UIH\textsc{ash}: Detecting Similar Android UIs through Grid-Based Visual Appearance Representation

Jiawei Li, Beihang University; National University of Singapore; Jian Mao, Beihang University; Tianmushan Laboratory; Hangzhou Innovation Institute, Beihang University; Jun Zeng, National University of Singapore; Qixiao Lin and Shaowen Feng, Beihang University; Zhenkai Liang, National University of Singapore

https://www.usenix.org/conference/usenixsecurity24/presentation/li-jiawei

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.
UIHASH: Detecting Similar Android UIs through Grid-Based Visual Appearance Representation

Jiawei Li†‡, Jian Mao†‡§*, Jun Zeng‡, Qixiao Lin†, Shaowen Feng†, Zhenkai Liang‡
†Beihang University ‡National University of Singapore §Hangzhou Innovation Institute, Beihang University

Abstract
User interfaces (UIs) is the main channel for users to interact with mobile apps. As such, attackers often create similar-looking UIs to deceive users, causing various security problems, such as spoofing and phishing. Prior studies identify these similar UIs based on their layout trees or screenshot images. These techniques, however, are susceptible to being evaded. Guided by how users perceive UIs and the features they prioritize, we design a novel grid-based UI representation to capture UI visual appearance while maintaining robustness against evasion. We develop an approach, UIHASH, to detect similar Android UIs by comparing their visual appearance. It divides the UI into a #-shaped grid and abstracts UI controls across screen regions, then calculates UI similarity through a neural network architecture that includes a convolutional neural network and a Siamese network. Our evaluation shows that UIHASH achieves an F1-score of 0.984 in detection, outperforming existing tree-based methods and image-based methods. Moreover, we have discovered evasion techniques that circumvent existing detection approaches.

1 Introduction
User interfaces (UIs) is the main channel for users to interact with mobile apps. As a result, many security problems arise from similar UIs designed to deceive users, such as phishing and spoofing [11]. For example, the Svpeng malware [12] mimics a fake FBI penalty notification UI to blackmail victims, which has infected 350,000 Google devices.

To prevent such attacks, existing techniques detect similar UIs by taking either layout trees or screenshot images of UIs as input. For layout trees, layout definitions specified in XML trees, including tree structure [66, 70] and node attribute [52] are widely adopted. Another research direction is to collect UI screenshot images at runtime for comparison [51, 56]. However, adversaries can generate visually similar UIs via actively adjusting layout trees and screenshot images to evade existing detection methods. We call such evasions as active evasion attacks. Due to this reason, existing methods based on layout trees or screenshot images are vulnerable to attackers’ evasion techniques. On one hand, inconsistency between layout trees and UI visual appearances significantly degrades the detection effectiveness. In particular, adversaries can build visually similar UIs on top of structurally non-similar layout trees, resulting in false negatives in tree-based detection. On the other hand, users usually show high tolerance when perceiving visual changes on UIs, e.g., color modification and logo rotation [50, 61]. Such tolerance enables evading image-based detection by permitting modifications to the on-screen UI imagery (e.g., at the pixel level).

Key Idea. To address active evasions, we aim to find a robust representation for UI similarity comparison, which abstracts UI visual appearances and tolerates variations that are not perceived by users. Gestalt Principles reveal how people recognize scenes in an image by grouping components of images [74]. For example, the proximity principle indicates that users naturally tend to group adjacent elements, perceiving them as a unified entity. There are other principals help to connect isolated elements, such as closure, common region and continuity. The proximity principal is found to be dominant according to prior study on vision [17]. Inspired by Gestalt Principles, we represent UI by controls (e.g., buttons) in different screen regions, which is used as the basis of recognizing a UI when users take a glance. After partitioning the UI screen based on a grid and extracting visual features for each grid region, we integrate visual features across grid regions as the UI representation. Since various visual features, including control sizes, positions, types, colors, styles, texts, and images/icons—have different impacts on users’ perception on UI, we conduct a user study to identify the visual features that users prioritize. The study shows that users are sensitive to changes on control position, control size, and control type. Therefore, after abstracting UI appearance based on grids, we encode these specific visual features in grids based on user perception, instead of using empirically preset features [46].

Our Approach. In this paper, we present UIHASH, a frame-
work for detecting Android UI similarities based on visual appearance. As the appearance of a UI is specified by its controls distributed in separate screen regions, we partition the UI screen into a #-shaped grid, encode each grid region based on its constituent controls, and aggregate individual regions as the representation of a UI’s appearance, which we call UI#. Compared with layout trees and screenshot images, UI# captures the visual appearance features that users prioritize, balancing the trade-off between inconsistent tree structures and inflexible image pixels when determining UI similarity.

To generate UI# from Android UI, we collect runtime UI visual appearance in UI# and extracting image visual features. Specifically, we encode control position, control size, and control type to represent a UI#. Taking UI# as a quantitative representation for measuring the visual appearance of UI, we develop a learning-based model tailored to detect similar UIs in Android apps. Specifically, we identify the connection between measuring visual appearance in UI# and extracting image visual features. Leveraging a convolutional neural network, we distill visual features from UI#, and employ a Siamese network to assess the similarities between different instances of UI#.

We conduct a systematic evaluation with 52,390 real-world Android apps. Our experimental result shows that UIHASH achieves a 0.984 F1-score in detecting similar UIs, outperforming existing detection techniques [44,51,52,56,70,85]. In particular, it improves tree-based and image-based methods by 13.3% and 20.1% on the recall rate, respectively, denoting that more similar UIs are revealed. Furthermore, we discover that the active evasion attacks have already spread in real-world UI mimicking practices—accounting for 55.3% of all the detected similar UI pairs—that evade tree-based or image-based detection. Our evaluation also demonstrates that UIHASH is efficient in supporting real-life Android UI similarity detection and robust against adversarial attacks [20]. In summary, this paper makes the following contributions:

- We present a novel grid-based representation, UI#, to describe the visual appearance of Android UIs. It abstracts the UI into a #-shaped grid and focuses on visual features that users prioritize, addressing the inconsistency between UI appearance and layout tree/screenshot image.
- We design UIHASH to detect visual similarities among Android UIs, using a convolutional neural network to distill visual features from UI# and a Siamese network to identify similar UIs.
- We implement UIHASH and conduct a systematic evaluation against large-scale real-world apps1. The results demonstrate UIHASH’s effectiveness and robustness in detecting similar UIs.

1We release UIHASH at https://github.com/DaweiX/UIHash.


2 Preliminaries

Android apps define user interface (UI) by a layout tree, as Listing 1 shows. Each node in the tree (called View [25]) is associated with a bundle of attributes describing its properties (e.g., visual appearance). In particular, we refer to the views (e.g., Button and ImageView) that interact with users or visualize information as controls. Besides, ViewGroups [26] are specific views that can contain other views as children in various ways. For example, a LinearLayout allocates its children linearly (horizontally or vertically). After an app starts, the Android system renders the UIs described by layout trees. The UIs might change by app code at runtime.

