videos as input is a key aspect of Avgust's novelty and it has three major advantages. First, videos are common artifacts that are easily collectible without requiring difficult tool configuration and setup, which are major barriers for adoption [43, 52]. Second, videos can be collected by crowd workers (e.g., real users) with no testing experience, enabling the opportunity to obtain much more training data to cover diverse and realistic usage scenarios across different apps. This can yield more generalized models to generate higher-quality tests. Finally, videos are agnostic to the underlying device and platform, meaning Avgust's design is not tied to Android platforms (where Avgust is evaluated on), but is applicable to any apps, devices, and platforms (e.g., websites) in principle.

The key research challenge that Avgust tackles is the automated synthesis of a generalized model of feature usages that can effectively map test scenarios across apps from a variety of domains, using only screenshots and video frames from screen recordings. The challenge lies in automating two key tasks: (1) screen understanding from pixels and (2) design of an IR Model that is general enough to capture diverse app usages yet specific enough to allow mapping actions to a given target app for test scenario generation. Avgust accomplishes the first task through the creation of a bespoke image classification technique, built on top of a neural auto-encoder representation [69] and BERT-based textual embeddings [28]. The classification operates at two granularity levels, (i) screen-level, and (ii) GUI widget-level. This classification procedure helps to provide the mapping to our IR Model, and makes use of a rich screen representation obtained by training our neural auto-encoder on the public RICO dataset [27]. Avgust accomplishes the second task by using the information from our classifiers to build a state machine capable of simultaneously capturing multiple scenarios from different usages. This yields a richer model of app usage than past test transfer techniques.

In order to build a community resource of usage-based tests, we conducted a user-study to collect 374 video recordings from 18 apps, covering 18 usage scenarios, wherein each usage scenario is exercised by three associated apps, for a total of 54 unique app-usage pairs [39, 85]. Using this data, we conducted an empirical evaluation to measure the efficacy of Avgust in generating usage-based tests that closely mirror those created by humans during our user study. First, we examined Avgust's test generation capability by simulating a developer interacting with Avgust's suggestions, and measured how closely generated tests matched analogous tests created by human users. Next, to gain a better understanding of Avgust's performance during test generation, we evaluated Avgust's classifiers compared to state-of-the-art techniques. Our results show that Avgust is able to generate tests that effectively exercise target-app features and closely match human tests in terms of the screens visited and actions performed. Additionally, Avgust's classifiers significantly outperform state-of-the-art techniques and show promising performance for our generated developer-in-the-loop recommendations.

We have developed Avgust with open science in mind, making it both practical to use for developers, and easily reusable by researchers to foster future research in this area. We make publicly

available all of our source code, trained models, and annotated evaluation data collected during our user study [10]. Avgust's pipeline can be easily adapted to create various usage models of interest by simply changing the input videos, such as including additional usages and apps. As such, Avgust not only lays a foundation for future work on usage-based test generation, but also represents a living repository for the software engineering community to study related problems.

In summary, this paper makes the following contributions:

(1) We introduce Avgust, the first technique capable of generating *usage-based* tests by learning from app videos.
(2) We develop a novel image classification technique to translate app videos into an app-independent intermediate representation based on vision-only information, which largely outperforms the state-of-the-art.
(3) We implement a reusable pipeline to train IR models based on app videos that can be applied to various downstream tasks, and further provide 125 pre-trained models that can be used by developers directly.
(4) We collect 374 app videos and conduct an empirical evaluation to demonstrate Avgust's effectiveness in assisting developers with generating usage-based tests.
(5) We provide a public repository [10] that contains Avgust's artifacts to foster future research, including Avgust's source code, our pre-trained models, labeled datasets, benchmarks used, and their corresponding results.

2 THE AVGUST APPROACH

Avgust is an automated approach that aims to assist developers with the generation of usage-based tests to mimic realistic usage scenarios. Avgust operates in three phases. (1) It processes recorded videos of different apps' usages by applying neural CV and NLP to detect user actions in individual video frames. (2) Avgust uses this information to generate an app-independent state machine-based IR Model. (3) Finally, Avgust leverages the IR Model to generate tests for a new (i.e., "target") app. In this section, we provide an overview of Avgust's workflow, and then detail its three phases.

2.1 Avgust Overview

Avgust functions as a human-in-the-loop tool to provide suggestions of input events for developers in the creation of usage-based tests. This design decision is guided by the nature of *usage-based* tests, since each usage scenario may have various correct ways of being tested. For instance, there may be different ways to execute the login scenario in an app, such as logging in using username and password or by using user's existing social media accounts. Thus, providing suggestions to a developer allows for flexibility in generating tests that are tailored to a given app usage and testing objective. Figure 1 depicts Avgust's workflow, which consists of three principal phases: ① *Video Collection & Analysis*, ② *IR Model Generation*, and ③ *Guided Test Scenario Generation*.

During the *Video Collection & Analysis* phase, crowdsourced workers are tasked with collecting videos of app usages. These videos are then analyzed in a fully automated process that involves deconstructing the video into constituent frames, identifying touch-based actions that were performed on the an app's UI (which builds upon past work in app-video analysis [21]), and eliminating sensitive information such as user passwords.

Next, in the *IR Model Generation* phase, Avgust assists a developer with labeling screens and individual GUI widgets from processed video frames into categories, which Avgust can then use to generate an app-independent IR Model. This is a semi-automated process wherein a developer is presented with a screen and Avgust provides top-k suggestions for the labels that should be applied to both screens an exercised GUI widgets. These suggestions are made using a combination of visual- and text-based classifiers that operate upon the video frames extracted in the prior phase. After the labels have been applied by the developer, Avgust is able to automatically generate the state machine-based IR Model for the usage, merging it with other similar usages in a shared database. This phase is intended to be a one-time cost, wherein developers contribute their crowdsourced IR Models of various app usages to a collective community database for future use.

Finally, in the *Guided Test Scenario Generation* phase, Avgust assists developers by providing top-k recommendations for actions that should be performed on given screens of a previously unseen target app in order to exercise a specified app feature (e.g., adding an item to the shopping cart). This process starts from the initial screen of the app and runs until the specified feature is exercised. This functions similarly to the IR Model generation phase, but in reverse order: the model is used in conjunction with Avgust's classification techniques to recommend event inputs to developers.

2.2 Video Collection & Analysis

Given a set of collected videos of app usages, Avgust's video analysis processes them into frames that serve as the inputs for Avgust' AI-assisted IR Model generation (Section 2.3). Specifically, Avgust first identifies the user actions in the videos and extracts their corresponding *event frames*, which are the key video frames that capture the user interactions via the *touch indicator*.[1] An example of event frame is shown in Figure 3, where the touch indicator points to the user interacting with the "app menu" button in the top-left corner. As a final step, Avgust filters the extracted event frames by eliminating the frames that contain sensitive user information.

2.2.1 *Action Identification & Event Frame Extraction.*

Avgust builds upon the analyses introduced by V2S [21, 36] to identify user actions and event frames. V2S is a recent technique that leverages neural object detection and image classification to identify the user actions in a video, and automatically translates these actions into a replayable scenario. We extended V2S to work with GPU clusters, to enable it to process large numbers of videos in parallel. The outputs of our extended video processing technique are (1) a sequence of *event frames* of a given video and their associated user actions (i.e., click, long tap, and swipe); and (2) the *coordinates* of the touch indicator in each event frame [45].

2.2.2 *Event Frame Filtering.*

Avgust filters the extracted event frames by eliminating the frames that are associated with the *typing* action, since they may expose user's private information such as password. Note that Avgust only eliminates the frames where the

[1] Avgust requires enabling the display of the *touch indicator*, which can easily be done in, both, the Android and iOS settings menus, even by inexperienced users.

