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*
[♯]*Tianmushan Laboratory* [§]*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-

*Corresponding author. Email:maojian@buaa.edu.cn.

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 hierarchy instead of relying on static definitions of layout trees or screenshot images. Then, we extract and aggregate visual features in different screen regions. 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 apps[1]. The results demonstrate UIHASH's effectiveness and robustness in detecting similar UIs.

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

```
1  <LinearLayout ...>
2    <FrameLayout ...>
3      <TextView ... android:text="Account"/>
4      <EditText ... />
5    </FrameLayout>
6    <ImageView ... android:layout_marginTop="120dp"/>
7  </LinearLayout>
```

Listing 1: A simple Android layout tree example.

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.

Layout trees and UI screenshot images are widely used to measure UI similarity. Specifically, prior works detect tree or image similarity to search similar UI, assuming that adversaries only adopt simple and minor substitutions on UI components like texts, as they aim to save time on counterfeiting while keeping the counterfeit UI as close to the original as possible. Such UI spoofing attacks are called *lazy attack* [67, 70, 83]. Since the layout tree structure or images remain unchanged, analysts can still detect these similar UIs. However, we observe that the prior lazy attack assumption is unreliable in practical scenarios due to active evasions that apply to both tree-based and image-based approaches. There are two types of active evasion attacks:

Applying changes to layout trees to evade tree-based detection methods. The first way is to apply different ViewGroup types. For example, we list an example of two buttons arranged from left to right in Listing 2. As another example, using different view groups results in polymorphic tree structures and view attributes but visually similar UIs, as Figure 1 shows[2]. Inserting invisible controls is another way to obfuscate trees. Specifically, adversaries can add controls small enough to perceive, located out of the screen, or with a transparent color (we provide examples in Appendix A). Besides, as apps can change UI appearance dynamically (e.g., by initializing a view instance at runtime and inserting it into the current tree), UI appearances are not always consistent with the pre-defined layout trees. Therefore, static tree structures are insufficient to represent UI visual semantics.

Changing images or tuning image features to evade image-based detection methods. Images like app icons [35, 76] and

[2]More examples in https://developer.android.com/training/improving-layouts/optimizing-layout.

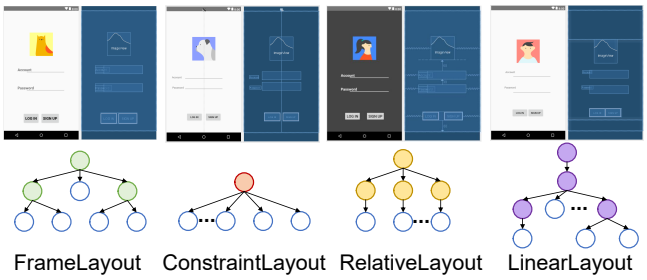


Figure 1: Similar UIs defined by different view groups. Nodes of different fill colors illustrate corresponding view groups.

```
<LinearLayout ...
  android:layout_gravity="center_horizontal">
  <Button .../>
  <Button .../>
</LinearLayout>
<androidx...ConstraintLayout ...>
  <Button ... app:
    layout_constraintEnd_toStartOf="@+id/g"/>
  <Button ... app:
    layout_constraintStart_toStartOf="@+id/g"/>
  <android.support.constraint.Guideline
    android:id="@+id/g" .../>
</androidx...ConstraintLayout>
```

Listing 2: Leveraging LinearLayout and ConstraintLayout with different view attributes to implement similar layouts.

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 UIHASH 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.

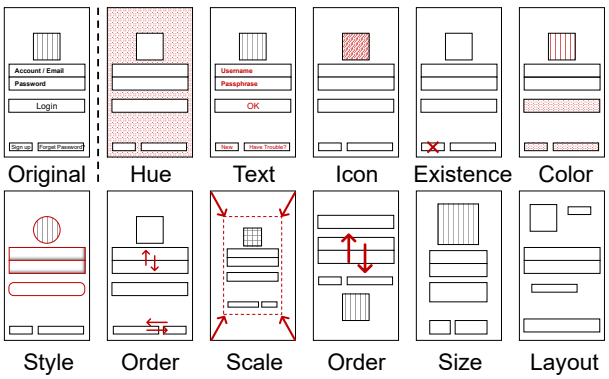


Figure 2: Modifications applied to an original login UI.

Feature	Score (1-5)	Rank
Control positions	3.77	1
Control sizes	3.41	2
Control styles	1.87	3
Background color or image contents	1.74	4
Texts on UI	1.45	5

Table 1: User ranking for features that prevent login.

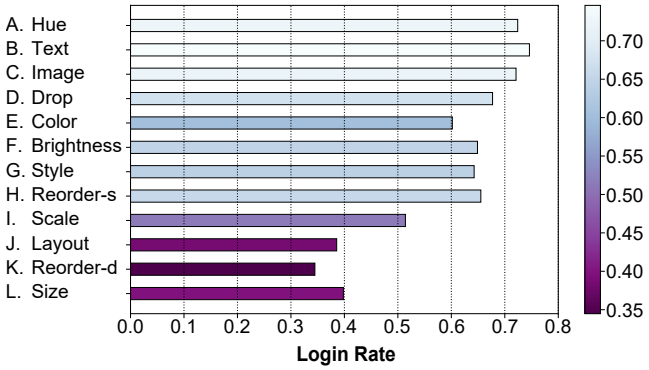


Figure 3: Login rates of modifications. H.Reorder-s and K.Reorder-d refers to reordering controls of the same type and different types, respectively.