Listing 1: A simple Android layout tree example.

```
<intent
<brace-open>
<LinearLayout
<brace-open>
<FrameLayout
<brace-open>
<TextView
<brace-open>
<Button
<brace-open>
<EditText
<brace-open>
<ImageView
<brace-open>
</LinearLayout
<brace-close>
</FrameLayout
<brace-close>
</TextView
<brace-close>
</Button
<brace-close>
</EditText
<brace-close>
</ImageView
<brace-close>
</LinearLayout
<brace-close>
```

Listing 1: A simple Android layout tree example.
UI screenshots [6, 8, 51, 56] have been widely adopted as powerful features to detect repackage, plagiarized or counterfeit apps via UI similarity. However, compared to layout trees, images are too strict for describing what a UI looks like. In practice, users are likely not to remember the accurate logos and icons of companies, brands, and apps [61]. Users are also surprisingly poor at noticing image changes due to change blindness [5, 15]. According to a questionnaire, 40% users still trust a fake Facebook login UI in brown color [50]. Therefore, UIs with modifications on images or colors can easily evade image-based methods while not raising user concerns.

3 User Perception Study

To detect similar UIs by active evasion attacks, we measure UI similarity based on a novel representation designed to capture UI appearance, preserving immediate human perception while remaining robust against deceptive alterations. Naturally, user perception of UI is influenced by various visual features (e.g., control size). We conduct a user study to explore the impact of these features on user perception when inspecting UI similarity. The study aims to identify visual features that users pay the most attention to. It also helps to align the design of UI HASH with user perception.

Survey Design. Our survey (details in Appendix C) contains three sections: ① Demographic survey. We prioritize user privacy and refrain from collecting personally identifiable information (PII), such as phone numbers and dates of birth. We also include “Prefer not to say” options. ② Similarity evaluation. UI design varies based on app categories (e.g., social networking apps, games) and the specific tasks to be performed [32]. Consequently, the flexibility for UI adaptation for adversaries may differ in different scenarios. For example, phishing UI should closely resemble the original to deceive users, whereas UIs in a plagiarized game can incorporate more variations without raising user suspicion. Our study focuses on the representative phishing scenario. It involves the evaluation of 12 login UIs, allowing users to assess their perception on various UI features. These UIs are modified versions of a login UI from a popular social networking app. To eliminate bias, we present an equal number of modified UIs to each participant and randomize the order in which UIs are displayed. Figure 2 illustrates some performed modifications. Naturally, the perceived similarity to the original UI is determined by the “login rate” from users. Additionally, we inquire about users’ specific concerns when refusing to log in and ask them to rank these aspects from most important to least important. ③ Exit survey. Participants in our study completed an exit survey regarding their mobile app experience and their perception on UI changes. Additionally, we use a simple attention check question to ensure users’ careful reading and accurate responses.

Recruitment. We use four separate URLs for data collection. To avoid bias, each URL presents different and randomized UIs. We run the online survey for one month before data merging and analysis, and recruit 350 participants through notice boards and mailing list advertisements. Each participant is directed to a random one of the four URLs.
To mitigate ethical risks, we consult researchers in the local community for best practices. We follow the transparent and user-consented procedures in line with the guidelines presented in prior works [13,29]. The participants are enrolled on a voluntary and anonymous basis and are informed about the study’s purpose. Besides, the form specifies a minimum age requirement of 18 years and assures participants that no personally identifiable information (PII) is asked or collected.

Results. We exclude surveys where participants do not use the app or failed the attention check, resulting in 319 completed surveys. Table 1 shows scores for features that discourage users from logging in. Control position and size receive higher scores than other features such as color, image content, and text. Figure 3 displays detailed login rates of users under different UI modifications (all users logged in successfully to the original UI). All users demonstrate significant tolerance for UI modifications, as reflected in the login rates. While the t-test reveals a difference in login rates for the 12 features between participants with over 8 frequently-used apps and those using fewer apps ($t = 2.07, p < 0.05$; $d = 0.88$), both user groups are most sensitive (exhibit the lowest login rates) to control size in each region. At last, it aggregates all regions into a same UI feature set. Specifically, users show heightened attention to the following features:

- Control size (D.Drop, I.Scale, and L.Size). Both global scaling (I) and separate control resizing (L) decrease the login rate drastically. However, users lack attention to smaller controls. 70% of the users who self-measured as skilled still trust the UI of a “sign up” button missing in D.
- Control type (H.Reorder-s, K.Reorder-d). We rearrange controls of the same and different types (e.g., swapping EditTexts or Buttons, replacing images with buttons). The results demonstrate that changing the control type in a specific screen area effectively increases users’ awareness.
- Control position (J.Layout, K.Reorder-d). J and K get the lowest login rates compared to other modifications.

Meanwhile, users show less attention (higher login rates) to other features, such as hue/brightness, control style, and text/image contents. The results complement existing study [50].

Insights. Recognizing that users exhibit a higher tolerance for UI changes compared to machines, we focus on comparing UIs based on the visual features that users prioritize, rather than relying on strict pairwise comparisons of layout trees or image pixels. Since features with lower login rates draw more user attention, such features (e.g., control size, type and position) are instrumental to differentiate UIs. Consequently, UIs that actively incorporate changes in high-login-rate features can be considered similar to the originals, making it challenging for evasive UIs to avoid detection.

4 UIHASH Design

In this section, we present UIHASH, our approach for UI similarity detection. We introduce how to abstract and encode visual semantics of UIs into the new representation UI# and how we use UI# to detect similar UIs.

Overview. As shown in Figure 4, UIHASH consists of three phases: UI Parsing, UI# Generation, and Similarity Detection. It extracts UIs from apps, transforms UIs to UI#, and measures UI# similarity. Specifically, UI Parsing extracts UIs from Android application packages (APKs). It produces tree-structured UI hierarchies and screenshots dumped at runtime. The UI hierarchies describe the boundaries of individual UI controls, specifying their positions and sizes. Besides, we utilize UI screenshots to extract control images and identify their types. UI# Generation takes UI hierarchies and control images as inputs. It first uses a convolutional neural network (CNN) to identify control types based on their visual appearance. Then, it divides the UI into an $n \times m$ grid, categorizes controls in each grid region by their types, and encodes control size in each region. At last, it aggregates all regions into a visually-inspired semantic representation, UI#. Similarity Detection measures UI pairwise similarities based on UI#. Specifically, it first embeds UI# via a CNN to infer visual semantics. A Siamese neural network is later applied to calculate the similarity score of two UI#. Finally, UIHASH uses the score as a metric to discover similar UIs across apps and reports them to security analysts.

We integrate UIHASH into an Android app analysis platform, shown in Figure 5. Utilizing this platform, we extract various app features beyond UI (e.g., manifest, static resources, and codes) to support cross-checking.

4.1 UI Parsing

Given an app, UIHASH first extracts UI information dynamically by Minicap [1] and UIAutomator [24]. Specifically, UIHASH launches the app and captures the main UI (i.e., the main activity). It then simulates tapping on each interactive view. If the app navigates to another UI, it repeatedly taps on the new UI and flags the UI as visited. This procedure...
Method-based Detection
Dex-based Detection
Resource-based Detection
Manifest-based Detection
UIHash
Feature Tool
Crosscheck
Analyst
Figure 5: The platform for analyzing Android apps.

terminates if the visited UI set stops expanding. Note that apps may have unreachable UIs and non-launcher/exported UIs, which require user authorization or other apps to trigger. To collect these UIs, UIHASH also launches apps separately from each Android activity registered in the manifest. Before capturing each UI, we wait for a short period (0.2 seconds) to complete animations and online resource loading. We note that the presence of the FLAG_SECURE flag results in UIs returning empty data upon capture. However, given its limited use in real-world scenarios [16], we choose not to bypass the flag. We mark UIs without layout data as unobtainable. We provide further discussion on this flag in Section 6.

