Understanding iOS-based Crowdturfing Through Hidden UI Analysis

Yeonjoon Lee[*], Xueqiang Wang[*], Kwangwuk Lee[1], Xiaojing Liao[1], XiaoFeng Wang[1], Tongxin Li[2], and Xianghang Mi[1]

[1]*Indiana University Bloomington*
[2]*Peking University*

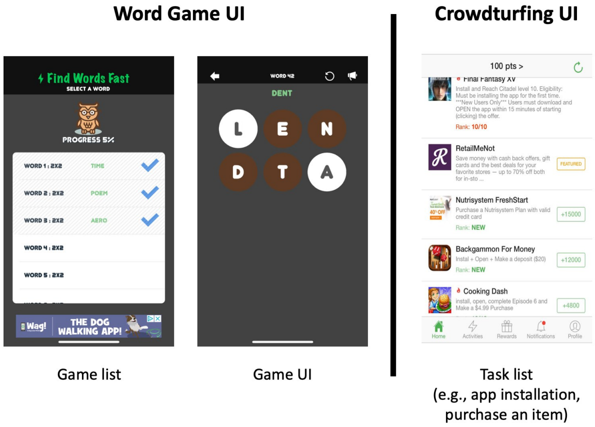
Abstract

A new type of malicious crowdsourcing (a.k.a., crowdturfing) clients, mobile apps with hidden crowdturfing user interface (UI), is increasingly being utilized by miscreants to coordinate crowdturfing workers and publish mobile based crowdturfing tasks (e.g., app ranking manipulation) even on the strictly controlled Apple App Store. These apps hide their crowdturfing content behind innocent-looking UIs to bypass app vetting and infiltrate the app store. To the best of our knowledge, little has been done so far to understand this new abusive service, in terms of its scope, impact and techniques, not to mention any effort to identify such stealthy crowdturfing apps on a large scale, particularly on the Apple platform. In this paper, we report the first measurement study on iOS apps with hidden crowdturfing UIs. Our findings bring to light the mobile-based crowdturfing ecosystem (e.g., app promotion for worker recruitment, campaign identification) and the underground developer's tricks (e.g., scheme, logic bomb) for evading app vetting.

1 Introduction

Crowdturfing is a term coined for underground crowdsourcing [44], in which an illicit actor (typically a cybercriminal) hires a large number of small-time workers to perform questionable and often malicious tasks online. Supporting such an operation is a crowdturfing platform, the underground counterpart of Amazon Mechanical Turk [1] that acts as an intermediary for the cybercriminal to recruit small-time workers for the hit jobs like creating fake accounts on an online store, posting fake Yelp reviews, spreading rumors through Twitter, etc. These attacks damage the quality of online social media, manipulate political opinions, etc., thereby threatening the public confidence in the cyberspace, which is the very foundation of the open web ecosystem.

Mobile crowdturfing. With the fast growth of mobile markets today, crowdturfing is extending its reach to mobile computing, serving illegal missions like inflation of an app's rating or mass collection of coupons or other bonus during a sales promotion. For this purpose, a mobile client (app) needs to be deployed to a large number of underground workers. Such an app, however, is prohibited by both Apple and reputable Android app stores according to their guidelines [21, 28], and will be taken down once detected. Although dissemination of the crowdturfing apps is still possible in the fragmented Android world, through less regulated third-party stores, on the Apple platform, cybercriminals find it hard to reach out to the iPhone users, due to the centralized app vetting and installation enforced by the Apple App Store. To circumvent this security check, it has been reported [52] that crowdturfing Trojans have been increasingly used to infiltrate the iOS App Store, through embedding stealthy crowdturfing user interfaces (UI) in innocent-looking iOS apps. An example is shown in Figure 1. Compared with *web-based crowdturfing* recently studied [37, 45, 46, 49], these apps are used to deliver mobile based crowdturfing tasks, such as fake app review and app ranking manipulation. Also, they are characterized by utilizing hidden UI techniques to bypass app vetting and deliver tasks for their small-time workers, which raise the challenges for finding them. So far, little has been done to systematically discover and analyze such hidden crowdturfing apps, not to mention any effort to understand the underground ecosystem behind them.



Final Fantasy XV
Install and Reach Citadel level 10. Eligibility: Must be installing the app for the first time. ***New Users Only*** Users must download and OPEN the app within 15 minutes of starting (clicking) the offer.
Rank: 10/10

RetailMeNot
Save money with cash back offers, gift cards and the best deals for your favorite stores — up to 70% off both for in-sto ...

Nutrisystem FreshStart
Purchase a Nutrisystem Plan with valid credit card
Rank: NEW

Backgammon For Money
Instal + Open + Make a deposit ($20)
Rank: NEW

Cooking Dash
install, open, complete Episode 6 and Make a $4.99 Purchase

Figure 1: A Word Game with hidden crowdturfing UIs.

Finding crowdturfing apps. In this paper, we report the first measurement study on iOS crowdturfing apps. The study relies on the discovery of such malicious apps from the Apple App Store, which is challenging, due to the difficulty in identifying their elusive hidden UIs. These UIs are under the cover of benign ones and can only be invoked under some

[*] The two lead authors are ordered alphabetically.

specific conditions (e.g., time, commands from C2 servers). Even when they indeed show up, likely they operate similarly as the legitimate UIs: no malware downloading, no illicit use of private APIs, etc. To capture their illegitimacy, one needs to read their content and understand their semantics. This, however, requires human involvement and therefore does not scale during app vetting. The attempt to detect such UIs becomes even more complicated for the third party, who does not have the source code of the related apps and therefore needs to work on binary executables. Indeed, our research has brought to light almost 100 such apps already published on Apple App Store, completely bypassing its vetting protection.

To address these challenges, we come up with a new triage methodology, *Cruiser*, that identifies the iOS apps likely to contain hidden crowdturfing UIs for further manual inspection. A key observation here is that such apps are characterized by their conditionally triggered UIs (e.g., triggered not by user actions but by network events), as demonstrated through UI transitions. Also, the content of such hidden UIs is related to crowdturfing semantically (e.g., app ranking manipulation), which is inconsistent with their hosting app's public description. These unique features make it possible to detect these iOS apps through a combination of binary, UI layout and content analyses. From 28,625 iOS apps covering 25 app categories, our system reports 102 most likely involving hidden crowdturfing UIs; considering the large scale of Apple App Store (2 million apps [3]) and the relatively high false detection rate (8.8%) of our tool, we manually examined all the 102 flagged apps, and found that 93 apps indeed contain hidden crowdturfing UIs.

Measurement and discoveries. Looking into the apps with hidden crowdturfing UIs reported by *Cruiser*, we are surprised to find that this new threat is indeed trending, with a big impact on today's mobile ecosystem. More specifically, from the 93 apps detected, we discover 67 different mobile crowdturfing platforms, which handle a variety of crowdturfing tasks, such as app ranking manipulation, fraud account registration, fake reviews, online blog reposting, and order scalping, etc. Also importantly, these apps are found to bypass app vetting several times and have a long lifetime. Such apps are popular, having been installed by a large number of users (32.4 million in total). Some of them even appear on the Apple leaderboards, with 25 of them ranked among the top 100 in their corresponding categories.

Also interesting is the ecosystem of mobile crowdturfing, as discovered in our study, which includes app promotion for worker recruitment, campaign identification, etc. In particular, crowdturfing platform owners are found to advertise their apps through multiple channels, including crowdturfing app gateway sites, in-app promotion and a pyramid (or referral) scheme that rewards the individuals for recommending crowdturfing apps to other users. In the crowdturfing app gateway sites, we observe that around 50% of hidden crowdturfing apps have been downloaded more than 18K times; there are 32.4 million downloads in total. Also, we find that the app with hidden UIs is in high demand from the underground market: e.g., cybercriminals are willing to pay hundreds of dollars for developing such an app to circumvent Apple's vetting.

