Understanding Illicit UI in iOS apps Through Hidden UI Analysis

Yeonjoon Lee[†1], Xueqiang Wang[†2], Xiaojing Liao[2], and XiaoFeng Wang[2]

[1]*Hanyang University,* [2]*Indiana University Bloomington*

Abstract—In *Chameleon* apps, benign UIs are displayed during Apple App vetting while their hidden potentially-harmful illicit UIs (*PHI-UI*) are revealed once they reached App Store. In this paper, we report the *first* systematic study on iOS *Chameleon* apps, which sheds light on a largely overlooked threat that the illicit activities are launched solely based on UI. Our research employed CHAMELEON-HUNTER, a new static analysis approach that determines the suspiciousness of a *PHI-UI* leveraging the *semantic features* generated from iOS app UI and metadata. The approach is based on the observation that *PHI-UI* not only is structurally hidden but also has notable semantic inconsistency with the benign UI. Our evaluation shows that CHAMELEON-HUNTER is highly effective, achieving 92.6% precision and 94.7% recall. From 28K Apple App Store apps, we found 142 new *Chameleon* apps, which were confirmed and promptly removed by Apple. Our work reveals that Chameleon apps can easily bypass the App store vetting and conduct a set of suspicious activities including collecting users' private information, swindling money with fake monetary services, and leading the user to a pirated app store.

Index Terms—Measurement of malware and spam, Mobile security, Evasive apps, Underground services.

1 INTRODUCTION

Apple App Store is well-known for its stringent App Review. They not only continuously update their review guidelines to stay on top of the trending malware but also quickly ban services or apps that are suspicious from their App Store. Further, they reject any apps that use private API which provides access to user's sensitive information; recently, they even blocked hot patching frameworks (e.g., JSPatch) because they can be used to dynamically deliver and invoke code after App Review. The effectiveness of their review process is well reflected in the low rate of suspicious apps on App Store [24]. To circumvent this security check, it has been reported [56] that iOS Trojans have been increasingly used to infiltrate the Apple App Store, through embedding stealthy illicit user interfaces (UI) in innocent-looking iOS apps. So far, little has been done to systematically discover and analyze such hidden-UI based Trojans, not to mention any effort to understand their impact.

Chameleons on iOS. We consider apps that conduct suspicious activities based on a hidden UI as *Chameleon* apps and conduct the first systematic study on them. As illustrated in Figure 1, the *Chameleon* app provides benign UIs (i.e., music list and song player) when under Apple's review, but then unveil its potential harmful UIs (i.e., crowdturfing [39] UIs that allow users to make money by performing *app downloading* tasks, which boosts the apps' ranking in a manner forbidden by Apple) once it reaches a user device. We believe that such apps are able to bypass App Review because they do not include obvious malicious signatures (e.g., private APIs), and hide their illicit UI behind the benign UI. The security implications of such apps, capable of carrying **p**otentially-**h**armful **i**llicit UI (*PHI-UI*), are significant, which, however, have never been investigated before.

Chameleon apps consist of two fundamental features: (1) *PHI-UI* delivers illicit services that significantly deviate from apps' declaration [51], which may cause potential damage to mobile users; (2) *PHI-UI* cannot be triggered by normal UI interactions, but by certain conditions (e.g., commands from C2 server, environmental states). Such illicit services, once infiltrate App Store, could cause harm to users, invading their privacy as well as swindling money out of their pocket. Examples include conducting sensitive information collection through phishing *PHI-UI*, suggesting to purchase premium services or fake lottery through illicit payment services, leading users to pirated third-party app stores, displaying pornographic contents.

Building a detection tool for *Chameleon* apps is non-trivial. First, instead of directly invoking privileged operations (e.g., collecting user's phone number with private APIs), *PHI-UI* uses operations, such as UI rendering, that commonly appear in benign UIs. Second, *PHI-UI* is stealthy since it is under the cover of benign UIs and only gets triggered when specific conditions are satisfied. Such characteristics place *Chameleon* apps beyond the reach of existing detection systems [23], [18] which rely on a known or predefined set of malware signatures (e.g., private API, API sequence).

CHAMELEON-HUNTER. In this paper, we make the first step towards automatic detection of *Chameleon* apps. Our detection tool, CHAMELEON-HUNTER, is based on the following key observations: a *PHI-UI* of a *Chameleon* app is not only structurally 'hidden' behind a benign UI but also shows notable semantic inconsistency with the benign one. Based on the observations, we build the following two main components of CHAMELEON-HUNTER: *Structure Miner* and *Semantic Analyzer*. The *Structure Miner* identifies *hidden* view controllers (*VC*) based on their unique structural features; e.g., not allowing the users to leave a suspicious UI. In our work, we call the hidden VCs checkpoints since they are potential *PHI-UI* which requires further investigation. Given the checkpoint VCs, the *Semantic Analyzer* measures the semantic inconsistency between each checkpoint and benign UI to determine whether a checkpoint VC indeed is a *PHI-UI*. However, developing such tool for iOS is non-trivial as static analysis on its apps accompanies various technical challenges. As we are the first

[†]*The two lead authors contributed equally to this work.*

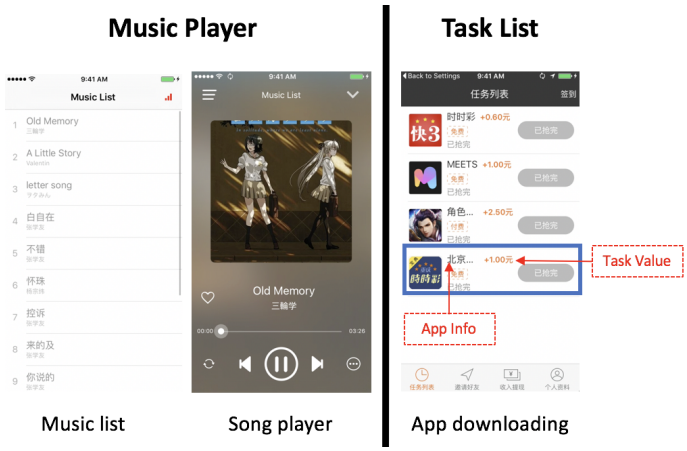

Fig. 1: A music player with hidden crowdturfing UIs.

to analyze navigation patterns of iOS apps by retrieving the VC hierarchy (i.e., UI graph) from an iOS app, there neither is a public documentation we can learn from nor tools we can utilize. To address these challenges, we carefully reverse engineered iOS apps (i.e., binary, compiled UI layout files) and the UI builder to understand how iOS app binary and UI layout files work together to present an app UI. Below, we outline the designs of the *Structure Miner* and *Semantic Analyzer*.

Structure Miner. By manually analyzing how *PHI-UI*s are hidden, we found that navigations from benign UIs to *PHI-UI*s indeed have distinctive features. For instance, *PHI-UI* is often preemptive, taking over the whole screen as a root UI and thus eliminating display of benign UI; *PHI-UI* also restricts users from navigating back to benign UI. Based on such navigation features, we design a set of rules (see Section 4.3) to identify structurally hidden UIs.

To analyze UI navigation features, we first build a UI graph which consists of VCs, VC properties, and transitions between them. iOS app UIs are found in both binary and UI layout files. To handle UI in binary, we build a call graph (CG) based on *Capstone* [4] to depict UI transitions defined by the iOS APIs. For UI layout files (e.g., storyboard, nib, plist) we built a tool that reestablishes the UIs and transitions between them (e.g., restore transition from *Segues*). Based on the *CG* and the reestablished UI information, we create a *labeled view controller graph (LVCG)*. From the *LVCG*, the *Structure Miner* identifies semantic checkpoint VCs (structurally hidden), based on the UI navigation features discussed above.