For each captured UI, we save its screenshot and dump its runtime hierarchy. Compared with static layout trees, the hierarchy reveals accurate runtime visual details like control boundaries [22]. We use it for bridging the semantic gap between static UI definition (i.e., layout tree) and dynamic UI visual appearance (i.e., images). In the meantime, we use UI screenshots to collect control images for subsequent type identification. It is worth mentioning that compared with image-based UI similarity detectors, the screenshots in UIHASH are not directly involved in the final UI similarity detection.

Figure 4: Overall architecture of UIHASH.

Figure 6: Six toggles implemented by different view types.

4.2 UI# Generation

UI Control Reidentification. We observe that controls may have a similar visual appearance even with different built-in View types defined by Android. For example, Figure 6 illustrates six apps that leverage controls labeled with different types to build visually similar “Toggles” on UIs. The UI similarity detection built on the declared control type is vulnerable if an adversary introduces visually similar controls with different built-in types, e.g., choosing an ImageButton instead of a TextView to show text. App developers can also name and implement custom Views in Android [25]. Besides, inline advertisements pasted on UIs can be shown in different forms such as buttons, texts, images, etc. However, this difference cannot be reflected by the control names of advertisements, so we also need to conduct reidentification for advertisements.

We aim to reidentify the types for all UI controls to capture UI’s visual features more precisely. More specifically, we assign new types to controls based on what they look like instead of their original types claimed in the UI hierarchies. Specifically, UIHASH classifies UI controls to input, button, checkbox, list, spinner, tab, text, toggle, and bar. We manually label 27,000 control images extracted from RePack [38] and Rico dataset [41] to provide supervision signals. We use OpenCV [72] to extract control images from UIs, convert them to grayscale, and unify their sizes. By backward propagating supervised signals from labeled UI controls, we optimize a CNN-based classifier to predict the probability of a UI control belonging to one of the nine types. If the probability of a type is beyond a pre-defined threshold (0.95 in our implementation), UIHASH identifies the control as the corresponding type. If no match exists, we annotate the control type as “Others”. Afterward, we split all UI controls into ten
Algorithm 1: Generate UI# for a UI

1. **Input:** hierarchy of a UI $H$, and type index of each control
2. **Output:** UI# $\mathcal{H}$
3. $\text{controls} \leftarrow \text{controls in } H$
4. for $i \leftarrow 1$ to $n \times m$
   5. cord$^i$ \leftarrow coordinates of the $i$-th region
   6. $v_i \leftarrow \text{zeros}(\text{shape} = (1, 10))$
   7. // split channels by reidentified control types
5. for $j \leftarrow 1$ to 10
   8. $E^j \leftarrow \{e \in \text{controls}|e\.typeindex = j\}$
8. foreach $e \in E^j$
   9. cord$^e$ \leftarrow coordinates of control $e$
   10. $iou^e \leftarrow \text{GET_IOU}(\text{cord}^e, \text{cord}^i)$
   11. // drop the tiny exceed parts
   12. if $iou^e > \text{threshold then}$
   13. $v_i^j \leftarrow v_i^j + iou^e$
   14. // revise hash values
   15. $v_i^j \leftarrow \text{LOG_REV}(v_i^j)$
8. $\mathcal{H} \leftarrow [v_1, v_2, ..., v_{n \times m}]$ // semantic aggregation

4.3 Similarity Detection

The next step in UIHASH is to compare visual semantics in UI# for UI similarity detection. Our insight comes from the grid representation UI# similarity is based on, which is comparable to that of images. To learn image visual (spatial) semantics, CNN is widely adopted [42, 64]. It can tolerate image distortions while generalizing visual features [62]. It is also invariant to image object translation, matching our intuitive understanding that partial UI view translations make a limited contribution to the overall UI visual appearance.

UIHASH utilizes CNN to embed UI# into a vector space (16 dimensions in our case). Intuitively, the relative distance between a pair of UI# indicates their visual similarities. In other words, the closer embeddings for two UI# to each other, the more similar appearances their related UIs share, and vice versa.

For example, the four login UIs in Figure 1 are located nearby in the embedding space.

To infer visual semantics, our CNN model takes UI# as input, goes through two convolutional layers and three fully connected (FC) layers, and generates vector-representation visual semantics. Based on vectorized UI#, we further use a Siamese neural network to measure the similarities of their visual semantics. Specifically, UIHASH calculates UI similarity scores using the cosine similarity considering its simplicity and effectiveness. We define the Siamese network output as the cosine similarity of a pair of UI# and optimize network parameters by minimizing the following loss function:

$$\mathcal{L} = \sum_{i=1}^{N} (\text{sim}(v_i^1, v_i^2) - y)^2,$$

where $\theta$ presents the shared model parameters of the two CNN models in the Siamese network. $N$ indicates the count of input UI# pairs, and $y$ is the ground truth label.

Note that the app set is commonly enormous in Android UI analysis. Directly applying traditional gradient
descent (e.g., stochastic gradient descent) will be limited due to computational inefficiency. To address this issue, we leverage the mini-batch gradient descent algorithm [33] to split the training dataset into small batches before calculating model training losses and optimizing model parameters. Specifically, we use an Adam optimizer [10] with mini-batch optimization for CNN model training. We apply a grid search to get the best hyperparameters for the Siamese network. The learning rate and batch size are tuned among \{0.01, 0.005, 0.0015, 0.001, 0.0005\} (with and without decay) and \{16, 32, 64, 128, 256, 512\}, respectively. We search the decision threshold for similar UI detection in \{0.5, 0.6, 0.7, 0.75, 0.8, 0.85, 0.9\}. In light of the best F1 score in our experimental environment, we show the results taking the initial learning rate as 0.001 (×0.1 every ten epochs), the batch size as 32, and the threshold as 0.6. We train the model for 36 epochs. Note that the configurable decision threshold illustrates the flexibility of the learning model because analysts can customize it according to different practical scenarios. For example, decrease it to satisfy high true positives or increase it to maintain low false positives. Based on the threshold, UIHASH differentiates similarity scores and detects UI similarity. If the similarity score of two UIs across apps exceeds the decision threshold, UIHASH categorizes them as similar UIs and reports to security analysts.

## Evaluation

In this section, we evaluate on:

i. How effective is UIHASH as a UI similarity detection system? (§ 5.1)

ii. How common are active evasion attacks in the wild? (§ 5.2)

iii. How robust is UIHASH against adversarial attacks? (§ 5.3)

iv. What benefits can analysts gain from our UI representation UI#, in addition to pairwise UI similarity detection? (§ 5.5)

### 5.1 Effectiveness of UIHASH

#### 5.1.1 Comparison Analysis

In this section, we evaluate the detection effectiveness of UIHASH by comparing it with state-of-the-art similar UI detectors. Besides, we investigate the impact of different design choices on UIHASH’s performance. Furthermore, we assess the effectiveness of UIHASH on new data by applying the pre-trained model to new apps.

