AURORA: Navigating UI Tarpits via Automated Neural Screen Understanding

Safwat Ali Khan
George Mason University
Fairfax, VA, United States
skhan89@gmu.edu

Wenyu Wang
University of Illinois
Urbana, IL, United States
wenyu2@illinois.edu

Yiran Ren
Dragon Testing Technology
Hangzhou, China
renyiran@dragontesting.cn

Bin Zhu
Dragon Testing Technology
Hangzhou, China
zhubin@dragontesting.cn

Jiangfan Shi
Dragon Testing Technology
Hangzhou, China
shijiangfan@dragontesting.cn

Alyssa McGowan
Thomas Jefferson High School
Alexandria, VA, United States
alyssamc38@gmail.com

Wing Lam
George Mason University
Fairfax, VA, United States
winglam@gmu.edu

Kevin Moran
University of Central Florida
Orlando, FL, United States
kpmoran@ucf.edu

***Abstract*—Nearly a decade of research in software engineering has focused on automating mobile app testing to help engineers in overcoming the unique challenges associated with the software platform. Much of this work has come in the form of Automated Input Generation tools (AIG tools) that dynamically explore app screens. However, such tools have repeatedly been demonstrated to achieve lower-than-expected code coverage – particularly on sophisticated proprietary apps. Prior work has illustrated that a primary cause of these coverage deficiencies is related to so-called *tarpits*, or complex screens that are difficult to navigate.**

In this paper, we take a critical step toward enabling AIG tools to effectively navigate tarpits during app exploration through a new form of automated semantic screen understanding. That is, we introduce AURORA, a technique that learns from the visual and textual patterns that exist in mobile app UIs to automatically detect common screen designs and navigate them accordingly. The key idea of AURORA is that there are a finite number of mobile app screen designs, albeit with subtle variations, such that the general patterns of different categories of UI designs can be *learned*. As such, AURORA employs a multi-modal, neural screen classifier that is able to recognize the most common types of UI screen designs. After recognizing a given screen, it then applies a set of flexible and generalizable heuristics to properly navigate the screen. We evaluated AURORA both on a set of 12 apps with known tarpits from prior work, and on a new set of five of the most popular apps from the Google Play store. Our results indicate that AURORA is able to effectively navigate tarpit screens, outperforming prior approaches that *avoid* tarpits by 19.6% in terms of method coverage. Our analysis of the results finds that the improvements can be attributed to AURORA's UI design classification and heuristic navigation techniques.

I. INTRODUCTION

Mobile application development (or app development) is a challenging endeavor. Developers working in this domain face a variety of unique challenges that range from rapidly evolving and fault-prone APIs [1], [2] to frequent user feedback [3] and highly competitive app marketplaces [4]. As such, software maintenance and testing techniques, which are critical to ensuring software quality, are often overlooked due to pressure to deliver features in the face of these external factors [5]. As such, the research community has worked to provide a range of automated techniques to help developers cope with these challenges, spanning tools that support tasks from bug management [6]–[16] to software evolution [17], [18].

In the past decade, an extremely popular area of work in the software engineering research community has aimed to automate *software testing* for mobile apps – more specifically, *GUI-based testing*. Given that mobile apps are GUI-centric in nature, UI testing is one of the most popular testing modalities for ensuring the correctness of functionality. However, creating UI tests manually is extremely time-consuming [5]. As such, the research community has developed numerous Automated Input Generation (AIG) tools that dynamically explore applications with the goals of exercising substantial portions of app functionality, while simultaneously uncovering crashes and other faults. These tools can be broadly grouped into *random-based* [19]–[24], *model-based* [20], [22], [25]–[28], and *machine learning-based* tools [29]–[31].

In controlled experimental settings, AIG tools often perform well and achieve reasonably high code coverage. However, in practice, these AIG tools are often prone to low effectiveness in certain testing scenarios, particularly on sophisticated proprietary apps [32]. One reason for this low effectiveness is that many proprietary apps often contain complex screens that are difficult for AIG tools to navigate, i.e., the semantics of the screen require a precise order of actions to navigate and bypass so that additional app states can be explored. Wang et al. recently performed a study that empirically illustrated this phenomena, wherein they observed different types of screens that caused AIG tools to halt exploration progress [33]. The authors of this recent study refer to such screens as *UI Exploration Tarpits*. In addition to demonstrating this phenomenon, the authors also introduced a preliminary technique for dealing with these tarpit screens, called VET. VET integrates with existing AIG tools and uses a learn-from-mistake strategy that first identifies exploration tarpits from runs of the AIG tool and later disables these screens in future runs of the AIG tool.

This work was supported in part by Dragon Testing Technology and NSF grants CCF-1955853 and CCF-2132285. Jiangfan Shi is also with the Institute of Computing Innovation (Zhejiang University), Hangzhou, China.

979-8-3503-0818-1/24/$31.00 ©2024 IEEE
DOI 10.1109/ICST60714.2024.00028

The VET approach introduced by Wang et al. has two major limitations. First, VET is inherently expensive to run in practice, as it requires running an AIG tool twice, once to detect potential tarpits and then again to explore the app with the tarpits disabled. Second, VET does not assist with *navigating* UI tarpits, it simply disables them, meaning that there will always be portions of an application's state that AIG tools enhanced with VET cannot explore. However, navigating *through* UI tarpits may be feasible (as suggested by the results of the manual analysis performed by Wang et al. [33]), exploration tarpits often fall into one of a limited number of categories, even across different apps and AIG tools. This finding suggests that there may be patterns that can be exploited to explore these complex UI tarpit screens.

Given the findings and current limitations of prior work, in this paper, we propose a novel technique called AURORA, that aims to effectively *navigate* tarpit screens during app exploration using a new form of automated semantic screen understanding. The key idea of AURORA is that there are a finite number of UI designs for screens that are likely to represent tarpits, and hence, general *design motifs* that can be learned will allow for the automated recognition and navigation of such screens. AURORA learns from both the visual and textual patterns present in UI screens to automatically identify tarpit screens, through a component we call the *screen recognizer*, and then navigates recognized screens using a set of flexible heuristics, via the *heuristic navigator* component.

To better understand UI design categories and their relationship with exploration tarpits, we first studied Android app screenshots and UI hierarchies from the RICO dataset [30], deriving a set of 21 mobile app *UI Design Motifs*, representing coherent design patterns. During this process, multiple authors jointly labeled a minimum of 60 screens exhibiting each of our 21 design motifs, for a dataset totaling 1369 UI screens. We then proceeded to study the correlation between these general categories of UI designs and the tarpit screens as manifested through the dataset of tarpits discovered by and published alongside the VET tool [33]. We found that eight of our 21 design motifs were identified as tarpits in the VET dataset.

This analysis of the relationship between various categories of UI designs and tarpits inspired our design of AURORA. During the app exploration process of an AIG tool, AURORA is able to detect when app exploration progress is hindered by a tarpit, and will then automatically categorize the tarpit screen into one of our identified categories using a neural screen understanding approach that analyzes *both* the visual patterns in a screenshot of the tarpit and the textual patterns on the UI. For example, AURORA's screen recognizer can categorize a given screen as a login screen if the screenshot and UI contain a username and a password **`EditText`**, and contain salient visual patterns that indicate the typical structure of a login screen, such as center-aligned text-box(es) and button(s).

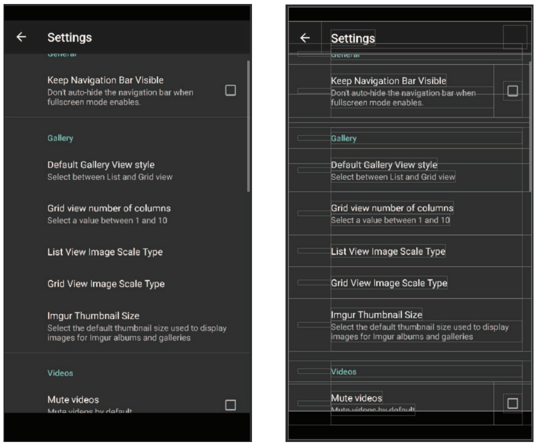
AURORA's screen recognizer is implemented using a multi-modal deep learning model [34], which is initialized through self-supervised learning on a set of 6000 app screenshots extracted from an online search engine. It is then trained and tested on an 80-20 split of our labeled dataset of 1369 UI images. Our evaluation finds that AURORA achieved 81.4% classification accuracy. AURORA's *heuristic navigator* implements eight input generation heuristics that aim to intelligently generate sequenced input scenarios to navigate through typical tarpit categories using a combination of neural text matching via transformer-based language models, and dynamic analysis.