Furthermore, we analyze the evasion techniques employed by the crowdturfing apps, and bring to light new techniques that utilize complicated conditions to trigger their malicious behaviors: such apps not only know whether they have passed Apple's review so they can change their behaviors accordingly, but also protect their hidden UIs with the conditions involving user interactions or communication with a malicious website. Up to our knowledge, such techniques have not been reported to the Apple platform before, and therefore bring new challenges to its vetting process. Further discovered in our research is the way underground developers reuse their product and work with each other: we see that different developers inject different crowdturfing UIs to similar apps, and the same developer hides the same UIs into her different products. Also interestingly, almost identical apps, with both over and cover UIs, are found to be submitted to the store under different developer IDs. We disclosed our findings to Apple, which acknowledged us and has removed all reported apps from the App Store, though new attack apps of this type continue to pop up due to Apple's lack of effective means to detect them; also upon Apple's request, we provided a list of fingerprints for eliminating the apps similar to the malicious ones.

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

• *New methodology*. We developed a novel approach that utilizes a binary-code analysis on UI hierarchy and Natural Language Processing (NLP) analysis on UI semantics to detect the iOS apps with hidden crowdturfing UI.

• *New findings*. *Cruiser* helps us gain new insights into the mobile crowdturfing ecosystem and exposes the underground developer's new tricks for evading Apple's app vetting. Also importantly, our study sheds light on a new attack vector that has long been ignored: use of hidden UIs to evade even most restrictive app vetting to distribute illicit content.

Roadmap. The rest of the paper is organized as follows: Section 2 provides background information for our study; Section 3 elaborates on the design of *Cruiser*; Section 4 presents our measurement study and new findings; Section 5 discusses the limitations of our current design and potential future research; Section 6 reviews related prior research and Section 7 concludes the paper

2 Background

Crowdturfing platform. As mentioned earlier, crowdturfing, also called malicious crowdsourcing, is an illicit business model, in which cybercriminals (i.e., *intermediaries*) recruit *small-time workers* to carry out malicious tasks (e.g., app

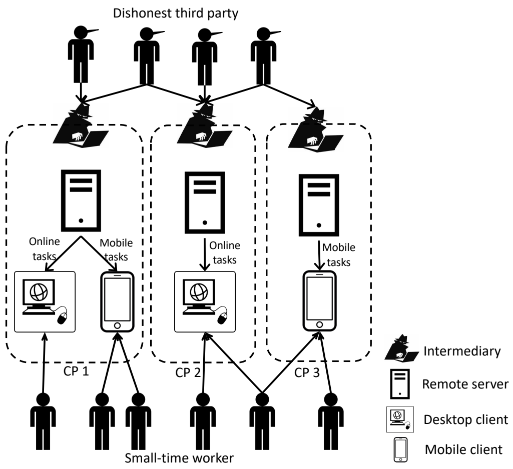


Figure 2: Overview of modern crowdturfing platforms, where "CP" represents a crowdturfing platform.

ranking manipulation) for *dishonest third parties* (e.g., app owner). Moving from the desktop browser-based clients (e.g., Zhubajie [5] and Sandaha [4]) to mobile devices, crowdturfing today increasingly happens through the apps deployed to workers' smartphones. As an example, consider a dishonest app owner who intends to inflate the app's installation volume and therefore seeks help from a crowdturfing platform; through the platform, the owner can pay workers to download and install his app so as to fake its popularity. Other hit jobs performed by the platform include the spread of fake reviews, defamatory rumors, etc.

Figure 2 illustrates modern crowdturfing platforms supporting both desktop browser-based clients and mobile clients. Such a platform, generally created and maintained by intermediaries, is designed to coordinate crowdturfing tasks and organize small-time criminals (workers) to do the tasks. As shown in the figure, a crowdturfing platform consists of servers to distribute crowdturfing tasks, and desktop browser-based clients or mobile clients to interact with the workers (e.g., publishing the tasks and checking the quality of the work). Unlike the platforms with browser-based clients, those with mobile clients mainly aimed at mobile-related crowdturfing (e.g., app ranking manipulation).

However, mobile crowdturfing clients, in the form of apps, are widely considered to be illicit by app stores, including Apple App Store [21] and reputable Android App stores like Google Play [28]. Especially for iOS crowdturfing clients, it is extremely hard for such apps to get through Apple's restrictive vetting process. Actually from the underground forum, we find that some intermediaries seek experienced developers to build apps capable of infiltrating the Apple store, by hiding their crowdturfing UIs (Section 4.4). Also interestingly, due to the difficulty in publishing crowdturfing apps, we find from the Apple store that multiple servers even share one client (Section 4).

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 crowturfing 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 3.2).

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* [50] 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) [13] 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* [48] for keyword extraction.

Threat Model. In our research, we consider an adversary who tries to publish iOS apps carrying hidden crowdturfing content on Apple App Store. Examples of such crowdturfing activities include fake review posting, app ranking manipulation and order scalping [15], etc. For this purpose, the adversary creates iOS apps with hidden crowdturfing UIs. These UIs are meant for displaying the tasks assigned by a crowdturfing platform and providing guidance on how to accomplish the tasks, so typically they do not ask for additional capabilities (guarded on iOS by entitlements). To publish such apps, the adversary is supposed to be knowledgeable about Apple's vetting process. Use of private APIs or side-loading are the focus of Apple's vetting and therefore not considered in our research. 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 Methodology

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

3.1 Overview

Architecture. Figure 3 illustrates the architecture of *Cruiser*, which includes a *Structure Miner* and a *Semantic Analyzer*. After fetching and decrypting iOS apps from App Store, *Structure Miner* takes as its input a set of decrypted iOS apps, and disassemble them. The disassembled apps are then utilized by the Structure Miner to construct a VC hierarchy for identifying the VCs with conditionally triggered UIs (e.g., two entry UIs). Here we define *checkpoint VCs* as all VCs associated with conditionally triggered UIs and their corresponding children VCs (see detail in Section 3.2). We also consider children VC, since VCs with conditionally triggered patterns

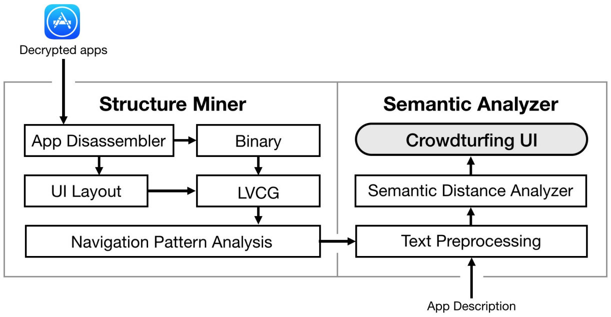
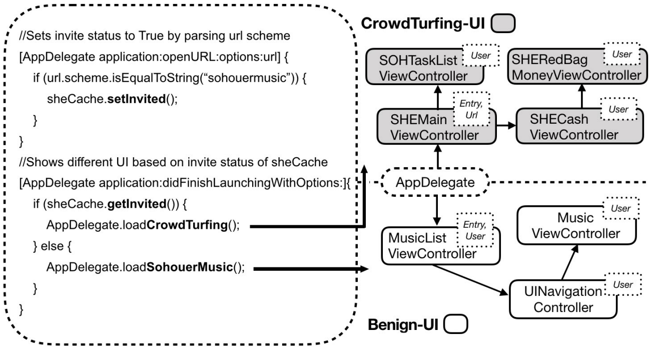


Figure 3: Architecture of *Cruiser*.



```
//Sets invite status to True by parsing url scheme
[AppDelegate application:openURL:options:url] {
    if (url.scheme.isEqualToString("sohouermusic")) {
        sheCache.setInvited();
    }
}
//Shows different UI based on invite status of sheCache
[AppDelegate application:didFinishLaunchingWithOptions:]{
    if (sheCache.getInvited()) {
        AppDelegate.loadCrowdTurfing();
    } else {
        AppDelegate.loadSohouerMusic();
    }
}
```

Figure 4: Pseudocode and simplified LVCG with conditionally triggered UIs.

sometimes may not contain sufficient texts for semantic analysis. On each checkpoint VC, the Semantic Analyzer further extracts texts from it, and evaluates its content through a set of NLP techniques to determine whether it is used for crowdturfing.