**Settings.** To conduct a ground-truth-based evaluation, we expand the RePack dataset [30] with similar apps provided by a security enterprise and obtain a total of 18,359 apps (2,816 original + 15,543 repackaged). This app set is referred to as RePack-e. We label 6,371 similar UIs with the same activity name derived from known similar applications. We also generate 6,371 non-similar UI pairs by randomly sampling two UIs that do not come from a similar app pair and have different names. Based on this dataset, to evaluate the effectiveness of UIHASH in detecting similar UIs, we first randomly select 80%, 10%, and 10% of UI pairs to constitute the training, validation, and testing sets. We follow the guidance proposed in [9] to responsibly report results for our learning models, and avoid data snooping and biased parameter risks on two fronts: (1) we ensure that there is no knowledge overlapping (e.g., repeated original apps/UIs) among the three subsets, considering that RePack-e includes cases with one original app and its many repackaged versions; (2) we optimize UIHASH solely based on the validation set [39, 40, 82]. We compare UIHASH with state-of-the-art UI-based similarity detection baselines (without overlap), including text-based [44, 56], image-based [51, 56] and layout-tree-based [52, 70] methods. We take precision \( p \) (true positives vs. detected positives), recall \( r \) (true positives vs. ground-truth positives), and F1-score (harmonic mean of precision and recall) as evaluation metrics. Besides, we search for the decision thresholds for prior works with the highest F1-scores under the dataset used (e.g., \( d = 10 \) for pHASH and \( \delta = 0.82 \) for GEMINI SCOPE).

We illustrate how our design choices contribute to similar UI detection. (1) To justify the grid-based design, we perform an ablation experiment where the original feature values (e.g., size, position, and type) of controls are concatenated and directly fed into the Siamese model without being encoded into grids. (2) To investigate the impact of re-identifying controls, we remove this step and instead label UI controls based on their literal names (e.g., directly regarding a control named **Button** as a button). (3) To explore the effect of encoded control size, we fine-tune the base number \( \gamma \), which controls control size encoding, among \{2, 5, 10\}. (4) To determine the optimal grid size, we apply various grid sizes \( \{1 \times 1, 2 \times 2, 3 \times 3, 4 \times 3, 5 \times 5, \text{ and } 10 \times 10\} \) for UIHASH.

To assess the pre-trained UIHASH’s effectiveness on new data, we test it on a different set from a leading security vendor, with 24/24 manually labeled similar/different UI pairs.

**Results.** Table 2 shows that UIHASH outperforms the prior UI similarity detection approaches on all metrics, especially on the recall rate and F1-score, indicating its stronger ability to find similar UIs. The receiver operating characteristic (ROC) curves in Figure 7 demonstrate a notable enhancement UIHASH gains on the area under the ROC curve (AUC). UIHASH performs better than methods based on single UI texts, which has the worst precision and F1-score, and obtains a 0.201 higher recall rate than the image-based method.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text-based detection [44, 56]</td>
<td>31.7%</td>
<td>83.0%</td>
<td>0.459</td>
</tr>
<tr>
<td>pHASH [51, 56]</td>
<td>85.1%</td>
<td>79.7%</td>
<td>0.825</td>
</tr>
<tr>
<td>DROID EAGLE [70]</td>
<td>96.8%</td>
<td>86.5%</td>
<td>0.914</td>
</tr>
<tr>
<td>GEMINI SCOPE [52]</td>
<td>99.0%</td>
<td>94.3%</td>
<td>0.949</td>
</tr>
<tr>
<td>UIHASH</td>
<td>97.0%</td>
<td>99.8%</td>
<td>0.984</td>
</tr>
</tbody>
</table>

1 Two UIs are flagged as similar if their layout hashes are the same.

Table 2: Effectiveness evaluation results.
The above results show that a too-small grid size cannot fully represent a UI’s visual appearance, while a too-large grid size decreases the tolerance to partial and minor UI changes. The $5 \times 5$ grid is used as a balanced trade-off between UI representation and tolerance.

On the new testing set, UIHASH successfully flags all the similar UIs, demonstrating its transferability across different datasets. Figure 11 shows two similar UI pairs in the dataset. Despite changes in UIs, UIHASH can still capture the visual similarity within individual pairs.

### 5.1.2 Effectiveness on Finding New Similar UIs

To explore how UIHASH performs on malicious apps and recent apps as a similar UI detection method, we apply the UIHASH model pre-trained by RePack-e on malicious apps and recent apps without retraining or fine-tuning.

**Malicious Apps.** We applied UIHASH on RMDROID, an Android malware dataset [77] with 9,133 malicious apps confirmed by VirusTotal [3]. Unfortunately, UI similarity labels are unavailable for this dataset. As such, instead of evaluating UIHASH’s effectiveness using ground-truth-based metrics like F1-score, we report interesting findings on identifying similar UIs using UIHASH. We successfully extracted 34,524 UIs from 8,879 out of 9,133 apps. These UIs contain 280,750 visible controls, of which 95,025 controls are re-identified (confidence score $\geq 0.95$). We further filtered out semantic-limited UIs with only four or fewer visible controls. In total, UIHASH generated 18,558 UI#. Then, we made pairwise in-app UI comparisons via UIHASH and dropped redundant UIs.

---

1 The rest 254 (2.8%) apps stop running as soon as we launch them.
Finally, we got 11,561 UI# that constitute 66,812,868 pairs. In all the UI# pairs, UIHASH identified 2,566,643 ones (3.8%) as similar. We group similar UIs and present the two largest groups in Figure 12. The left group contains 700 UIs from 700 apps developed by 387 different developers. A representative UI is named com.kuguo.kuzai.Boutique.Activity from the app com.xiaobo.baobaogushihui. These UIs all show “Elaborate App Recommendation” texts, implicitly inducing users to download apps from untrusted sources. These apps belong to the kuguo malware family and exhibit the family’s characteristics including installation of apps from unknown or unverified download sites [54]. The second most frequent UI pattern (right) appears in 570 airpush malwares designed by 183 developers. An example of this pattern is the UI com.google.android.gms.ads.AdActivity from the app com.appsmirnity.litres.book121140. These UIs require user agreement to perform actions such as changing the browser’s homepage, which is a representative behavior of the airpush family [18]. These results underscore the essential role that detecting similar UIs plays in identifying malicious app families—these apps not only implement the functions of their corresponding malware families through UIs but reflect the core traits specific to malicious families by UIs. Hence, despite inherent noise in UI-based similarity detection (e.g., not all apps with similar UIs are malicious), UI is important for discerning the inherently malicious nature of these apps. Analyzing UI provides a unique perspective for understanding malicious behaviors across various malware families.

Recent Apps. We collected 8,963 recent apps (uploaded from March 2022 to July 2022) from six app markets in Table 3 and extracted 84,972 UIs. These apps do not appear in all the above datasets. On average, we collected 10.8 UIs from one app. After dropping similar UIs within each app according to UI# comparison, we compared UIs between every two markets to explore cross-market and same-market UI similarities.

We summarize the pairwise UI similarity in Table 4. For a pair of UIs, if one is from market A and the other is from market B, we mark the UI pair as a member in “market A - market B”. Of all UI pairs, similar pairs account for 8.6% (Vxat - Vxat) to 13.4% (2345 - 2345). Given cross-market similar UIs, we manually confirm that 50 apps (different files from three out of five markets) share at least five similar UIs with their peers. Among the 25 app pairs, 22 pairs hold certificates from different issuers. For 23 out of 25 pairs, the apps differ in runtime behaviors, e.g., one of the apps displays ads or generates SMS payments while the other does not.