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 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

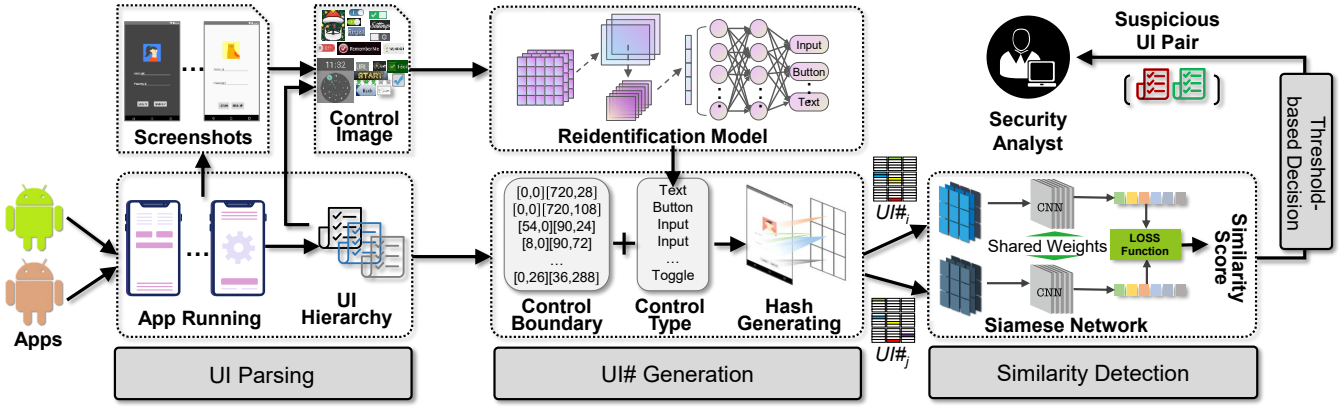


Figure 4: Overall architecture of UIHASH.

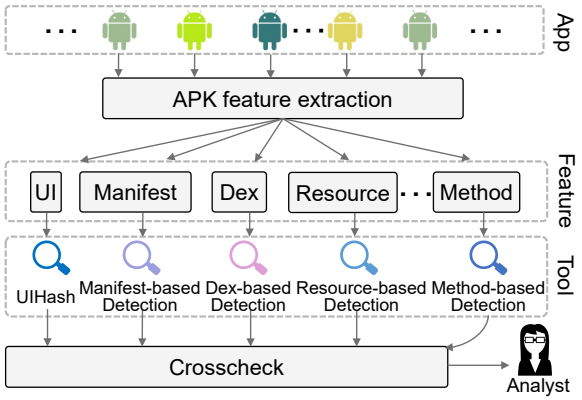


Figure 5: The platform for analyzing Android apps.

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.

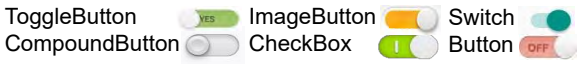


Figure 6: Six toggles implemented by different view types.