Example. To explain how *Cruiser* works, here we walk through its workflow (Figure 3) using a Music Player app with a hidden app ranking manipulation UI, *com.sohouer.music*. *Cruiser* first automatically decrypts the app and disassembles it into binary, UI layout files and resource files. Meanwhile, we also crawl the app's metadata (i.e., description for the Music Player app) from the App Store as the input for the Semantic Analyzer.

The Structure Miner processes the binary and UI layouts of the *com.sohouer.music* app and creates a VC hierarchy in the form of a labeled view controller graph (LVCG) (as shown in Figure 4). From the LVCG, our approach extracts VCs with conditionally triggered UIs and marks them as the checkpoint VCs. More specifically, the Structure Miner identifies VCs and VC transitions from the app binary and UI layout files to construct the LVCG (Figure 4). From the LVCG, the Structure Miner discovers the conditionally triggered UIs: two root VCs of *MusicListViewController* and *SHEMainViewController*, which indicate that there are two entry UIs for the app. Depending on whether the app has received a particular scheme invocation before, different main UIs will be displayed when the app is launched. Therefore, these two VCs are labeled as checkpoint VCs for the follow-up semantic analysis.

Once the checkpoint VCs are found, the Semantic Analyzer then processes their text data to identify semantic features: the *MusicListViewController* VC contains a series of *Music Player* related words, such as $\{album, singer, shuffle, song, music, radio\}$, which are consistent with the app's description. On the other hand, the topic words under *SHEMainViewController* are $\{task, cash, earn, withdrawl, join, pay, reward\}$. Given the semantic inconsistency discovered, the Semantic Analyzer flags the app as a crowdturfing client.

Data collection. In our research, we collected 28,625 iOS apps for discovering new hidden crowdturfing apps, which we call *unknown set*. Specifically, we scanned the entire iOS app list from iTunes Preview website [34] using an app crawler running on an iPhone, and then selected the apps updated after Jan. 1, 2016 to download and decrypt. This is because apps with hidden crowdturfing UI is an emerging threat, and recently updated apps tend to have more active users. In this way, we collected 28,625 iOS apps, which cover 25 app categories.

3.2 Structure Miner

The Structure Miner is designed to identify the VCs with conditionally triggered UIs 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 4, 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 $G = (V, E, \alpha)$ over a node label space Ω, 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 $\alpha : V \to \Omega$ marks each node with its UI properties and text data. Each node is given four property labels: *entry*, *user*, *url*, *others*. 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 VC, and in the meantime, all the UIs built through storyboard can only be found in the layout files, including the transitions between them. To address this complexity, *Cruiser* builds two LVCGs, one from the binary and the other from the layout files, before combining them together.

Specifically, 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 flows from a VC to another to recover the transitions between the detected VCs. For this purpose, our approach first maps the addresses in the binary code to symbols (e.g., class name, method name) using a binary analysis tool Capstone [7], 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-flow analysis using an implementation similar to the prior techniques [18, 23], to connect the transition APIs (*performSegueWithIdentifier:sender:*) discovered in a VC to another one, the transition target.

To construct a LVCG on the layout files under the storyboard folder generated by Apple's interface builder, we need to extract VCs and VC transitions from the files. The former can be found from the storyboard plist file that includes the mappings from VC names to the obfuscated names of nib files. The latter is recorded by the nib files, 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 file and further detects each VC's nib files from the mappings it records. More challenging here, however, is to identify the transitions between the VCs, since objects included in a nib file are undocumented. To enable the Structure Miner to interpret the file, we reverse-engineered part of its format relevant to the transition and content extraction. Specifically, we started from the interface builder, through which one can define one or multiple scenes to represent a UI and a *Segue* to describe a transition. Through a differential analysis, we compared the compiled nib files with and without a specific transition to pinpoint the nib objects corresponding to different *Segue* types (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 files, 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 defined in the code, two LVCGs can then

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

Property	Definition	Method/Class names
entry	root VC	setRootViewController:
user	VC triggered by a user interaction	addTarget:action:forControlEvents:
url	VC rendering web content	openURL:, UIWebViewController
others	other properties (e.g., self-dismiss)	dismissViewControllerAnimated:completion:

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 find 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 3.1).

Conditionally triggered UI extraction. Given 17 apps with hidden crowdturing UI collected from *91ssz* [8] (see detail in Section 3.4), without loss of generality, in our study, we consider two types of conditionally triggered UIs on the LVCG, as elaborated below:

• *More than one root VCs*. We consider an LVCG to be suspicious if it has more than one root VCs, i.e., app has two entry points, that is, two different root UIs. The root VC is the first one launched (by *AppDelegate*) when an app starts running. One evasion trick the adversary often plays is to run two root VCs, one legitimate and the other illicit, depending on some trigger conditions (e.g., the app's execution environment). For example, in the app *com.sohouer.music* (see Section 3.1), besides the benign UI (i.e., MusicListViewController), the hidden crowdturfing UI (i.e., SHEMainViewController) can also be invoked by *AppDelegate*). Such a pattern can be described as $|\alpha(v) == `root'| \geq 2$. In this case, we label the two VCs and their corresponding children VCs as checkpoint VCs for further semantic analysis.

• *VC not triggered by users*. If an *entry* VC or intermediate VC is not triggered by the user, but by other external events (e.g., network), i.e., $\alpha(v)[`entry'] = True \wedge \alpha(v)[`user'] = False$ or $\alpha(v)[`user'] = False \wedge \alpha(v)[`url'] = True$, we consider it as suspicious, since such UI is difficult to be triggered during app vetting. In such a case, we mark such a VC v and its children VCs as checkpoint VCs.

Looking into all 28,625 apps, we discover 34,679 checkpoint VCs using conditionally triggered UIs. These VCs are further evaluated by the Semantic Analyzer. Our evaluation (see Section 3.4) shows that the Structure Miner maintains a good coverage on hidden crowdturfing UIs while filtering out most legitimate apps.

3.3 Semantic Analyzer

The Semantic Analyzer determines whether checkpoint VCs are crowdturfing UIs. Serving this purpose is a set of NLP based semantic analysis techniques: we first extract UI texts from the VCs, and then find out whether they are related to crowdturfing by calculating the semantic distance between the texts and crowdturfing keywords.

Text discovery. As mentioned earlier, the format of the UI layout files (the nib files) is undocumented. However, they can be converted into the XML form using ibtool [42]. From their XML content, we can find plain-text strings under *NSString* objects, a property of UI element objects like button, table, textbox, font, color, etc. Some of these strings are part of the content a UI displays, while the others are not, depending on the type of the UI element objects. For example, *UIFont* and *UIColor* carry strings such as *".HelveticaNeueInterface-Regular"* and *"blackColor"* for defining fonts and UI color, respectively. To extract UI content from the nib files, we come up with a blacklist of UI element objects that do not include UI texts, and use that list to filter out irrelevant text strings. More specifically, we randomly sampled 70 iOS apps from our unknown set, which gives us 1,307 nib files including 28,469 *NSString* objects. We clustered them based on the types of their UI element objects, and manually went over all 103 types discovered. In this way, we constructed a blacklist with 21 patterns that cover 64 object types that do not contain any meaningful UI texts. Table 8 in Appendix shows the blacklist. When analyzing a given app, the Semantic Analyzer locates all *NSString* objects from its checkpoint VCs and further recovers their host UI element objects from the app's UI object tree (i.e., a tree built on layout files). If the element is on the blacklist, we ignore its *NSString* object.