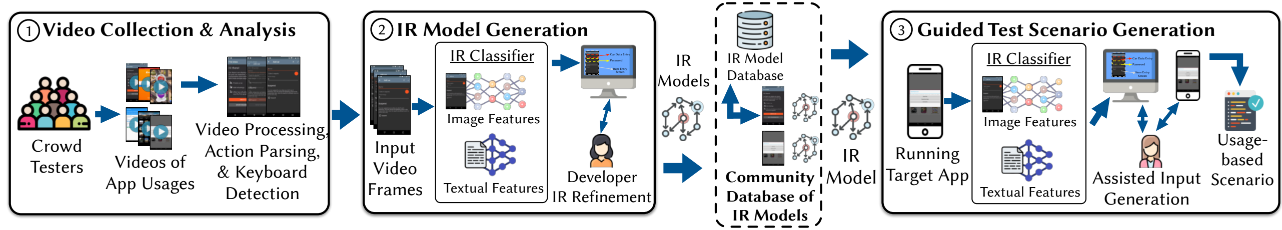
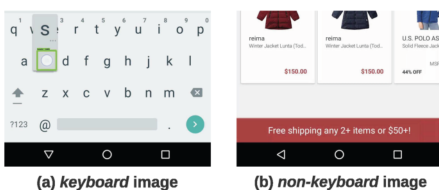


Figure 1: AVGUST's three-phase workflow.

user types the text content on a keyboard, but still keeps the frames where the user selects which input field she intends to enter the text content. For example, the sequence of video frames related to "typing user password" consists of (1) a frame associated with clicking the password field, and (2) a group of consecutive frames associated with typing each individual character in the password. AVGUST only eliminates the latter, while maintaining the former to represent "typing user password" action in the usage scenario.

To do so automatically, we trained a binary image classifier to recognize whether an event frame contains a keyboard image. Our classifier is based on a CNN architecture with 4 blocks, each consisting of a Convolution, a BatchNorm, a ReLU, and a Dropout layer [79]. The CNN is trained on cropped screenshots that depict the area of the screen where keyboard may appear, since this area is standard for mobile devices, as shown in Figure 2. We decided to focus on the region of the screen where the keyboard appears, as opposed to the entirety of the screen, based on empirical evidence collected while tuning our classifier, as the former setting dramatically improved the classifier's accuracy. We sourced *non-keyboard* training data by randomly selecting 4,926 app screenshots without the keyboard from the publicly available RICO dataset [27]. The non-keyboard training data do not contain any subject apps used in our evaluation. Because screens that display a keyboard cannot be automatically identified using the GUI metadata provided in the RICO dataset, we additionally sourced 5,605 *keyboard* training data images from the video frames in the dataset collected for AVGUST's evaluation. Our training data relies on standard Android keyboard images, but can be easily extended to additional keyboard types.

Next, the cropped screen region where a keyboard may appear for each event frame is fed to the trained keyboard classifier, and will be classified as either a *keyboard* or *non-keyboard* frame. For the keyboard frames, AVGUST further verifies whether the associated action is *typing*, based on whether the touch indicator falls in the keyboard region. This is determined by the touch indicator's coordinates on the screen [45], which are obtained from V2S. In the end, the event frames that contain a typing action are eliminated.



(a) *keyboard* image (b) *non-keyboard* image

Figure 2: Examples of the training data used in AVGUST's keyboard classifier during the event frame filtering.

Note that this frame filtering process not only addresses the privacy issue as discussed earlier, it also largely reduces the number of event frames used to represent an app's usage. For example, a two-minute sign-in video from the app 6pm contained over 3,000 video frames originally, but only 8 *filtered* event frames. These 8 frames are sufficient to represent all relevant user actions without the duplicated or privacy-exposing frames in the original video frames.

2.3 AI-Assisted IR Model Generation

AVGUST uses the filtered event frames from the previous phase as inputs and translates them into app-independent IR Models of app usages. The key technical challenge in this phase stems from AVGUST's use of video inputs, which forces us to rely solely on visual information encoded into the pixels of the video frames. We address this challenge in two steps: (1) we break down event frames into GUI events and (2) use image classification techniques to assist developers in translating GUI events to their corresponding IR Models. As an illustration, Figure 3 demonstrates the key artifacts in this process using a single event frame extracted from a popular shopping app 6pm as an example. We now detail these two steps.

2.3.1 Transforming Event Frames to GUI Events. We define a *GUI event* as a triple (s, w, a), where s is the app screen that shows a snapshot of the app's execution state; w is the GUI widget the user interacts with; and a is the corresponding action the user performs, such as `click` or `swipe`. The GUI widget w is optional since certain actions (e.g., `swipe`) are not associated with any widgets. AVGUST converts event frames into GUI event triples in a two-step process: widget extraction and action identification.

Widget Extraction. Extracting individual widgets from a screen presents two major challenges. First, each widget's bounding box must be identified without reliance on source code-level information that is available on platforms like Android [7]. Second, AVGUST

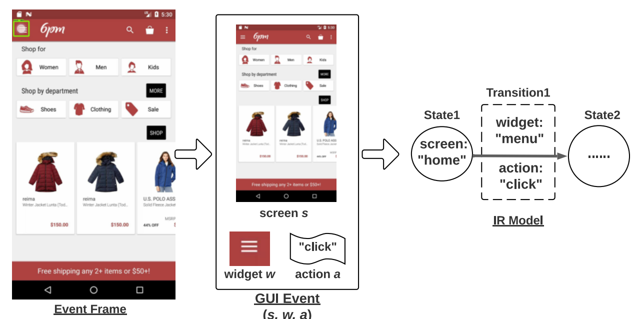


Figure 3: An example of converting a 6pm's Event Frame into a GUI Event triple, and an app-independent IR Model.

needs to isolate the precise bounding box of the widget with which the user is interacting, such as the app-menu button in the top-left corner of the screen in Figure 3.

To detect the bounding boxes of GUI widgets, we modified UIED [24, 82], a state-of-the-art tool that combines unsupervised CV and deep learning, and applied it on the screens extracted from AVGUST's previous phase. Given an input screen, UIED detects textual and visual GUI elements and produces their bounding boxes, as depicted with solid rectangles in Figure 4. However, UIED treats each visual and textual GUI element separately, which can lose important semantic information. For example, if the touch indicator refers to a checkbox, the corresponding GUI element detected by UIED in Figure 4 would be one of the two checkboxes only, leaving unclear whether the extracted widget is intended to be associated with "Show password" or "Keep me signed in".

To remedy this, we modified UIED to group the visual GUI elements together with their surrounding textual elements, if any. AVGUST iterates through all visual elements detected by UIED and identifies their closest GUI elements. If the closest GUI element is both a textual element and is in the same line as the visual element—defined as being vertically collocated based on a customizable threshold—then the bounding box of the visual element will be updated to include the textual element as well. In Figure 4, this results in the two checkboxes being grouped with their corresponding labels, as depicted by the dashed rectangles.

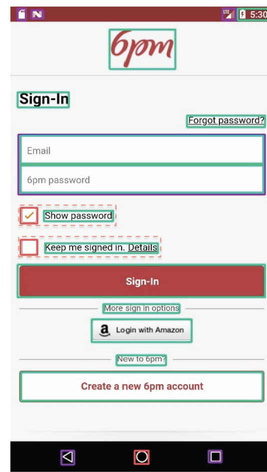


Figure 4: AVGUST adjusts the GUI-element bounding boxes detected by UIED, depicted by the two dashed rectangles.

Next, the detected GUI elements' bounding boxes are used by AVGUST to automatically crop out the widget w to which the touch indicator refers. AVGUST combines the widgets detected by its modified UIED with the coordinates of the touch indicator obtained from the modified V2S (recall Section 2.2) to identify all candidate widgets for cropping, covering three possible cases: (1) The simplest case is when only one widget's bounding box covers the touch indicator, in which case AVGUST crops that widget as-is. (2) If no widgets' bounding boxes cover the touch indicator, AVGUST repeatedly expands each widget's bounding box based on a customizable threshold, until a suitable widget is found. AVGUST's default threshold is set at 10 pixels. (3) When multiple widget candidates are found, AVGUST first eliminates the "coarse-grained" candidates whose boundaries completely cover any of the other candidates (e.g., "sign-in form" that covers "username" widget), and then selects the widget whose center point is closest to the touch indicator's coordinates.