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 `ImageView` 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 to humans 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 gray-scale, 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# ℋ
3  controls ← controls in H
4  for i ← 1 to n × m do
5      cord^i ← coordinates of the i-th region
6      v_i ← zeros(shape = (1,10))
       // split channels by reidentified control types
7      for j ← 1 to 10 do
8          E^j ← {e ∈ controls | e.typeindex = j}
9          foreach e ∈ E^j do
10             cord^e ← coordinates of control e
11             iou_i^e ← GET_IOU(cord^e, cord^i)
               // drop the tiny exceed parts
12             if iou_i^e > threshold then
13                 v_i^j ← v_i^j + iou_i^e
           // revise hash values
14         v_i^j ← LOG_REV(v_i^j)
15 ℋ ← [v_1, v_2, ..., v_{n×m}] // semantic aggregation
```

types. We illustrate their example images in Appendix B.

Hashing UIs. we partition a UI into a grid with $n \times m$ regions of equal size. For each region, UI# encodes the size and type of its constituent controls into a k-dimension vector v_i, where $i = 1, 2, ..., n \times m$. Specifically, v_i^j, the value of the j-th dimension ($j = 1, 2, ..., 10$) in v_i, is derived based on the overlapping between the controls in the j-th type group and the i-th region. To measure the overlapping between a region and a control, we take Intersection over Union (IoU, also known as Jaccard index [63]) as the metric. Given A_r and A_c, namely the region area and the control area, we calculate IoU by $IoU = |A_r \cap A_c| / (|A_r \cup A_c| \times A_r)$.

Algorithm 1 shows the detailed steps of UI# generation, which includes three main steps. First, the algorithm splits the UI to $n \times m$ regions. Coordinates of the i-th region (located in the a-th row and b-column) are calculated as follows:

$$cord^i = (\lfloor S_h/n \rfloor \times (a-1), \lfloor S_w/m \rfloor \times (b-1)),$$

where $a = 1, 2, ..., n$, $b = 1, 2, ..., m$, and $i = (a-1) \times n + (b-1)$. S_h and S_w represent the height and width of the screen. The $n \times m$ grid also indicates the spatial relationships among regions, e.g., we can deduce that "the i-th region is at the top left of the $(i+m+1)$-th region".

The second step is to extract visual semantics for each grid region. The algorithm first calculates IoU between a region and a control group in Line 11. IoU, ranging from 0 to 1, describes how a UI control overlaps with a locality. Note that we do not necessarily encode each control's IoU. The value is meaningful only if it exceeds a threshold (Line 12); otherwise, we pad it with zeros. In particular, when a UI control lies its principal part in a grid region, UI# ignores the exceeded areas in other regions. The design makes it difficult for adversaries to confuse the algorithm by fine-tuning control sizes to relocate them in different regions without changing the UI layout, even with the knowledge of the algorithm's underlying logic. Besides, we adjust the original IoU metric since a control's visual impact on the overall UI layout is not linear to its size. For example, after a 50% scaling on the original UI, the login rate drops 25% instead of being cut in half. Accordingly, UI# focuses more on the existence of controls and compresses the impacts of their sizes. Inspired by log transformation, which is widely used in digital image processing for detail reveal and image enhancement via compressing the dynamic range of an image [60], we define a normalized log-based transformation to revise UI#'s components: $LOG\_REV(x) = [f(x) - f(0)]/[f(1) - f(0)]$, where $f(a) = log_\gamma(a + 0.01)$.

Finally, we aggregate the vectors—representing individual grid regions—to generate UI# (Line 15). Aggregating UI semantics, UI# captures a high-level layout characteristic of the overall UI.

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:

$$loss = \min_{\theta} \sum_{i=1}^{N} (sim(v_1^i, v_2^i) - y^i)^2,$$

where θ 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 $\{0.5, 0.6, 0.7, 0.75, 0.80, 0.85, 0.9\}$. In light of the best F1 score in our experimental environment, we show the results taking the initial learning rate as 0.001 ($\times 0.1$ 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.

5 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**. How efficient is UIHASH? (§ 5.4) **v**. 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

Approach	Precision	Recall	F1-score
Text-based detection [44, 85]	31.7%	83.0%	0.459
pHash [51, 56]	85.1%	79.7%	0.823
DROIDEAGLE [70] [1]	96.8%	86.5%	0.914
GEMINISCOPE [52]	95.6%	94.3%	0.949
UIHASH	**97.0%**	**99.8%**	**0.984**

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

Table 2: Effectiveness evaluation results.

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, 85], 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 GEMINISCOPE).

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 γ, which controls control size encoding, among $\{2, 5, 10\}$. (4) To determine the optimal grid size, we apply various grid sizes (1×1, 2×2, 3×3, 4×3, 5×5, and 10×10) 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

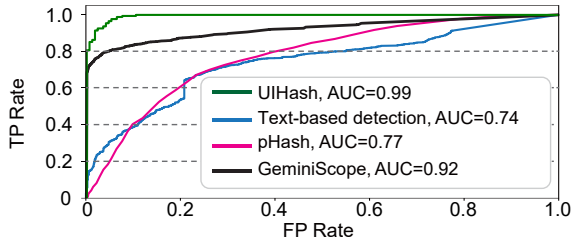


Figure 7: ROC curves of UI similarity detection methods.

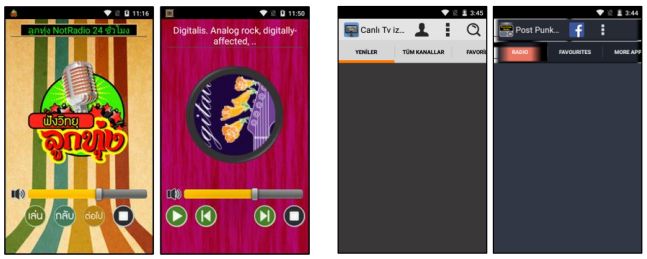


Figure 8: UI pairs that bypass screenshot-based approaches.

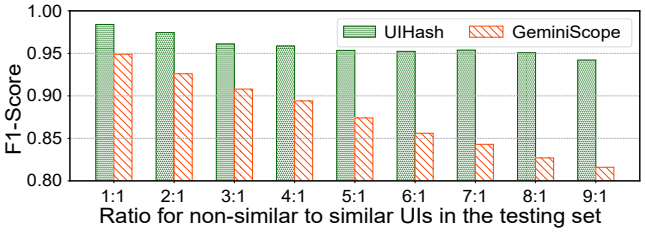


Figure 9: Effectiveness of similar UI detection under different ratios for non-similar to similar UIs in the testing set.

pHash [51, 56], reducing false negatives considerably. Figure 8 shows two UI pairs that bypass pHash but are detected by UIHASH as similar. Based on tree similarity, DROIDEAGLE does not consider that different layout trees might render similar UIs. In contrast, UIHASH achieves a 13.3% higher recall rate and a higher F1-score. Compared to GEMINISCOPE [52], which detects similar UIs based on pairwise View matching, UIHASH surpasses in all metrics. Considering that the amounts of non-similar and similar UIs are not equal in practice, we adapt the ratio for non-similar UIs to similar UIs in the testing set to explore its impact on similarity detection. Figure 9 illustrates that UIHASH's F1-scores against unbalanced datasets are lower than a balanced one, as prior research reveal [9, 59]. Nevertheless, its F1-score drops less than GEMINISCOPE's as the testing set becomes unbalanced. Specifically, even under a 9:1 ratio, UIHASH still performs better than all prior works. These results show that by relaxing the comparisons sticking to tree or control details, UIHASH gains better detection performance.