AURORA is designed to run alongside and enhance existing AIG tools by quickly identifying and navigating through identified exploration tarpits. As an AIG tool is exploring, AURORA will periodically check if the exploration is stuck. If the AIG tool appears to be stuck, AURORA will analyze the current screen to determine if it is a tarpit. If it is, then AURORA will pause the AIG tool and activate a corresponding input generation heuristic to navigate the current screen. We combine AURORA with a state-of-the-art AIG tool, APE [21], and test on 17 popular Android apps. We find that AURORA helps improve code coverage substantially, with an average improvement of 11.0% over APE and 19.6% over VET on two separate comparative analyses. Through a qualitative analysis, we observe that these improvements arise due to the effectiveness of AURORA's screen recognizer and heuristic navigation strategies – the latter of which exhibits an 88.8% success rate in navigating through UI tarpit screens.

AURORA was developed in cooperation with Hangzhou Dragon Testing Technology Co., Ltd. who is focused on building AI-enhanced software testing products, and customers of the company include internationally recognized clients, such as WeChat, a messenger app with over one billion monthly active users. Dragon Testing has deployed a proprietary version of AURORA, that closely mirrors the components and workflow described in this paper, to its automated software testing product offerings. In this context, AURORA has enabled automated navigation of several types of tarpit screens for thousands of automated test cases, including pop-ups and forms, that had previously hindered the testing of app business logic for Dragon Testing's customers. This deployment of AURORA further illustrates both its effectiveness and practicality.

In summary, this paper makes the following contributions:

- **A Study** identifying prevalent *design motifs* of Android UI screens and the categories that constitute UI exploration tarpits.
- **A Multi-modal Deep Learning-based Approach** for classifying a given UI screen into the design motifs identified in our study.
- **Automated Heuristics** that can be used to navigate prevalent UI tarpits.
- **A Framework** implemented as AURORA [35], which can be combined with automated input generation (AIG) tools to automatically categorize screens and apply relevant heuristics to help AIG tools bypass UI tarpits.
- **An Evaluation** on the (1) accurateness of our deep learning model at predicting UI screen categories, (2) effectiveness of our heuristics at bypassing UI tarpits, and (3) code coverage improvements that AURORA can help AIG tools achieve.



Settings

Keep Navigation Bar Visible
Don't auto-hide the navigation bar when fullscreen mode enables.

Gallery

Default Gallery View style
Select between List and Grid view

Grid view number of columns
Select a value between 1 and 10

List View Image Scale Type

Grid View Image Scale Type

Imgur Thumbnail Size
Select the default thumbnail size used to display images for Imgur albums and galleries

Videos

Mute videos
Mute videos by default

(a) Imgur App Screenshot

(b) Visualization of Imgur App UI Hierarchy

Fig. 1: A visualization of the spatial properties of components captured within Android UI hierarchy of the Imgur [36] app.

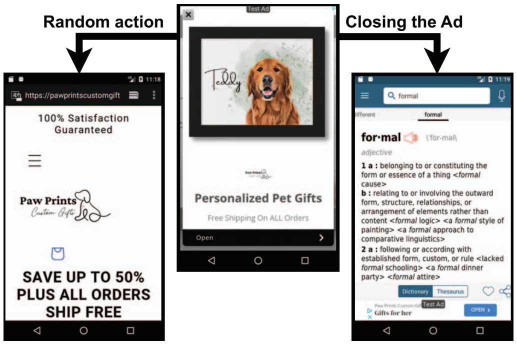
- **Artifacts** made publicly available from this work, which includes AURORA and our labeled dataset of UI categories, to help aid future research [35].

II. BACKGROUND

A. Mobile App UI Hierarchies and Frameworks

A *UI Hierarchy* represents the contents of an app UI rendered to the screen of a mobile device. UI hierarchies are comprised of UI elements that each exhibit several properties (e.g., location, size, component type), wherein UI elements are arranged in a hierarchical fashion and children can inherit design and logical properties from their parents. In Android, the **`ViewServer`** generates and maintains runtime information about app UIs, and can be queried using the **`uiautomator`** framework, which extracts a representation of the UI hierarchy in **`xml`** format. Figure 1 illustrates a visualization of the spatial properties of components captured as output in **`uiautomator xml`** files. Programmatically, Android screens are primarily made up of constructs called *Activities* and *Fragments*, where Activities typically represent a single logical screen and Fragments represent smaller components of a screen, such as a pop-up menu. For the purposes of discussion in this paper, when we refer to a "UI screen", we are effectively referring to a single Activity or Fragment that is rendered to the UI screen, such as the "Settings Activity" illustrated in Figure 1.

UI hierarchies effectively capture information that is relevant to the *structure* of a UI screen, which we will illustrate is important for learning abstract representations of UIs to aid in AURORA's screen classification capabilities. In addition to UI hierarchies, screenshots can be easily captured from mobile apps using the **`screencap`** utility built into the Android Debug bridge (**`adb`**) framework. This captured UI information is often used by AIG tools for decision-making and is especially critical to model-based testing tools. As we describe later, AURORA extracts runtime UI hierarchies and screenshots via the **`uiautomator`** and **`adb`** frameworks, which are fed as input to both AURORA's *screen recognizer* and *heuristic navigator*.



for·mal \ˈfȯr-məl\
adjective
1 a : belonging to or constituting the form or essence of a thing <*formal* cause>
b : relating to or involving the outward form, structure, relationships, or arrangement of elements rather than content <*formal* logic> <a *formal* style of painting> <a *formal* approach to comparative linguistics>
2 a : following or according with established form, custom, or rule <lacked *formal* schooling> <a *formal* dinner party> <*formal* attire>

Fig. 2: Example of an Advertisement screen tarpit.

B. UI Exploration Tarpits and the VET Approach

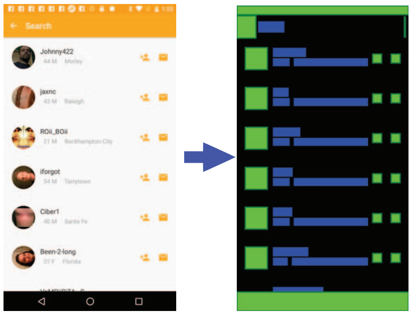
UI Exploration Tarpits (or UI Tarpits) is a term coined by Wang et al. [33], that describes a phenomenon that occurs when an AIG tool explores a single UI screen (i.e., *Activity*) for an excessive amount of time, hindering the progress of the tool in exploring undiscovered UI states of a given app. Figure 2 shows an example advertisement tarpit screen in the Merriam-Webster app. Random actions on these screens typically cause an AIG tool to get stuck – in this example, the advertisement launches a WebView that can only be closed through a specific series of actions.

To overcome issues with these screens, Wang et al. proposed the VET tool [33] to detect and cope with UI exploration tarpits. VET works in three stages: First, it runs the AIG tool on the target app without any restriction and records the testing process, yielding traces that record how the AIG tool has interacted with the target app. Second, it analyzes the collected traces against pre-defined low-level patterns that characterize repetitions to discover potential exploration tarpits. A ranking strategy is utilized to reduce false positives. Third, it reruns the AIG tool on the target app with the discovered exploration tarpits disabled, achieved through dynamically blocking the entry points to the exploration tarpits or via restarting the target app. Evaluation results show that VET can improve the testing effectiveness of multiple Android AIG tools, and can help improve bug detection capabilities.

Despite its advancements, VET has two major limiting factors. First, VET must first run an AIG tool *comprehensively* to identify and build a model of potential UI tarpits. This requirement means that testing time for apps is often *doubled* due to the need to run once to detect tarpits, and another time to explore the app with tarpits disabled. Second, VET does not facilitate *navigating* through tarpit screens, but instead, simply *disables* tarpit screens. This limitation means that VET is effectively incapable of exploring certain areas of an app that were identified as tarpits. AURORA aims to overcome both of these shortcomings by analyzing UI screens in real-time to detect, classify, and navigate through tarpit screens using new forms of automated neural UI screen understanding.