Semantic Analyzer. The *Semantic Analyzer* focuses on determining whether the given checkpoint VC indeed is a *PHI-UI*. As discussed, *PHI-UI* and benign UI show notable semantic inconsistency. More specifically, the texts in *PHI-UI* (checkpoint VC and its children VC) are irrelevant to their connected UIs (e.g., sibling or parent UI); for example, an illicit lottery UI (i.e., checkpoint) carrying words such as "money, gamble", is suspicious when it appears with a music player UI involving "genre, equalizer, playlist". Based on such observation, the *Semantic Analyzer* generates semantic features and further uses them to determine *PHI-UI* with a supervised learning classifier. The unique UI design of iOS platform makes it challenging to collect informative text from resource files and app binary. To resolve the issue, we adopted a series of noise-removing strategies (e.g, word embeddings, affinity propagation, tf-idf). Besides, a group of intuitive features, including cohesion (consistency of words within UIs) and distance (separation of words between UIs), are designed to infer the semantic difference between benign UI and *PHI-UI*. In our study, CHAMELEON-HUNTER achieved a precision of 92.6% and a recall of 94.7%.

Discoveries. Our large-scale measurement study on over 28K iOS apps is the first step towards systematically uncovering illicit UIs (*PHI-UI*s) on iOS: 142 apps were found to hide illicit UIs that perform suspicious activities, such as providing a crowdsourcing platform for malicious purposes (e.g., e-commerce reputation manipulation), delivering unauthorized content to users (e.g, third-party app stores, fake news), collecting sensitive information, and fraudulent online advertising, etc. (see Section 6.1). Surprisingly, the ranking data indicates that *Chameleon* apps are impacting a large number of users: they were discovered from the official App Store, as well as appearing on the leaderboards, with 14 apps ranked among the top 100 of their categories.

In addition, we found that triggering suspicious UIs of *Chameleon* apps is extremely difficult: besides traditional evasion techniques adopted on Android platform, such as remote commands from C2 servers, new techniques are observed to trigger iOS apps based on complicated conditions. Particularly, the malicious crowdsourcing apps not only detect whether they have passed Apple's review, but also have a triggering condition that involves user interaction and collusion with a malicious website; Such techniques were not reported on any other platforms, and bring new challenges to app vetting. Also our study shows that even in Apple App Store, repackaging and app clones are widely adopted in *Chameleon* apps. *Chameleon* developers leverage a series of techniques to keep their apps from being removed from App Store (e.g., repackaging existing apps and hiding the same *PHI-UI* under multiple bundle ids). On top of that, to infect more users, *Chameleon* developers were found to promote their apps through multiple channels, which includes in-app promotion and pyramid (or referral) scheme, providing users with benefits for referring *Chameleon* apps to their friends. Also, we found that *PHI-UI* is an in-demand product in the underground market: e.g., cybercriminals pay hundreds of dollars for *Chameleon* apps development. *Chameleon* apps are clearly harmful to users and prohibited by Apple: we disclosed our findings to Apple, which appreciated our work and promptly removed most of the *Chameleon* apps from App Store; also upon Apple's request, we provided a list of fingerprints of detected apps to assist them in eliminating the risk of the repackaged *Chameleon* apps. The video demos and other materials are provided on a private website [5].

Contributions. The contributions of the paper are outlined as follows:

Semantic-based detection of Chameleons apps. We designed an effective technique which detects iOS *Chameleon* apps, based on the observation that illicit UIs not only are structurally hidden but also have semantic inconsistency from other parts of the app. Building such a tool is non-trivial as it requires serious effort on analyzing iOS binary as well as demystifying iOS UI system.

New findings. Our study shows that *Chameleon* apps indeed exist in the wild and are involved in various illicit services. Most importantly, our study sheds light on a new attack vector that has long been ignored: conducting suspicious activities based solely on UI.

2 BACKGROUND

iOS UI design. The UIs of an iOS app include view, view controller (VC) and data: *view* defines the UI elements to be displayed (e.g.,

button, image, and shape). *Data* is the information delivered through the defined UI elements; and a *VC* controls both views and their data to present a UI. All the VCs of an app and their relations, which describe the transitions between different UIs, form a *VC hierarchy* with its root (called *anchor*) being the initial VC of the app or the VC launched by the iOS object *AppDelegate*. Implementing a VC hierarchy can be done using either VC transition APIs (e.g., pushViewController:animated:), or storyboard [19], a visual tool in the Xcode interface builder. In the storyboard, a sequence of scenes are used to represent VCs, and they are connected by *segue objects* which describe transitions between VCs. iOS employs *layout files* (a.k.a., nib files) to implement UIs, which can be generated using storyboard.

Over a VC hierarchy, developers commonly define two kinds of transitions between a pair of VCs: *Modal* and *Push*. A modal VC does not contain any navigation bar or tab bar, and is used when developers create outgoing connections between two UIs. To present a modal VC, the developer can directly use APIs (e.g., presentViewController:animated:completion:), or define a modal segue object [20] in a storyboard. An API needs to be called in order to dismiss such a modal VC. On the other hand, *Push* uses a navigation interface for VC transitions. Selecting an item in the VC pushes a new VC onscreen, thereby hiding the previous VC. Tapping the back button in the navigation bar removes the top VC and reveals the background VC. More specifically, developers can display the view of a VC by pushing it to the navigation stack using the pushViewController:animated: API, or define a push segue in a storyboard. In the meantime, tapping the back button will pop up the top VC from the navigation stack and makes the new top displayed.

In our research, we observe that hidden crowdturfing UIs exhibits conditionally triggered navigation patterns in an app's VC hierarchy, including multiple root VCs as entry UIs, entry VC not triggered by the users nor dismissed by itself, etc. (Section 4.3).

Natural language processing The semantic information our system relies on is automatically extracted from UIs using Natural Language Processing (NLP). Below we briefly introduce the key NLP techniques used in our research.

Word embedding Word Embedding is an NLP technique that maps text (words or phrases) to high-dimensional vectors. Such a mapping can be done in different ways, e.g., using the continual bag-of-words model or the skip-gram technique to analyze the context in which the words show up. Such a vector representation ensures that synonyms are given similar vectors and antonyms are mapped to different vectors. Tools such as Word2vec [54] could be used to generate such vectors. Word2vec takes a corpus of text (e.g., Wikipedia dataset) as inputs, and assigns a vector to each unique word in the corpus by training a neural network. In our study, we leverage Word2vec to quantify the semantic similarity between the words based on the cosine distance of their vectors.

Topic model for keyword extraction Topic model is a statistical model for finding the abstract "topics" of a document, and topic modeling is a common text-mining tool for discovering keywords from corpora. Among various topic modeling approaches, Latent Dirichlet Allocation (LDA) [10] is one of the most popular methods. The basic idea is that documents are represented as random mixtures over latent topics, where a topic is characterized by a distribution over words, and the statistically significant words are selected to represent the topic. In our study, we leverage the LDA implementation of Stanford Topic Modeling Toolbox [30] for keyword extraction.

Fig. 2: Pseudocode and VC hierarchies of a Chameleon app.

Threat Model. In our research, we consider an adversary who tries to publish Chameleon apps, iOS apps carrying hidden illicit content, on Apple App Store. In order to circumvent app vetting, the adversary hides suspicious UIs behind a fully-functional legitimate app. Such suspicious UIs present unsolicited contents and require no additional app capabilities. The adversary has partial knowledge about app vetting process (e.g., guidelines) and is able to compose different Chameleon apps and submit them for review. Suspicious apps that use private APIs or side-loading are out of the scope of this paper. Also, in our research, we only cover native iOS apps. The cross-platform framework (e.g., react native) based apps, which are built using different languages (e.g., javascript), are out of the scope of this work.

3 A Motivating Example

To illustrate our approach, we summarize the characteristics of Chameleon apps through a motivating example, the com.sohouer.music app (see Figure 1). com.sohouer.music is a Chameleon that carries both the benign music player UI and the PHI-UI that allows users to make money by performing crowdturfing (e.g., app downloading) tasks. The former is displayed during App Review, while the latter is hidden until the app runs on the user's device.

The triggering condition and underlying VC hierarchies (Benign and PHI-UI) are shown in Figure 2. Once the app starts to run, it first checks whether its PHI-UI has ever been displayed on the current device by examining a variable [theCache getInvited], which is set once the app receives a Custom URL Scheme (i.e., sohouermusic://). The result of this check determines which execution path the app follows. One such path is [AppDelegate loadMusicUI], which provides users with a Music Player. The other is [AppDelegate loadPhiUI], which leads users to a Task List that allow them to download apps. As shown in Figure 2, both paths come with a VC hierarchy. For instance, the Music Player component allocates a new UIMusicListViewController and sets it as the root VC, and further displays MusicViewController upon a user click. Similarly, for the Task List, the root VC is set to SHEMainViewController, and the other PHI-UIs (e.g., SOHTaskListViewController) would be shown following the transition.