Figure 10a shows the ROC curves of UIHASH with different parameter settings. Results show that the grid-based UI# surpasses non-grid-based original features (p: 0.970 vs. 0.928, r: 0.998 vs. 0.881) in detecting similar UIs. Besides, UIHASH's effectiveness decreases without control re-identification. For size encoding, the base number (γ) only affects UIHASH slightly. However, if we remove the IoU revision in Algorithm 1 (Line 14), the effectiveness decreases. The experimental results demonstrate the essential roles grid-based encoding, control re-identification, and IoU revision play in UIHASH's UI similarity detection. For grid size selection, we illustrate the detection results of different gird sizes in Figure 10b. We observe that the size of 1×1 has the worst performance, especially on the recall rate. The reason is that one grid region cannot characterize the positions of UI controls and the layout of UI. The grid size of 3×3 achieves much better effectiveness. A possible reason is that it can represent the semantic of "center" vertically, horizontally, or both, which depicts a UI more accurately. In general, when we increase the size of a grid, the performance rises accordingly at first (e.g., 4×3 and 5×5) but turns down when the grid size becomes too large (e.g., 10×10). A reasonable explanation is that UI# loses its ability to *abstract* UI appearance and becomes sensitive to controls' exact position and size.

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 5×5 grid is used as a balanced trade-off between UI representation and tolerance.

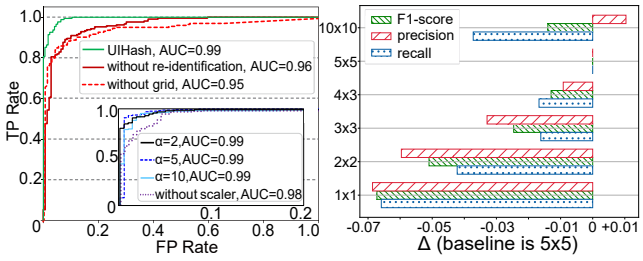
On the new testing set, UIHASH successfully flags all the similar UIs, demonstrating its transferability across different datasets. Figure 11 shows two similar UI pairs in the dataset. Despite changes in UIs, UIHASH can still capture the visual similarity within individual pairs.

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 RMVDROID, 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[3]. These UIs contain 280,750 visible controls, of which 95,025 controls are re-identified (confidence score ≥ 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.

[3]The rest 254 (2.8%) apps stop running as soon as we launch them.



(a) Grid encoding (b) Grid size

Figure 10: UIHASH's performance with different setups.

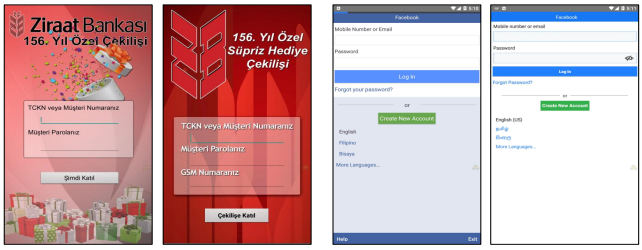


Figure 11: Similar UI samples in the new app set.

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.appsministry.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.

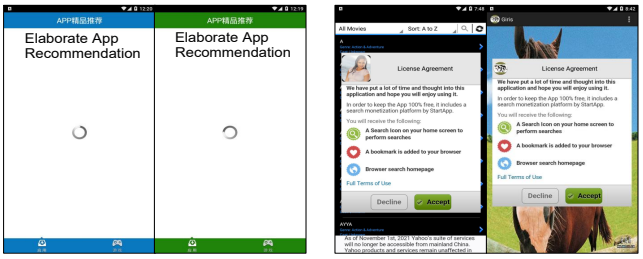


Figure 12: The two most frequently-used UIs in RmvDroid.

Market	App	Size	UI	Market	App	Size	UI
Kuan	1,231	80 GB	19,303	Mumayi	339	16 GB	2,331
2345	172	10 GB	826	Leyuan	6,649	245 GB	59,070
Vxat	224	12 GB	1,886	Appfun	348	10 GB	1,556

Table 3: Details of the recent app set.

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 counterfeit 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: `13AD2E7F`

[6]from *appfun.cn*, package: com.tencent.mmweix, CRC32: `AE8C46BD`

		Mumayi	Leyuan	2345	Vxat	Kuan
Kuan	T	4,160K	66,036K	1,112K	2,639K	10,981K
	S	494K	7,000K	140K	351K	1,378K
	S/T	11.9%	10.6%	12.6%	13.3%	12.6%
Vxat	T	253K	26,368K	36K	42K	392K[1]
	S	31K	2,874K	4.4K	3.6K	49K[1]
	S/T	12.3%	10.9%	12.2%	8.6%	12.4%[1]
2345	T	206K	32,960K	26K	84,284K[2]	98,546K[3]
	S	25K	3,691K	3.5K	7,754K[2]	10,446K[3]
	S/T	12.0%	11.2%	13.4%	9.2%[2]	10.6%[3]

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

Table 4: Total UI pairs (T) compared by UIHASH and detected similar UI pairs (S) between markets.

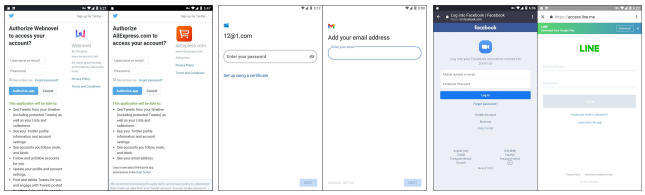
Figure 13: Three similar UI pairs we detect in recent apps.

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.



(a) Twitter SSO (b) Outlook-GMail (c) Facebook-Line

Figure 14: Similar UIs UIHASH detected in the top 100 apps.

UI Pair	Similar Rating (%)*					Average Rating
	1	2	3	4	5	
Facebook SSO	2.2	1.7	9.0	28.7	58.4	4.45
Twitter SSO	2.5	0	2.5	33.8	61.3	4.51
Outlook-GMail	7.5	8.8	5.0	38.8	40.0	3.95
Facebook-Line	16.3	25.6	21.3	26.7	10.1	2.89

* : 1: Not -, 2: Hardly -, 3: Hard to say, 4: Pretty -, 5: Very - (-: similar)

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

Active Evasion Attack	Ratio	Image	Tree	UIHASH
Empty text	17.9%	✓	✗	✓
Flexible use of view groups	17.7%	✓	✗	✓
Image changing	14.3%	✗	✓	✓
Off-screen control	2.9%	✓	✗	✓
Micro control	1.4%	✓	✗	✓
Hue modification	1.1%	✗	✓	✓

Table 6: Identified techniques for active evasion attacks and whether they are flagged as similar by different methods.

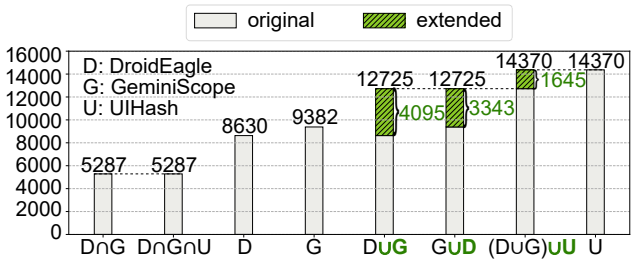


Figure 15: Similar app pairs (app pairs with similar UIs) flagged by different UI-based methods.

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 UIHASH. In addition to all the true positive UIs the combination detects, UIHASH 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 UIHASH'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, α, is the ratio of the TED to the minimum tree size (i.e., node number) in a pair of UIs. The vertical axis, β, presents the percentage of the UI pairs whose α is not greater than the corresponding α. 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

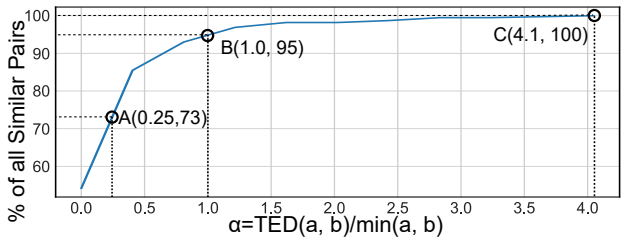


Figure 16: Tree edit distances of similar UI pairs.

learning-based detection evasion. Therefore, it is important to know how robust UIHASH is against adversarial examples.