III. DERIVING MOBILE APP UI DESIGN MOTIFS

One of the key ideas underlying AURORA is that certain designs for mobile app UI screens are *reused*, with varying



(a) Original App screenshot (b) Screenshot "Silhouette"

Fig. 3: A visualization of a UI Silhouette Screen.

degrees of change and variability, across applications such that the structural and lexical patterns of these designs can be automatically identified. In this paper, we refer to these patterns as *UI Design Motifs*. To better understand the types and prevalence of UI Design Motifs across Android applications, we conducted a preliminary investigation to derive a set of motifs that will serve as the focus of our AURORA approach.

A. Isolating Structural Screen Patterns with Silhouette Screens

To empirically derive a set of UI design motifs, we analyzed a set of randomly sampled screens from the RICO dataset [30]. RICO is currently the largest dataset of Android app UI screenshots and corresponding runtime UI metadata, spanning over 66,000 UI screens collected from over 9,000 free Android apps available on the Google Play Store. This screen information was collected via a combination of automated UI exploration and crowdsourced UI exploration on virtual Android devices. The UI metadata for these screens was collected using the **`uiautomator`** framework.

Given the sheer scale of this dataset, manually analyzing even small portions of the dataset to discover UI design motifs would be a time-consuming proposition. Furthermore, as observed by the authors of the RICO dataset, there likely exist certain common UI patterns followed by a long tail of unique or one-off UI screen designs. Given that our aim is to identify and categorize common UI design patterns into categories we term as *motifs*, we introduced lightweight automation to facilitate the manual labeling process. As such, we conducted an initial step of unsupervised, computer vision-based clustering of screens into broad categories that exhibited visual similarities.

Using "raw" screenshots to group together visually similar UI screens for the purposes of deriving design motifs presents certain challenges. First, two screens that share a design motif (i.e., Settings Screen) may *instantiate* that screen using a similar screen *structure*, but have widely varying colors, fonts, and other stylistic properties. As such, clustering screens according to raw image similarities is likely to be greatly impacted by similarities in style. To focus our analysis on *structural* UI Design Motifs that occur across a diverse set of apps, we developed a process to "abstract" raw UI screenshots, by stripping out stylistic information and creating what we refer to as *Silhouette Screens*. Figure 3 illustrates this process. In essence, we divide all UI components into one of two categories: (i) textual components, and (ii) non-textual components. We then draw textual components on a black canvas as blue boxes and non-textual components as green boxes, parsing the spatial component size and locations from RICO's **`uiautomator`** metadata that accompanies each screenshot. This methodology allows us to effectively capture each screen's abstract structure, while ignoring the stylistic variations different screens may exhibit. We discuss different potential variations of Silhouette Screen creation in Section IV.

B. Structural Clustering of UI Screens

After developing our process for creating Silhouette Screens, we needed a reliable methodology to create a robust representation of our UI screens so that they could be clustered according to their structural UI features. While simple image similarity measures could be used to accomplish this goal, such measures often rely on handcrafted heuristics that may not transfer well to UI screens. Furthermore, deep learning-based computer vision techniques that incorporate convolutional neural networks have been shown adept at learning robust representations of image features [37], [38]. As such, we implemented a 2D convolutional autoencoder, consisting of 6 encoder and 7 decoder layers, and trained it on 32,338 UI screens collected by randomly sampling from 50% of the RICO dataset. Generally speaking, an autoencoder is a neural network that encodes an image into a high-dimensional vector representation and then is trained to decode this representation back into the original image, with differences between the input and output being used by a loss function to update model weights during the training process. We trained our autoencoder model on a subset of RICO to avoid overfitting and to improve the model's generalization to handle a wider range of real-world data. Through experimentation, we also found that this amount of data was sufficient to train our autoencoder to convergence.

We then applied a K-means clustering technique to 645 screens, roughly representing $\approx$1% of the RICO dataset sampled from outside the autoencoder's training set. Using the elbow technique [39], we found 30 clusters to be optimal.

C. Manual Analysis and Derivation of Design Motifs

After this clustering procedure, three authors of this paper manually analyzed each of the 30 clusters, refined the categorizations of screens, and then collectively provided labels to the finalized clusters. This process proceeded as follows: First, separate authors would look at a defined set of clusters (i.e., five clusters), and they would examine each screen in these clusters and re-cluster them or form new clusters to better group screens that share a common structural UI pattern. Two authors would examine these sets of clusters, and then all three authors would meet to discuss the results and come to an agreement on the newly formed clusters. This process proceeded until all 30 clusters had been examined. The end result of this process was a refined set of 21 clusters, representing UI *design motifs* with the following

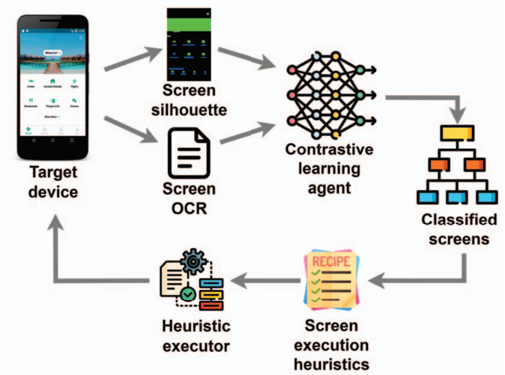


Fig. 4: Overview of AURORA's workflow.

labels: *Advertisement*, *Calendar time and weather*, *Catalog*, *Feed*, *Form*, *Home menu*, *List*, *Log-in*, *Map*, *Onboarding*, *Player*, *Pop up*, *Product*, *Search*, *Settings*, *Splash*, *Terms and conditions*, *Travel booking*, *Type message*, *Viewer*, and *Web browser* screens. We provide complete descriptions and examples of these categorizations in our artifact [35].

D. Identifying Tarpit Screens and Motifs from the VET Dataset

The study conducted by Wang et al. [33] contains a dataset of over 95,000 UI screens obtained from testing 16 different industrial apps using different AIG tools, including APE [21], Monkey [20], and WCTester [40]. These UI screens are organized into 127 test traces, with each trace comprising a collection of screenshots and associated metadata derived from a specific app-tool pairing. These traces include timing information as well as the number of actions executed on various UI screens, allowing us to identify screens for which the tools encounter obstacles that may represent tarpits.

To find examples of tarpit UI screens, we perform a UI metadata comparison using the **`JSONComparison API`** [41] on the **`uiautomator`** metadata included with each screenshot from the VET traces. We set a minimum threshold of five action repetitions *and* 10 seconds of elapsed time to consider a screen as a tarpit in the context of the traces. Said differently, we only consider a screen to be a tarpit if at least five consecutive actions were entered on the screen and the screen did not change for at least 10 seconds. Applying these criteria to the screens from the traces, we extract 238 screens that likely represent UI tarpits. Upon further inspection of this dataset, we observed that there were screens with fewer actions but very large elapsed time. Therefore, we additionally include the top 200 screens on which the tools used in VET's evaluation spent the most time. In the end, we take the union of these two sets which in total contains 348 tarpit screens.

E. Tarpit Screen Analysis

We aimed to better understand two aspects of the tarpit screens from the VET dataset including: (i) reasons for obstructed AIG tool navigation, and (ii) the labels of these screens according to the design motifs we derived earlier. To carry out this process, two authors of the paper independently analyzed each of the 348 tarpit screens and investigated the two factors above. We present these findings below.

Reasons for AIG Tool Obstructions:

1) Inability to input relevant text in a designated field.
2) Difficulty locating specific components or series of components for screen progression.
3) Failure to identify interactive elements on the screen.
4) Accidental engagement with advertisements, resulting in difficulty exiting the web view screen.
5) Tool initiated closure of the application.
6) App unresponsiveness.

Tarpit Screen Design Motifs and Navigation Strategies:

- **Log-in:** Input some predefined text and tap certain buttons.
- **Onboarding:** Identify and interact with certain buttons.
- **Player:** Perform sequential actions to access different screens.
- **Advertisement:** Locate and tap the close button to exit.
- **Viewer:** Tap on the screen to expose interactive elements or use the back button for navigation.
- **Form:** Input relevant text, interact with spinners and buttons.
- **Web browser:** Tap the back button or restart the app.
- **Search:** Input relevant text in search fields.