Compared to other previous malware that hides suspicious operations (e.g., sensitive APIs) through evasive techniques, this Chameleon example delivers illicit content by PHI-UI. In the following, we summarize the characteristics of Chameleon apps which we utilize for detection in Section 4.

Fig. 3: Overview of CHAMELEON-HUNTER.

Distinctive Navigation Patterns. First, to serve different purposes, a Chameleon app tends to own multiple VCs that serve as the app entries (i.e., SHEMainViewController and MusicListViewController). Second, the entry of a PHI-UI is not reachable by normal user interactions and instead triggered by certain conditions. Third, once the user enters the PHI-UI, it does not allow the user to navigate back to the main screen or benign UI.

Clear Semantic Difference. The different functionalities of benign UI and PHI-UI are well reflected in the words extracted from the VC hierarchies. As indicated by the com.sohouer.music case, the benign UI contains Music Player related words, such as {album, singer, shuffle, song, music, radio}, while the PHI-UIs are filled with content such as {task, cash, earn, withdrawl, join, pay, reward}.

4 DESIGN AND IMPLEMENTATION

Here we elaborate on the design and implementation of a new technique for identifying apps with hidden UIs. We begin with an overview of the idea behind CHAMELEON-HUNTER, and then present the design details of each component.

4.1 Overview

Design and Architecture. The design of CHAMELEON-HUNTER, illustrated in Figure 3, is built based on the following key observations: a PHI-UI of a Chameleon app not only is structurally hidden behind a benign UI but also has notable semantic inconsistency compared to the benign one. As shown, the tool consists of three modules: Data Preparation, Structure Miner and Semantic Analyzer. Their responsibilities are as follows: The Data Preparation module crawls data (app files, app metadata) and decrypts apps. The Structure Miner focuses on searching for suspicious checkpoint VCs through UI navigation pattern analysis. The Semantic Analyzer generates semantic features for each checkpoint VCs and uses those features with a supervised learning classifier to predict whether the suspicious checkpoint VCs are indeed a PHI-UI; for checkpoint VCs that lack text data, the Semantic Analyzer determines its suspiciousness by analyzing the web UI.

How it works. Here we walk through the workflow of the system. As the Data Preparation module finishes crawling the metadata and the app file from App Store, the metadata is sent to the Semantic Analyzer for text processing whereas the app file is decrypted and disassembled into app binary, UI layout files and resource files. To conduct UI navigation pattern analysis, the Structure Miner retrieves UI data (i.e., text on UI, transitions) from the app binary and UI layout files, and creates a labeled view controller graph (LVCG); LVCG is a comprehensive UI graph that consists of view controllers (VC), UI transitions defined between the VCs and the texts (and attributes) of the VCs. Based on the LVCG, the Structure Miner searches for checkpoint VCs that matches the suspicious navigation patterns discussed in Section 3 (e.g., multiple entry UIs, reachability of UI, etc.). The Semantic Analyzer then processes the texts of the checkpoint VCs and the metadata of the app to generate semantic features. Lastly, the generated features are passed to the classifier for PHI-UI detection. For the checkpoint VCs with a dominant web UI (e.g., webview), because of the insufficient amount of text data, the Semantic Analyzer leverages URL scanner (i.e., VirusTotal) results and determines whether the given checkpoint VCs are PHI-UIs.

4.2 Data Preparation

To find Chameleon apps in the wild, we collected 28,625 iOS apps from Apple App Store with a Sikuli-based [57] crawler. After scanning the entire iOS app list from iTunes Preview website [38], we selected the apps that were updated after Jan. 1, 2016; those apps were chosen because Chameleon is a newly emerging threat and recently updated apps tend to affect more active users. The difficulty is that Apple is quick in disabling aggressive crawlers' accounts. To avoid violating their policy, we slowly crawled apps for 6 month switching in between 35 accounts every 900 apps we downloaded. Besides downloading the apps, we also collected their metadata from iTunes Preview website and App Annie [3]. We include data from App Annie as they provide additional information such as app version history, app ranking, etc.

4.3 Structure Miner

The Structure Miner is designed to identify the VCs with suspicious navigation patterns from an app's disassembled code and UI layout files. Examples of such patterns include two different main UIs, as discovered from com.sohouer.music, and the UI that can only be invoked by a specific network or other events, not directly by the user, indicating the potential presence of evasive behaviors. To discover such patterns, we first construct a VC hierarchy in the form of an LVCG through analyzing the app's binary and retrieving UIs from the UI layout files to identify their corresponding VCs and establish their transition relations among them. Then, from the LVCG, we search for predefined conditionally triggered UIs and mark those having these UIs as checkpoint VCs for further analysis.

LVCG. LVCG is a directed graph as shown in Figure 2, in which each node is a VC and each directed edge describes a transition from one VC (corresponding to a UI) to another.

> **Definition 1.** An LVCG is a directed graph $G = (V, E, \ell)$ over a node label space $\mathcal{L}$, where:
> 1) V is a node set, with each node being a VC;
> 2) Edge set $E \subseteq V \times V$ is a set of transitions between VCs;
> 3) Node labeling function $\ell : V \to \mathcal{L}$ marks each node with its UI properties and text data. Each node is given four property labels: {entry, user, url, block}. Table 1 shows the definition of each property and the corresponding method names.

LVCG construction. The construction of an LVCG requires both an app's binary and its UI layout files. This is because the VC of a UI is in the code and even the UI itself can be programmed through APIs (e.g., initWithFrame: API in UIView) so becoming part of the

TABLE 1: LVCG node properties and their corresponding method names.

Property	De nition	Method/Class Names
entry	root VC	setRootViewController:
user	VC triggered by a user interaction	addTarget:action:forControlEvents:
url	VC rendering web content	openURL:, UIWebViewController
block	VC that blocks further user navigation	dismissViewControllerAnimated:completion: setNavigationBarHidden:animated:

VC, and in the meantime, all the UIs built through storyboard can only be found in the layout les, including the transitions between them. To address this complexity, CHAMELEON-HUNTER builds two LVCGs, one from the binary and the other from the layout les, before combining them together.

Speci cally, on the binary code, we look for system VC class names (e.g., UIViewController) and method names (e.g., setNavigationBarHidden), which help identify individual VCs and their properties (see Table 1). Then we track the data ows from a VC to another to recover the transitions between the detected VCs. For this purpose, our approach rst maps the addresses in the binary code to symbols (e.g., class name, method name) using a binary analysis tool Capstone [4], and then uses a set of targeted system VC class names (e.g., UIViewController) and method names (e.g., setNavigationBarHidden) to recognize VCs and their properties (e.g., entry) from the symbols. After that, the Structure Miner performs a data- ow analysis using an implementation similar to the prior techniques [23], [18], to connect the transition APIs (performSegueWithIdenti er:sender:) discovered in a VC to another one, the transition target.

To construct a LVCG on the layout les under the storyboard folder generated by Apple's interface builder, we need to extract VCs and VC transitions from the les. The former can be found from the storyboard plist le that includes the mappings from VC names to the obfuscated names of nib les. The latter is recorded by the nib les, each of which carries a subset of a VC's properties, e.g., the types of some elements (such as botton, textbox, etc.) and the transitions between VCs.

Our approach directly recovers VCs from the plist le and further detects each VC's nib les from the mappings it records. More challenging here, however, is to identify the transitions between the VCs, since objects included in a nib le are undocumented. To enable the Structure Miner to interpret the le, we reverse-engineered part of its format relevant to the transition and content extraction. Speci cally, we started from the interface builder, through which one can de ne one or multiple scenes to represent a UI and a Segue to describe a transition. Through a differential analysis, we compared the compiled nib les with and without a speci c transition to pinpoint the nib objects corresponding to different Seguetypes (e.g., push, modal, unwind), such as ClassSwapper. From such objects, the Structure Miner is then able to collect the transitioning data, in the form of src, dst, type, etc. This allows us to restore the recorded transition information and build up the LVCG of an app.