Settings. We conducted two popular white box adversarial attacks via CLEVERHANS [57] on UIHASH 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 δ to each UI# element, where $|\delta| \leq \varepsilon$. Since attackers may adjust the counterfeit UI to evade UIHASH (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 ε is critical to attack for it determines the magnitude of perturbations, we explore different ε 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 ε. 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 $UI_{P \rightsquigarrow N}$, 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, UIHASH 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 UIHASH on multiple Android devices [36]. On the RmvDroid dataset, UIHASH 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

ε	Precision (P)		AUC (A)		$UI_{P\rightsquigarrow N}$ to all Pairs (N)	
	FGM	PGD	FGM	PGD	FGM	PGD
0.05	0.965	0.969	0.990	0.982	0.045	0.067
0.10	0.950	0.934	0.962	0.943	0.190	0.271
0.15	0.914	0.852	0.881	0.778	0.218	0.393
0.20	0.899	0.798	0.829	0.742	0.482	0.558

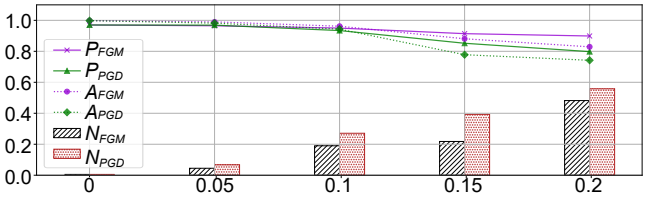


Figure 17: Evaluating UIHASH with RePack-e, against two adversarial attacks (FGM and PGD) with different ε.

Approach	Collect	Detect	Overall	F1	AUC
DROIDEAGLE	static	1.3 h (0.4)	9.6 h (1.9)	0.91	—
GEMINISCOPE	8.3 h (1.5)	1.4 h (0.4)	9.7 (1.9)	0.95	0.92
pHash-based	dynamic	0.2 h (0.05)	27.1h (5.3)	0.82	0.77
UIHASH	26.9 h (4.8)	1.5 h (0.5)	28.5 h (5.6)	0.98	0.99

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).

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

We list the time overhead[7] 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

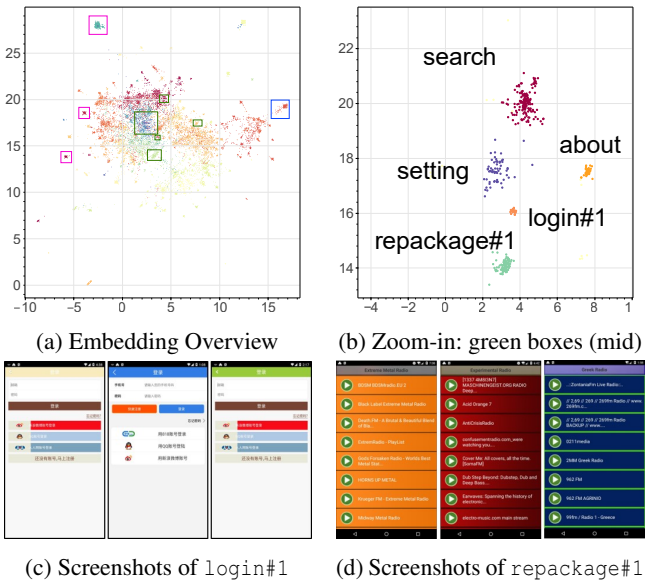
UI# can serve for clustering analysis besides pairwise UI similarity detection. Empirically, we can cluster similar UIs into groups, supporting two functionalities as follows:

- *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 `repackage#1`, 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

[7] 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 $\tau = 2$.



(a) Embedding Overview (b) Zoom-in: green boxes (mid)

(c) Screenshots of `login#1` (d) Screenshots of `repackage#1`

Figure 18: UMAP visualization of the UI#. (a) shows the embedding space of 20,000 UIs. Each dot denotes a UI#.

Method	LibRadar	SimiDroid	Manifest	Resource
$\frac{filtered*}{\text{UIHash } FP}$	42/48 (87.5%)	48/48 (100%)	48/48 (100%)	48/48 (100%)
$\frac{\text{UIHash } TP}{missed*}$	87/92 (94.6%)	461/526 (87.6%)	314/450 (69.8%)	6,039/6,643 (90.9%)

Table 8: UIHash works with other app similarity measures. The first row shows our false positive app pairs and pairs filtered by other methods, and the second row lists the missed app pairs of other methods and the pairs we detected.

the clustering analysis can mitigate the burden of searching for similar UIs on a large scale by removing mundane UIs.

5.5.2 Complementary Strength with Related Solutions

Settings. We take all the 15,297 similar app pairs in RePack as ground truth to evaluate the complementary strength of the solutions. In the Android app analysis platform (Figure 5), we deploy UIHash and four other methods, namely LibRadar [47], SimiDroid [37], Manifest-based, and Resource-based, to detect app similarity (We provide more details in Appendix B). We incorporate UIHash with the above methods in two ways. First, as a UI-based method, UIHash can detect similar apps that other methods fail to discover. Second, other methods can improve UIHash by filtering its false positives. Note that for UIHash, if any UI pair in two apps are similar, we flag the app pair as similar. For other detection approaches, two apps are potentially similar when their similarity score is not less than 0.5.

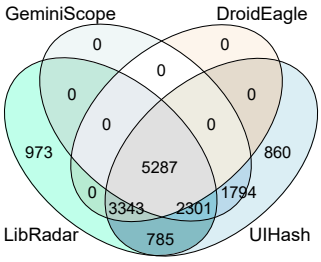
Results. Table 8 lists the results of corporating UIHash with other app similarity detection methods. As the first row shows, all the methods effectively filter out UIHash's false positives. Specifically, by incorporating SimiDroid, Manifest-based comparison, or Resource-based comparison, UIHash's precision reaches 1.0 as all the 48 false positives generated by UIHash are similar considering beyond-UI features. On the other hand, by taking UIHash as a complimentary detector, all the other four methods reduce their false negatives significantly (the second row): the drops are from 69.8% to 94.6%. 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 GeminiScope and DroidEagle. 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.

APK Pair	L	S	M	R	RP
pet.app46 imdroid.dina.hamel	1	0.99	0.88	0.44	✗
com...floppyspace com...stone	0.63	0.54	1	0.27	✗
yong...player com...stone	0.50	0.72	0.2	0.52	✗
osa.app21 gextreme.app33	1	0.99	0.88	0.50	✗

[1] **L**, **S**, **M** and **R** denotes LibRadar, SimiDroid, Manifest-based, and Resource-based comparisons, respectively.
[2] **RP** indicates whether the pair is disclosed in RePack.

Table 9: New similar app pairs in RePack found by UIHash and their similarity scores given by other approaches.



(G ∪ D ∪ L) ∪ U	U ∪ (G ∪ D ∪ L)
14,483 **+860** (+5.9%)	14,370 **+973** (+6.8%)

Figure 19: Similar app pairs detected by GeminiScope (G), DroidEagle (D), LibRadar (L), and UIHash (U).

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 Vadility. Android apps can use `FLAG_SECURE` flag to prevent specified UIs from being captured. While we can gain root access and employ an 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 benifit 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

[1] minicap. https://github.com/openstf/minicap.

[2] ProGuard. https://github.com/Guardsquare/proguard.

[3] VirusTotal. https://www.virustotal.com/.

[4] Xposed-Disable-FLAG_SECURE. https://github.com/VarunS2002/Xposed-Disable-FLAG_SECURE.

[5] Devdatta Akhawe, Warren He, Zhiwei Li, Reza Moazzezi, and Dawn Song. Clickjacking revisited: A perceptual view of UI security. In *USENIX Workshop on Offensive Technologies*, 2014.

[6] Alessandro Aldini, Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. Detection of repackaged mobile applications through a collaborative approach. *Concurrency and Computation: Practice and Experience*, 2015.

[7] Mohammed K Alzaylaee, Suleiman Y Yerima, and Sakir Sezer. Emulator vs real phone: Android malware detection using machine learning. In *ACM International Workshop on Security and Privacy Analytics*, 2017.

[8] Benjamin Andow, Adwait Nadkarni, Blake Bassett, William Enck, and Tao Xie. A study of grayware on google play. In *IEEE Security and Privacy Workshops*, 2016.