In addition to the text strings in the *NSString* objects, other UI content can be embedded in images and therefore cannot be easily extracted. To collect more semantic information for crowdturf UI detection, we utilize an app's meaningful variable names (e.g., *_album_id*), class names (e.g., *TicketDetailViewController*) and method names (e.g., *setSongIdsArrayM*), which are preserved in the binary's symbol table by the Object-C compiler. These human-readable symbols are recovered by our approach from the variables, class names, etc. output by Capstone [7] for each checkpoint VC. Also for the VCs with Web UIs (e.g., *UIWebViewController*), we include the text content collected from the URL embedded in the VC. An example of the data gathered from both UI layouts and a binary is presented in Table 2.

Crowdturfing UI identification. Given the UI content recovered from each checkpoint VC, we analyze whether such data is semantically associated with crowdturfing: to this end,

Table 2: Sample text data.

Object Type	Text Data
UILabel	"Proceed to checkout"
NSLocalizableString	"start making money"
Class Name	"TaxiViewController", "GameView" "TicketDetailViewController"
Method Name	"setSongIdsArrayM:", "setBuyAllProductId:"
Instance Variables	"_album_id", "_uploadMedia ", "_btnPaid"
CFString	"Select photo from photo library" "more clear free voice calls"
URL	*booking.com* "hotel" "city", "trip", "taxi"

we first preprocess the texts to address the issues like multi-language, noisy words, and then identify the keywords representing their semantics. In the meantime, we crawl a set of popular crowdturfing websites (e.g., Zhubajie [5] and Sandaha [4]) to build a crowdturfing word list. Words on the list are compared with the UI keywords using Word2vec [50] to find out their semantic distances. When such a distance becomes sufficiently small, the checkpoint VC is then flagged as a hidden crowdturfing UI. In the following, we elaborate on each step of this analysis.

At the preprocessing step, our approach runs Google Translate [2] to convert content in other languages into English. For the text in the languages without delimiters, Chinese in particular, we first use open source tools [27, 30] to segment texts into words before the translation; for the class/method names extracted from the binary, we tokenize them using regular expressions that cover common naming conventions (e.g., *CamelCase* style). Further, we drop all common stop words (e.g., NLTK stop words), and the frequent words from iOS frameworks and programming languages (e.g., "UIViewController", "ignoreTouch:forEvent:" and "raiseException"), as well as program language and debugging related texts (e.g., "socket", "connection", "memory", "allocation"). These words come from 74 framework-libraries of iOS 8.2.1, and are gathered in our research from sections such as *__cfstring* and *__objc_methname*. Selected from these documents are 1,806 frequent words whose inverse document frequency (IDF) values are larger than a threshold (we use $log(5)$ in our implementation). Also 1,031 program language and debugging related words are hand-picked for *Objective-C*, *Swift*, and *Javascript*.

After removing these words from a checkpoint VC, the remaining words are then analyzed using affinity propagation [26], which clusters them based upon their semantics (represented by the vectors computed using an embedding technique) and reports the most significant cluster. The words in such a cluster are then used by our approach to represent the semantics of their hosting VC.

To collect crowdturfing keywords, we crawl 280 web pages from the popular crowdturfing websites (i.e., Zhubajie [5] and Shandaha [4]). From these pages, we identify their topic keywords using the Latent Dirichlet Allocation (LDA) method. In this way, we build a crowdturfing list of 214 words. A problem for directly using these words is the observation that some of the crowdturfing words may also appear in legitimate apps: for example, "coupon" is certainly a meaningful word for a shopping app, not necessarily referring to the illicit task of bounty hunting. To address this problem, we compare these words with the keywords extracted from an app's description, dropping those related to the app's publicly stated functionality before the comparison below.

Given keywords discovered from the checkpoint VCs and the list of crowdturfing words, we run *Word2vec* [50] on each of these words, which maps the word to a vector that describes its semantics. Using these vectors, our approach measures the semantic relations between the UI keywords and the crowdturfing keywords by calculating their vectors' cosine similarities. For each UI keyword, its average similarity with all the crowdturfing keywords is used to determine its relevance with crowdturfing. We find that when the average relevance score of all the keywords of a checkpoint VC reaches 0.525 or above, the VC is nearly certain to be a crowdturfing UI.

3.4 Challenges in Identification

Here we evaluate *Cruiser* and elaborate on the challenges in crowdturfing app identification.

Evaluation with ground-truth set and unknown set. We evaluated *Cruiser* over the following *ground-truth datasets*: for the bad set, we collected the apps with hidden crowdturfing UIs from *91ssz* [8]. *91ssz* is a website that hosts the apps with the features (e.g., spam forums, earn money) violating Apple's guidelines. We manually examined 290 apps and confirmed 17 with hidden crowdturfing UIs (the other 273 apps do not have hidden UIs and are *only* accessible through third-party black markets). The good set were gathered from the top paid app list found from Apple App Store charts, which are considered to be mostly clean. We randomly sampled 17 of them (the same size of the bad set) to build the good set. Note that we manually examined those apps and verified that they are indeed benign. Running on these sets, *Cruiser* shows a precision of 88.9% and a recall of 94.1%.

Next we further report the results when running our approach on the unknown set, including all the apps collected from the Apple App Store (Section 3.1), at each stage of our analysis pipeline. We statically analyzed disassembled code and UI layout files over the 28,625 iOS apps, and discovered 34,679 checkpoint VCs, which are related to 3,999 (14.0%) apps using conditionally triggered UIs. Then, we executed the Semantic Analyzer, which flagged 102 apps. We manually examined all of them and found that 93 apps indeed contain hidden crowdturfing UIs. This gives us a precision of 91.2%. The 9 falsely detected apps, though not including crowdturfing UIs, also turned out to be less legitimate. Below we elaborate on the missed apps and the falsely detected apps.

Missed apps. On the ground-truth set, only one crowdturfing app was missed by *Cruiser*. The app fell through the cracks due to inadequate semantic content extracted from their UIs. It is found to construct the URL for the content to be displayed during its runtime and dynamically loads crowdturfing pages through the URL. While *Cruiser* can find the suspicious view controller, it cannot statically gather semantic content from the crowdturfing pages and therefore fail to provide enough semantic information for the Semantic Analyzer to make a decision.

Determining the number of missed crowdturfing apps in the unknown set (with 28K iOS apps) is challenging. Given the low density of such malicious apps in the dataset, we could not randomly sample from the set hoping to capture ones missed by our methodology. So what we did in our study is to lower down the threshold used by the Semantic Analyzer for detection, which improved the recall, at the expense of precision. With the threshold decreasing from 0.525 to 0.513, our approach flagged 313 more apps. We manually analyzed all these apps and found only 3 new crowdturfing apps (false negatives), while the remaining 310 were all false positives. Looking into these 3 missed apps, interestingly we found that they were all web-based apps that dynamically download crowdturfing content from the web during their runtime, as we observed on the ground-truth set.

Falsely detected apps. All false detections reported come from the apps indeed carrying conditionally triggered UIs. These apps are not only structurally but also semantically related to a true crowdturfing app. More specifically, their hidden UIs all contain monetary content, which is one of the semantic features for crowdturfing apps. For example, among the 9 false detections, 7 are about “Health & Fitness” but actually include hidden lottery UIs. The remaining two are “Education” apps, which declare to be free but later display a remotely controllable UI asking for payment. Note that all these UIs are potentially unwanted, since they are undocumented (in the apps’ description) and forbidden by Apple’s guideline [21]. We consider these apps (with illicit UIs) as false detections, just because they are not directly related to crowdturfing.

Legitimate use of conditionally triggered UI. In Section 3, we report the observation of 14% apps including conditionally triggered UIs. Through a manual analysis, we found that these apps use two entry UIs to display notifications, a tour or a guide for the app, special events (e.g., New Year) and etc. All their hidden UIs cannot be reached through user interactions. This demonstrates the importance of the Semantic Analyzer, which utilizes NLP to determine the irrelevance of these apps to crowdturfing, thereby controlling the FDR of our approach.