Given the LVCGs generated from the binary and the layout les, our approach automatically combines them together, based on the relations between the VCs on these graphs: particularly, when a transition is found from a VC in the layout to the one de ned in the code, two LVCGs can then be linked together through this VC pair. On the combined LVCG, further we remove the dead VCs introduced by the part of libraries and other shared code not used by an app. To this end, our approach performs a test to nd out all the VCs that cannot be reached from the app's entry points (such as AppDelegate, the initial VC of the main storyboard) and drops them. In this way, we remove 1,053,161 dead VCs (55.4%) from the 28,625 iOS apps we collect (see Section 4.2).

Mining Semantic Checkpoints. Based on the discussed LVCG, we formalize the concept of Semantic Checkpoints as follows:

De nition 2. A semantic checkpoint is a vertex v that meets either of the following conditions:

$$
\begin{aligned}
&1)\ (v)[\text{`user}^0] = F \wedge (v)[\text{`entry}^0] = T;\\
&2)\ (v)[\text{`user}^0] = F \wedge (v)[\text{`block}^0] = T;\\
&3)\ (v)[\text{`user}^0] = F \wedge (v)[\text{`url}^0] = T;\\
&4)\ \forall e \in E; e[\text{`dst}^0] = v \rightarrow e[\text{`src}^0] = \text{AppDelegate}.
\end{aligned}
$$

Where: T is True, F is False

Our manual analysis on known Chameleon apps indicates that all PHI-UIs fall into above categories: 1) an entry VC which is not triggered by the user, but by other external events (e.g., network), as the SHEMainViewController in Figure 2, 2) a VC which is not triggered by the user and blocks further user navigation (e.g., by hiding its navigation bar); this VC is suspicious as the user is trapped and not allowed to navigation to other UIs, 3) a non-user triggered VC with web content (e.g., webview); different from legitimate apps that trigger a webview by user interactions, Chameleon apps often pop up a webview with illicit content when speci c conditions are met (see Section 6), 4) a VC only triggered during startup by AppDelegate, to take over the entire UI hierarchy, Chameleon developers could present a PHI-UI before any benign UI is rendered (i.e., com.sohouer.music, the PHI-UI is created in AppDelegate). As measured in Section 5.2, these rules not only greatly reduce detection noise by narrowing down the set of suspicious VCs, but also achieve a good coverage.

4.4 Semantic Analyzer

The Semantic Analyzer focuses on determining PHI-UI: As shown in Figure 3, it uses a classi er which takes series of semantic features as input. The semantic features are generated as follows: Firstly, it compares the semantics of PHI-UI and benign UI of (checkpoint VCs, initial UI shown at app launch time). Secondly, it utilizes the app's metadata (e.g., app's description, version history). To compare the semantics of PHI-UI and benign UI, the Semantic Analyzer begins with de ning two view controller (VC) sets: VC_l, VC_r; PHI-UI is represented by VC_l which consists of the checkpoint VC and its children VCs, while the benign UI is denoted by VC_r which includes the VCs of the sibling-UI of checkpoint VCs or the initial UI. Note that the VCs can be collected by analyzing the LVCG. The Semantic Analyzer then collects text data from VC_l, VC_r and preprocesses them with the other strings collected from the app's metadata and generate semantic features from them. Finally, the semantic features are used by the classi er to determine Chameleon apps. In the following, we discuss the implementation details of Semantic Analyzer

Preprocessing Text. The collected text data are further processed to solve a series of problems; i.e., text data in multiple languages, handling text from code, noise in text data, determining the word set that represents the dominant (or main) topic of VCs.

Translation The text data collected from apps are in multiple languages: Chinese, Japanese, Russian, etc. Analyzing such text is non-trivial; e.g., unlike English, Chinese is written without delimiters, and thus requires segmentation. To solve this problem, we use open source tools [31], [29] to split words and then translate them into English.

Text Tokenization The text data extracted from binary code needs to be preprocessed due to their special format (e.g., uploadMedia). We handle such text by designing regular expressions that cover common naming conventions; e.g., CamelCase style, hyphen ("-") or underscore ("_") separated variables, etc.

Noise Removal Removing the noise from the collected text data is an important step as the effectiveness of semantic analysis (i.e., understanding the topic of UI) depends on how well the noisy data is removed. In this study, we remove not only common stop words (e.g., NLTK stop words), but also the frequent words from iOS frameworks and programming languages. Specifically, we analyzed 74 framework-libraries of iOS 8.2.1, and collected text from sections such as cfstring and __objc_methname. Further, from the collected text data, we removed words that appeared in more than 1=5 frameworks utilizing the IDF value. In addition, we remove program language and debugging related text; for example, "socket", "connection", "memory", "allocation" are not related to the functionality (i.e., topic) of the hosting VC CheckoutViewController. To address such problem, from [6], we manually selected and blacklisted 1,031 frequent words that cover Objective-C, Swift, and Javascript.

Determining core word list of VCs Despite removing noisy words, word lists of VCs are not in line with the main topic of a VCs. For example, consider the words in a music list VC, whose main topic is Music; song titles such as "Can't Buy Me Love", or "Butterfly Effect" deviate from the Music topic. To better understand the main topic (or semantics) of VCs, we further get the "core" of the word list (W) by utilizing tf-idf, Word2vec, and affinity propagation (AP): First, we use Word2vec to create a vector representation (V) of the words appearing in W; then, to measure the importance of each word, we utilize tf-idf to assign a weight for each word; further, through AP, we divide W into several clusters based on closeness in its vector space and sort the clusters according to their sum of weights. Finally, we collect the words that considerably represent the main topic by selecting the most significant cluster (from the sorted clusters). In the following, we discuss how each feature is generated.

Feature Generation In this part, we describe 4 categories of 18 features used for PHI-UI detection. The features are selected and categorized based on the following observations: 1) the semantic inconsistency between PHI-UI and the benign UI, 2) the semantic inconsistency between PHI-UI and the app's description, 3) the potentially harmful illicit behaviors of Chameleon apps have similar goals (e.g., phishing, advertisement, adult content) with known malicious activities (e.g., email spam), 4) the metadata (e.g., version history) of Chameleon apps show dissimilar patterns from normal apps. Below we discuss each feature in detail.

Comparing against benign UI The semantic inconsistency of PHI-UI and benign UI are well reflected in their word lists (W_l; W_r) and vectors (V_l; V_r). We begin with, analyzing how closely VC_l and VC_r are related to each other by measuring the cohesion of the two vectors; the lower the cohesion, the higher the similarity of words in W. The cohesion of a V is measured by Weighted Sum of Squared Error (W-SSE): $\sum_{v \in V} \phi(v) \cdot \text{cosine}(v, c)^2$, where ϕ is a tf-idf based weight function, and c centroid for vectors in V. W-SSE considers not only within-cluster distance but also the importance of each word. In addition, we measure how distinct and well-separated V_l is from V_r. The difficulty is that words in PHI-UI and benign UI may appear in each other's to a limited extent; UI transitions and the related UI components may exist when the two UI are connected. To smooth out impacts, we use weighted average distance (WAD) as it considers the word frequency. WAD is calculated as follows: $\sum_{l \in W_l} \phi(l) \cdot \phi(r) \cdot \text{cosine}(l, r)^2 / \sum_{l \in W_l} \phi(l) \cdot \phi(r)$, where $l \in V_l$ and $r \in V_r$. We also take into account the sum of weights of $W_l \setminus W_r$; the larger the intersection, the higher the similarity of the word lists. Lastly, we compare W_l and W_r with 14 explicit topics (e.g., Arts, Business, Sports, Shopping) of ODP-239 dataset [12]. We use such approach because measuring the distance between words in a vector space is not enough; long-distance in vector space indicates dissimilar contexts in the corpus (in our case, Wikipedia), but does not necessarily mean they serve different topics. We generate the feature by calculating the difference between two probability distributions (i.e., $prob(W_l)$ and $prob(W_r)$).

Comparing against description While the description of a Chameleon app includes accurate description of its legitimate functionality, it does not state the illicit contents of the app. Based on such observation, we measure the separation between vectors of UI ($\{V_l; V_r\}$) and description (V_{desc}). The challenge is that developers often describe multiple or even unrelated app features in description; due to that, WAD approach is not suitable. Instead, we design an asymmetrical distance approach, extended single-linkage (ESL) as follows:

$$ESL(A; B) = \sum_{a \in A} \phi(a) \min_{b \in B} \text{cosine}(a, b)^2 \Big/ \sum_{a \in A} \phi(a)$$

For each vector $a \in A$, ESL finds the nearest vector in B, and then calculates an average weighted distance. In this case, $ESL(V_{l \text{ or } r}; V_{desc})$ gives a good estimation of whether words within a UI are similar (or close) to a part of the app's description.