IV. THE AURORA APPROACH

In this section, we define the methodology of our Android UI exploration tool, AURORA [35]. Our approach leverages a multi-modal computer vision-based *screen recognizer* that uses structural and lexical patterns in UI screens to detect different UI Design Motifs. When an AIG Tool encounters a tarpit screen, AURORA classifies this screen using the screen recognizer and then applies one of several pre-defined *navigation heuristics* to navigate through the screen to uncover additional states of the application. An overview of the approach is shown in Figure 4. We describe the components of AURORA in detail in the following subsections.

A. Approach Workflow

AURORA [35] operates in conjunction with any AIG Tool that extracts **`uiautomator`** metadata and screenshots as part of its exploration process. The integrated AIG tool can be random-based, model-based, or machine learning-based in nature. During an AIG tool's app exploration, we need a mechanism to detect when the tool may be "stuck" in a tarpit screen. We derived a suitable elapsed time to determine whether a screen is a tarpit empirically through a small set of experiments where we varied the "tarpit trigger" time from 10 seconds to 30 seconds in 5-second intervals, and inspected the number of identified tarpits during a one-hour execution of the APE [21] automated testing tool on the set of 16 apps from the VET dataset. Upon inspection of the identified screens at each tarpit threshold level, we found that 10 seconds allowed AURORA to detect the highest number of true tarpit screens with a reasonably small number of false positives. As such, AURORA polls the activity/window combination queried from an Android emulator's view-server every second — if the activity/window combination remains the same after 10 seconds AURORA is triggered to bypass the potential tarpit.

Once a tarpit screen has been identified, a screenshot and the corresponding **`uiautomator`** metadata for the screen are saved, and then a Silhouette Screen is created to capture the structural patterns, and the EAST Optical Character Recognition (OCR) technique [42] from the Google Cloud Vision API [43] is used to extract text from the screen. The **`uiautomator`** metadata, Silhouette Screen, and OCR data are then converted to visual and textual embeddings and passed to AURORA's *screen recognizer*. This component generates a ranked list of UI Design Motifs for the current screen.

Given the ranked set of potential UI Design Motifs for a given screen, AURORA checks if the screen falls into one of the eight Tarpit UI Design Motifs derived in Section III-E. If so, then AURORA triggers the execution of a *navigation heuristic* for that Design Motif, which performs a predefined set of UI actions. To carry out the actions of a heuristic, AURORA uses the SentenceBERT model [44] to determine where to input predefined text and where to click. It is possible that a given tarpit screen could be mis-classified or that a given heuristic might fail. In these scenarios, if AURORA recognizes that it has not navigated through a given UI screen after the execution of a heuristic, it then executes heuristics of the next *two* tarpit categories from the ranked list of predicted UI Design Motifs. Given that AURORA's heuristics generate fewer actions per given unit of time than many AIG Tools, it is necessary to limit the number of heuristics executed on a given tarpit screen to small, reasonable number (three). If no progress is made after three heuristic execution attempts, the app is restarted.

B. UI Screen Recognizer

As part of our development of AURORA, we worked with Dragon Testing to develop two different UI screen classifiers. The first classifier combines image embeddings learned from a neural autoencoder, with lexical embeddings of screen text from a large language model. These embeddings are concatenated and then passed into a classifier that predicts a ranked list of screens. The second classifier uses a multi-modal CLiP [34] model to encode images and text. We describe each of these classifiers below. We explore these two classifiers as they require very different numbers of parameters, with the CLiP-based approach requiring a much larger number of parameters than the Autoencoder-based approach. Given that past research has suggested the exploration of machine learning techniques of varying degrees of complexity when applied to software data [45], we opted to explore both a "simpler" and "more advanced" machine learning technique. We present the results of an empirical comparison of these two classifiers in Section V.

1) Autoencoder-based UI Design Motif Classifier: The first classifier that we constructed for AURORA uses both a visual and textual classifier before combining the output of these two techniques to produce a final ranked list of categorized UI Design motifs for a given target screen.

For the visual classifier, we utilize the encoder portion of our pre-trained autoencoder framework described in Section III-B. The encoder transforms high-dimensional image data into compact feature vectors. These feature vectors contain essential information from the images while reducing their dimensionality. Leveraging these feature vectors, we applied traditional machine learning models, such as Random Forest, Multi-Layer Perceptron (MLP), and Naive Bayes to conduct classification tasks on a set of labeled UI screens. In our experiments, we found the Random-forest classifier to achieve the highest accuracy. To derive a training set for this approach, we had two authors label an additional 599 screenshots from the RICO dataset into our 12 Design Motif categories, distributed as evenly as possible.

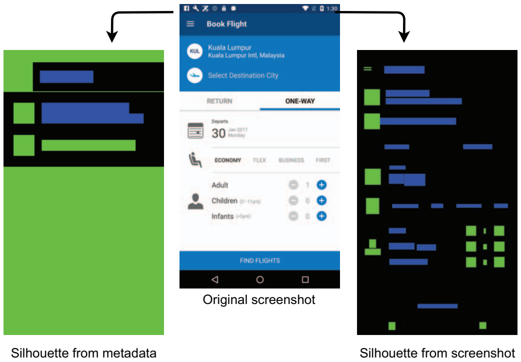


Fig. 5: Comparing Silhouette Screens - blue color represents text and green color represents non-text components.

For the textual classifier, we use both the UI metadata and the OCR output from the UI screenshots and create a single paragraph out of each UI screen. To encode the text from the UI metadata, we convert six different component attributes for each component in the UI hierarchy into sentences. The attributes that AURORA encodes are (i) component class, (ii) ancestor class, (iii) label, (iv) position, (v) width, and (vi) height. In cases where *labels*, which typically represent text rendered to the screen, are missing in the UI metadata, we utilize OCR information using Google's Tesseract OCR engine [46] as an alternative source of data. We applied various classifiers, including Naive Bayes, KNN, Random Forest, and Multi-Layer Perceptron (MLP), in conjunction with a TF-IDF vectorizer. The TF-IDF vectorizer combined with the MLP classifier produced the highest accuracy. We utilized the same training set of 599 screenshots for this textual classifier.

Finally, AURORA combines the output of these two classifiers together. It uses a combined probabilistic approach to predict screen class by extracting 21 attributes from both the visual and textual classifiers, each denoting a probability value for a specific class. Subsequently, we utilize a Random Forest classifier for prediction.

2) Transformer-based UI Design Motif Classifier: AURORA's transformer-based approach uses CLiP (Contrastive Language-Image Pre-Training) [34], a model known for its ability to perform transfer learning on a wide array of classification-based downstream tasks. The CLiP model works by creating two separate embeddings for the image and the provided text. It creates image embedding using a convolutional neural network (CNN) based on the ResNet architecture. For text embedding, it uses a transformer-based neural network architecture similar to the GPT language model [47].

Additionally, for this classifier, we improved upon the concept of Silhouette Screens we introduced earlier in the paper. That is, our prior technique for deriving Silhouette Screens relied *solely* on **`uiautomator`** metadata to create the Silhouette Screen. However, as illustrated in Figure 5, there may be certain textual elements that are not properly captured in the UI metadata – for instance if the text is displayed through an image or web view. As such, we use the Google Cloud OCR Engine to detect text on the screen, relate it to the spatial properties of components, and create more accurate UI Silhouette Screens, as shown in the right-hand side of Figure 5.

To train the CLiP model, we conduct unsupervised pre-training on a set of 6,000 UI screenshots that we crawl from Google. These screens were collected by searching the terms of each of our identified design motifs in conjunction with the terms "mobile app screen" using Google image search, and downloading the resulting UI screenshots until we had 6000 screens distributed across our design motif categories. We then train and test CLiP using an 80-20 split of our new set of 1369 RICO images (expanding from the set of 599 used to train the Autoencoder-based model) evenly distributed across our 21 Design Motif categories.

To encode the textual information, we use the CliP model to extract textual embeddings from all the text displayed on a given UI screen as extracted by Google's Cloud OCR engine.