Besides, UIHASH finds the prevalence of similar apps in the same market. The dominant proportion of similar app clusters is found in Mumayi, where 51 apps (15.0%) gather because their main UIs are similar\(^4\). We note that app similarity might be ambiguous according to app features besides UI.

For example, the two app pairs on the left in Figure 13 are not similar in none of their package names, developers, certs, and icons. Codes for the center app pair are not similar either. Finally, UIHASH reports a similar UI pair associated with WeChat, a popular app for instant messaging\(^5\) (Figure 13 right). The malicious version (left)\(^6\) attempts to mimic the original splash screen and obfuscates it by modifying text contents (“English” to “Language”), substituting the background image, and fine-tuning button sizes and colors. VirusTotal\(^3\) reports that the counterfeited app is a high-risk malware.

---

\(^4\)There are another 55 apps with similar UIs. However, we manually confirm that it is the market instead of individual developers who customize and insert the UIs. Since the app-business-related UIs are not alike, we do not consider these apps similar.

\(^5\)from wandoujia.com, package: com.tencent.mm, CRC32: 13A3227F

\(^6\)from appfun.cn, package: com.tencent.mmweix, CRC32: AEBC46BD

---

![Figure 10: UIHASH’s performance with different setups.](image1)

![Figure 11: Similar UI samples in the new app set.](image2)

![Figure 12: The two most frequently-used UIs in RmvDroid.](image3)

<table>
<thead>
<tr>
<th>Market</th>
<th>App</th>
<th>Size</th>
<th>UI</th>
<th>Market</th>
<th>App</th>
<th>Size</th>
<th>UI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kuan</td>
<td>1,231</td>
<td>80GB</td>
<td>19,303</td>
<td>Munayi</td>
<td>339</td>
<td>16GB</td>
<td>2,331</td>
</tr>
<tr>
<td>2345</td>
<td>172</td>
<td>10GB</td>
<td>826</td>
<td>Leyuan</td>
<td>6,649</td>
<td>245GB</td>
<td>59,070</td>
</tr>
<tr>
<td>Vxat</td>
<td>224</td>
<td>12GB</td>
<td>1,886</td>
<td>Appfun</td>
<td>348</td>
<td>10GB</td>
<td>1,556</td>
</tr>
</tbody>
</table>

Table 3: Details of the recent app set.
Table 4: Total UI pairs (T) compared by UIHASH and detected similar UI pairs (S) between markets.

<table>
<thead>
<tr>
<th>UI Pair</th>
<th>Mumayi</th>
<th>Leyuan</th>
<th>2345</th>
<th>Vxat</th>
<th>Kuan</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>4,160K</td>
<td>66,036K</td>
<td>1,112K</td>
<td>2,639K</td>
<td>10,981K</td>
</tr>
<tr>
<td>S</td>
<td>494K</td>
<td>7,000K</td>
<td>140K</td>
<td>351K</td>
<td>1,378K</td>
</tr>
<tr>
<td>S/T</td>
<td>11.9%</td>
<td>10.6%</td>
<td>12.6%</td>
<td>13.3%</td>
<td>12.0%</td>
</tr>
<tr>
<td>T</td>
<td>253K</td>
<td>26,308K</td>
<td>36K</td>
<td>42K</td>
<td>392K</td>
</tr>
<tr>
<td>S</td>
<td>31K</td>
<td>2,874K</td>
<td>4.4K</td>
<td>3.6K</td>
<td>49K</td>
</tr>
<tr>
<td>S/T</td>
<td>12.3%</td>
<td>10.9%</td>
<td>12.2%</td>
<td>8.6%</td>
<td>12.4%</td>
</tr>
</tbody>
</table>

* Superscripts 1, 2, 3 indicate data for Mumayi-Mumayi, Leyuan-Mumayi, and Leyuan-Leyuan, respectively.

Figure 13: Three similar UI pairs we detect in recent apps.

5.1.3 Generalization of UI# for Practical Use Cases

In addition to its capability to detect similar UIs, we assess the extent to which UIHASH produces false alarms for security analysts, confirming whether our similarity measure exhibits excessive generalization for practical use cases.

**Settings.** As benign apps do not mimic other apps to deceive users and have unique UIs, UI pairs detected as similar are potentially false alarms. Therefore, we collected 75 login UIs from the top 100 Google Play apps and evaluated similarity among 2,484 UI pairs. We then analyze UIHASH’s false alarms by examining similar UIs detected among benign apps.

**Results.** UIHASH flagged 58 UI pairs (2.7% of all) as similar. We confirmed that the majority of them are single-sign-on (SSO) UIs provided by famous enterprises such as Facebook and Twitter. However, the right two pairs in Figure 14 are not SSO UIs and belong to different online service providers. We show the positive UIs to the participants of our user study and ask for their similar ratings for the UIs. As Table 5 shows, SSO UIs are considered similar by users with high confidence (4.5 out of 5), though there are differences in icons and texts. Pair (b) also receives a high score (4 out of 5), which has more differences like EditText styles and the existence of micro controls. However, users have mixed options on whether pair (c) is similar. The score is close to the borderline (i.e., 3) but slightly lower, so we mark (c) as a false positive. Finally, of all the 2,484 UI pairs from the 100 apps, UIHASH raises eight false positives (Facebook login UIs to Line login UIs, 0.3% to all UI pairs). We believe that a 0.3% false positive rate is promising to scale up UIHASH for similar UI detection. In summary, UIHASH generates few false alarms and has a reasonable and not excessive similarity tolerance.

<table>
<thead>
<tr>
<th>UI Pair</th>
<th>UI Pair</th>
<th>Average Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facebook SSO</td>
<td>2.2</td>
<td>28.7</td>
</tr>
<tr>
<td>Twitter SSO</td>
<td>2.5</td>
<td>0.0</td>
</tr>
<tr>
<td>Outlook-GMail</td>
<td>7.5</td>
<td>8.8</td>
</tr>
<tr>
<td>Facebook-Line</td>
<td>16.3</td>
<td>25.6</td>
</tr>
</tbody>
</table>


Table 5: User rating for the similarity of positive pairs.

5.2 Active Evasion Attacks in the Wild

Among all the similar UIs we detected and confirmed in Repack-e, malicious apps, and recent apps, we further investigate the popularity of strategies adversaries may perform for active evasion attacks. Then, we quantify how different similar UI detectors perform in detecting similar apps.

**Settings.** We take six active evasion scenarios as examples. We identify a flexible usage of view groups evasion case when two visually similar UIs use different sets of view groups, where adversaries with Android UI domain knowledge leverage various view groups to achieve similar UIs and evade tree-based detection. Besides, we identify micro controls when their on-screen pixels are not greater than ten and identify off-screen controls according to their positions. We also identify image-based evasion attacks that target background images or hue changes. After rebuilding each UI pair by removing their background images (if available), we identify an evasion case of image changing if the new pair is detected by the pHash algorithm but the original one bypasses the detection. Hue modification is identified if a UI pair is classified as similar by pHash when traversing the color gamut. In the case of empty text, attackers insert TextViews without any visible content (e.g., whitespace characters) to obfuscate layout trees.