Comparing against known malicious behavior A critical challenge for PHI-UI detection is the lack of ground truth; the only source we can refer to is the App Store Review Guidelines [21] as it describes the illicit behaviors not allowed in the App Store. The problem is that the guidelines are often abstract and does not contain concrete information (e.g., a benchmarking dataset). Surprisingly, we found that in many aspects, the guidelines are consistent with email spam; e.g., spam emails provide unsolicited promotional services for some products, whereas review guidelines do not allow apps primarily made up of advertisements; fraudsters often trick a user into entering personal information through spam emails, and the guidelines state that such behavior is not allowed. Based on such findings, we compare spam activities with the word list, we collected 304 unique spam trigger words [35] (e.g., easy cash, job offer) and measure the ESL between vectors of $\{V_l; V_r\}$ and spam (V_{spam}).

Distinctive patterns in metadata We also generate features utilizing the version history of an app on App Store as it reflects the number of times an app was inspected by Apple; we assume that the higher the version number, the less likely an app is a Chameleon app. Moreover, we check whether the author type of an app is individual or enterprise. Chameleon apps tend to be developed using an individual account; enterprise developer accounts are difficult to apply and expensive. We carefully assume that developers would

TABLE 2: Feature vector examples.

VC_l	VC_r	Comparing against benign UI						Comparing against description				Comparing against known malicious behaviors				Distinctive patterns in metadata				Chameleon
		F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}	F_{18}	
PlayCatVC	LEQEarnVC	0.43	0.78	0.91	0.0	0.91	N	0.31	0.0	0.85	0.0	0.27	0.0	0.35	0.18	1.0	Individual	N	N	Y
MusicVC	SHECleanVC	0.55	0.64	0.81	0.0	0.83	N	0.23	0.14	0.51	0.0	0.33	0.0	0.24	0.14	1.5	Individual	N	N	Y
ReadingVC	ReportVC	0.62	0.79	0.29	0.57	0.36	Y	0.19	0.38	0.34	0.14	0.38	0.0	0.40	0.0	4.3	Enterprise	N	N	N
SFMContainer	SFMHomeVC	0.69	0.55	0.28	0.71	0.36	Y	0.36	0.0	0.25	0.14	0.34	0.0	0.28	0.0	4.3	Enterprise	N	N	N

F_1 = cohesion(V_l), F_2 = cohesion(V_r), F_3 = WAD (V_l; V_r), F_4 = weight ($W_l \setminus W_r$), F_5 = dist (prob(W_l); prob(W_r));
F_6 = index (max (prob(W_l))) == index (max (prob(W_r))), F_7 = ESL (V_l; V_{desc}) F_8 = weight ($W_l \setminus W_{desc}$), F_9 = ESL (V_r; V_{desc});
F_{10} = weight ($W_r \setminus W_{desc}$), F_{11} = ESL (V_l; V_{spam}), F_{12} = weight ($W_l \setminus W_{spam}$), F_{13} = ESL (V_r; V_{spam}), F_{14} = weight ($W_r \setminus W_{spam}$);
F_{15} = version, F_{16} = author type, F_{17} = in-app purchase(VC_l), F_{18} = in-app purchase(VC_r)

less likely risk (account may get disabled) their enterprise account for developing Chameleon apps.

Classification of PHI-UI. To predict whether the suspicious checkpoint VCs are indeed PHI-UIs, the Semantic Analyzer utilizes the generated features with a supervised learning classifier. Specifically, the Semantic Analyzer uses an SVM classifier as it tends to be resilient to over fitting. Table 2 shows feature vector examples. The VC_l and VC_r represent the UIs being compared. The features are sorted according to their categories. F_1–F_6 show the semantic comparison result obtained by comparing potential PHI-UI against benign UI; F_7–F_{10} indicate whether potential PHI-UI is consistent with app's description; F_{11}–F_{14} indicate whether PHI-UI are related to known malicious activities; F_{15}–F_{18} reveal whether the app's metadata patterns are similar to those of Chameleon apps.

The structurally hidden checkpoint VCs identified by Structure Miner may include VCs that deliver contents mostly through WebView, those VCs that 1) lack text data, 2) embed URLs are difficult to analyze without manual analysis (e.g., web pages require authentication, URL only works under specific parameters). We handle such cases by relying on VirusTotal [17]: we submit the URL and use the results to measure the suspiciousness of the web content. The URLs are collected by tracking APIs including openURL: loadRequest:. Such VCs are considered as Chameleon since suspicious contents are indeed delivered through structurally hidden UIs.

5 Evaluation

5.1 Experiment Setup

App Collection. We collected 28K apps, covering 25 app categories. Release dates of these apps range from Jul. 11, 2008 to Jul. 19, 2017, all of which have been updated after Jan. 1, 2016. Duplicates are removed based on bundle id and version. To decrypt these apps, we ran Clutch on an iPhone 5s (8.1.2) and iPhone 5c (9.0.1). Apps that failed to decrypt are not included in the app set.

Confirmed Apps. Our study is based on 17 manually collected apps; the apps were collected from VirusTotal and Jssz [1], a website which provides apps with illicit features that are against Apple's guidelines (e.g., spam forums, earn money). All 17 confirmed apps were manually verified to structurally hide PHI-UI behind a benign UI.

Labeled and Unknown Dataset. To create the labeled-bad-set, from the 17 confirmed cases, we manually collected 41 PHI-UI VCs. However, 41 PHI-UI VCs are not enough; to address the problem we collected more pairs of VCs from the LVCG. As discussed in Section 4, Chameleon-Hunter generates features by comparing a checkpoint VC to its parents (p_c) or siblings (s_c). We expand the bad set as follows: for $\{c; p_c\}$ pair, we follow outgoing LVCG edges of c to construct a forward spanning tree (FST), and incoming edges of p_c for a backward spanning tree (BST); each pair of VCs from FST and BST are used to generate a feature vector. Through such approach $\{c; p_c\}$ gets enlarged to $|FST| \times |BST|$ vectors; the bad set of 17 apps are increased to 399 vectors. Another simple approach is to use over-sampling (e.g., SMOTE) to augment the bad set by including nearby vectors. However, our experiments show that given the sparseness of existing bad set, SMOTE tends to push the decision boundary towards unwanted directions and leads to more false positives. On the other hand, to create the labeled-good-set, we first randomly selected 800 from 28k apps and generated 996 feature vectors out of 800 apps. Lastly, for the unknown dataset, we generate 35,696 feature vectors from the 34,639 checkpoint VCs identified by the Structure Miner.

TABLE 3: Types of semantic checkpoints.

# of VC / UI	Type 1	Type 2	Type 3	Type 4
Benign UI	31	7	0	4
PHI-UI	19	2	18	2

5.2 Effectiveness and Performance

Checkpoint Mining. From 28K apps, Structure Miner identified 34,639 (1.7%) semantic checkpoints out of 2,005,030 VCs; while Structure Miner cannot pinpoint PHI-UI, it significantly decreases the number of VCs that needs to be analyzed by the Semantic Analyzer. In addition, we ran it on 17 confirmed Chameleon apps; 83 VCs were identified as checkpoints which included all 41 VCs manually confirmed as PHI-UI. In Table 3 we further show how the 83 checkpoint VCs distribute over the five checkpoint types (See Definition 2 of Section 4).

Classifier Evaluation Results. To determine PHI-UI, we build an SVM classifier of C-SVM type with a linear kernel; we use SVM as it tends to be resistant to over fitting. We leverage GridParameterSelection to optimize the cost parameter C. Specifically, we search C in space $(0.02, 1.0)$ with 50 steps and accordingly set it to $C = 0.94$. The ten-fold cross-validation results show that Chameleon-Hunter is highly effective, with a precision of 92.6% and recall of 94.7% on PHI-UI class (threshold=0.50). With a threshold of 0.80, the precision rises to 98.3%, while recall remains above 85%.