[9] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. Dos and don'ts of machine learning in computer security. In *USENIX Security Symposium*, 2022.

[10] Jimmy Ba and Diederik Kingma. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

[11] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. What the app is that? Deception and countermeasures in the android user interface. In *IEEE Symposium on Security and Privacy*, 2015.

[12] Thomas Brewster. Alleged 'nazi' android fbi ransomware mastermind arrested in russia. https://www.forbes.com/sites/thomasbrewster/2015/04/13/alleged-nazi-android-fbi-ransomware-mastermind-arrested-in-russia/.

[13] Weicheng Cao, Chunqiu Xia, Sai Teja Peddinti, David Lie, Nina Taft, and Lisa M Austin. A large scale study of user behavior, expectations and engagement with android permissions. In *USENIX Security Symposium*, 2021.

[14] Sen Chen, Lingling Fan, Chunyang Chen, Minhui Xue, Yang Liu, and Lihua Xu. GUI-squatting attack: Automated generation of android phishing apps. *IEEE Transactions on Dependable and Secure Computing*, 2019.

[15] Thomas Davies and Ashweeni Beeharee. The case of the missed icon: Change blindness on mobile devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2012.

[16] Mohamed El-Serngawy and Chamseddine Talhi. Captureme: Attacking the user credential in mobile banking applications. In *IEEE International Symposium on Recent Advances of Trust, Security and Privacy in Computing and Communications*, 2015.

[17] James H Elder and Richard M Goldberg. Ecological statistics of gestalt laws for the perceptual organization of contours. *Journal of Vision*, 2002.

[18] F-Secure. Trojan:Android/Airpush. https://www.f-secure.com/v-descs/trojan-android-airpush.shtml.

[19] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. Stack overflow considered harmful? the impact of copy&paste on android application security. In *IEEE Symposium on Security and Privacy*, 2017.

[20] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[21] Google. Chrome DevTools. https://developer.chrome.com/docs/devtools.

[22] Google. Optimizing layout hierarchies. https://developer.android.com/training/improving-layouts/optimizing-layout.

[23] Google. SafetyNet Attestation API. https://developer.android.com/training/safetynet/attestation.

[24] Google. UIAutomator. https://developer.android.com/training/testing/ui-automator.

[25] Google. View. https://developer.android.com/reference/android/view/View.

[26] Google. ViewGroup. https://developer.android.com/reference/android/view/ViewGroup.

[27] Jiaqi Guo, Shuyue Li, Jian-Guang Lou, Zijiang Yang, and Ting Liu. Sara: self-replay augmented record and replay for android in industrial cases. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019.

[28] Shifu Hou, Yanfang Ye, Yangqiu Song, and Melih Abdulhayoglu. Hindroid: An intelligent android malware detection system based on structured heterogeneous information network. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017.

[29] Hang Hu, Steve TK Jan, Yang Wang, and Gang Wang. Assessing Browser-level Defense against IDN-based Phishing. In *USENIX Security Symposium*, 2021.

[30] Yangyu Hu, Guosheng Xu, Bowen Zhang, Kun Lai, Guoai Xu, and Miao Zhang. Robust app clone detection based on similarity of UI structure. *IEEE Access*, 2020.

[31] Chun-Ying Huang, Ching-Hsiang Chiu, Chih-Hung Lin, and Han-Wei Tzeng. Code coverage measurement for android dynamic analysis tools. In *IEEE International Conference on Mobile Services*, 2015.

[32] Zhao Huang and Morad Benyoucef. A systematic literature review of mobile application usability: addressing the design perspective. *Universal Access in the Information Society*, 2023.

[33] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, 2015.

[34] Sibei Jiao, Yao Cheng, Lingyun Ying, Purui Su, and Dengguo Feng. A rapid and scalable method for android application repackaging detection. In *International Conference on Information Security Practice and Experience*, 2015.

[35] Naveen Karunanayake, Jathushan Rajasegaran, Ashanie Gunathillake, Suranga Seneviratne, and Guillaume Jourjon. A multi-modal neural embeddings approach for detecting mobile counterfeit apps: A case study on google play store. *IEEE Transactions on Mobile Computing*, 2020.

[36] Taeyeon Ki, Chang Min Park, Karthik Dantu, Steven Y Ko, and Lukasz Ziarek. Mimic: UI compatibility testing system for android apps. In *International Conference on Software Engineering*. IEEE, 2019.

[37] Li Li, Tegawendé F Bissyandé, and Jacques Klein. SimiDroid: Identifying and explaining similarities in android apps. In *IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2017.

[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.

[39] Jiahao Liu, Jun Zeng, Xiang Wang, Kaihang Ji, and Zhenkai Liang. TeLL: log level suggestions via modeling multi-level code block information. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022.

[40] Jiahao Liu, Jun Zeng, Xiang Wang, and Zhenkai Liang. Learning graph-based code representations for source-level functional similarity detection. In *International Conference on Software Engineering*, 2023.

[41] Thomas F. Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. Learning design semantics for mobile apps. In *ACM Symposium on User Interface Software and Technology*, 2018.

[42] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European Conference on Computer Vision*, 2016.

[43] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. Fill in the blank: Context-aware automated text input generation for mobile gui testing. In *International Conference on Software Engineering*, 2023.

[44] Fang Lyu, Yapin Lin, Junfeng Yang, and Junhai Zhou. SUIDroid: An efficient hardening-resilient approach to android app clone detection. In *IEEE International Conference on Trust, Security, and Privacy in Computing and Communications*, 2016.

[45] Fang Lyu, Yaping Lin, and Junfeng Yang. An efficient and packing-resilient two-phase android cloned application detection approach. *Mobile Information Systems*, 2017.

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