**Results.** Table 6 shows that 17.9% mimicry UIs utilize empty texts to obfuscate layout trees. The second popular option (17.7%) is the flexible usage of view groups. 15.4% evasion cases use image-based techniques (14.3% for image changing and 1.1% for hue modification). Evasions using invisible or microscopic controls account for 4.3%. Considering that the above active evasion techniques occupy more than half (55.3%) UI pairs, we point out that the active evasion attacks
Table 6: Identified techniques for active evasion attacks and whether they are flagged as similar by different methods.

<table>
<thead>
<tr>
<th>Active Evasion Attack</th>
<th>Ratio</th>
<th>Image</th>
<th>Tree</th>
<th>UIH</th>
<th>ASH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty text</td>
<td>17.9%</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Flexible use of view groups</td>
<td>17.7%</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Image changing</td>
<td>14.3%</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Off-screen control</td>
<td>2.9%</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Micro control</td>
<td>1.4%</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Hue modification</td>
<td>1.1%</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Figure 15: Similar app pairs (app pairs with similar UIs) flagged by different UI-based methods.

are already widely used in real-world apps. Figure 15 shows similar app pairs detected in RePack-e by different approaches. A combination of different prior work flags more similar apps than a single method, but is still inferior to UIH. In addition to all the true positive UIs, the combination detects, UIH detects 1,645 more apps with similar UIs—which employ multiple evasion attacks to counter both tree-based and image-based methods. These results highlight the pivotal role that considering user perception plays in the detection of similar UIs. It stands as a key factor contributing to UIH’s effectiveness compared to previous methods.

We also investigate the tree edit distance (TED) [84] of the similar UI pairs, which forms the basis for tree-based detection. The smaller TED is, the more similar two layout trees are. Figure 16 illustrates the results. The horizontal axis, $\alpha$, is the ratio of the TED to the minimum tree size (i.e., node number) in a pair of UIs. The vertical axis, $\beta$, presents the percentage of the UI pairs whose $\alpha$ is not greater than the corresponding $\alpha$. We find 27% UI pairs meet $\alpha > 0.25$ (point A), indicating that in 27% cases, the minimal cost of node edit operations to convert the smaller layout tree to the other is at least a quarter of the smaller tree size. Moreover, 5% of UI pairs require a TED not less than the smaller tree size (point B). In extreme cases, the operations are four times the smaller tree size (point C). These results demonstrate the prevalence of mimicry UIs in adopting evasion techniques on trees.

5.3 Robustness to Adversarial Attacks

With the growth of neural networks, adversarial examples (i.e., crafted inputs designed to deceive a neural network and result in mistakes) have become increasingly common for learning-based detection evasion. Therefore, it is important to know how robust UIH is against adversarial examples.

**Settings.** We conducted two popular white box adversarial attacks via CLEVERHANS [57] on UIH with the RePack-e dataset, namely fast gradient method (FGM) [20] and projected gradient descent (PGD) [48], where the attacker is assumed to have complete access to the similarity detection model. Both attacks subtract or add a value $\epsilon$ to each UI# element, where $|\epsilon| \leq \epsilon$. Since attackers may adjust the counterfeit UI to evade UIH (i.e., be detected as non-similar to the original one) while cannot modify the UIs in the imitated apps, we do not add perturbations on both the input UI#. Instead, we replace one of the UI# in the input UI# pair with an adversarial example and then calculate the similarity score for the new UI# pair. As $\epsilon$ is critical to attack for it determines the magnitude of perturbations, we explore different $\epsilon$ selections, namely 0 (i.e., no adversarial attack), 0.05, 0.1, and 0.2.

**Results.** We show the results in Figure 17. The precision and AUC gradually decrease with the increase of $\epsilon$. In particular, PGD has a more significant effect than FGM. Guided by the attacker’s goal of using adversarial attacks to keep a visually similar UI from being detected, we take a further step to manually check UI$_{p\cdots N}$, i.e., true-positive UI pairs becoming false-negative after adversarial attacks. Interestingly, we observe that all these false-negative pairs are, in effect, no longer visually similar. A reasonable explanation is that UI# is an abstraction of UI rather than a pixel-grained representation. Consequently, perturbations to UI# work on the overall UI’s visual appearance rather than locality. As such, UIH is generally robust to adversarial attacks.

5.4 Performance

We conducted all the experiments with a two-core@2.90 GHz CPU, 16 GB memory, and a GPU with 2 GB memory. The overhead mainly comes from UI dynamic collecting, which consists of time for installing/uninstalling apps and a 0.2s delay for each UI loading its contents. The collecting time can be reduced by parallelizing UIH on multiple Android devices [36]. On the RmvDroid dataset, UIH uses 1.3 h to extract 280,750 view images and takes 3.0 min to generate UI# for 18,558 UIs. It costs 6.2 min to filter similar UI# in each app, and the time cost for a brute pairwise search among
We note that when generating UI#, re-identifying control elements is related to our implementations. All the implementations are Python-based. Specifically, for DroidEagle, we set its parameter $\tau$ to 2.

Table 7: Runtime (collect and detect), F1-score, and AUC on RePack of different detection approaches. The average seconds are indicated in parentheses (per UI, pair, and app for column Collect, Detect, and Overall, respectively).

<table>
<thead>
<tr>
<th>Approach</th>
<th>Collect</th>
<th>Detect</th>
<th>Overall</th>
<th>F1</th>
<th>AUC</th>
</tr>
</thead>
<tbody>
<tr>
<td>DROIDEAGLE</td>
<td>static</td>
<td>1.3h (0.4)</td>
<td>9.6h (1.9)</td>
<td>0.91</td>
<td>—</td>
</tr>
<tr>
<td>GEMINISCOPeE</td>
<td>dynamic</td>
<td>2.8h (4.8)</td>
<td>27.5h (5.3)</td>
<td>0.82</td>
<td>0.77</td>
</tr>
<tr>
<td>pHash-based</td>
<td>dynamic</td>
<td>2.8h (4.8)</td>
<td>27.5h (5.3)</td>
<td>0.82</td>
<td>0.77</td>
</tr>
<tr>
<td>UIHASH</td>
<td>dynamic</td>
<td>2.8h (4.8)</td>
<td>27.5h (5.3)</td>
<td>0.82</td>
<td>0.77</td>
</tr>
</tbody>
</table>

Figure 17: Evaluating UIHASH with RePack-e, against two adversarial attacks (FGM and PGD) with different $\epsilon$.

Figure 17: Evaluating UIHASH with RePack-e, against two adversarial attacks (FGM and PGD) with different $\epsilon$.

We list the time overhead\footnote{Note that all the baseline tools are not publicly available, so their overheads are related to our implementations. All the implementations are Python-based. Specifically, for DroidEagle, we set its parameter $\tau = 2$.} of different methods for processing the RePack dataset in Table 7. We divide the above approaches into two groups, depending on whether they collect UIs statically or dynamically. Although dynamic UI collection incurs more than three times overhead compared with the static collection, we highlight that only by capturing UIs at runtime can a detector bridge the semantic gap between layout trees and UI appearances (further discussion in Section 6). Considering the overhead for extracting UI features and measuring UI similarity, pHash-based methods outperform their peers significantly. However, they perform the worst F1-score.

We note that when generating UI#, re-identifying control images costs 94.8% time, but improves UIHASH’s AUC from 0.96 to 0.99. Training the Siamese model takes much less time (3.1 min). With a pre-trained model, UIHASH finishes detecting tens of thousands of UI pairs in seconds. To conclude, UIHASH achieves a trade-off between overhead and effectiveness compared to the baselines.