Action Identification. To identify the action a in the GUI event triple, AVGUST leverages V2S's action identification procedure, which analyzes the coordinates of touches detected in consecutive video frames and classifies actions according to a set of heuristics [21, 36]. V2S is able to identify clicks, long taps, and swipes. We reused V2S's heuristics for click and long tap, and extended its swipe detection heuristic to additionally detect the *direction* of the swipe.

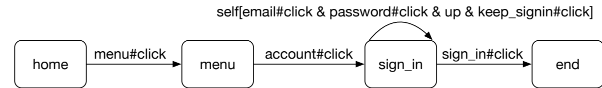


Figure 5: The IR Model generated from a sign-in video collected from the app 6pm.

2.3.2 Transforming GUI Events to IR Models. AVGUST's IR Model generation is a developer-in-the-loop process. This section explains how AVGUST provides recommendations to assist developers in translating GUI events into their app-independent IR representations (recall the example in Figure 3).

AVGUST's IR Model is defined as a finite state machine (FSM) that captures app usages. Figure 5 shows an example IR Model converted from one of 6pm's sign-in videos. Each state in the IR Model represents a particular app screen and is captured as an app-independent *canonical screen*, while each transition represents a user interaction with a *canonical widget* and its corresponding action. A self-transition (e.g., shown in the "sign_in" state in Figure 5) means that the app stays on the same screen during certain user interactions.

The key challenge in translating a GUI event triple (s, w, a) into AVGUST's IR Model is to properly abstract away and capture in an app-independent manner the app-specific screens s and widgets w. Each action a in the GUI event triple is translated as-is, including click, long tap, and swipe up/down/left/right. With the translated canonical screens, canonical widgets, and actions, the final IR Model can be constructed by iterating through the sequence of GUI events of a particular usage.

We formulate the translation of screens and widgets as a classification problem, where app-specific screens and widgets are classified into their canonical counterparts (categories) that are shared across different apps. To this end, we build upon and extend app-independent categories defined by previous work [39, 85], resulting in 37 canonical screens and 74 canonical widgets. Example canonical screens are "home screen", "password assistant page", and "shopping cart page". Example canonical widgets are "account", "help", and "buy". The complete sets of canonical screens and widgets are available [10]. Note that these categories are not directly tied to our subject apps used in the evaluation, but can generalize across diverse apps. To facilitate the extensibility of new canonical screens/widgets, we have created a data labeling tool using Label Studio [12], allowing future research to further tailor and improve the canonical categories for both screens and widgets.

We note that our classification problem provides a unique challenge as it relies only on information from app screenshots. There are no existing techniques in the mobile-app domain that have previously addressed this problem [27, 39, 47] in a context similar to ours. Moran et.al. [64] were one of the first to use a CNN for widget classification from GUI component images. However, their classifier only functioned on 15 general widget categories. Our

larger number of 74 categories represents a more challenging classification problem that requires the use of both textual and visual features to achieve reasonable accuracy. Another recent technique for screen classification in this domain is Screen2Vec (S2V) [47], which aims to produce embeddings for app screens that can be used for downstream classification tasks, such as ours. However, S2V cannot be applied on screenshots alone as it requires the UI layout information of an app screen that is specific to the Android platform [7]. Obtaining such information requires extraction using third-party tools (e.g., Appium [4], UIAutomator [1]), and would sacrifice the practicality of usage collection through videos.

Screen Classifier. To classify a given app-screen image into its canonical screen, AVGUST leverages both visual and textual features, wherein textual features are extracted from the image using the Tesseract OCR engine [73]. More precisely, we make use of a pre-trained autoencoder model to encode the screen's visual information, and a pre-trained BERT language model [28] to encode the screen's textual information. AVGUST uses a three-layer convolutional autoencoder with max pooling [33] and is tasked with encoding an image into a high dimensional vector space, and then decoding the image vector to reconstruct the original image, hence employing a self-supervised training process.

As past work has shown [48], learning features or patterns directly from the pixels of UI screens can be difficult due to the variability in GUI designs across apps. Therefore, to train and use our auto-encoder to learn *app-agnostic* visual patterns, we re-implemented the screen segmentation approach introduced by REMAUI [66], and use the segments to generate abstracted versions of screens from the RICO dataset [27]. As illustrated in Figure 6, in these abstracted screens text components are transformed into yellow boxes and non-text components into blue boxes, on a black background. We trained AVGUST's autoencoder on 33,000 abstracted images from the RICO dataset [27], and to classify an incoming screen, we run it through this abstraction process, and then through the encoder of our autoencoder network to extract the feature vector.

AVGUST's screen classifier leverages linear layers to combine the autoencoder and BERT embeddings and classify the screens. The architecture for the screen classifier consists of three blocks, each containing a linear layer, BatchNorm, a ReLU activation function, and a dropout layer. These blocks are followed by a fully connected output layer that applies softmax function to predict the probability distribution of different screen classes. We then train the screen classifier on partitions of data collected for our evaluation, introduced in Section 3.1, where individual classifiers for each app were trained on data sourced from other apps. This process produced 18 pre-trained screen classifiers for each of our subject apps, and will be reused in AVGUST's test generation phase (see Section 2.4).

Widget Classifier. To classify a given app-widget image, AVGUST leverages its textual, visual, contextual, type, and spatial information: (1) the widget's text is extracted from the widget's image using Tesseract [73] and then encoded using the pre-trained BERT model; (2) the canonical screen of the screen image to which the widget belongs is mapped to an `id` and transformed into a continuous vector via an embedding layer; (3) the visual features of the widget are encoded with the pre-trained ResNet model [37], widely used for encoding images; (4) the UI widget class type (e.g., `EditText`, `ImageButton`) is obtained using the classification method introduced by ReDraw [64] and refined by S2V [47], then mapped by an embedding layer into a continuous vector; and (5) the widget's location on the screen is obtained by dividing the screen into 9 zones and then transformed to a continuous vector via an embedding layer.

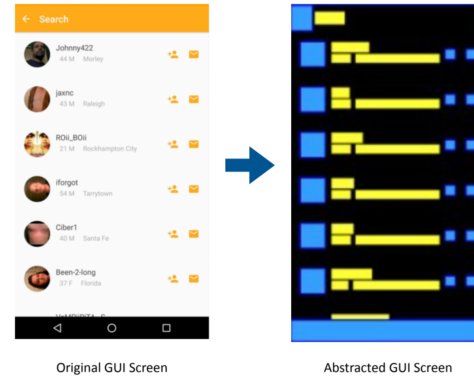


Figure 6: AVGUST's screen abstraction process.

AVGUST's widget classifier then adapts a similar architecture to its screen classifier—three blocks of linear layers followed by a fully connected output layer applying softmax—to combine the different feature vectors discussed above. Note that the embedding layers for the canonical screen, widget location, and widget class type features are optimized during the training phase of the widget classifier to generate meaningful embeddings for each of these input features.

Similarly to the screen classifier, AVGUST's widget classifier is trained on our dataset (Section 3.1), and produces 18 pre-trained models that are reused in AVGUST's test generation phase.

AVGUST's screen and widget classifiers are able to provide the standard top-k labels with different confidence levels. The top-k labels are then recommended to developers in labeling the screens and widgets and, in turn, the IR Models are constructed using the specific labels selected by developers. Note that the IR labels refined by developers and the generated IR Models can be reused by future work. We have thus created a database [10] to serve as a living repository for this problem domain, as also depicted in Figure 1.

2.4 Guided Test Scenario Generation

AVGUST assists developers in generating usage-based tests for their ("target") apps by leveraging the IR models described above. Given a usage of interest, AVGUST selects the relevant IR Model(s) from the IR Model Database (recall Figure 1), and uses them to guide the test generation. Internally, IR Models of the same usage are represented as a single merged model where multiple scenarios for a given usage populate the same state machine. Specifically, the merged model is constructed by using the union of all the edges from all the IR Models of the same usage, demonstrating "all possible transitions". A simplified example of the merged IR model for the sign-in usage is shown in Figure 7. This unified model of scenarios for a single usage gives AVGUST the ability to generate multiple test scenarios for a target usage on an unseen app. AVGUST's test scenario generation phase has three principal components: (1) State Extractor, (2) State Matcher, and (3) Event Generator. We first describe the test generation workflow, and then discuss the three components.