3.5 Comparison to Other Approaches

***NaiveCruiser*: Semantic analysis on all VCs**. *Cruiser* is characterized by a two-step analysis (by the Structure Miner and then the Semantic Analyzer), first filtering out the VCs with normal navigation pattern and then analyzing the semantics of suspicious VCs. This strategy is designed to minimize the overheads incurred by the Semantic Analyzer, which is crucial for making our system scalable for analyzing the 28K apps in the wild. In the meantime, there is a concern whether the performance benefit comes with an impact on the technique’s effectiveness, making it less accurate. To understand the problem, we compared our implementation of *Cruiser* with an alternative solution, called *NaiveCruiser*, which conducts a semantic analysis on all VCs in the app. This approach is fully tuned toward effectiveness, completely ignoring the performance impact.

In particular, we also evaluated the *NaiveCruiser* over the same *ground-truth datasets* we used to evaluate *Cruiser*. Running on these sets, *NaiveCruiser* shows a precision of 90.9% and a recall of 93.2%, which is in line with *Cruiser* (precision of 88.9% and recall of 94.1%). This indicates that our two-step design does not affect the effectiveness of detection. We also show the large performance degrade of *NaiveCruiser*, compared to *Cruiser*, in Appendix.

Crowdturfing keyword search. Simply searching for crowdturfing keywords is not effective. This is because the words used in crowdturfing UI (e.g., money, withdrawal, cash) are common, which often appear on other legitimate UIs (e.g., stock apps, accounting apps). Therefore, a simple keyword-based approach would bring in a high FDR (see below). Our approach utilizes a suite of techniques (e.g., looking for structural features of conditionally triggered UIs and corresponding VCs, removing words related to app descriptions) to avoid false reporting of legitimate UIs.

To understand how effective these techniques are, we evaluated the baseline – the naive keyword search on the 28K iOS apps. Specifically, we automatically extracted keywords from crowdturfing content collected from our ground-truth set, and then manually crafted a list of 32 most representative keywords for crowdturfing tasks (e.g., reward, task and installation). In the experiment, we studied the effectiveness of these keywords by first searching for the apps containing individual words and then analyzing their combinations (those including 2, 3, $\cdots$, 32 words). The more keywords an app includes, the more likely it is problematic but the fewer such apps would be found. In the end, we did not see any app involving more than 8 keywords. Among those carrying no more than 8 words, the highest precision achieved was 15.38% (an FDR of 84.62%), for those with 8 words. In this case, only 5 apps were reported. By comparison, our approach achieved a precision of 91.2%, reporting 93 malicious apps on the unknown set. This result demonstrates that the naive keyword search is indeed inadequate.

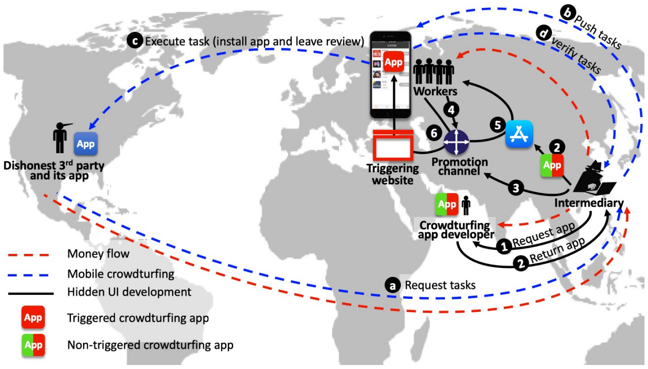


Figure 5: Overview of modern crowdturfing value chain, which consists of hidden crowdturfing app development (❶-❻) and mobile crowdturfing operations (ⓐ-ⓓ).

4 Understanding iOS-based Crowdturfing

Based on the detected crowdturfing apps, we further performed a measurement study to understand the iOS-based crowdturfing ecosystem. In this section, we first present as an example a real value chain of modern crowdturfing (Section 4.1), and then describe the scope and magnitude of this malicious activity as discovered in our research (Section 4.2), before elaborating the two key components of the value chain: *crowdturfing app development and promotion* (Section 4.3) and *mobile crowdturfing operations* (Section 4.4).

4.1 Mobile-Crowdturfing Value Chain

Before coming to the details of our measurement findings, first let us summarize the mobile-crowdturfing value chain discovered in our research.

A cybercriminal (i.e., intermediary), who owns a modern crowdturfing platform *chinazmob*, intends to publish a mobile client, which is downloadable from the App Store, to publish crowdturfing tasks and coordinate with small-time workers. Hence, the intermediary seeks underground app developers to build an app with hidden crowdturfing UIs (❶, see Section 4.3). The hidden crowdturfing UI will only be triggered when app users visit the website *ioswall.chinazmob.com*. Once done, the app *Pleasant Music* (id115****781), which disguises as a music player, passes the vetting of the App Store and is published (❷). Then, the intermediary promotes this app on social networks (❸) with links to the App Store and the triggering website *ioswall.chinazmob.com*. Small-time workers, who observe the promotion (❹) and download the *Pleasant Music* app (❺), will access the mobile crowdturfing client after triggering the hidden UI (❻) to execute crowdturfing tasks. Meanwhile, in the underground business of mobile crowdturfing, a dishonest mobile app owner of *Anjuke* who plans to inflate the app's installation volume reported by the App Store, pays for a crowdturfing platform *chinazmob* to manage crowdsourced app downloading tasks (ⓐ). Then, the intermediary will publish a task on its mobile client and recruit small-time workers (ⓑ) to do the task. These workers will install *Anjuke* and write fake reviews for the app (ⓒ). Once done and verified by the crowdturfing platform (ⓓ), the workers will get commissions from the platform.

In the rest of the section, we discuss the security implication introduced by these hidden UI apps, considering both crowdturfing app development and promotion and mobile crowdturfing operations in the value chain. As evidence for their impacts, those apps successfully infiltrated the App Store, even reached a high rank and bypassed the app vetting multiple times. In addition, we discovered various hidden UI techniques and the underground services that support the development of such apps. In particular, we revealed a set of techniques (e.g., logic bomb, scheme) deployed by the cybercriminals, as well as the underground services that are willing to pay $450 for developing such iOS apps. For app promotion, we identified 40 crowdturfing app gateway sites used by cybercriminals to promote 67.7% of such apps, which also enabled us to estimate the volume of the users. Furthermore, we report the findings related to *mobile-based* crowdturfing and discuss their insights, which have never been done before. For example, in contrast to the web-based crowdturfing dominated by a small number of platforms, on the mobile side we observed a fragmented crowdturfing market and a stealthy iOS crowdturfing ecosystem: we detected 93 hidden crowdturfing apps related to 9 campaigns, after clustering them based on similar app information, code structure and network behavior. Finally, we report a case study on an app with a hidden app ranking manipulation UI.

4.2 Landscape

Scope and magnitude. Our study reveals that apps with hidden crowdturfing UI are indeed trending in the Apple App store. Altogether, *Cruiser* detected 93 apps with hidden crowdturfing UIs, which are related to 67 crowdturfing platforms. To the best of our knowledge, this is the largest finding on mobile crowdturfing ever reported.

Apps with hidden crowdturfing UI, as discovered in our experiment, are found in 15 categories of the Apple App Store. As shown in Table 3, over 77.4% of the apps are in the categories of *Music*, *Utilities*, *LifeStyle*, and *Entertainment*. These apps are often built upon existing open source projects (see Section 4.4). Surprisingly, we found that some crowdturfing apps are of high ranks: six apps, including the wifi helper app (*cn.qimai2014.polarbearwifi*), the recorder utility app (com.amzhushou.app), the Temple Run style app (com.funinteract.ballgame) and several word guessing game apps reached top 20 of the leaderboard across different countries (e.g., *China*, *Laos*), based on the ranking data available from App Annie [6]; also, we observed that at least 14 apps were once ranked within the top 50, and 25 apps were in the

Table 3: Top 5 app store categories of apps with hidden crowdturfing UI.

Category	# apps	Benign UI examples
Music	32 (34.4%)	Ringtones, Piano Pieces
Utilities	15 (16.1%)	Recorder, File Manager
LifeStyle	15 (16.1%)	Story Teller
Entertainment	10 (10.8%)	Web Browsers, *Jeopardy*-style Quiz
Games	5 (5.4%)	Word Guess, Fruit Cutting

top 100 of their corresponding categories.