[47] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. Libradar: fast and accurate detection of third-party libraries in android apps. In *International Conference on Software Engineering Companion*, 2016.

[48] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[49] Luka Malisa. *Security of User Interfaces: Attacks and Countermeasures*. PhD thesis, ETH Zurich, 2017.

[50] Luka Malisa, Kari Kostiainen, and Srdjan Capkun. Detecting mobile application spoofing attacks by leveraging user visual similarity perception. In *ACM Conference on Data and Application Security and Privacy*, 2017.

[51] Luka Malisa, Kari Kostiainen, Michael Och, and Srdjan Capkun. Mobile application impersonation detection using dynamic user interface extraction. In *European Symposium on Research in Computer Security*, 2016.

[52] Jian Mao, Jingdong Bian, Hanjun Ma, Yaoqi Jia, Zhenkai Liang, and Xuxian Jiang. Robust detection of android ui similarity. In *IEEE International Conference on Communications*, 2018.

[53] Leland McInnes, John Healy, Nathaniel Saul, and Lukas Grossberger. Umap: Uniform manifold approximation and projection. *The Journal of Open Source Software*, 2018.

[54] Microsoft. PUA:AndroidOS/Kuguo.A!MTB. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=PUA:AndroidOS/Kuguo.A!MTB.

[55] Omid Mirzaei, Guillermo Suarez-Tangil, Jose M. de Fuentes, Juan Tapiador, and Gianluca Stringhini. Andrensemble: Leveraging api ensembles to characterize android malware families. In *ACM ASIA Conference on Computer and Communications Security*, 2019.

[56] Thanh Nguyen, Jeffrey Mcdonald, William Glisson, and Todd Andel. Detecting repackaged android applications using perceptual hashing. In *Hawaii International Conference on System Sciences*, 2020.

[57] Nicolas Papernot, Fartash Faghri, Nicholas Carlini, Ian Goodfellow, Reuben Feinman, Alexey Kurakin, Cihang Xie, Yash Sharma, Tom Brown, Aurko Roy, et al. Technical report on the cleverhans v2.1.0 adversarial examples library. *arXiv preprint arXiv:1610.00768*, 2018.

[58] Akshay Gadi Patil, Manyi Li, Matthew Fisher, Manolis Savva, and Hao Zhang. Layoutgmn: Neural graph matching for structural layout similarity. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021.

[59] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. TESSERACT: Eliminating experimental bias in malware classification across space and time. In *USENIX Security Symposium*, 2019.

[60] Yunliang Qi, Zhen Yang, Wenhao Sun, Meng Lou, Jing Lian, Wenwei Zhao, Xiangyu Deng, and Yide Ma. A comprehensive overview of image enhancement techniques. *Archives of Computational Methods in Engineering*, 2021.

[61] Sajal Rastogi, Kriti Bhushan, and BB Gupta. Android applications repackaging detection techniques for smartphone devices. *Procedia Computer Science*, 2016.

[62] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2016.

[63] Hamid Rezatofighi, Nathan Tsoi, JunYoung Gwak, Amir Sadeghian, Ian Reid, and Silvio Savarese. Generalized intersection over union: A metric and a loss for bounding box regression. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019.

[64] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 2015.

[65] SciPy. API reference:scipy.cluster.hierarchy.linkage. https://docs.scipy.org/doc.

[66] Yuru Shao, Xiapu Luo, Chenxiong Qian, Pengfei Zhu, and Lei Zhang. Towards a scalable resource-driven approach for detecting repackaged android applications. In *Annual Computer Security Applications Conference*, 2014.

[67] Charlie Soh, Hee Beng Kuan Tan, Yauhen Leanidavich Arnatovich, and Lipo Wang. Detecting clones in android applications through analyzing user interfaces. In *IEEE/ACM International Conference on Program Comprehension*, 2015.

[68] Dawn Song, Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Florian Tramer, Atul Prakash, and Tadayoshi Kohno. Physical adversarial examples for object detectors. In *USENIX Workshop on Offensive Technologies*, 2018.

[69] Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai. One pixel attack for fooling deep neural networks. *IEEE Transactions on Evolutionary Computation*, 2019.

[70] Mingshen Sun, Mengmeng Li, and John CS Lui. Droideagle: seamless detection of visually similar android apps. In *ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2015.

[71] Symantec. Five ways android malware is becoming more resilient. https://www.symantec.com/connect/blogs/five-ways-android-malware-becoming-more-resilient.

[72] OpenCV team. OpenCV. https://opencv.org/.

[73] Ke Tian, Danfeng Yao, Barbara G. Ryder, Gang Tan, and Guojun Peng. Detection of repackaged android malware with code-heterogeneity features. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[74] Dejan Todorovic. Gestalt principles. *Scholarpedia*, 2008.

[75] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G. Bringas. Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *IEEE Symposium on Security and Privacy*, 2015.

[76] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 2014.

[77] Haoyu Wang, Junjun Si, Hao Li, and Yao Guo. Rmvdroid: towards a reliable android malware dataset with app metadata. In *International Conference on Mining Software Repositories*, 2019.

[78] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART. In *USENIX Security Symposium*, 2017.

[79] Shengtao Yue, Weizan Feng, Jun Ma, Yanyan Jiang, Xianping Tao, Chang Xu, and Jian Lu. RepDroid: an automated tool for Android application repackaging detection. In *IEEE/ACM International Conference on Program Comprehension*, 2017.

[80] Christoph Zauner. Implementation and benchmarking of perceptual image hash functions. 2010.

[81] Jun Zeng, Zheng Leong Chua, Yinfang Chen, Kaihang Ji, Zhenkai Liang, and Jian Mao. WATSON: Abstracting behaviors from audit logs via aggregation of contextual semantics. In *Network and Distributed System Security*, 2021.

[82] Jun Zeng, Xiang Wang, Jiahao Liu, Yinfang Chen, Zhenkai Liang, Tat-Seng Chua, and Zheng Leong Chua. ShadeWatcher: Recommendation-guided cyber threat analysis using system audit records. In *IEEE Symposium on Security and Privacy*, 2022.

[83] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. ViewDroid: Towards obfuscation-resilient mobile application repackaging detection. In *ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2014.