We further discuss the effectiveness of each category of features; we ran the classifier again only using each category of features at a time. As shown in Table 4, each feature gives a certain confidence for detection, with F-measure ranging from 61.9% to 77.7%. Among the features, "Patterns in metadata" plays an important role (F-measure 77.7%). This result is intuitive: for instance, consider the author feature, the benign apps often release bug fixes and new features frequently to keep them top of mind, while the confirmed Chameleon apps stay stealthy, without raising

TABLE 4: Evaluation of features.

Category	FP	FN	Precision	Recall	F-measure
Overall	30	21	92.6%	94.7%	93.7%
Comparing against benign UI	155	122	64.4%	69.4%	66.8%
Comparing against description	44	142	85.4%	64.4%	73.4%
Known malicious behaviors	69	189	75.3%	52.6%	61.9%
Patterns in metadata	113	74	74.2%	81.5%	77.7%

TABLE 5: Evaluation of suspicious cases.

Reachability of PHI-UI in Restricted Settings	Semantic Consistency of PHI-UI and App Description	# of Apps	Chameleon
7	7	99	m
Failed[1]	7	5	m
3	7	6	s
3	3	2	l

3 : Yes, 7: No, m: TP, l : FP, s : indeterminable
[1] Apps that are not responding or require an invitation code.

Apple's attention. Meanwhile, "Comparing against benign UI" and "Comparing against descriptions", which describe the semantic difference between PHI-UI and the expected behavior of the app, are effective, with an F-measure of 66.8% and 73.4% respectively.

Detection Results. In our study, CHAMELEON-HUNTER detected 142 new Chameleon apps; among them, 104 apps were determined by the SVM classifier, while 38 apps were identified by inspecting the URLs of web UIs embedded in structurally hidden (checkpoint) VCs.

From the unknown set (35,696 vectors), with threshold=0.80, the SVM classifier determined 112 new suspicious apps. Due to the absence of ground truth (e.g., virus scanner), we manually validate the apps based on simple but effective rules: 1) we check whether reported PHI-UIs are indeed hidden. Such rule is based on the assumption that a Chameleon hides its PHI-UI by default, and only activates it under certain conditions (e.g., through a remote message). For that purpose, we examine whether the PHI-UI is reachable through regular UI navigation during app's first launch after app installation under the restricted settings. 2) We check the semantic consistency of PHI-UI and the app's description; if the PHI-UI is described in the app description, we consider such case as benign. The experiment was conducted on an iPhone 6, in airplane mode with all third-party apps removed. The results are shown in Table 5. Among the 112 apps that were determined by the classifier, 92.9% (104 apps: 99+5) are true positives (TP): we failed to test 5 suspicious apps dynamically as they either won't open or do not respond at all on initial UIs under restricted settings. Further inspection shows that they hang when syncing with online servers, while the PHI-UIs are guarded by a certain response (e.g., server time < "2015-05-03", "isInReview"). We consider the 8 remaining apps as false positives (FP: 6 and 2). Among them, 2 apps ( ) are clearly benign as they include a suspicious UI that is not only reachable but also consistent with the app's description. CHAMELEON-HUNTER fails to perceive the semantic similarity. On the other hands, the other 6 ( ) apps are tricky as PHI-UI are reachable but semantically inconsistent with the app's description. We consider these cases as false positives, since the inconsistent content (e.g., "recruiting messages" left by the app developer) is not harmful to mobile users. Additionally, we checked 3,189 URLs extracted from 34,639 structurally hidden checkpoint VCs, and 12 URLs were identified as harmful. These URLs lead to the discovery of 38 Chameleon apps that deliver PHI-UI through web UI in hidden VCs. As discussed in Section 4, we determine the suspiciousness of the web UI contents based on VirusTotal, which aggregates more than 60 antivirus products. Note that VirusTotal has been used to obtain ground truth in security research [40], [34], [37], [47] and involved in industry security projects [7]. Most importantly, we reported the Chameleon apps to Apple; since then, the reported apps are quickly being removed (109 apps so far) from the App Store. We also checked the updated version of the leftovers and found that suspicious payloads of 5 apps are completely removed. We are unable to get further information for the other ones, since Apple's investigation, including its communication with the app developers, is a hidden process.

The false positive rate (FPR) could further be reduced by vetting the app description more strictly during Apple's App Review process. For instance, the critical factor for misclassification of DBRechargeVC, a discount variety store app (main topic is "recharge account"), is due to the semantic inconsistency between the app UI and the app's description; while the word list of VC included monetary service related words, the app description did not. Furthermore, there are no indicators (e.g., shopping) even in the other part of the app's metadata.

Performance. To evaluate the performance of CHAMELEON-HUNTER, we measured the time it takes to process all the apps in the unknown dataset on a Red Hat server using 14 processes. On average, 29.11 seconds (18.88 seconds for Structure Miner and 10.23 seconds for Semantic Analyzer) were spent on each app. The results demonstrate that CHAMELEON-HUNTER scales much better than other techniques (e.g., dynamic analysis, symbolic execution) [11], [48] and can easily process a large number of iOS apps.

6 MEASUREMENT

Based on the detected Chameleon apps, we further performed a measurement study to understand the illicit UI in iOS apps based on hidden UI. In this section, we first present the scope and magnitude of this malicious activity as discovered in our research (Section 6.1), then we describe the infiltration techniques (Section 6.2), and lastly provide case studies of interesting Chameleon apps (Section 6.3).

6.1 Landscape

Hidden Suspicious Activities. Table 6 summarizes suspicious activities launched by Chameleon apps and the corresponding review guideline each activity violates, as elaborated below.

Malicious crowdsourcing platforms. We observe 38 Chameleon apps' PHI-UIs are used as malicious crowdsourcing platforms. More specifically, those PHI-UI provides a list of illicit paid tasks for users. The illicit tasks include fake review and paid download to increase app ranking; fake transaction for e-commerce reputation manipulation.

Unauthorized content. Another large portion (58 apps) of Chameleon apps deliver unauthorized content through a structurally hidden UI. For instance, two apps are reported to serve as the entries for third-party app stores, which mainly deliver jailbreak tools and pirated apps. Moreover, an app called "joking & minesweeper", com.apushi.wan, recommends users to install an unprotected and unrelated enterprise app at http://115.29.198.162. Also surprisingly, we observe 11 Chameleon apps that distribute fake news. In particular, these apps, with a benign facet (Photo&Picture), display fake news of McDonald's (i.e., selling overdue foods,

TABLE 6: Suspicious activities of PHI-UI.

Activity	# of Apps	Cases	Apple's Guideline
malicious crowdsourcing	38	app ranking manipulation platform hidden in a music player	2.3.1
		a service to promote E-commerce seller's reputation	
unauthorized content	58	display content of compromised website or a third-party app store	2.5.6
		show bad news about a corporation (e.g., McDonald's)	2.3.1
		spread p2p player with adult content	3.2.2
		show lottery content in a Health & Fitness app	2.3.1
personal data collection	14	get your blood pressure in a Radio FM app	5.1.1
		in Travel app, show online dating site that requests DOB, SSN, etc.	
ad fraud	7	a Temple Run style app replaced by `watching ads, get lucky money'	3.3.1/3.1.4
Others	25	in-app promotion (e.g., referring user to shopping items)	2.3.1
		a "totally-free" app hides remote controllable payment UI	

which is confirmed by its official microblog [2]). Such fake news, although different from traditional malicious apps, has a tremendous potential to cause real-world and long-lasting impacts on individuals and businesses, as indicated by [38].

Sensitive information collection. 14 apps are inspected to aggressively collect sensitive information. For instance, we observed that a radio broadcasting app cn.doradio.radio collects health data (i.e., blood pressure) without a clear reason. Additionally, we found a delivery tracking app hiding a mortgage/income calculator inside. These cases, not necessarily malicious though, are suspicious since the sensitive data is collected stealthily.

Ad fraud. Furthermore, interestingly, we found that several apps fraudulently represent online advertisement to increase ad watching time and gain profit. In the pay-per-ad-view business model, an advertiser pays a publisher (i.e., the app owner) based on the number or time of ad view. For instance, com.funinteract.ballgame, a Temple Run style game app, once activated on a user's device, completely turns into a red envelope [53] app; whenever the red envelope is activated, the user is required to watch several lengthy video ads. Red envelope is a tradition to give lucky money to friends during certain events (e.g., weddings) and is popular in Asia.