Impact of hidden crowdturfing apps. Furthermore, our study shows that the apps with hidden crowdturfing UIs have indeed successfully infiltrated App Store. Figure 6 illustrates the *Version* distribution of the crowdturfing apps. Most of them (53%) have only few updates, with a version number in the range from 0 to 1.5. However, still a large portion of apps (40% apps have *Version* $\geq$ 2.5) seem to be capable of carrying their suspicious payloads even to their higher versions. This is interesting since apps need to go through Apple's inspection for every new version submitted to the App Store.

Then, we analyzed the trend of the infiltration performed by the crowdturfing apps. Figure 7 shows the distribution of the number of such apps on the Apple App store over their release date. The trend-line based on the linear forecast regression indicates that those apps are still on the rise and require further attention. We observed that the newly-released apps with hidden crowdturfing UI have increased by 150% from Jan. 2015 to Jun. 2017.

4.3 App Development and Promotion

App development. Apparently, the development of crowdturfing apps is in strong demand on the underground market. Our research shows that one could get an illicit app, with desired hidden UIs, on the App Store for $450 [25]. Specifically, a quick search on Google yields dozens of recruitment posts for such app development; e.g.,*freelancer* [24,25], *Code Mart* [17], *witmart* [51], *dongcoder* [22], *Code4App* [16]. As shown in the task description [25], the illicit app to be developed should be capable of displaying a benign UI during app vetting, and switching to an illicit UI once it is published on the App Store.

Also illicit app developers tend to minimize the effort to develop the benign UIs for covering the crowdturfing ones. One common approach they take is to hide the crowdturfing UIs to the app built upon an open source project ([31,35,43]). In particular, we extracted strings from the benign VCs of the detected crowdturfing apps, and then searched them in leading code repositories (e.g., Github). Interestingly, we found that the benign UIs of six crowdturfing apps come from two open source projects: *ESTMusicPlayer* and *LittleFrog-MusicPlayer*. Note that according to Apple's guidelines [21] (4.3 and 4.2.6), such template apps should have been rejected. However, we observe that Apple seems to loosen its policy, which makes developing such illicit apps easier. To verify the observation, we designed a hidden crowdturfing app by utilizing one of the open source projects, ESTMusicPlayer [35], as the benign template. The app successfully got into the App Store in two days (we removed the app immediately before any user downloaded it).

UI hiding techniques: Triggers. We found that such illicit apps utilize a spectrum of UI hiding techniques to evade app vetting, which are described as follows:

• *Logic bomb*. Apparently, the adversary tends to trigger hidden crowdturfing UI when certain conditions are met (e.g., after app vetting). Some detected hidden crowdturfing apps contain logic bombs; e.g., the app sets off the hidden crowdturfing UI when a specified time (e.g., after "2017-01-18 00:00:00"), location (e.g., "isCN"), or device information (e.g., connected to cellular) conditions are met. For instance, the crowdturfing UI in *cn.music.s3b* is only activated when the device is connected to network and has its area/language code set to "zh".

• *C2 server*. Like bots, apps with hidden crowdturfing UI are also found to leverage command and control servers (C2 servers) to trigger their hidden UIs. For instance, *com.catTestPlay.app* retrieves a "status" code from its web server *http://[domain]/itunes_app/sound_dog* to decide whether to switch to its hidden UI.

• *Scheme*. Another interesting observation is that the app developers utilize extremely sophisticated triggering conditions, which even require the user to take certain actions. An interesting example is that a hidden crowdturfing UI can only be invoked by a specific scheme. Those apps promoted themselves on the social networks or websites; when users download those hidden crowdturfing apps from the App Store, the promoted sites provide the users an activation link to trigger the hidden crowdturfing UIs. More specifically, when the activation link is clicked, a scheme (e.g., *babyforring://[params]*), that releases the illicit UI, is sent to the app.

• *Others*. Several other techniques are also used to differentiate normal users' devices and vetting environment. As an example, we observe that a UI is hidden by the combination of scheme and logic bomb: the app *com.qianying.music* will first determine whether a user has logged into her WeChat app on the device, and then release its illicit UI only when receiving a scheme from a specific website.

App Clones. We observed that illicit app owners resubmitting clones of removed or existing illicit apps by only changing their bundle IDs through different Apple developer IDs; e.g., after *com.cloud.NHCore* was removed from App Store, it was quickly resubmitted as *com.good.jingling*. Developers also submitted multiple repackaged apps containing the same hidden crowdturfing UI; e.g., two apps, music player *com.yueyuemusic* and eBook reader *com.Qingyu* app,

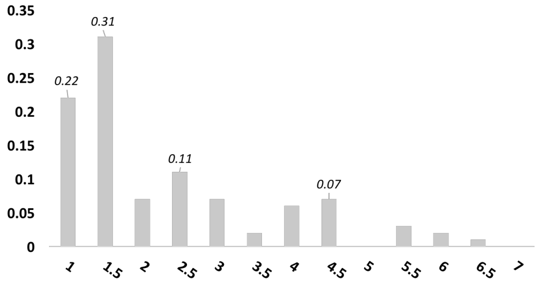


Figure 6: Version distribution of apps with hidden crowdturfing UIs.

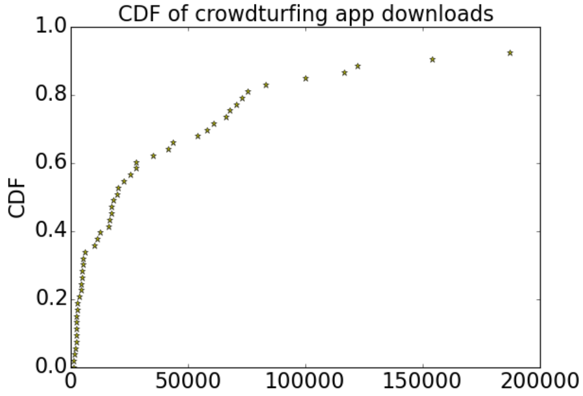


Figure 8: Cumulative distribution of crowdturfing app downloads.

were found to integrate the identical crowdturfing platform (i.e., *rehulu.com*). To mitigate the threat of such persistent infiltration attempts, we provided a list of words that could help to fingerprint such apps upon Apple's request, and meanwhile are actively collecting resubmitted/repackaged hidden crowdturfing apps.

App Promotion and worker recruitment. To understand how crowdturfing platform owners disseminate such apps and recruit workers, we searched for the apps' names on the search engine and manually analyzed top-10 results to identify their promotion websites. In this way, we gathered 50 websites advertising 78 (83.9%) hidden crowdturfing apps. We found that the owners of these hidden crowdturfing apps promote their apps through multiple channels: advertising on the online communities (e.g., *BBS*, *tieba*), social networks (e.g., youtube, weibo), and crowdturfing app gateway sites (e.g., *app522.com*, *i8i3.com*).

Of particular interest is the crowdturfing app gateway sites, which refer the visitors to multiple hidden crowdturfing apps. We identified 40 such gateway sites that promoted 63 (67.7%) hidden crowdturfing apps. For example, the *com.cq.diaoqianyaner.pro.bookstore* app was found to be promoted on eight crowdturfing sites: *qisw123.com*, *ydzapp.com*, *eshiwan.com*, etc. Most intriguing is the discovery that all the apps *actively* promoted on those gateway websites have been detected by *Cruiser* from the unknown set. Since those websites record apps' download volume, we were able to estimate the number of these apps' users. Figure 8 illustrates the cumulative distribution of the number of downloads per crowdturfing app. As shown in the figures, around 50% of the crowdturfing apps were downloaded more than 18K times, with 32.4 million downloads in total.