```
1  <!--Make a view invisible by its attributes-->
2  <Button android:id="@+id/c" ... android:
       layout_marginTop=5000dp/>
3  <Button android:id="@+id/d" android:alpha="0"/>
4  <Button android:id="@+id/e" android:padding=6000dp>
5  <!--Make a view invisible in a view group-->
6  <LinearLayout ...>
7    <TextView android:layout_weight="0" ...
8      android:layout_width="match_parent"
9      android:layout_height="match_parent"/>
10   <Button android:id="@+id/f"
11     android:layout_weight="1"
12     android:layout_width="match_parent"
13     android:layout_height="match_parent"/>
14 </LinearLayout>
15 <RelativeLayout ...>
16   <Button android:id="@+id/g" ...
17     android:layout_toLeftOf="@id/z"
18     android:layout_toRightOf="@id/z"/>
19   <Button android:id="@+id/z" .../>
20 </RelativeLayout>
```

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

[84] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 1989.

[85] Yury Zhauniarovich, Olga Gadyatskaya, Bruno Crispo, Francesco La Spina, and Ermanno Moser. FSquaDRA: Fast detection of repackaged applications. In *Annual IFIP WG 11.3 Working Conference*, 2014.

A Active 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.

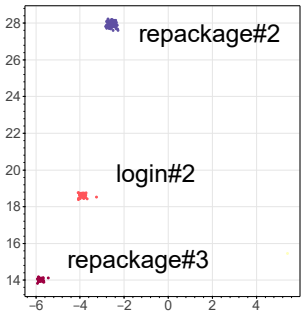
B Design and Evaluation Details

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).

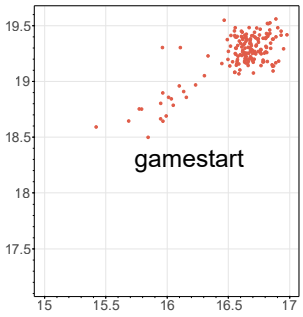
Baselines for App Similarity. We provide more details of the baselines for calculating app similarity as follows: 1) LibRadar [47]: an accurate and anti-obfuscation tool for scanning third-party libraries. Given two library set L_a and L_b, their similarity score is given by $\max\{[L_a \cup L_b]/[L_a - L_b], [L_a \cup L_b]/[L_b - L_a]\}$. 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-

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

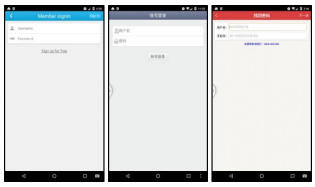
Table 10: Control types for reidentification.



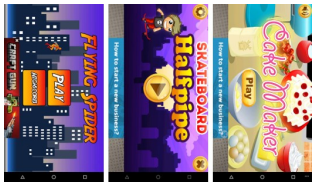
(a) Zoom-in: pink boxes (left)



(b) Zoom-in: blue box (right)



(c) Screenshots of `login#2`



(d) Screenshots of `gamestart`

Figure 20: UI samples for different clusters.

ity as that of LibRadar. 4) Resource-based comparison: we apply pHash [80] to find similar images and icons between two apps and search for identical static resources of other types by checksum comparison. Given the resource file sets of two apps F_a and F_b, the number of similar images P_s, and the number of other identical resource files, R_s, the similarity score is given by $\max\{[P_s+R_s]/[F_a-F_b],[P_s+R_s]/[F_b-F_a]\}$.

More Details of UI# Clusters. We illustrate more zoom-in plots and UI screenshots for the clustering results in Figure 18a. We find another *login* UI pattern as Figure 20c shows, which is different to `login#1` (Figure 18c). Besides, UIs in `repackage#2` and `repackage#3` correspond to two repackaging bundles (with 24 "India Newspapers" apps and 18 "Software Stills" apps, respectively). For UIs in the `gamestart` cluster, despite different background images, colors, texts, and fonts, they are still grouped as they share a similar appearance to achieve the same design goal (starting game and showing an advertisement at the bottom of the screen).

Age	<20	$[20,40)$	$[40,60)$	≥ 60	Others
Male	15	120	25	2	5
Female	12	56	54	5	2
Others	4	8	0	3	8

Table 11: Gender and age demographics of participants.

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 "perfer not to say")
A2. What is your gender? (gender or "perfer 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 in ○ 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 phone apps?
○ <1 h ○ ≥ 1 h but <2 h ○ ≥ 2 h but <4 h ○ ≥ 4 h
C2. How many apps do you use frequently?
○ 0 ○ 1–3 ○ 4–6 ○ 7–8 ○ >8
C3. How many apps you have used have UI changes with version updates?
○ All / Almost all (80%–100%)
○ Most (60%–80%)
○ About one–half (40%–60%)
○ A few (20%–40%)
○ None / Hardly any (0–20%)
○ I do not concern about app UIs
C4. Are you good at manipulating your smartphone?
○ No, I am not good at manipulating my smartphone
○ Yes, I can manipulate my smartphone smoothly

Listing 4: User study questions.