66,812,868 UI pairs is 3.2 h (0.17 ms per pair on average).

We list the time overhead of different methods for processing the RePack dataset in Table 7. We divide the above approaches into two groups, depending on whether they collect UIs statically or dynamically. Although dynamic UI collection incurs more than three times overhead compared with the static collection, we highlight that only by capturing UIs at runtime can a detector bridge the semantic gap between layout trees and UI appearances (further discussion in Section 6). Considering the overhead for extracting UI features and measuring UI similarity, pHash-based methods outperform their peers significantly. However, they perform the worst F1-score.

We note that when generating UI#, re-identifying control images costs 94.8% time, but improves UIHASH’s AUC from 0.96 to 0.99. Training the Siamese model takes much less time (3.1 min). With a pre-trained model, UIHASH finishes detecting tens of thousands of UI pairs in seconds. To conclude, UIHASH achieves a trade-off between overhead and effectiveness compared to the baselines.

5.5 Additional Applicability of UI#

5.5.1 Clustering Analysis

UI# can serve for clustering analysis besides pairwise UI similarity detection. Empirically, we can cluster similar UIs into groups, supporting two functionalities as follows:

- **Mundane layout filtering.** Some app UIs may share similar or even identical UI layouts, e.g., splash screens (a single Image) and dialogs (e.g., Text and Button). These UIs with redundant layouts bring a tedious burden to analysts when exploring UI similarity.

- **Targeted similar UI search.** Given specific benign apps, clustering analysis helps pinpoint UIs with a similar visual appearance at a large scale. When UIs are outliers or belong to mundane clusters with no sensitive information (e.g., “about” and “settings”), the underlying apps are less likely to be repackaged or cloned by adversaries for profit or phishing. Otherwise, we should inspect the apps whose UIs are neighbors of benign ones.

**Settings.** We conducted a UI#-based similar UI clustering analysis among 20,000 UIs randomly sampled from all UIs. Specifically, we use the agglomerative hierarchical clustering analysis (HCA). It performs well for other security applications, such as system behavior clustering from audit logs [81]. Initially, each UI# belongs to its cluster. HCA then iteratively calculates similarity scores between every two clusters and combines the nearest clusters until the maximum similarity is below a merge threshold. We calculate the semantic relationships between two UI# based on their Euclidean distance and evaluate the distance between two clusters by the Ward variance minimization [65]. To understand the internals of UI#, we further projected a multiple-dimensional UI# to a plane and visualize the cluster results. We adopt Uniform Manifold Approximation and Projection (UMAP) [53] for dimension reduction while preserving structural information among UI#.

**Results.** We show the visualization for UI#-based clustering analysis in Figure 18. In general, UI# with similar visual appearance group together while UI# with different appearances are separated with clear boundaries. For each cluster, we perform manual verification by reviewing UI screenshots and identifying the UI functionality (e.g., login, setting, and gamestart) as its descriptor. The dense clusters in the visualization usually indicate app cloning or repackaging. For example, we identify 64 repackaged “Radio” apps in repackages1, they have identical Java codes but try to bypass UI-based detection by changing UI background colors. We also identify several clusters related to sensitive UIs, including different UI patterns frequently used for login UIs (e.g., login#1). Besides, we pinpoint some other clusters with various high-level semantics, like setting, about, search, and gamestart. We illustrate more examples in Figure 20 (we provide more details in Appendix B). Finally, by identifying mundane UI clusters, including simple dialogs (a text and one or more buttons at the bottom) and splash screens (a single full-screen image), we filter out 3,724 (18.6%) UIs. After further deploying the filter on RePack-e, we reduce the time cost for UI similarity detection by 21.8%. The results show that
...by taking UIHASH as a complimentary detector, all the other four methods reduce their false negatives significantly (the second row): the drops are from 69% to 96%. It is also worth noticing that by combining UIHASH and other methods, we improve the ground truth in the RePack dataset. Table 9 shows four app pairs detected as “false” positives by UIHASH. By checking their similarity via other tools, we confirm that they are similar app pairs missed by RePack.

Finally, we combine LIBRadar—which detected the most similar apps among non-UI-based methods in our experiment, and UI-based baselines including GEMINI SCOPE and DROID EAGLE. Then, we examine whether UIHASH can strengthen the combination. Figure 19 shows that UIHASH and the combination helps each other detect 5.9% and 6.8% more similar apps in RePack-e. It suggests that when studying app similarity, utilizing a reasonable combination of multiple app features will outperform a single-feature-based detection.

6 Discussion

Necessity for Dynamic Analysis. Despite runtime UI collecting still has work to do in terms of performance and coverage [31], we choose dynamic UI information as the detection basis, considering: ① Active evasion attacks. According to...
our empirical study, adversaries can build layout files with advanced evasion techniques. Therefore, we cannot capture accurate UI visual appearances with only static layout trees; ② Runtime UI adjustments. It is a common practice to update UIs at runtime, and dynamic analysis works for obtaining UI properties with runtime updates; ③ Protected app source code. While benign apps adopt obfuscation [2] and packers [75] to protect codes from reverse-engineering, Android malware also increasingly leverages these techniques to evade detection [71]. Besides, Android apps could mask their runtime behaviors via anti-emulator techniques [7, 78]. As a countermeasure, we make UIHASH’s UI extraction module easy to deploy on both emulators and physical devices.

Defense against GUI Squatting Attack. UIHASH by design can detect spoofing UIs generated by GUI squatting attacks [14], the state-of-the-art method for developing phishing apps. To fabricate a UI, the attack specifies control attributes (e.g., width, height, and margins) with fixed values analyzed from the original screenshot. Each layout file resulting from this attack contains a single root ViewGroup with all the remaining controls as its children. This new layout structure usually differs significantly from the original one. Thus, GUI squatting attacks can easily evade tree-based detection. Nevertheless, UIHASH can detect these spoofing UIs based on UI# similarity, raising the bar of GUI squatting attacks.

Threat to Validity. Android apps can use FLAG_SECURE flag to prevent specified UIs from being captured. While we can gain root access and employ Xposed mod [4] to bypass this flag, we refrain from implementing such a solution, as apps involved in privacy-sensitive tasks may require device integrity (e.g., by Attestation API [23]). To extend UI capture capabilities, we can integrate UIHASH with heuristic GUI testing tools [27, 43]. UIHASH can also benefit from Chrome DevTools [21] when handling WebView.

For generality, we divide the screen equally when abstracting UI. It’s worth mentioning that optimizing grid parameters, such as employing unevenly distributed grids to prioritize specific screen areas, could improve UIHASH’s effectiveness in some scenarios. We leave this as a subject for future research.

7 Related Work

There has been active research inspecting app similarity based on different app features, including code [19, 28, 55, 73], static assets (e.g., images, icons, and XML files) [35, 44, 85] and UI. We categorize Android UI similarity detections into two groups based on their detection basis (static layout tree structure or runtime screenshot image). To our best knowledge, we are the first to discuss active evasion attacks from both tree-based and image-based aspects.