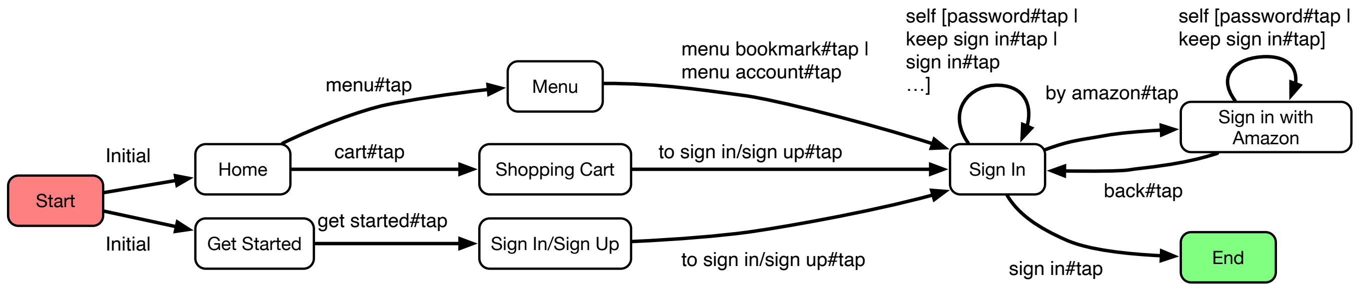


Figure 7: A simplified example of the merged IR Model learned from three sign-in videos of the 6pm and Etsy apps.

Avgust's test scenario generation phase is an iterative process that continues to generate the test inputs based on the target app's current state, until the end condition is met (i.e., the target feature is executed). The process begins by launching the target app and running Avgust's Screen Classifier to retrieve the most likely canonical categories of the target app's starting screen. Avgust then presents the developer with these top-k classification results, and the canonical category selected by the developer is used to match the target app's current device screen to the canonical screen states in the IR Model. Next, Avgust recommends the top-k app widgets for developers to interact with by using its Widget Classifier to map the canonical widgets in the IR Model to the widgets on the target app's current screen. Avgust then checks whether the test should complete based on whether the widget chosen by the developer will lead to the end state in the IR Model. If not, the chosen widget is triggered, the target app's next state becomes its current state, and this process repeats until the end state in the unified IR Model for a given usage is reached.

2.4.1 State Extractor. Avgust's State Extractor extends recent work on the MAPIT [77] test case transfer tool. Specifically, for a given target app Avgust extracts (1) the bitmap of the current screen, (2) the graph representation of the app screen's UI layout hierarchy [7], and (3) the boundaries of each UI widget and their corresponding cropped images. The UI layout hierarchy is an XML file that contains the information of all the UI widgets on the target app's current screen, such as their position, size, textual attributes (e.g., "Sign In"), and class name (e.g., `ImageButton`). The extracted information is used by Avgust to generate tests, and also to explore different variants of Avgust's classifiers as discussed in Section 3.

2.4.2 State Matcher. As discussed previously, Avgust uses its Screen Classifier to suggest the top-k candidates for the canonical category of a target app's given screen. Once the developer selects from one of the suggested categories, Avgust maps the current screen to the corresponding state in the IR Model. Since all possible transitions captured in the IR Models are known, Avgust is able to recommend the target app's widget(s) with which the developer should interact by using a combination of the Widget Classifier (recall Section 2.3.2), information obtained from the State Extractor (recall Section 2.4.1), and a set of pre-defined heuristics.

The number of widgets to be recommended by Avgust is determined by a configurable threshold. Avgust first checks whether the target widgets match the expected canonical widgets from the IR Model based on a set of heuristics that can be divided into two categories. The first category are heuristics that infer a widget's type based on the UI class of its parent widget. This allows Avgust to bypass the noise that may be present in the data associated with an individual widget. For example, Avgust identifies a widget that represents a menu item, not by trying to capture all possible menu items, but much more simply by comparing its parent widget's UI class to *ListView*. The second category are heuristics that correlate the textual data of a widget with similar terms associated with each of the canonical widgets (e.g., the terms from our set of canonical widgets [10] discussed in Section 2.3.2 and their synonyms).

If the heuristics alone yield a number of recommendations below the set threshold, as the next step Avgust will predict the top-1 classification of the canonical category for each interactive widget on the target app's screen. If the target-app widgets that match the expected canonical widgets in the IR Model bring the total number of matched widgets above the threshold, the process terminates and the identified widgets are presented to the developer. Otherwise, as the final step, the matching criteria are relaxed and the process switches from the top-1 to the top-5 classifications of each widget's canonical category.

2.4.3 Event Generator. With a chosen widget, Avgust generates an executable event to trigger based on whether the widget requires user input. This is determined by the widget type. For example, `EditText` [11] is a widget type that requires an input from the user, such as entering the email address. In such cases, Avgust prompts the developer for text inputs, as these typically do not generalize across apps. If the selected widget does not require user input, the Event Generator automatically executes the touch event (e.g., tap, swipe) stored in the transition of the IR Model.

This event generation process requires minimal effort from the developer and provides the flexibility to test the same usage with different desired text inputs of the developer's choice. A test scenario is generated when the end condition is met, as discussed earlier, and each test consists of a sequence of events triggered by the Event Generator.

2.5 Avgust's Implementation

Avgust is implemented in Python with 10,700 SLOC, of which the screen and widget classifiers are stand-alone modules totaling 2,800 SLOC, and include the autoencoder model we developed. Avgust additionally extended several research tools, including the modified V2S (500 SLOC in Python), UIED (300 SLOC in Python),

and the re-implemented REMAUI (5,000 SLOC in Java). Avgust employs the `pytransitions` library [70] to manipulate its state-machine IR Models, and uses the Appium testing framework [4] for its test generation.

3 EVALUATION OF AVGUST

To demonstrate Avgust's effectiveness at generating usage tests, and its improvement on the state of the art, we answer two research questions:

RQ1 How effective is Avgust at generating tests that exercise the desired usage?
RQ2 How accurate are Avgust's vision-only screen and widget classifiers?

3.1 Evaluation Context

To evaluate Avgust as a whole, app videos are needed as its input. To collect these videos, we relied on the apps and usages defined by the FrUITeR benchmark [85], which contains 20 popular apps and 18 most-common app usages. We then designed a user study to collect screen recordings of these app usages, which resulted in the collection of 374 videos. To evaluate Avgust's image classification component, we developed a semi-automated pipeline to annotate the video frames of screen recordings collected by users with ground-truth canonical categories for both screens and GUI widgets. This process is detailed in Section 3.1.2.

3.1.1 Video Collection. We designed a large-scale user study to collect videos of app usages from participants. We recruited and assigned the 18 usages and 20 apps to 61 computer science students in a Master's level course at the authors' institution, and asked them to record videos of themselves exercising scenarios that triggered the features associated with the usages. We assigned usages such that each student was assigned 2 applications, each with 2 different usages, for a combination of 4 app-usage pairs. We balanced the assigned apps and usages evenly across participants. We then asked them to collect two screen-recording videos for each app-usage pair, for a *potential* total of 8 videos per participant. We asked for *two* videos per app-usage pair in order to capture different ways of exercising a given feature (e.g., adding an item to a shopping cart by searching vs. by browsing categories). However, if a participant deemed that there were not two distinct scenarios for exercising a given feature, they were allowed to provide only one usage.

This study was conducted remotely due to COVID-19, and participants were given detailed instructions for installing and setting up an Android emulator (Nexus 5X, API24), the `.apk` files required to install their assigned apps, and a short textual description of the assigned use cases (illustrated in Table 1). Additionally, we provided a small desktop application that allowed participants to record the screens and a usage trace of their scenarios. This application makes use of the `adb screenrecord` and Linux `getevent` command line utilities. We provide these instructions, the app `.apk` files, usage descriptions, device recording tool, and anonymized collected data in our online appendix [10]. This study was approved by the Institutional Review Board (IRB) at the authors' university (IRBNet 1666261-1).