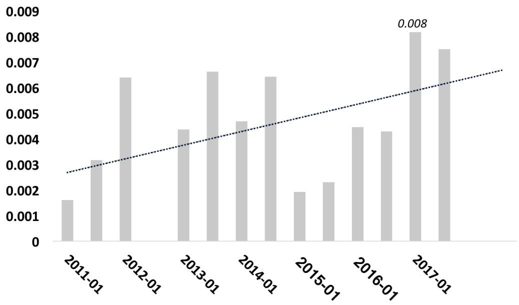


Figure 7: Release date distribution of apps with hidden crowdturfing UIs.

Another interesting promotion channel is the referral bonus policy, which is provided through the app: the app's owner pays workers (users) if they invite other workers to use this app for crowdturfing. We found that 23% of the crowdturfing apps are using such a channel to recruit workers.

4.4 Mobile Crowdturfing Operations

Crowdturfing tasks. Table 4 illustrates the top-6 most common illicit crowdturfing tasks found in the apps with hidden crowdturfing UIs. As we can see here, most of them are mobile based crowdturfing tasks. According to our findings, app ranking manipulation is supported by a significant portion (88.2%) of crowdturfing apps, followed by fraud account registration, and fake review. Figure 9 illustrates the cumulative distribution of the task categories per app. We observe that about 62.5% apps only provide one kind of crowdturfing tasks, among which 86.7% are designed for iOS app ranking manipulation. Surprisingly, when analyzing apps seeking crowdturfing for iOS app ranking manipulation, we observe several popular and reputable apps. Examples include a calendar app, which ranked Top 10 in the App Store category of Utilities across 15 countries, and a restaurant review app, which ranked Top 10 in Lifestyle category across 49 countries.

To measure the task volume of an app (i.e., number of tasks × number of required workers per task), we crawled five apps' task information and the number of required workers through their crowdturfing UIs. Table 5 presents the average daily task

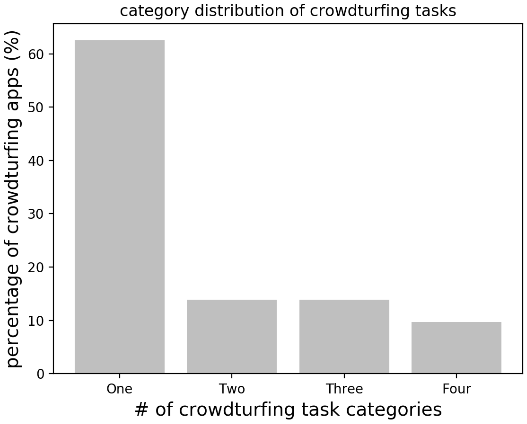


Figure 9: Distribution of the categories of crowdturfing tasks per app.

Table 4: Top-6 most common illicit crowdturfing tasks in apps with hidden crowdturfing UIs.

Crowdturfing tasks	# apps	# download (K)	Highest ranking
App ranking manipulation	82	32,268	5
Fraud account registration	28	15,618	64
Fake review	13	1,218	79
Bonus scalping	11	13,990	18
Online blog reposting	9	14,602	19
Order scalping	9	601	122

volume for each app. For instance, the app ranking manipulation app *com.zhang.samusic* has a daily task volume of 42,064 for manipulating 24 apps. Given an average task price of $0.14, the revenue for all those tasks is around $5.88K.

Furthermore, we analyze network traffic of such apps to study their servers, which distribute the tasks to the apps (see Figure 2). Interestingly, due to the difficulty in publishing crowdturfing apps, we find that multiple servers even share one client. In particular, besides their own servers, six apps are found to receive crowdturfing tasks from seven other servers (e.g., *qumi.com* and *domob.cn*) and all these tasks are related to app ranking manipulation.

Campaign discovery. In contrast to the web-based crowdturfing platforms [49], which are dominated by a few popular websites, we observed that the iOS-based crowdturfing platforms are more diverse. To study the relations among these crowdturfing apps, we built a graph for campaign discovery and further manually analyzed large campaigns identified. In the graph, each app is regarded as a node, and an edge connecting two apps represents that they are all from the same developer, with similar code or similar network behaviors. In particular, we crawled apps' developer information from the iTunes Preview website [34]. Then, we checked the common strings referenced by different apps' hidden crowdturfing UIs. If the strings from two different apps have more than 90% in common, we link them together. To capture the network behavior, we triggered all these apps by signing onto their platforms. If two apps' hidden crowdturfing UIs connect to the same server, we consider them to belong to the same campaign.

Table 5: Task volume and price of five apps with hidden crowdturfing UIs

App	# tasks	task volume	Per task price
com.zhang.samusic	24	42,064	0.14
com.roidmi.mifm	29	29,000	0.12
com.miaolaierge.iosapp	12	7,500	0.13
com.applyape.yycuimian	15	16,715	0.11
com.jialiang.weka	8	10,000	0.14

Table 6: Top-3 campaigns with most apps with hidden crowdturfing UI.

Campaign	# apps	Remote server
uxiaowei	9	uxiaowei.com
apptyk	6	apptyk.com laizhuan.com diaoqianyaner.com.cn
rehulu	6	rehulu.com

Table 6 shows top-3 campaigns with most crowdturfing apps. The largest one includes nine apps with hidden app ranking manipulation UIs, and all of them connect to the server *uxiaowei.com*. Interestingly, we observe that seven crowdturfing app owners (e.g., id109****906, id110****416, id110****262, id114****820) are related to this campaign. This campaign enjoyed a long lifetime, from May 2016 to March 2018.

4.5 Case Study

Here we introduce a typical app with hidden crowdturfing UI *sohouermusic*, which disguises as a music player, but also receives app ranking manipulation tasks (download, install, make up fake reviews, etc.). 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 redirect users to a website (*play.sohouer.com*). Only when a user visits the website on his iPhone and requires an invitation scheme *sohouermusic://invite=[serial number]* to be sent, will the app load its hidden UI. However, before the UI is actually rendered, the *sohouer* app checks whether it has passed the vetting process via its server, and the hidden crowdturfing UI shows up only when the remote server responds with *"isreview: 0"* and a *scripturl*. Besides acting as a client of a crowdturfing platform, such an app also stealthily collects user's data ; e.g., device type, version, jailbreak status, location. Another interesting observation is that the *sohouermusic* developers are persistent: after the *sohouermusic* app was removed (after we reported to Apple), the hidden crowdturfing UI was quickly repackaged into a *sohouercamera* app and was submitted through a different developer account.

5 Discussion

Evasion. The current implementation of *Cruiser* is based on identifying two types of conditionally triggered UIs for further semantic analysis (see Section 3.2). Hence, to evade *Cruiser*, the adversary may use the hidden crowdturfing UI, which is triggered by users and also avoids the root UI. Such evasion techniques, however, will cause the possible crowdturfing UIs to be triggered during app vetting. This is because all clickable elements may be triggered by Apple employee's manual or automatic analysis during app vetting [47]. This defeats the purpose of hidden UI.

The adversary may play other evasion tricks, by hiding semantic texts on the hidden crowdturfing UI to downgrade the accuracy of the Semantic Analyzer. In particular, the adversary can show crowdturfing related texts in the images, or obfuscate class names and method names, even dynamically fetch the crowdturfing related content. One possible solution is to run an Optical Character Recognition (OCR) tool [36] to extract the texts from images in the resource files, which enables to identify enough UI semantic even when the code is obfuscated. Considering the dynamically fetched hidden crowdturfing content, the adversary may deliver it on runtime using dynamic code loading (e.g., JSPatch [12]). However, Apple regulates and carefully monitors those dynamic code enabling techniques (e.g., hot patching frameworks) to minimize the attack vector; recently, Apple even bans or rejects any apps that use hot patch [39] from their App Store.