Others. Chameleon apps are also observed to include other activities, such as offering suspicious monetary services. As required by Apple, developers must use in-app purchase (IAP) to secure users' transactions (e.g., subscriptions to premium content). However, in our study, developers of 8 apps labeled them "totally free", but meanwhile hide a remote controllable third-party payment page.

Scope of impacts. The Chameleon apps discovered in our experiment are found from 16 App Store categories. Note that the app category corresponds to the functionality of the benign UI since it is the UI shown during Apple's App Review process. As shown in Table 7, over 65% of Chameleon apps fall into Utilities, Music, and Entertainment. Also, we found that benign UIs, such as recorder, piano pieces, and file manager, are favored by Chameleon developers (possibly) as they are relatively simple to develop and maintain. Moreover, a group of Books apps were discovered to simply include a different list of books and share most of the code. This indicates that repackaged iOS apps could be a potential source for conducting malicious activities. Furthermore, the ranking data available from App Annie [3] demonstrates that the impacts of Chameleon apps are critical, affecting a large number of users. For instance, 4 apps, including com.funinteract.ballgame (red envelope under game UI) and cn.qimai2014.polarbearwifi (malicious crowd-sourcing under wifi helper), reached top 20 of the leaderboard across different countries (e.g., China, Laos). Despite

TABLE 7: Top 5 App Store categories of Chameleon apps.

Category	# of Apps	Benign UI Examples
Utilities	36 (25.4%)	Recorder, File Manager
Music	30 (21.1%)	Ringtones, Piano Pieces
Entertainment	27 (19.0%)	Web Browsers, Jeopardy-style Quiz
Reference	14 (9.9%)	WiFi Helper, Funny Pics/Jokes
Books	7 (4.9%)	Book Reader, List of Novels

of the incomplete ranking data from App Annie, we found that at least 8 apps were once reported in top 50 and 14 apps in top 100 within their corresponding categories (check [5] for more details). This is an indicator that chameleon app developers are motivated to promote their apps' ranking in order to attract more victims and achieve greater revenue. We are unable to measure the exact number of users being affected, as Apple provides no clue for number of downloads in any way. However, the aforementioned ranking data implies that Chameleon apps are affecting a large group of users.

In addition, we found that most Chameleon apps (53%) have only few updates, with a version number in the range (0, 1.5]. However, still a large portion of Chameleon apps (40% apps have Version 2.5) are capable of carrying their suspicious payload even to higher version; this observation is interesting as Chameleon apps need to undergo Apple's inspection for every new version submitted to the App Store. We carefully infer that malicious actors could either hide illicit content in app's initial submission, or flexibly insert it in following app updates, since Apple is incapable of detecting them in any phase. We also analyze the distribution of Chameleon apps over the release date. During the 6 months period (Jul. 2016, Jan. 2017), the probability of an app being Chameleon is as high as 0.8%. The linear forecast (regression) trend-line also indicates that Chameleon is still on the rise and requires further attention.

6.2 Understanding Chameleon Infiltration

Chameleon app Development. Chameleon app developers tend to create the benign UI, which covers the PHI-UI, with less effort as it is not their source of profit. One common approach is to add PHI-UI to existing app templates or open source projects ([36], [49]). According to Apple's guidelines [21] (4.3 and 4.2.6), such template apps should have been rejected. However, Apple seems to relax the policy, which makes Chameleon developing easier. To verify such observation, we designed a Chameleon app with ESTMusicPlayer [16] as the benign template; the app indeed is malicious as it shows a phishing UI once activated on user's device.

Amazingly, it got into App Store within two days (we removed it immediately).

More interestingly, Chameleon app development is in demand in the underground market. Based on our research, one could get a Chameleon app, with the desired PHI-UI, on the App Store for only $450 [28]. Specifically, a quick search on Google yields dozens of recruitment posts for Chameleon app development; e.g., freelancer [28], [27], Code Mart [16], witmart [55], dongcoder [22], Code4App [15].

Circumventing Apple's App Check. To bypass Apple's App Review process, Chameleon apps show PHI-UI only under specific conditions. We found that Chameleon apps use various traditional triggers, which includes commands from C2 servers (e.g., "uitype:2"), geographic locations ("isCN"), time, device states (e.g., jailbroken device, connected to cellular), device information (IP address, language), etc. Particularly, triggering conditions of several apps (e.g., com.91luo.91Ring app) were found to be far more complicated than others; e.g., a Chameleon developer first encourages users to download his app from App Store. Simultaneously, he provides an activation link on his website. When the activation link is clicked, a scheme (e.g., babyforring://[params]) that releases the app's illicit UI is send from Safari to the Chameleon app. This trick is by no means a novel technique. However, the existence of such complicated triggers brings a new challenge for app vetting. (see Section 6.3 for details)

Tactics to Reach Users and Reside in App Store. To attract more users, Chameleon developers promote their apps: using App Store Optimization (ASO); advertising on forums, media-sharing websites and social-networking services (e.g., AppKarma [25]). Another interesting channel is the pyramid scheme which provides rewards for finishing tasks. Chameleon app developers pay users if they refer the app to their friends; users are further rewarded for their friends' referrals as well.

Moreover, Chameleon developers resubmit clones of removed or existing Chameleons by only changing its bundle ID through a different Apple developer ID; e.g., after com.cloud.NHCore was removed from App Store, it was quickly resubmitted as com.good.jingling. Developers also submit multiple repackaged apps containing the same PHI-UI (i.e., suspicious activity); e.g., 7 book-reading apps from developer 1098099338 were found to integrate the same malicious crowd-sourcing platform. To mitigate the threat of such persistent activities, we provided a list of words that could help fingerprinting such apps upon Apple's request, and meanwhile are actively collecting resubmitted/repackaged Chameleon cases.

6.3 Case Studies

Here we elaborate on a few real-world Chameleon cases found in our study to help understand their behaviors in detail.

A Magic "Music Player". As mentioned in Section 6.1, a group of Chameleon apps are reported to provide illicit crowdsourcing services to device users. Among them, the most typical is sohouermusic, whose innocent side is a music player, also serving as an illicit crowdsourcing platform to distribute app ranking manipulation tasks (download, install, make up fake reviews, etc.) to individuals. We observed that triggering the illicit service is surprisingly difficult, and such triggering process is designed to evade app vetting. Specifically, the sohouermusic app is promoted on popular social networks (e.g., WeChat), which will redirect users to a website (play.sohouer.com). Only when a user visits the website on his iPhone and achieves an invitation scheme sohouermusic://invite=[serial number] sent, the app tries to load the Chameleon UI. However, before the UI is actually rendered, the sohouer app checks whether it has passed vetting process via its remote server, and the Chameleon UI shows up when the remote server responds with "isreview: 0" and a script url. Note that the response from remote server is set by the app developer once he notices the app enters the App Store. Besides acting as an illicit crowdsourcing platform, the Chameleon app also stealthily collects user data (e.g., device type, version, jailbreak, location). Another interesting observation is that the sohouermusic developers are so persistent. After the sohouermusic app was removed, the Chameleon UI was quickly repackaged into a sohouercamera app and submitted under different developer account.

Lucky Money in Rolling Ball Game. We found that a Chameleon app is hidden behind a rolling ball game, in which the user needs to avoid obstacles with a rolling ball, like the well-known Temple Run game. Similar to sohouermusic, for its Chameleon UI triggering, the app talks with its server 203.195.143.105 to determine whether it has passed the vetting process. An affirmative reply turns the app into a "lucky money" app, rewarding users for ad view. Interestingly, besides ad fraud, the "lucky money" app tries to collect users' phone number by popping up another malicious phishing UI. Further investigation shows that the "lucky money" facet provides a tutorial to unlock a pornographic video player on http://74.121.149.110/. Such kind of Chameleon apps, designed to provide illicit content to users, may have potential impacts on individuals and require attention from App Store.