C. Heuristics Design

Our heuristics are formulated through the analysis of tarpit screens within the VET dataset. Considering the reasons for getting stuck at the end of Section III-E, we created heuristics in executable Python code. The heuristic code makes use of the Python wrapper for Android Debug Bridge commands [48] for sending commands to a virtual device on an Android emulator. While looking for specific components or input fields, it uses the SentenceBERT model [44] to match with the closest on-screen component. To validate the functionality and reliability of our executable heuristic code, we tested it rigorously on three Android applications, ensuring that they can execute without any operational issues or errors. AURORA focuses on eight specific types of UI screens, as detailed in Section III-E, for which it has developed specific heuristics that are generalizable across various applications. We provide a description of two of AURORA's heuristics below as examples, and refer readers to our artifact for additional examples [35].

Form screen: Form screens usually contain multiple text fields, spinner components, and one submit button. Random-based tools often cannot go past these screens, as such screens require relevant text input. AURORA can provide the necessary knowledge for entering relevant input using its preset spreadsheet values. The values represent predefined column headers and associated data, which is static during application testing. We use a SentenceBERT model [44] to match input fields on UI screens with our preset spreadsheet values and input the top match to the UI screen. This model is essential for matching on-screen labels (e.g., "last name") with the relevant spreadsheet column names (e.g., "surname"), which we then take a value from. The SentenceBERT model can resolve nuanced differences by capturing the semantic relationships between the labels and the spreadsheet column names.

TABLE I: Apps used for our evaluation. #DLs represents the approximate number of downloads.

App name	Version	Category	#DLs
AccuWeather	7.4.1	Weather	50m+
AllTrails	14.2.0	Travel & Local	10m+
AutoScout24	9.8.0	Auto & Vehicles	10m+
CarMax	2.56.1	Auto & Vehicles	5m+
Duolingo	3.75.1	Education	100m+
Flipboard	4.1.1	News & Magazines	500m+
Fox News	4.50.0	News & Magazines	10m+
KAYAK	176.2	Travel & Local	10m+
Merriam-Webster	4.1.2	Books & Reference	10m+
Spotify	8.4.48	Music & Audio	100m+
TripAdvisor	25.6.1	Travel & Local	100m+
trivago	4.9.4	Travel & Local	10m+
Walmart	22.31	Shopping	50m+
Wattpad	6.82.0	Books & Reference	100m+
WEBTOON	2.4.3	Comics	10m+
wish	4.16.5	Shopping	100m+
YouTube	17.33.42	Video Player & Editor	1b+

Player screen: Player screens usually have a play and resume button and other smaller buttons. AIG tools can get stuck in these screens, as the probability of hitting bigger buttons (e.g., play and resume) are higher, which does not necessarily result in a screen change. AURORA searches for other buttons, such as the settings or share button using its pre-defined heuristics and interacts with them. In this way, it helps with moving on to a different screen, so AIG tools can explore other parts of the application.

V. EVALUATION

To understand how well AURORA can explore a given app, we ask the following research questions (RQs):

1) How well do AURORA's Screen Classifiers function in relation to a baseline?
2) How often do automated input generation tools get stuck, and what kind of screens are more difficult to explore?
3) How much improvement does AURORA offer over APE, Monkey, and VET regarding method coverage?
4) How often do the heuristics get executed successfully, and how many additional methods can they cover?
5) How effective are the heuristics in navigating the intended tarpit screens?

A. Evaluation Context

Datasets and Baselines: We evaluate the two AURORA classifiers using the 1369 labeled RICO images derived as part of our UI Design Motif study. We compare our classifiers against Screen2Vec [49], which is a textual screen embedding technique that can be used to classify Android screens using a neural representation of UI metadata. We evaluate the performance of APE, Monkey, VET, and AURORA on an emulator operating within the Android 6.0 environment. We run AURORA in conjunction with APE as the exploration tool due to its superior method coverage rate observed during the VET experimentation conducted by Wang et al. [33]. Additionally, APE has demonstrated a remarkable capacity to attain higher test coverage [21] compared to alternative tools, such as Monkey [20] or STOAT [50]. Our experimental analysis focuses on a carefully selected set of 12 apps derived from the VET experiments. To ensure the validity of our findings, we exclude 4 apps from the previous study due to their lack of support on Android 6.0, which could potentially introduce inconsistencies in the results. Additionally, we expand our investigation to encompass five additional apps beyond the scope of the original VET experiments, resulting in a total of 17 apps under examination, one more than the number of apps assessed in the VET experiments [33].

The inclusion of the five additional apps is meant to help assess the generalizability of our heuristic-based approach. The additional five applications are AllTrails, CarMax, Fox News, KAYAK, and Walmart. These supplementary apps are among the most popular apps with over five million downloads each. They are incorporated into the study to examine the broader applicability and generalizability of our proposed method. We also updated two apps (AutoScout24 and YouTube) to a newer version than the one used in the VET paper because almost all functionalities of those apps were disabled in the older versions at the time of our experiments. Table I shows the statistics of the apps we used for our evaluation.

Experimental Procedure: Our experiment starts by executing Monkey and APE on 17 pre-selected apps. We run a single instance of the emulator at a time to collect the app traces. The emulator is allocated 2 GB of RAM and 2 GB of internal storage space. To ensure the emulator remains responsive and efficient, we avoid running more than three 1-hour traces simultaneously. We conduct three 1-hour runs of Monkey and APE. VET learns from its built-in tarpit identification process from the APE runs and then adds three more 1-hour runs, giving us a total of six 1-hour runs. AURORA is executed for six 1-hour runs for each app. We compare the first three 1-hour runs of AURORA with APE and Monkey and compare the total six 1-hour runs with VET. Due to experiment costs, we do not conduct six 1-hour runs for Monkey and APE, therefore we present their comparison in a separate table. After each 1-hour run, our automated script restarts and wipes the emulator's data, preventing the emulator memory from filling up due to the screenshots taken during AURORA's runtime.

Metrics: For **RQ**$_1$, we use the classic definitions of Precision, Recall, F1-score, and Accuracy for multi-class classification problems. For **RQ**$_2$, we consider a screen to be a tarpit if a tool gets stuck on the screen for more than 10 seconds, which we felt appropriate given the high number of actions tools like APE can generate. For **RQ**$_3$, we use MiniTrace [51] to calculate the method coverage of our selected industrial apps. MiniTrace collects method coverage using the Android runtime and does not require app instrumentation. As MiniTrace requires Android 6.0 to run, we employ this Android version for our evaluation. We measure coverage by calculating the union of the method coverage over the set of three runs for each tool, which we refer to as "set-union" method coverage. We also calculate the area under the curve for our method coverage. This measurement tells us how soon a tool can achieve more coverage. The formula we use is AUC $= \sum_{i=1}^{n} \frac{1}{2}\left(R_{i-1} + R_i\right) \cdot \Delta t$, where n is the total number test runs, R_i is the method coverage at hour i, and Δt represents the time interval for each one-hour test run.

TABLE II: Performance of AURORA's Motif Classifiers.

	Precision	**Recall**	**F1-Score**	**Accuracy**
RICO	0.717	0.686	0.690	0.689
Extended	0.830	0.812	0.809	0.813

TABLE III: Top 10 UI categories identified by AURORA in real-time and their associated bypassing rates. Bolded rows represent tarpit categories.

	APE stopped	% passed by AURORA
Search screen	633	93.7%
Settings screen	523	77.8%
Viewer screen	479	97.1%
Home menu screen	473	78.9%
Onboarding screen	444	90.3%
Pop up menu	418	99.5%
Web browser	357	100%
Catalog screen	342	85.4%
Player screen	312	81.7%
Log-in screen	294	90.1%
Average	275.1	88.8%

B. RQ1: Accuracy of Screen Classifiers?

As described in Section IV-B, AURORA was evaluated with Autoencoder-based and CLiP-based models. We find that AURORA's Autoencoder-based model achieved ≈60% accuracy, whereas Screen2Vec, our baseline, was able to achieve an overall accuracy of only ≈38%. We get an even more sizeable increase in accuracy over our baseline with AURORA's CLiP-based models. Table II illustrates the classification effectiveness of AURORA's CLiP-based models on the 1369 labeled RICO images. The RICO variant is the performance without the unsupervised pre-training on the screens collected from Google, whereas the Extended model does include this process. Our results show that pre-training achieves an 81.3% accuracy compared to the other model's 68.9% accuracy.

C. RQ2: How often AIG Tools Get Stuck

Table III shows the top 10 categories of UI screens considering the number of halts faced during the entire experimentation run. Search screens represent the most prevalent tarpit with 633 halts, but AURORA managed to navigate through 93.7% of these screens using its heuristic-based approach. AURORA has an average of 88.8% passing rate across all tarpit screens.