This data collection process spanned two semesters, and in total, 31 of the originally recruited 61 students completed the study, some with partial data, hence the imbalance of videos across apps and usages shown in Tables 1 and 2. In the end, we obtained a dataset of 374 screen recordings of 18 usages from 18 of the 20 apps (shown in Table 2) from the FrUITeR benchmark [85] discussed earlier.

3.1.2 Ground-Truth Annotation. Recall from Section 2 that Avgust's classification involves (1) the screen classifier that maps an app screen to an abstract screen IR category, and (2) the widget classifier that maps a cropped widget image to an app-independent canonical widget category. To establish the ground-truth labels (i.e., the correct categories needed to generate Avgust's IR Models), we developed a pipeline to import pairs of screen-widget images into Label Studio [12], and trained four human annotators to label the data. Specifically, the screen-widget image pairs are sourced from the *GUI Event Frames* that Avgust converted during its *Video Analysis* phase (recall Figure 1).

To establish our ground truth categorizations, we provided detailed instructions to and trained the four annotators to label the data based on the 37 screen and 74 widget canonical categories we defined (recall Section 2.3.2). Each image is labeled by at least two annotators, and discrepancies are resolved by negotiated agreement [23] with the annotators and one author.

Table 1: The 18 usages used in Avgust's evaluation.

Usage ID	Test Case Name	Tested Functionalities	#Videos
U1	Sign In	provide username and password to sign in	21
U2	Sign Up	provide required information to sign up	76
U3	Search	use search bar to search a product/news	29
U4	Detail	find and open details of the first search result item	17
U5	Category	find first category and open browsing page for it	27
U6	About	find and open about information of the app	15
U7	Account	find and open account management page	18
U8	Help	find and open help page of the app	17
U9	Menu	find and open primary app menu	12
U10	Contact	find and open contact page of the app	16
U11	Terms	find and open legal information of the app	20
U12	Add Cart	add the first search result item to cart	13
U13	Remove Cart	open cart and remove the first item from cart	10
U14	Address	add a new address to the account	11
U15	Filter	filter/sort search results	14
U16	Add Bookmark	add first search result item to the bookmark	15
U17	Remove Bookmark	open the bookmark and remove first item from it	20
U18	Textsize	change text size	23

Table 2: The 18 subject apps used in Avgust's evaluation.

App ID	App Name	#Downloads (mil)	#Videos
A1	AliExpress	100	27
A2	Ebay	100	15
A3	Etsy	10	25
A4	Dailyhunt	1.7	8
A5	Geek	10	17
A6	Groupon	50	53
A7	Home	10	53
A8	6PM	0.5	25
A9	Wish	100	44
A10	The Guardian	5	8
A11	ABC News	5	18
A12	USA Today	5	26
A13	Zappos	0.054	11
A14	BuzzFeed	5	8
A15	Fox News	10	11
A16	BBC News	10	6
A17	Reuters	1	12
A18	News Break	0.322	7

This data collection process was time-consuming and intensive, spanning ~8 person-months of effort. At the end of the process, we derived a comprehensive labeled dataset containing 2,478 ground-truth labels for screens and 2,434 labels for widgets across 18 apps. Given these labels, Avgust was able to automatically generate the IR Models for the usages needed for our evaluation. Our labeled dataset, as well as the annotation pipeline we developed can be easily reused or extended by future work in this area [10].

3.2 RQ1: Avgust's Test Quality

Avgust's main goal is to generate a test for a target app that accomplishes the usage encoded by the videos of other apps. To evaluate how well Avgust accomplishes that goal, for each of the 18 app usages, we randomly selected 3 apps under test (AUTs) as the target apps. (For three of the usages, we selected only 2 apps because we were unable to extract data from certain commercial apps due to security reasons or limitations of the underlying used testing framework [4].) For each of those apps, we use the merged IR Model Avgust learned from all the other 17 apps to guide Avgust's test generation, aiming to demonstrate Avgust's ability to generate tests for *unseen* apps. As discussed in Section 2.4, Avgust's test generation is a developer-in-the-loop process. Specifically, four of the authors served as the developers interacting with the tool during this process. We use the first test Avgust generates for the evaluation, resulting in a total of 51 tests across 18 app usages. (We limit our evaluation to a single test per usage per app due to the significant manual effort involved in the evaluation process.)

We examined each of the tests manually to consider whether it accomplished the intended usage. Note that this judgement is objective. For example, it is straightforward and unambiguous to determine whether the generated test accomplishes the Sign In usage — either the tests signs into the app or it does not. We found that 35 of the 51 tests (68.6%) accomplished the usage, meaning that Avgust successfully generated a correct test.

For the remaining 16 tests, we measured how similar each generated test was to the closest human test, to evaluate whether it would potentially save human effort in writing the test. The is due to the nature of *usage-based* tests, as there are usually multiple correct paths of exercising the same usage. Thus, to enable fair comparison, we compare Avgust's test with the *closest* human test. We measured similarity using two metrics: precision and recall in matching the human test's behavior. Precision measures the fraction of the states and transitions in the generated test that occur in the most-similar human test from the relevant videos. Recall measures the fraction of the states and transitions in the most-similar human test that occur in the generated test. The *closest* human test is chosen using the precision similarity metric.

Table 3 lists the similarity results for the 16 tests across 11 usages that do not satisfy that usage, and the cloest human test. On average, 79% of the states and 47% of the transitions in the generated tests is captured by the most similar human test. This means that Avgust rarely visited an incorrect state, but often triggered inputs for GUI widgets not triggered by humans. Meanwhile, on average, the generated tests capture 68% of the states and 37% of the transitions in the closest human test. This means that Avgust was able to visit a majority of the screens seen in the human test, but correctly exercised comparatively fewer expected GUI widgets that trigger proper transitions. This suggests that while the 31.4% of the tests Avgust generates do not fully exercise the intended usage, they may be at least partially helpful for developers writing tests. Our future work will examine the effort reduction Avgust's tests produce for developers.

Table 3: We compare the 16 Avgust-generated tests that do not satisfy their intended usage with the most-similar human test, to indicate how much work these tests may save a developer.

		precision		recall	
	Usage	states	transitions	states	transitions
U4	Search	1.00	0.50	1.00	0.50
U5	Terms	0.71	0.29	0.63	0.33
U9	About	1.00	0.50	1.00	0.25
U10	Contact	0.75	0.37	0.72	0.33
U11	Help	0.67	0.50	0.67	0.33
U12	AddCart	0.75	0.59	0.55	0.37
U13	RemoveCart	1.00	0.69	0.62	0.42
U14	Address	0.83	0.69	1.00	0.75
U15	Filter	1.00	0.50	1.00	0.40
U17	RemoveBookmark	1.00	0.75	0.60	0.50
U18	Textsize	0.00	0.00	0.00	0.00
average		0.79	0.47	0.68	0.37

> RA1: We find that 69% of Avgust's generated tests successfully accomplish the desired usage, saving the developer from having to manually write the test from scratch. For the remaining 31% of the tests, we found that those tests capture significant portions of the behavior in the most-similar test a human would write, again, potentially saving human effort.

3.3 RQ2: Avgust's Classification Accuracy

RQ2 compares Avgust's vision-only screen classification and widget classification accuracy to the state-of-the-art S2V [47]. First, Section 3.3.1 evaluates Avgust's classification independently, as a stand-alone tool. Then, Section 3.3.2 evaluates Avgust's classification in the context of test generation.

3.3.1 Evaluating Avgust's Stand-Alone Classification. To evaluate Avgust's vision-only classification module as a stand-alone technique, we use our labeled dataset from Section 3.1.2. We use leave-one-out cross-validation [22] to evaluate the accuracy of Avgust's screen and widget classifiers. For each of the 18 apps, we train our model on the data from all the other apps, and test on that app.