7 Discussion

Chameleon apps, detected in our research, circumvent app vetting by hiding the PHI-UI behind a legitimate UI. However, there are other approaches adversaries can utilize to reach the App Store without getting detected. PHI-UI can be delivered on runtime using dynamic code loading which includes hot patch frameworks (e.g., JSPatch [9]) that allow developers to modify the native code of an app at runtime. Chameleon apps can be obfuscated (e.g., class/method names and strings) using tools such as PPiOS-Rename [45]. Such approaches can be tackled through dynamic analysis as the UI structure and the semantics of the UIs delivered through dynamic payload, web-content and obfuscated code could be analyzed on runtime. However, dynamic analysis may suffer from the code coverage issue and would not be able to detect PHI-UI of Chameleon apps without handling the complicated trigger conditions. Techniques, such as concolic execution, can be complementary to our semantic inconsistency analysis, and the two could be combined to provide a better CHAMELEON-HUNTER. On the other hand, Apple regulates and carefully monitors those dynamic code enabling techniques (e.g., hot patching frameworks) to minimize the attack vector; recently, Apple even decided to ban or reject any apps that use hot patch [42] from their App Store. Also, while obfuscation can become popular in the future, our experiments show that it is yet rare on iOS: none of 500 manually inspected apps (including Chameleon cases) were obfuscated. In addition to the cases mentioned above, CHAMELEON-HUNTER fails to detect PHI-UIs that have similar meanings to benign UIs. For instance, a Finance app (e.g., finance related forum, news, etc.) that shows PHI-UI which tricks users to enter personal information (e.g., name, phone number, address) for lower interest rate could not be detected.

8 Related Work

Detecting harmful iOS apps Compared to Android, iOS apps have not been studied much. PiOS [23] uses control flow analysis to detect privacy leaks in iOS apps. iRiS [38] combines binary instrumentation with static analysis and detects private API abuse in iOS apps. Chen et al. [14] determines potentially harmful iOS libraries by looking for their counterparts on Android. Nevertheless, none of the prior research notices the threats of iOS hidden UI, which bypasses App Store vetting and delivers suspicious content (e.g., phishing) to iOS users. Many works, such as [13], [52], focus on detecting privacy leaks in Android apps; similar to PiOS, they are ineffective in hidden UI identification.

Evasive mobile apps Chameleonapp falls into the general category of evasive malware targeting App Review. While there are several prior works on Android, we are the first to study evasive behavior on iOS. Moreover, prior works on evasive malware focus on handling triggering conditions and simply rely on sensitive APIs to determine its suspiciousness, while our work focuses on suspicious activities conducted only through UI. TriggerScope [26] relies on symbolic execution to detect suspicious triggering conditions of Android apps and check whether the path from the condition leads to a sensitive API. HsoMiner [42] classifies hidden sensitive operations on Android with a set of evasion-specific features. Recently, a report [50] discussed a Chameleon-like malware from Google Play: a multi-stage Android app that hides its malicious activities (i.e., phishing bank service) behind its legitimate-looking UI. Similarly, Google has removed these independent cases; such report shows that Google Play Protect could be circumvented, and a systematic understanding of Chameleon is necessary. Besides, there has been no report of evasive malware in filtration in enterprise-owned app markets [7], [8], which is reasonable given the strict store administration and the less benefit malware may get from the small user base.

Identifying suspicious activities using NLP NLP has been developed over decades and adopted in various fields (e.g., search engines, machine translation). In recent years, it has also been widely adopted in security analysis. WHYPER [44] and AutoCog [46] checks whether an Android app indicates its permission usage in app's description. SUPOR [32] and UIPicker [41] aim at identifying sensitive user inputs within user interfaces by NLP techniques. Liao et al. [40] checks semantic inconsistency between a sponsored top-level domain (sTLD) and its content, to detect whether the sTLD is compromised by promotional infection. Different from these works, our study focuses on detecting suspicious PHI-UIs in an iOS app by measuring semantic inconsistencies between its UIs.

9 Conclusion

In this paper, we conduct the first systematic study on Chameleon apps. Our detection tool, Chameleon-Hunter, identifies PHI-UIs with a set of semantic features generated by a suite of nontrivial techniques: e.g., UI transitioning pattern and semantic analysis. Our discovery of 142 Chameleon apps on the official App Store, sheds light on the various suspicious activities conducted by Chameleon apps. Most importantly, we disclosed our findings to Apple, which promptly investigated the newly-exposed threat and removed most of the Chameleon apps or disabled the suspicious UIs. For more details (e.g., demos and impacts) please check our website [5].

As future work, we intend to study the hidden UIs introduced by dynamic code loading using dynamic analysis. Also, to boost accuracy of our approach to be suitable for a market-scale detection, we plan to make a variation of Chameleon-Hunter which targets specific activities (e.g., hidden UIs that collect user health data) instead of detecting all suspicious activities. Lastly, as Chameleon-Hunter relies heavily on retrieval of app meta-data, the accuracy of our approach would improve according to Apple's provision of more accurate and detailed app meta-data (e.g., app's description).

10 Acknowledgements

We are grateful to our the anonymous reviewers for their insightful comments. This work is supported in part by NSF CNS-1801365, 1527141, 1618493, 1801432, 1838083, ARO W911NF1610127, and Hanyang University HY-2019-N (Project number: 201900000002835). Yeonjoon Lee is the corresponding author of this paper.

References

[1] Making money with your iphone. http://www.91ssz.com.
[2] Official statement of mcdonald's. https://www.weibo.com/1947211342/ya3l4kf5u.
[3] App annie. https://www.appannie.com/en/, Mar. 2010.
[4] Capstone: The ultimate disassembler. http://www.capstone-engine.org, Nov. 2013.
[5] Supplement materials: Chameleons on app store (ios). https://sites.google.com/site/ioschameleons/, 2018.
[6] anvaka. Common words in programming languages. https://anvaka.github.io/common-words, Dec. 2016.
[7] A. Armando, G. Costa, A. Merlo, and L. Verderame. Enabling byod through secure meta-market. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pages 219–230. ACM, 2014.
[8] A. Armando, G. Costa, L. Verderame, and A. Merlo. Securing the" bring your own device" paradigm. *Computer*, 47(6):48–56, 2014.
[9] bang590. Jspatch: bridging objective-c and javascript using the objective-c runtime. https://github.com/bang590/JSPatch, May 2015.
[10] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.
[11] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.
[12] C. Carpineto and G. Romano. Optimal meta search results clustering. In *SIGIR*, pages 170–177. ACM, 2010.
[13] H. Chen, H.-f. Leung, B. Han, and J. Su. Automatic privacy leakage detection for massive android apps via a novel hybrid approach. In *ICC*, pages 1–7. IEEE, 2017.
[14] K. Chen, X. Wang, Y. Chen, P. Wang, Y. Lee, X. Wang, B. Ma, A. Wang, Y. Zhang, and W. Zou. Following devil's footprints: Cross-platform analysis of potentially harmful libraries on android and ios. In *SP*, pages 357–376. IEEE, 2016.
[15] Code4App. Code4app: Looking for ios chameleon app developer. http://www.code4app.com/thread-14820-1-1.html, Sep. 2017.
[16] coding mart. Recruitement for ios chameleon app developer. https://mart.coding.net/project/11325, Nov. 2017.
[17] V. Community. Virustotal: Credits & acknowledgements. https://www.virustotal.com/en/about/credits/, Sep. 2012.
[18] Z. Deng, B. Saltaformaggio, X. Zhang, and D. Xu. iris: Vetting private api abuse in ios applications. In *CCS*, pages 44–56. ACM, 2015.
[19] A. Developer. Storyboard: Guides and sample code. https://developer.apple.com/library/content/documentation/General/Conceptual/Devpedia-CocoaApp/Storyboard.html, Sep. 2013.
[20] A. Developer. Using segues. https://developer.apple.com/library/content/featuredarticles/ViewControllerPGforiPhoneOS/UsingSegues.html, Sep. 2015.
[21] A. Developer. App store review guidelines. https://developer.apple.com/app-store/review/guidelines/, Dec. 2017.
[22] dongcoder. In demand of chameleon for app vetting. http://www.dongcoder.com/detail-678294.html, Sep. 2017.
[23] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *NDSS*, pages 177–183, 2011.
[24] F-Secure. Another reason 99% of mobile malware targets androids. https://safeandsavvy.f-secure.com/2017/02/15/another-reason-99-percent-of-mobile-malware-targets-androids/, Jan. 2017.