Previously, in our analysis of the VET dataset, we have identified certain categories that demonstrate characteristics akin to tarpits. These categories are denoted with bolded font in Table III. Notably, among the 8 UI classes previously identified as tarpit screens, 6 of them prominently feature within the top 10 UI screen categories. This observation highlights the high potential for AURORA to improve AIG tools as tarpit category screens frequently occur.

One tarpit category not in the top 10 is "Advertisements", which exhibited a relatively lower frequency in our experiments. This finding is reasonable, considering our experiment

TABLE IV: Set union method coverage comparison. Bolded cells represent the highest coverage for an app across all tools.

	Monkey	APE		AURORA		APE1-AURORA2	
App	Coverage	Coverage	% inc	Coverage	%inc	Coverage	%inc
AccuWeather	16711	21264	27.2%	22641	35.5%	**22714**	35.9%
AllTrails	28691	43132	50.3%	**67231**	134.3%	59548	107.5%
AutoScout24	29763	**40857**	37.3%	38554	29.5%	39136	31.5%
CarMax	11002	11619	5.6%	17260	56.9%	**17452**	58.6%
Duolingo	**15328**	14355	-6.3%	14805	-3.4%	14993	-2.2%
Flipboard	8652	10646	23.0%	13345	54.2%	**13569**	56.8%
Fox News	27705	29924	8.0%	30574	10.4%	**31375**	13.2%
KAYAK	44593	55688	24.9%	57555	29.1%	**59327**	33.0%
Merriam-Webster	7668	8621	12.4%	9112	18.8%	**9175**	19.7%
Spotify	12510	19533	56.1%	**28552**	128.2%	27071	116.4%
TripAdvisor	23390	**30548**	30.6%	27728	18.5%	30047	28.5%
trivago	19296	20096	4.1%	**20393**	5.7%	20343	5.4%
Walmart	27322	40435	48.0%	44041	61.2%	**51149**	87.2%
Wattpad	13324	23426	75.8%	23648	77.5%	**24690**	85.3%
WEBTOON	19310	27628	43.1%	22819	18.2%	**27750**	43.7%
wish	7544	9175	21.6%	**9192**	21.8%	8450	12.0%
YouTube	32428	**38372**	18.3%	34738	7.1%	36030	11.1%
Average			**28.2%**		**41.4%**		**43.8%**

uses an older version of Android, that may no longer support certain ads for the applications [52] we evaluated. Similarly, "Form" screens are not in the top 10 due to some apps no longer allowing sign-ups on older Android versions.

D. RQ3: Method Coverage Improvement?

To evaluate the effectivness of AURORA, we compare the total number of unique methods covered over its three 1-hour runs to those of Monkey and APE. From Table IV, we can see that AURORA gets an average of 41.4% increase in coverage compared to Monkey. APE, on the other hand, gets a 28.2% increase. If we combine a 1-hour APE run with 2-hour AURORA runs (denoted as APE1-AURORA2), we get the best performance, a 43.8% increase from Monkey's method coverage. We also performed an experiment with two hours of APE runs combined with one hour of AURORA. However, the results were worse than just AURORA or the shown combination. The reason for the difference in improvement is likely due to the fact that APE1-AURORA2 best harnesses the strengths of each technique. That is, APE is able to exercise a large number of actions in a shorter period of time, whereas AURORA can more effectively explore tarpits, but benefits from the extra time budget to do so – due to its online classification and heuristic execution.

We can see that Monkey performs better than all of the other tools for the Duolingo app. In this app, the screens typically only require taps and the UI components cover large areas of the screen. As Monkey works by generating random events like taps or gestures without considering UI layouts/metadata, it has a higher action per second rate than APE or AURORA. While Monkey often suffers from empty space tap issues on other apps, it does not suffer this issue for Duolingo given its large components, and the high action rate leads to higher coverage. All in all, AURORA gets higher than APE in set union coverage for 13 out of 17 apps, while APE1-AURORA2 also gets higher coverage than APE for 13 apps.

Considering AURORA vs APE1-AURORA2, we see that the latter is clearly ahead in set union method coverage and area under the curve. This result indicates that, for a 3-hour run, AURORA should be combined with APE to get the best possible coverage.

TABLE V: Set union method coverage of VET vs. AURORA. Bolded cells represent the highest coverage between the tools.

App	VET	AU	% inc	Comm.	V ex (%)	AU ex (%)
AccuWeather	23456	**28929**	23.3%	22105	1351 (4.5%)	6824 (22.5%)
alltrails	43765	**68829**	57.3%	43256	509 (0.7%)	25573 (36.9%)
AutoScout24	41258	**42953**	4.1%	38478	2780 (6.1%)	4475 (9.8%)
CarMax	12331	**19725**	60.0%	11876	455 (2.3%)	7849 (38.9%)
Duolingo	14704	**15628**	6.3%	14291	413 (2.6%)	1337 (8.3%)
Flipboard	11754	**14705**	25.1%	10830	924 (5.9%)	3875 (24.8%)
Fox News	31140	**31586**	1.4%	29601	1539 (4.6%)	1985 (6.0%)
KAYAK	57641	**77567**	34.6%	55103	2538 (3.2%)	22464 (28.0%)
Merriam-Web	**10547**	9734	-7.7%	9328	1219 (11.1%)	406 (3.7%)
Spotify	19918	**32111**	61.2%	19555	363 (1.1%)	12556 (38.7%)
TripAdvisor	32014	**32200**	0.6%	29973	2041 (6.0%)	2227 (6.5%)
trivago	20265	**20944**	3.4%	20032	233 (1.1%)	912 (4.3%)
Walmart	43334	**46232**	6.7%	35194	8140 (15.0%)	11038 (20.3%)
Wattpad	24053	**33192**	38.0%	23176	877 (2.6%)	10016 (29.4%)
WEBTOON	28059	**31789**	13.3%	20940	7119 (18.3%)	10849 (27.9%)
wish	9923	**10305**	3.8%	8178	1745 (14.5%)	2127 (17.7%)
YouTube	41518	**42466**	2.3%	37993	3525 (7.7%)	4473 (9.7%)
Average			**19.6%**	25288.8	2104.2 (**6.3%**)	7587.4 (**19.6%**)

TABLE VI: Heuristics success rate across all apps.

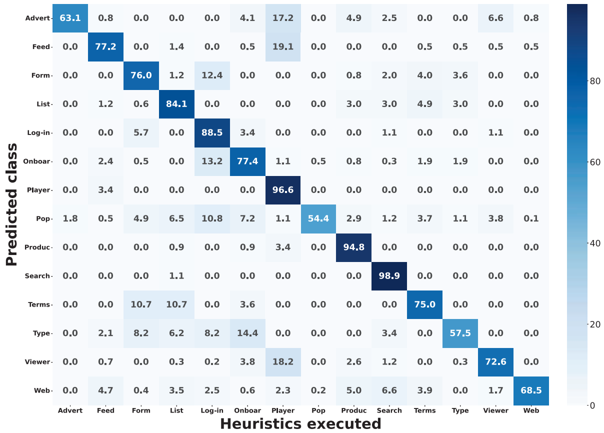
App	Passed	Failed	Total	% Pass
AccuWeather	421	144	565	74.5%
AllTrails	308	35	343	89.8%
AutoScout24	420	36	456	92.1%
CarMax	275	18	293	93.9%
wish	242	21	263	92.0%
Duolingo	364	31	395	92.2%
Fox News	279	12	291	95.9%
YouTube	532	5	537	99.1%
KAYAK	251	39	290	86.6%
Merriam-Webster	438	31	469	93.4%
WEBTOON	219	68	287	76.3%
Spotify	278	40	318	87.4%
TripAdvisor	177	49	226	78.3%
trivago	330	53	383	86.2%
Walmart	320	32	352	90.9%
Flipboard	346	49	395	87.6%
Wattpad	301	30	331	90.9%
Total	5501	693	6194	**88.8%**

To run VET, we must first run three hours of APE, and then, learning from the actions that end up in a stuck region, VET prevents them from happening in its additional 3-hour run. To make a fair comparison, we run AURORA for 6 hours. Table V compares the set union methods and exclusive methods for VET and AURORA. Considering coverage, AURORA gets an average of 19.6% increase compared to VET. We can also see that AURORA gets higher coverage for 16 out of 17 apps. Considering orthogonality, AURORA provides an average of 19.6% exclusive methods compared to 6.3% from VET.