Screen Classification: We evaluate three variants of Avgust's screen classifier. The first, the standard Avgust as introduced in Section 2, and two other classifiers that use only Avgust's autoencoder (AE) model and classify with two widely-adopted methods KNN [14] (AE + KNN) and MLP [32] (AE + MLP), respectively. As Avgust's AE model only encodes the screen's visual features, the results aim to demonstrate the impact of visual-only information on the classification tasks. We did not evaluate the text-only model since it contains app-specific noises (e.g., news content, product

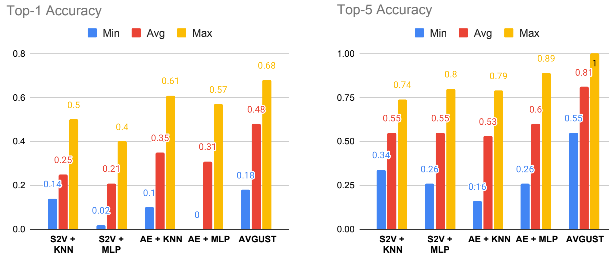


Figure 8: AVGUST's vision-only screen classification outperforms SV2 [47] and the other AVGUST classifier variants in both top-1 and top-5 accuracy.

description) that do not generalize, and text-only information has already been shown insufficient for classification tasks [39].

To compare AVGUST with S2V, we adapt S2V to learn from the information on the screen images only, and obtain the UI layout information [7] that S2V requires as its input. To do so, we reverse engineered app screen images using REMAUI [66], a research tool to convert app screen image into its corresponding UI layout [7]. This is the same process AVGUST uses (recall Section 2), aiming to ensure that S2V and AVGUST learn from the same raw information on the app screen. We then apply KNN and MLP to S2V's screen embeddings, resulting in two S2V's variants (S2V + KNN, S2V + MLP).

Figure 8 shows that AVGUST's classifier consistently outperforms all versions of SV2 and the other AVGUST classifier variants in both top-1 and top-5 accuracy. AVGUST's composite classifier that uses both visual and textual screen features outperforms both autoencoder-only variants by more than 20%, suggesting that although visual features are important in encoding a UI screen, adding textual features significantly improves the quality of the generated embeddings and results in higher classification accuracy.

While using S2V's screen embeddings is effective for downstream tasks when the dynamic UI layout information is available [47], we were unable to achieve high classification accuracy using the pre-trained S2V model by reverse engineering app screen images into S2V's required format. This suggests that the existing pre-trained models cannot be used for vision-only tasks effectively.

Widget Classification: To compare AVGUST's widget classification with S2V [47], we studied S2V's implementation and isolated its underlying model that encodes the widget's information. We then applied both KNN and MLP to S2V's widget embeddings.

Figure 9 shows that AVGUST's widget classifier outperforms both S2V variants that use the pre-trained UI widget encoder for two reasons. First, S2V's UI widget encoder only uses a widget's textual information and class type, whereas AVGUST's widget classifier takes into account many other widget features, such as its location on the screen and visual features. Second, S2V's UI widget encoder is trained using the textual information available on dynamically extracted UI layout, which is not available for the widgets in AVGUST's vision-only classification task.

3.3.2 Evaluating AVGUST's Classification for Test Generation. We next evaluate AVGUST's classification accuracy in the context of test generation. During the test generation phase in Section 3.2, we recorded AVGUST's Top-1 and Top-5 recommendations at each step, and evaluate the accuracy of those recommendations.

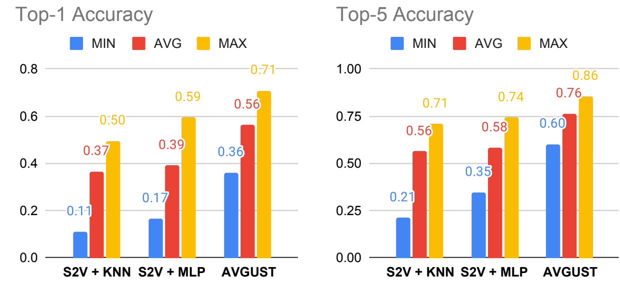


Figure 9: AVGUST's vision-only widget classification consistently outperforms S2V [47].

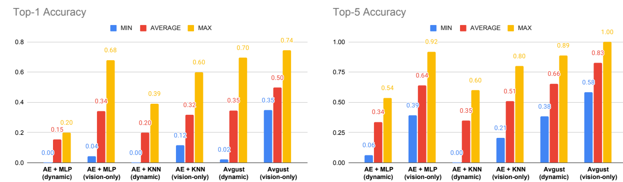


Figure 10: The accuracy of AVGUST's vision-only screen classifier variations with vision-only and dynamic input data.

Evaluation AVGUST's Screen Classification: Besides AVGUST's built-in screen classifier introduced in Section 2, we further implemented 5 variants that incorporate app's runtime information, aiming to get insights on whether runtime information can improve the classification's accuracy. This is inspired by S2V [47], which relies on the screen's runtime UI layout information [7], such as the *Activity name* [6] and *content description* of the widgets on the screen [8]. As AVGUST's test generation phase interacts with the target app at runtime, AVGUST can crawl the UI for this layout information. We thus relaxed the vision-only constraint, and modified AVGUST to use this dynamic UI layout information. We term the modified version AVGUST-Dynamic. Note that the fundamental difference between AVGUST-Dynamic and S2V is the training phase. AVGUST-Dynamic still uses vision-only information (screen images) to train its models, while S2V requires the dynamic UI layout information in the training data. In practice, as classification tasks usually require a large amount of training data, AVGUST-Dynamic makes the training process significantly easier by only requiring screen images, while S2V requires crawling the UI layout information at app runtime for every app screen in the training set.

Figure 10 shows AVGUST's screen classifier variants' accuracies during the test generation phase. Comparing Figures 10 and 8, we observe that in both cases, AVGUST always outperforms the two autoencoder-only variants. All the classifier variants are more accurate when using the vision-only data, compared to also using dynamic app information captured during runtime. While this might seem counterintuitive, one possible explanation is that the features extracted from the vision-only input data are similar to the data AVGUST's built-in classifier were trained with, whereas the dynamically obtained information might expose much more textual information (e.g., content description) that is not consistent with the OCR-based textual information used in the training phase.

Evaluation AVGUST's Widget Classification: To evaluate AVGUST's widget classifier during test generation, we assess whether

Avgust can faithfully recommend widgets to match the transitions suggested by its IR Model. We recorded the next transitions suggested by the IR Model at each step of the test generation (recall Section 2.4), as well as the crops of Avgust's recommended widgets. Three annotators then manually inspected the cropped widgets and determined whether their canonical categories match one of the suggested transitions. In total, over all the generated tests, Avgust's widget classifier correctly recommended widgets 77.4% of the time (175 out of 226 steps).

> RA2: Avgust's classifiers consistently outperform the state-of-the-art S2V tool. We find that using textual and visual features together improves accuracy, but adding runtime features decreases accuracy, perhaps because these features are too different from the ones used to train Avgust's vision-only models.

4 RELATED WORK

Automated Input Generation for Mobile Apps: Existing automated test generation techniques share a complementary objective to ours: they mainly focus on generating tests to maximize code coverage and detect crashes, as opposed to generating *usage-based* tests to test a certain functionality. The large body of existing work includes model-based testing [15, 16, 30, 34, 55, 56, 72, 74, 75], random testing [3, 57, 84], and systematic testing [13, 17, 58, 59]. Recently, Su et al. proposed Genie [76], which is the first automated testing technique to detecting non-crashing functional bugs in Android apps. However, Genie is a random-based fuzzing technique, thus does not generate usage-based tests. Besides the differences in testing objectives, Avgust's model is app-independent (representing usage scenarios learned from different apps) and is derived purely from visual data, which differs from existing model-based testing techniques. Furthermore, in comparison to recent human-in-the-loop techniques, e.g. NaviDroid [54], Avgust's model and recommendations differ by providing suggestions for GUI actions that fulfill a given use case, as opposed to uncovering unexplored areas of an app.