Tree-based Detections. Sun et al. propose DroidEagle [70], a system for similar UI detecting by measuring their tree edit distance (TED) or tree hashing. TED is also the metric for similarity measuring in other works such as FUIDroid [45] and RepDroid [79]. In addition to comparing trees, researchers also conduct pairwise similarity comparison on UI components [52], and feature engineering based on tree node attribute [67] or hierarchy structure [30] to compare UI similarity. Patil et al. [58] build a tree-based graph and train a graph matching network (GMN) for UI similarity detection.

All the above approaches take layout trees or hierarchy trees of UI as input. However, they do not use visual-appearance-related features like control visual type and size. Moreover, various active evasion attacks are available to modify trees, resulting in tree-based detection bypassing.

Screenshot-based Detections. Another research stream of UI-based similar app detection is based on extracting and comparing screenshot features. Perceptual hash (pHash) is a popular image descriptor [34, 49, 51, 56]. For example, Malisa [49] detects repackaging apps by calculating hash values of screenshots for a rough sampling and making further confirmation via layout trees.

Screenshot-based methods are not robust against simple UI modifications (e.g., changing background color or image contents) and adversarial noises (or tiles) [68, 69]. In contrast, UIHASH focuses on representing UI’s visual appearance features instead of making strict pixel-level comparison.

8 Conclusion

In this paper, we propose a novel approach, UIHASH, to detect similar Android UIs based on their visual features perceived by users. Aiming to sketch UI appearances, we abstract them into #-shaped grids, which we refer to as UI#. We then develop a neural network architecture to distill visual features from UI# representations and compare their similarity. We evaluate UIHASH using 52,390 real-world apps. The experiment results demonstrate UIHASH’s accuracy and robustness in detecting similar UIs. We also discover that similar UIs generated by active evasion attacks are already prevalent in the wild, invalidating existing tree-based and screenshot-based detection methods.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their insightful comments. This research/project is supported by the National Natural Science Foundation of China (No.62172027), the National Research Foundation, Singapore under its Industry Alignment Fund - Pre-positioning (IAF-PP) Funding Initiative, Zhejiang Provincial Natural Science Foundation of China (No.LZ23F020013), and CSC Scholarship. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.
References


[38] Li Li, Tegawendé F Bissyandé, and Jacques Klein. Rebooting research on detecting repackaged android apps: Literature review and benchmark. IEEE Transactions on Software Engineering, 2019.


[46] Jun Ma, Qing-Wei Sun, Chang Xu, and Xian-Ping Tao. Gridroid-an effective and efficient approach for android repackaging detection based on runtime graphical user interface. Journal of Computer Science and Technology, 2022.


Listing 3: Five ways to insert invisible controls (underlined).

```xml
<!--Make a view invisible in a view group-->
<LinearLayout ...
    android:layout_weight="0"
    android:id="@+id/d"
    android:alpha="0"/>
</LinearLayout>

<!--Make a view invisible by its attributes-->
<Button android:id="@+id/z" ...
    android:layout_toLeftOf="@id/z"
    android:layout_toRightOf="@id/z"/>
<Button android:id="@+id/z" ...
    android:layout_weight="1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
<Button android:id="@+id/z" ...
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
</LinearLayout>
```

A Effective Invasion Attack via Invisible Views

An effective way for adversaries to implement an active invasion attack is by inserting any number of invisible views (view groups or controls) into a layout tree, as Listing 3 shows. The imperceptible views contribute nothing to UI’s visual appearance but break down the tree-based detections. Note that many view attributes are available for such active invasion attacks. For example, the last Button in Listing 3 makes its size zero by position-related (instead of size-related) attributes.

B Design and Evaluation Details

Control Types for Reidentification. We illustrate the control types for reidentification in Table 10. All images in a specific type share specific image characteristics, except for the type Others, which consists of images shown on UIs and uncommon controls (e.g., rating bar).

Baselines for App Similarity. We provide more details of the baselines for calculating app similarity as follows: 1) LibRadar [47]: an accurate and anti-obfuscation tool for scanning third-party libraries. Given two library sets $L_a$ and $L_b$, their similarity score is given by $max(|L_a \cup L_b|/|L_a - L_b|, |L_a \cup L_b|/|L_b - L_a|)$. 2) SimiDroid [37]: an app comparison framework that yields method signatures and representations of statements at the source code level. 3) Manifest-based comparison: app comparison based on metadata (e.g., package), components, and permissions. We measure app similar-
Table 10: Control types for reidentification.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>areas to enter texts</td>
<td></td>
</tr>
<tr>
<td>Button</td>
<td>clickable rectangles or icons</td>
<td></td>
</tr>
<tr>
<td>CheckBox</td>
<td>small areas to check off</td>
<td></td>
</tr>
<tr>
<td>List</td>
<td>consecutive items in one template</td>
<td></td>
</tr>
<tr>
<td>Spinner</td>
<td>areas with a mark to show a drop list</td>
<td></td>
</tr>
<tr>
<td>Tab</td>
<td>banners for navigating</td>
<td></td>
</tr>
<tr>
<td>Text</td>
<td>areas to show text contents</td>
<td></td>
</tr>
<tr>
<td>Toggle</td>
<td>switches to make a boolean choice</td>
<td></td>
</tr>
<tr>
<td>Bar</td>
<td>bars to show progress or fix values</td>
<td></td>
</tr>
<tr>
<td>Others</td>
<td>controls that do not belong to the</td>
<td></td>
</tr>
</tbody>
</table>

Table 11: Gender and age demographics of participants.

<table>
<thead>
<tr>
<th>Age</th>
<th>&lt; 20</th>
<th>[20,40)</th>
<th>[40,60)</th>
<th>≥60</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>Male</td>
<td>15</td>
<td>120</td>
<td>25</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Female</td>
<td>12</td>
<td>56</td>
<td>54</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Others</td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

C User Study Details

We list the survey questions of our user study in Listing 4, and summarize participant demographics in Table 11. Our exit survey reveals that 81.2% of users have experienced UI changes in their frequently-used apps, while 8.5% do not notice any UI changes. Specifically, all users spending less than one hour per day on apps or have no frequently used apps report a lack of awareness of app UI changes.

A Demographic Survey

A1. What is your age? (age range or “prefe not to say”)
A2. What is your gender? (gender or “prefe not to say”)
A3. Are you skilled in using your smartphone?

B UI Similarity Evaluation

B1–B12. Would you perform login on this screen? (pic)
- Yes, I accept to log
- No, I reject logging in

B13. What factors prevented you from signing in? Select (zero to all) options in order of importance.
- Background color or image contents
- The position relationship of interface elements (such as buttons and input boxes)
- Texts on the screen
- Size of interface elements
- Style (such as fonts, borders, and colors) of interface elements (such as buttons and input boxes)
- Else: (please fill in the blank)

C Exit Survey

C1. How much time do you spend per day on apps?
- <1 h
- 1 h ≥ 2 h ≤ 4 h
- 4 h ≤ 8 h

C2. How many apps do you use frequently?
- 0–1
- 2–5
- 6–10
- 10+
- None / Hardly any

C3. How many apps have you used have UI changes with version updates?
- All / Almost all
- Most (60%–80%)
- About one-half (40%–60%)
- A few (20%–40%)
- None / Hardly any
- I do not concern about app UIs

C4. Are you good at manipulating your smartphone?
- Yes, I can manipulate my smartphone smoothly
- No, I am not good at manipulating my smartphone

Listing 4: User study questions.