E. RQ4: Successful Heuristic Execution?

Table VI shows the success rate of our heuristics. Ranging from 74.5% to 99.1% with an average of 88.8% of the executed heuristics being successful. We consider our heuristics successful when any of the heuristics from AURORA's top 3 predictions are successful in changing the app screen.

If we compare actions per second, using AURORA will generate fewer actions than not using AURORA, as it needs to classify screens and run heuristics during runtime. However, even with less actions generated, AURORA still achieves an improvement in code coverage. This result suggests that at tarpit screens, a properly curated heuristic is often better than randomly clicking around to increase code coverage.



	Advert	Feed	Form	List	Log-in	Onboar	Player	Pop	Produc	Search	Terms	Type	Viewer	Web
Advert	63.1	0.8	0.0	0.0	0.0	4.1	17.2	0.0	4.9	2.5	0.0	0.0	6.6	0.8
Feed	0.0	77.2	0.0	1.4	0.0	0.5	19.1	0.0	0.0	0.0	0.5	0.5	0.5	0.5
Form	0.0	0.0	76.0	1.2	12.4	0.0	0.0	0.0	0.8	2.0	4.0	3.6	0.0	0.0
List	0.0	1.2	0.6	84.1	0.0	0.0	0.0	0.0	3.0	3.0	4.9	3.0	0.0	0.0
Log-in	0.0	0.0	5.7	0.0	88.5	3.4	0.0	0.0	0.0	1.1	0.0	0.0	1.1	0.0
Onboar	0.0	2.4	0.5	0.0	13.2	77.4	1.1	0.5	0.8	0.3	1.9	1.9	0.0	0.0
Player	0.0	3.4	0.0	0.0	0.0	0.0	96.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Pop	1.8	0.5	4.9	6.5	10.8	7.2	1.1	54.4	2.9	1.2	3.7	1.1	3.8	0.1
Produc	0.0	0.0	0.0	0.9	0.0	0.9	3.4	0.0	94.8	0.0	0.0	0.0	0.0	0.0
Search	0.0	0.0	0.0	1.1	0.0	0.0	0.0	0.0	0.0	98.9	0.0	0.0	0.0	0.0
Terms	0.0	0.0	10.7	10.7	0.0	3.6	0.0	0.0	0.0	0.0	75.0	0.0	0.0	0.0
Type	0.0	2.1	8.2	6.2	8.2	14.4	0.0	0.0	0.0	3.4	0.0	57.5	0.0	0.0
Viewer	0.0	0.7	0.0	0.3	0.2	3.8	18.2	0.0	2.6	1.2	0.0	0.3	72.6	0.0
Web	0.0	4.7	0.4	3.5	2.5	0.6	2.3	0.2	5.0	6.6	3.9	0.0	1.7	68.5

Fig. 6: Confusion matrix of heuristics succeeding.

AURORA would likely offer similar improvements to systematic testing approaches, as they may also get "stuck" on screens. However, the VET dataset, and identified tarpits, were derived using random-based techniques, thus we oriented our analysis toward these techniques as well. Future work should explore AURORA 's effect on other types of techniques.

F. RQ5: Effectiveness of Individual Heuristics

The 88.8% success rate of heuristics shows us the collective effectiveness of using a heuristic-based approach. To understand the performance of each of these heuristics to effectively navigate their respective UI tarpit screens, we conducted an analysis using AURORA's post-run logs.

Our heuristic based approach works by iterating through the top three predictions of a tarpit screen. AURORA runs the heuristic designed for the initial predicted UI category, and when it does not result in a change in the tarpit screen, it proceeds to the subsequent predictions in a sequential manner. Figure 6 shows the frequency with which various heuristics successfully made changes to a predicted screen, presented in a percentage format. The diagonal values refer to the heuristic affecting changes being the one intended for the first prediction. As the figure suggests, we find that AURORA was able to navigate all the predicted screens using their intended heuristics most of the time. Outside the diagonals, we can see that player heuristics also successfully navigated a handful of advertisement, feed, and viewer screens. This result is due to player heuristic's ability to find on-screen components and resume random exploration on a different app screen.

VI. Threats to Validity

Our initial study on design motifs involved manual effort in classifying screens, identifying UI tarpits, and finding ways to overcome them. Any manual process can include biases. We limit the potential for bias by examining only a portion of tarpit screens during heuristic construction and using semantic text matching to make our heuristics generalizable.

Another threat to our work's external validity is that we use only Android 6.0. We utilize the MiniTrace mechanism for collecting method coverage without needing code instrumentation, and MiniTrace works with only Android 6.0.

VII. Related Work

Studies on Android Testing: Vásquez et al. [53] compiled a body of knowledge that can help researchers focus on new automated testing approaches tailored to developer needs. Choudhary et al. [54] analyzed various modern test generation tools in a systematic way and illustrated that, surprisingly, the simpler Monkey tool surpassed more sophisticated tools in terms of code coverage, ease of use, and fault detection.

Random-based Testing Tools: Random based testing approaches construct test cases in a pseudo-random manner from the set of all possible program inputs [19]. Random-based tools excel in adaptability, as they target the app under test only on a per-screen basis. This technique was popularized in the Android testing tool, Monkey [20], and was later adapted by APE [21], Dynodroid [22], Intent Fuzzer [23], and VANARSena [24].

Model-based Testing Tools: MonkeyLab [25] uses the GUI-based models extracted from Android application execution traces to generate usage scenarios. The results demonstrate that MonkeyLab is able to generate effective and fully replayable scenarios. Moran et al. [25] studied the importance of crashes during Android application testing. The authors developed CrashScope, a tool that can automatically discover, report, and reproduce crashes. They executed their tool on 61 Android apps and compared their tool with A3E, DynoDroid, MobiGUITAR, Monkey, and Puma [20], [22], [55]–[57].

Dong et al. [28] proposed time-travel testing for Android, which works to maximize exploration efficiency by resuming to the most progressive states observed in the past. They evaluated their approach against Sapienz [27] and Stoat [50] and it outperformed them in coverage and crashes discovered.

Machine Learning-based Testing Tools: Li et al. [29] propose Humanoid, a testing tool that uses a combination of CNN and Residual LSTM in their approach to generate automated tests. They use the RICO dataset [30] and perform CNN on the screenshots to predict action location on a given UI screen. Residual LSTM is used to predict the type of action performed - such as tap, long tap, swipe, etc. Q-testing [58] is an AIG tool that uses reinforcement learning for input generation.

QTypist [59] is a tool designed to automate the generation of input text for mobile applications by interacting with a large language model. Compared to AURORA, which handles various different UI exploration challenges, QTypist focuses on only text-related challenges.

VIII. Conclusion

In this paper, we proposed AURORA, a framework that runs alongside AIG tools and can categorize and navigate around UI screens when an AIG tool is stuck using multimodal techniques for neural screen understanding. Our evaluation illustrates that AURORA can effectively recognize different types of screens and effectively navigate around them, increasing the effectiveness of AIG tools. To aid future research, we make AURORA and our labeled dataset of UI categories publicly available [35].

REFERENCES

[1] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "API Change and Fault Proneness: A Threat to the Success of Android Apps," in *ESEC/FSE*, 2013.

[2] G. Bavota, M. Linares-Vásquez, C. E. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk, "The Impact of API Change- and Fault-Proneness on the User Ratings of Android Apps," *TSE*, 2015.

[3] F. Palomba, M. Linares-Vásquez, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "User Reviews Matter! Tracking Crowdsourced Reviews to Support Evolution of Successful Apps," in *ICSME*, 2015.

[4] "Android and Google Play Statistics, Development Resources and Intelligence," 2023. [Online]. Available: https://www.appbrain.com/stats

[5] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, "Understanding the Test Automation Culture of App Developers," in *ICST*, 2015.

[6] O. Chaparro, C. Bernal-Cárdenas, J. Lu, K. Moran, A. Marcus, M. Di Penta, D. Poshyvanyk, and V. Ng, "Assessing the Quality of the Steps to Reproduce in Bug Reports," in *ESEC/FSE*, 2019.