Test Reuse in Mobile Apps: The area of research that most closely aligns with usage-based test generation is the work on *UI test reuse*, which has been steadily growing over the past few years [19, 20, 50, 61, 62, 71, 77, 85]. These techniques can transfer an existing usage-based test from a *source app* to its equivalent test of a *target app* that shares the same functionality, but cannot generate usage-based tests from scratch. Furthermore, as discussed in Section 1, existing test-reuse techniques have three important limitations that we directly address in this paper.

Learning Patterns from Crowdsourced Tests: Similar to our objective, another line of work aims to learn patterns from crowdsourced tests for automated test generation. However, while such techniques learn from crowdsourced data, their test objectives are to increase coverage or fault-finding ability as opposed to generating usage-based tests. For example, Replica [81] compares existing in-house tests with the user traces in the field, and generates new tests to mimic field traces that are not covered by the in-house tests. Replica relies on pre-existing in-house tests that may not be available, as well as app instrumentation. Ermuth et al. proposed an approach to generate "macro events" that group multiple low-level events into logic steps performed by real users [31], such as filling and submitting a form. However, these macro events are recurring patterns across *all* the user traces collected when exercising the *entire* app, thus do not capture fine-grained user behaviors exercised in specific usages. Similarly, MonkeyLab and Polariz [53, 60] mines users' event traces to generate combinations of low-level events representing natural scenarios (similar to "macro events"), as well as untested corner cases (similar to Replica's objective). ComboDroid [80] aims to reach complex app functionality by combining independent short "use cases", such as toggling a setting, or switching to a different screen. Humanoid [49] leverages a deep neural network model to learn input actions based on real-user traces. However, the generated tests from all the work mentioned above again focus on maximizing the code coverage, but do not aim to generate tests of specific usages.

Specification-based Testing for Mobile Apps: Finally, this type of testing aims to generate tests that cover specific functionalities (similar to our definition of *usages*), guided by manually-written specifications. For example, FARLEAD-Android [44] requires the developer to provide UI test scenarios written in Gherkin [5] in order to generate tests using reinforcement learning. Similarly, AppFlow [39] requires the developer to first create a test library that covers the common functionalities in a certain app category (e.g., shopping apps) using a Gherkin-based language that AppFlow defines. AppFlow then synthesizes app-specific tests according to the test library. Augusto [63] uses GUI ripping to explore popular app-independent functionalities (referred to as "AIFs"), and generates functional tests accordingly. However, developers have to manually define UI patterns and Alloy semantic model [40] to describe the AIFs. Avgust attempts to advance upon such work by simplifying the specification process by relying purely on videos that specify desired test behaviors.

5 CONTRIBUTIONS

We have presented Avgust, a method for generating usage-based tests for the Android platform. By targeting usage-based tests, Avgust solves what mobile developers identify as a major need [52] but that the state of the art has failed to address [35, 43, 52]. Avgust uses user-generated videos of app usages to learn a model of a usage, and then applies that model to a new target app to generate tests. Evaluating on 374 videos of common uses of 18 popular apps, we show that 69% of the tests Avgust generates successfully execute the desired usage, that the remaining generated tests have potential for reducing developer effort in writing tests, and that Avgust's classifiers outperform the state of the art. Our work suggests a promising direction of research into usage-based test generation, and outlines outstanding problems in classification accuracy that future research should address.

ACKNOWLEDGEMENT

This work is supported by the U.S. National Science Foundation under grant no. CCF-1717963, CCF-1763423, CNS-1823354, CCF-1955853, and CCF-2030859 (to the Computing Research Association for the CIFellows Project), as well as the U.S. Office of Naval Research under grant N00014-17-1-2896. Additionally, we would like to thank Arthur Wu for his help on data collection and annotation.

REFERENCES

[1] 2016. uiautomator | Android Developers. https://android-doc.github.io/tools/help/uiautomator/index.html
[2] 2019. GATOR: Program Analysis Toolkit For Android. http://web.cse.ohio-state.edu/presto/software/gator
[3] 2020. UI/Application Exerciser Monkey | Android Developers. https://developer.android.com/studio/test/monkey
[4] 2021. Appium: Mobile App Automation Made Awesome. http://appium.io
[5] 2021. Gherkin Syntax - Cucumber Documentation. https://cucumber.io/docs/gherkin
[6] 2021. Introduction to Activities | Android Developers. https://developer.android.com/guide/components/activities/intro-activities
[7] 2021. Layouts | Android Developers. https://developer.android.com/guide/topics/ui/declaring-layout
[8] 2021. Make apps more accessible | Android Developers. https://developer.android.com/guide/topics/ui/accessibility/apps
[9] 2021. Soot. http://soot-oss.github.io/soot/
[10] 2022. AVGUST's public repository. https://doi.org/10.5281/zenodo.7036218
[11] 2022. EditText | Android Developers. https://developer.android.com/reference/android/widget/EditText
[12] 2022. Label Studio – Open Source Data Labeling. https://labelstud.io [Online; accessed 9. Mar. 2022].
[13] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic execution of android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 83–93.
[14] Naomi S Altman. 1992. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician* 46, 3 (1992), 175–185.
[15] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. 2012. Using GUI ripping for automated testing of Android applications. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 258–261.
[16] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M Memon. 2014. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE software* 32, 5 (2014), 53–59.
[17] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 641–660.
[18] G. Bavota, M. Linares-Vásquez, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. 2015. The Impact of API Change- and Fault-Proneness on the User Ratings of Android Apps. *IEEE Transactions on Software Engineering (TSE)* (2015).
[19] Farnaz Behrang and Alessandro Orso. 2018. Test migration for efficient large-scale assessment of mobile app coding assignments. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*.
[20] Farnaz Behrang and Alessandro Orso. 2019. Test Migration Between Mobile Apps with Similar Functionality. In *34th International Conference on Automated Software Engineering (ASE 2019)*.
[21] Carlos Bernal-Cárdenas, Nathan Cooper, Kevin Moran, Oscar Chaparro, Andrian Marcus, and Denys Poshyvanyk. 2020. Translating video recordings of mobile app usages into replayable scenarios. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 309–321.
[22] Christopher M Bishop and Nasser M Nasrabadi. 2006. *Pattern recognition and machine learning*. Vol. 4. Springer.
[23] K. Charmaz. 2006. *Constructing Grounded Theory*. SAGE Publications Inc.
[24] 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*. 1202–1214.
[25] A. Ciurumelea, A. Schaufelbühl, S. Panichella, and H. C. Gall. 2017. Analyzing Reviews and Code of Mobile Apps for Better Release Planning. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER) (SANER'17)*. 91–102. https://doi.org/10.1109/SANER.2017.7884612
[26] Nathan Cooper, Carlos Bernal-Cárdenas, Oscar Chaparro, Kevin Moran, and Denys Poshyvanyk. 2021. It Takes Two to Tango: Combining Visual and Textual Information for Detecting Duplicate Video-Based Bug Reports. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 957–969. https://doi.org/10.1109/ICSE43902.2021.00091
[27] 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*. 845–854.
[28] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
[29] Andrea Di Sorbo, Sebastiano Panichella, Carol V. Alexandru, Junji Shimagaki, Corrado A. Visaggio, Gerardo Canfora, and Harald C. Gall. 2016. What Would Users Change in My App? Summarizing App Reviews for Recommending Software Changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. ACM, Seattle, WA, USA, 499–510. https://doi.org/10.1145/2950290.2950299
[30] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of Android apps. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 481–492.
[31] Markus Ermuth and Michael Pradel. 2016. Monkey see, monkey do: Effective generation of GUI tests with inferred macro events. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 82–93.
[32] Matt W Gardner and SR Dorling. 1998. Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. *Atmospheric environment* 32, 14-15 (1998), 2627–2636.
[33] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.
[34] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 269–280.
[35] Roman Haas, Daniel Elsner, Elmar Juergens, Alexander Pretschner, and Sven Apel. 2021. How can manual testing processes be optimized? developer survey, optimization guidelines, and case studies. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1281–1291.
[36] Madeleine Havranek, Carlos Bernal-Cárdenas, Nathan Cooper, Oscar Chaparro, Denys Poshyvnayk, and Kevin Moran. 2021. V2S: A Tool for Translating Video Recordings of Mobile App Usages into Replayable Scenarios. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 65–68.
[37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.
[38] G. Hu, X. Yuan, Y. Tang, and J. Yang. 2014. Efficiently, effectively detecting mobile app bugs with AppDoctor. In *Ninth European Conference on Computer Systems (EuroSys'14)*. Article No.18.
[39] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: using machine learning to synthesize robust, reusable UI tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 269–282.
[40] Daniel Jackson. 2002. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11, 2 (2002), 256–290.
[41] N. Jones. 2013. *Seven best practices for optimizing mobile testing efforts*. Technical Report G00248240. Gartner.
[42] 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*. 1–4.
[43] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. 2015. Understanding the Test Automation Culture of App Developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 1–10. https://doi.org/10.1109/ICST.2015.7102609
[44] Yavuz Koroglu and Alper Sen. 2021. Functional test generation from UI test scenarios using reinforcement learning for android applications. *Software Testing, Verification and Reliability* 31, 3 (2021), e1752.
[45] Greg Lee. 2018. Android View Measurement - The Inside Scoop. *Medium* (Jun 2018). https://blog.takescoop.com/android-view-measurement-d1f2f5c98f75
[46] Kanglin Li and Mengqi Wu. 2006. *Effective GUI testing automation: Developing an automated GUI testing tool*. John Wiley & Sons.
[47] Toby Jia-Jun Li, Lindsay Popowski, Tom Mitchell, and Brad A Myers. 2021. Screen2Vec: Semantic Embedding of GUI Screens and GUI Components. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–15.
[48] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A Deep Learning-Based Approach to Automated Black-box Android App Testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1070–1073. https://doi.org/10.1109/ASE.2019.00104
[49] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A deep learning-based approach to automated black-box android app testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1070–1073.
[50] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test Transfer Across Mobile Apps Through Semantic Mapping. In *34th International Conference on Automated Software Engineering (ASE 2019)*.
[51] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. 2013. API Change and Fault Proneness: A Threat to the Success of Android Apps. In *ESEC/FSE'13*. 477–487.

[52] Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2017. How do developers test android applications?. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*.

[53] Mario Linares-Vásquez, Martin White, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2015. Mining android app usages for generating actionable gui-based execution scenarios. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 111–122.

[54] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2022. Guided bug crush: Assist manual gui testing of android apps via hint moves. In *CHI Conference on Human Factors in Computing Systems*. 1–14.

[55] Nikola Lukić, Saghar Talebipour, and Nenad Medvidović. 2020. AirMochi: a tool for remotely controlling iOS devices. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1273–1277.

[56] Nikola Lukić, Saghar Talebipour, and Nenad Medvidović. 2020. Remote control of ios devices via accessibility features. In *Proceedings of the 2020 ACM Workshop on Forming an Ecosystem Around Software Transformation*. 35–40.

[57] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 224–234.

[58] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 599–609.

[59] 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*. 94–105.

[60] Ke Mao, Mark Harman, and Yue Jia. 2017. Crowd Intelligence Enhances Automated Mobile Testing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 16–26.

[61] Leonardo Mariani, Ali Mohebbi, Mauro Pezze, and Valerio Terragni. 2021. Semantic Matching of GUI Events for Test Reuse: Are We There Yet?. In *The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[62] Leonardo Mariani, Mauro Pezzè, Valerio Terragni, and Daniele Zuddas. 2021. An Evolutionary Approach to Adapt Tests Across Mobile Apps. In *The 2nd ACM/IEEE International Conference on Automation of Software Test (AST 2021)*.

[63] Leonardo Mariani, Mauro Pezzè, and Daniele Zuddas. 2018. Augusto: Exploiting popular functionalities for the generation of semantic gui tests with oracles. In *Proceedings of the 40th International Conference on Software Engineering*. 280–290.

[64] Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2018. Machine learning-based prototyping of graphical user interfaces for mobile apps. *IEEE Transactions on Software Engineering* 46, 2 (2018), 196–221.

[65] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2016. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 33–44. https://doi.org/10.1109/ICST.2016.34

[66] Tuan Anh Nguyen and Christoph Csallner. 2015. Reverse engineering mobile application user interfaces with remaui (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 248–259.

[67] Fabio Palomba, Mario Linares-Vásquez, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. 2015. User Reviews Matter! Tracking Crowdsourced Reviews to Support Evolution of Successful Apps. In *Proceedings of 31st IEEE International Conference on Software Maintenance and Evolution (ICSME'15)*. to appear.

[68] Fabio Palomba, Pasquale Salza, Adelina Ciurumelea, Sebastiano Panichella, Harald Gall, Filomena Ferrucci, and Andrea De Lucia. 2017. Recommending and Localizing Change Requests for Mobile Apps Based on User Reviews. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 106–117. https://doi.org/10.1109/ICSE.2017.18

[69] Yunchen Pu, Zhe Gan, Ricardo Henao, Xin Yuan, Chunyuan Li, Andrew Stevens, and Lawrence Carin. 2016. Variational Autoencoder for Deep Learning of Images, Labels and Captions. In *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (Eds.), Vol. 29. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2016/file/eb86d510361fc23b59f18c1bc9802cc6-Paper.pdf

[70] pytransitions. 2021. transitions. https://github.com/pytransitions/transitions

[71] Xue Qin, Hao Zhong, and Xiaoyin Wang. 2019. Testmig: Migrating gui test cases from ios to android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 284–295.

[72] Ibrahim-Anka Salihu, Rosziati Ibrahim, Bestoun S Ahmed, Kamal Z Zamli, and Asmau Usman. 2019. AMOGA: A static-dynamic model generation strategy for mobile apps testing. *IEEE Access* 7 (2019), 17158–17173.

[73] Ray Smith. 2007. An overview of the Tesseract OCR engine. In *Ninth international conference on document analysis and recognition (ICDAR 2007)*, Vol. 2. IEEE, 629–633.

[74] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 245–256.

[75] Ting Su, Yichen Yan, Jue Wang, and Zhendong Su. 2020. Automated Functional Fuzzing of Android Apps. *arXiv preprint arXiv:2008.03585* (2020).

[76] Ting Su, Yichen Yan, Jue Wang, Jingling Sun, Yiheng Xiong, Geguang Pu, Ke Wang, and Zhendong Su. 2021. Fully Automated Functional Fuzzing of Android Apps for Detecting Non-Crashing Logic Bugs. In *Proceedings of the 28th ACM Joint Meeting on European Software ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

[77] Saghar Talebipour, Yixue Zhao, Luka Dojcilovic, Chenggang Li, and Nenad Medvidovic. 2021. UI Test Migration Across Mobile Platforms. In *36th International Conference on Automated Software Engineering (ASE 2021)*.

[78] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224.

[79] Akshaj Verma. 2021. PyTorch [Vision] — Binary Image Classification - Towards Data Science. *Medium* (May 2021). https://towardsdatascience.com/pytorch-vision-binary-image-classification-d9a227705cf9

[80] Jue Wang, Yanyan Jiang, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu. 2020. ComboDroid: generating high-quality test inputs for Android apps via use case combinations. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 469–480.

[81] Qianqian Wang and Alessandro Orso. 2020. Improving Testing by Mimicking User Behavior. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 488–498.

[82] Mulong Xie, Sidong Feng, Zhenchang Xing, Jieshan Chen, and Chunyang Chen. 2020. UIED: a hybrid tool for GUI element detection. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1655–1659.

[83] Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. 2018. Static window transition graphs for Android. *Automated Software Engineering* 25, 4 (2018), 833–873.

[84] Faraz YazdaniBanafsheDaragh and Sam Malek. 2021. Deep GUI: Black-box GUI Input Generation with Deep Learning. In *36th International Conference on Automated Software Engineering (ASE 2021)*.

[85] Yixue Zhao, Justin Chen, Adriana Sejfia, Marcelo Schmitt Laser, Jie Zhang, Federica Sarro, Mark Harman, and Nenad Medvidovic. 2020. FrUITeR: A Framework for Evaluating UI Test Reuse. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE '20)*. ACM, New York, NY, USA. https://doi.org/10.1145/3368089.3409708