[7] J. Mahmud, N. De Silva, S. A. Khan, S. H. Mostafavi, S. M. H. Mansur, O. Chaparro, A. A. Marcus, and K. Moran, "On Using GUI Interaction Data to Improve Text Retrieval-based Bug Localization," in *ICSE*, 2024.

[8] Y. Song, J. Mahmud, N. De Silva, Y. Zhou, O. Chaparro, K. Moran, A. Marcus, and D. Poshyvanyk, "Burt: A Chatbot for Interactive Bug Reporting," in *ICSE-Companion*, 2023.

[9] Y. Yan, N. Cooper, O. Chaparro, K. Moran, and D. Poshyvanyk, "Semantic GUI Scene Learning and Video Alignment for Detecting Duplicate Video-based Bug Reports," in *ICSE*, 2024.

[10] Y. Song, J. Mahmud, Y. Zhou, O. Chaparro, K. Moran, A. Marcus, and D. Poshyvanyk, "Toward Interactive Bug Reporting for Android App End-users," in *ESEC/FSE*, 2022.

[11] M. Fazzini, K. Moran, C. Bernal-Cárdenas, T. Wendland, A. Orso, and D. Poshyvanyk, "Enhancing Mobile App Bug Reporting via Real-Time Understanding of Reproduction Steps," *TSE*, 2023.

[12] C. Bernal-Cárdenas, N. Cooper, M. Havranek, K. Moran, O. Chaparro, D. Poshyvanyk, and A. Marcus, "Translating Video Recordings of Complex Mobile App UI Gestures into Replayable Scenarios," *TSE*, 2023.

[13] J. Johnson, J. Mahmud, T. Wendland, K. Moran, J. Rubin, and M. Fazzini, "An Empirical Investigation into the Reproduction of Bug Reports for Android Apps," in *SANER*, 2022.

[14] N. Cooper, C. Bernal-Cárdenas, O. Chaparro, K. Moran, and D. Poshyvanyk, "It Takes Two to Tango: Combining Visual and Textual Information for Detecting Duplicate Video-Based Bug Reports," in *ICSE*, 2021.

[15] M. Havranek, C. Bernal-Cárdenas, N. Cooper, O. Chaparro, D. Poshyvanyk, and K. Moran, "V2S: A Tool for Translating Video Recordings of Mobile App Usages into Replayable Scenarios," in *ICSE-Companion*, 2021.

[16] C. Bernal-Cárdenas, N. Cooper, K. Moran, O. Chaparro, A. Marcus, and D. Poshyvanyk, "Translating Video Recordings of Mobile App Usages into Replayable Scenarios," in *ICSE*, 2020.

[17] K. Moran, C. Watson, J. Hoskins, G. Purnell, and D. Poshyvanyk, "Detecting and Summarizing GUI Changes in Evolving Mobile Apps," in *ASE*, 2018.

[18] S. Salma, S. H. Mansur, Y. Zhang, and K. Moran, "GuiEvo: Automated Evolution of Mobile App UIs," in *MSR*, 2024.

[19] A. Orso and G. Rothermel, "Software Testing: A Research Travelogue (2000-2014)," in *Future of Software Engineering Proceedings*, 2014.

[20] "UI/Application Exerciser Monkey," 2022. [Online]. Available: https://developer.android.com/studio/test/other-testing-tools/monkey

[21] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical GUI Testing of Android Applications via Model Abstraction and Refinement," in *ICSE*, 2019.

[22] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An Input Generation System for Android apps," in *FSE*, 2013.

[23] R. Sasnauskas and J. Regehr, "Intent Fuzzer: Crafting Intents of Death," in *WODA PERTEA*, 2014.

[24] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan, "Automatic and Scalable Fault Detection for Mobile Applications," in *MobiSys*, 2014.

[25] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically Discovering, Reporting and Reproducing Android Application Crashes," in *ICST*, 2016.

[26] Y. M. Baek and D. H. Bae, "Automated Model-based Android GUI Testing using Multi-level GUI Comparison Criteria," in *ASE*, 2016.

[27] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective Automated Testing for Android Applications," in *ISSTA*, 2016.

[28] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel Testing of Android Apps," in *ICSE*, 2020.

[29] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A Deep Learning-Based Approach to Automated Black-box Android App Testing," in *ASE*, 2019.

[30] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. Kumar, "Rico: A Mobile App Dataset for Building Data-driven Design Applications," in *UIST*, 2017.

[31] Y. Zheng, Y. Liu, X. Xie, Y. Liu, L. Ma, J. Hao, and Y. Liu, "Automatic Web Testing using Curiosity-driven Reinforcement Learning," in *ICSE*, 2021.

[32] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie, "An Empirical Study of Android Test Generation Tools in Industrial Cases," in *ASE*, 2018.

[33] W. Wang, W. Yang, T. Xu, and T. Xie, "VET: Identifying and Avoiding UI Exploration Tarpits," in *ESEC/FSE*, 2021.

[34] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever, "Learning Transferable Visual Models from Natural Language Supervision," in *ICML*, 2021.

[35] "AURORA Replication Package," 2024. [Online]. Available: https://sagelab.io/aurora

[36] "Imgur Google Play Store Page." [Online]. Available: https://play.google.com/store/apps/details?id=com.imgur.mobile

[37] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *NeurIPS*, 2012.

[38] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *CVPR*, 2016.

[39] D. Marutho, S. Hendra Handaka, E. Wijaya, and Muljono, "The Determination of Cluster Number at k-Mean Using Elbow Method and Purity Evaluation on Headline News," in *iSemantic*, 2018.

[40] H. Zheng, D. Li, B. Liang, X. Zeng, W. Zheng, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated Test Input Generation for Android: Towards Getting There in an Industrial Case," in *ICSE-SEIP*, 2017.

[41] G. Karpushkin, "The JSON Comparison package," 2023. [Online]. Available: https://pypi.org/project/jsoncomparison

[42] X. Zhou, C. Yao, H. Wen, Y. Wang, S. Zhou, W. He, and J. Liang, "EAST: An Efficient and Accurate Scene Text Detector," in *CVPR*, 2017.

[43] "Google Cloud Vision API." [Online]. Available: https://cloud.google.com/vision/docs/ocr

[44] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," 2019.

[45] W. Fu and T. Menzies, "Easy over Hard: A Case Study on Deep Learning," in *ESEC/FSE*, 2017.

[46] "Tesseract OCR," 2023. [Online]. Available: https://github.com/tesseract-ocr/tesseract

[47] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving Language Understanding by Generative Pre-Training," *OpenAI*, 2018.

[48] Swind, "pure python adb," 2023. [Online]. Available: https://github.com/Swind/pure-python-adb

[49] T. J.-J. Li, L. Popowski, T. Mitchell, and B. A. Myers, "Screen2Vec: Semantic Embedding of GUI Screens and GUI Components," in *CHI*, 2021.

[50] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, Stochastic Model-based GUI Testing of Android Apps," in *ESEC/FSE*, 2017.

[51] "APE and Mini Trace Documentation," 2023. [Online]. Available: http://gutianxiao.com/ape

[52] "Why Am I Not Seeing Any Ads?" [Online]. Available: https://support.applovin.com/hc/en-us/articles/4403932179597-Why-Am-I-Not-Seeing-Any-Ads

[53] M. Linares-Vasquez, M. White, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk, "Mining Android App Usages for Generating Actionable GUI-based Execution Scenarios," in *MSR*, 2018.

[54] S. R. Choudhary, A. Gorla, and A. Orso, "Automated Test Input Generation for Android: Are We There Yet?" in *ASE*, 2015.

[55] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps," in *MobiSys*, 2014.

[56] T. Azim and I. Neamtiu, “Targeted and Depth-First Exploration for Systematic Testing of Android Apps,” in *OOPSLA*, 2013.

[57] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, “MobiGUITAR: Automated Model-Based Testing of Mobile Apps,” *IEEE Software*, 2015.

[58] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, “Reinforcement Learning Based Curiosity-driven Testing of Android Applications,” in *ISSTA*, 2020.

[59] Z. Liu, C. Chen, J. Wang, X. Che, Y. Huang, J. Hu, and Q. Wang, “Fill in the Blank: Context-Aware Automated Text Input Generation for Mobile GUI Testing,” in *ICSE*, 2022.