Limitations. Although *Cruiser* can already achieve a precision around 90%, still human involvement is needed to ensure that the apps reported are indeed problematic. Therefore, in the current form, it can only serve as a triage tool, instead of a full-fledged detection system. Also, as mentioned earlier, our current design is focused on iOS based apps, since cybercriminals have more intentions to utilize hidden UI to infiltrate the iOS app store than that of Android: centralized app vetting and installation make it hard for the crowdturfing app to reach out to the iPhone users. In the meantime, based on our observations, such hidden crowdturfing apps exist, though less pervasive, in the Android world. In particular, we conducted a small-scale study to find whether our detected apps have Android versions by searching for app names on Google Play, third-party stores and app download portals, and further manually examining them. We did not find any hidden crowdturfing apps, but did observe blatant crowdturfing apps (without hidden UIs) in less regulated third-party Android app stores.

Moreover, besides crowdturfing, we do think that cybercriminals can use hidden UI techniques for other abusive services, such as delivering unauthorized content, or even malware. When looking into such apps (those found in our research to carry hidden UIs but not perform crowdturfing), we found instances such as covering an phishing UI behind a travel app. A natural follow-up step is to investigate all abusive services exploring hidden UI to infiltrate the iOS app store and characterize the underground markets behind them. We will leave this as our future work.

Ethical issue. Our research only involved analysis of preexisting code and app content and did not collect new data during the study. Therefore, it is just a secondary analysis of already published materials, which does not constitute human-subject research. Another ethical concern comes from the potential that *Cruiser* could be used to identify possible benign hidden UIs; e.g., for censorship circumvention. Here we clarify that *Cruiser* is just a methodology for discovery and understanding of a new type of cybercrime, and during our study, we did not observe any such censorship evasion attempts. We acknowledge that any evasion detection techniques, including ours, could also be used for censorship. In the meantime, our methodology has been tailored towards crowdturfing detection: e.g., the features used by the structure miner are based upon the structures of real-world crowdturfing apps, the Word2vec model and other NLP components are all built on crowdturfing data. We are not sure how effective our approach would be when applying it to detect other types of hidden content, and how much additional effort is needed to make it a full-fledged censorship tool.

Responsible disclosure. Since the discovery of apps with hidden crowdturfing UI, we have been in active communication with Apple. So far, we have reported all the apps detected in our research to Apple, who has removed all of them from the App Store; also upon Apple's request, we provided a list of fingerprints for eliminating the similar apps.

6 Related Work

Study on crowdturfing. The ecosystem of *web-based* crowdturfing has been studied for long. Motoyama et al. [37] identified the labor market Freelance involved in service abuse (e.g., fraud account creation) and characterized how pricing and demand evolved in supporting this activity. Wang et al. [49] studied two Chinese online crowdturfing platforms and also revealed the impact of the crowdturfing followers task on those platforms to microblogging sites. Stringhini el al. [45] investigated five Twitter follower markets to study the size of these markets and the price distribution of their service.

Su et al. [46] studied the spamming activity of "Add To Favorites" by collecting the several "Add To Favorites" tasks information from one crowdturfing platform. In our research, to the best of our knowledge we for the first-time investigate the crowdturfing platforms on the mobile devices, and reveal several unique characteristics; e.g., fragmented crowdturfing markets, mobile targeted crowdturfing tasks, stealthy worker recruitment channel, hidden crowdturfing UI techniques.

Illicit iOS app detection. Compared with Android, the Apple platforms are much less studied in terms of their security protection. Egele et al [23] proposed PiOS, which uses control flow analysis to detect privacy leaks in iOS apps. Deng et al [18] presented an approach to detect private API abuse by binary instrumentation and static analysis. Chen et al. [14] determines potentially harmful iOS libraries by looking for their counterparts on Android. Bai et al. [11] and Xing et al. [53] uncovered several zero configuration and cross-app resource sharing vulnerabilities, and proposed the corresponding detection methods. Understanding the security implications of hidden crowdturfing UI in iOS apps has never been done before. Also, none of the prior research provides a UI based detection mechanism to identify illicit iOS apps with hidden UI.

Text analysis for mobile security. Numerous studies have looked into *apps' UI texts* to detect mobile threats such as task jacking, mobile phishing attack, ransomware, or to protect user privacy. AsDroid [33] checks the coherence between the semantics of the UI text (e.g., text of button) and program behavior associated with the UI (e.g., button) to detect malicious behavior (e.g., sensitive API) in Android apps such as sending short messages and making phone calls. Heldroid [10] uses a supervised classifier to detect threatening *sentences* from Android apps to detect ransomware. SUPOR [32], UIPicker [38] and UiRef [9] identify sensitive user inputs within user interfaces to protect user privacy. In particular, SUPOR [32] extracts layouts by modifying the static rendering engine of the Android Developer Tool (ADT). UIPicker [38] operates directly on the XML specification of layouts. UiRef [9] resolves the semantics of user-input widgets by analyzing the GUIs of Android applications. It improves the accuracy of SUPOR by addressing ambiguity of descriptive text through word embedding. In addition to UI texts, researchers intensively leverage Natural Language Processing (NLP) to process *app descriptions* for mobile security research. Examples include WHYPER [40] and AutoCog [41], which check whether an Android app properly indicates its permission usage in its app description, CHABADA [29] applied topic modeling technique on an app's text description to help infer user's expectation of security and privacy relevant actions. Different from previous works, our work compared the semantics of conditionally triggered UI texts of iOS apps, crowdturfing keywords and app descriptions to identify hidden crowdturfing apps. Also, sensitive or private API are not used for detection in our work as the illicit behavior of the app we detect are based on UI not API. Also, different from SUPOR, UIPicker and UiRef, we extract UI texts from UI hierarchies (LVCG) we generated from iOS apps.

7 Conclusion

In this paper, we report our study on illicit iOS apps with hidden crowdturfing UIs, which introduce conditionally triggered UIs and a large semantic gap between hidden crowdturfing UI and other UIs in the app. Exploiting these features, our crowdturfing UI scanner for iOS, *Cruiser*, utilizes iOS UI hierarchy analysis technique and NLP techniques to automatically generate a UI hierarchy from binary and UI layout files and investigate conditionally triggered UI and the semantic gap to identify such illicit apps. Our study shows that *Cruiser* introduces a reasonable false detection rate (about 11.1%) with over 94.1% coverage. Running on 28K iOS apps, *Cruiser* automatically detects 93 apps with hidden crowdturfing UIs, which brings to light the significant impact of such illicit apps: they indeed successfully infiltrate App Store, even bypassing app vetting several times. What is worse, we observed an increasing trend of the number of such apps in App Store. Our research further uncovers a set of unique characteristics of iOS crowdturfing, which has never been revealed before: for example, we observe several remote crowdturfing servers share one iOS crowdturfing app as a client, which may be due to the difficulty of infiltration; also, such illicit apps were promoted by crowdturfing gateway sites to recruit workers, etc. Moving forward, we further investigate the hidden UI techniques providing by illicit app developers, including logic bomb, command and control infrastructure, and scheme technique etc.

8 Acknowledgements

We are grateful to our shepherd Gianluca Stringhini and the anonymous reviewers for their insightful comments. This work is supported in part by NSF CNS-1801365, 1527141, 1618493, 1801432, 1838083 and ARO W911NF1610127.

References

[1] Amazon mechanical turk: Access a global, on-demand, 24x7 workforce. https://www.mturk.com.

[2] Google translate. https://translate.google.com.

[3] Number of apps available in leading app stores 2018. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[4] Sandaha. http://sandaha.cc.

[5] Zhubajie. https://www.zbj.com.

[6] App annie. https://www.appannie.com/en/, Mar. 2010.

[7] Capstone: The ultimate disassembler. http://www.capstone-engine.org, Nov. 2013.

[8] 91ssz. A website that provides ios apps with illicit features. http://www.91ssz.com/app/iphone/, Mar. 2017.

[9] B. Andow, A. Acharya, D. Li, W. Enck, K. Singh, and T. Xie. Uiref: analysis of sensitive user inputs in android applications. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 23–34. ACM, 2017.

[10] N. Andronio. *Heldroid: Fast and Efficient Linguistic-Based Ransomware Detection*. PhD thesis, 2015.

[11] X. Bai, L. Xing, N. Zhang, X. Wang, X. Liao, T. Li, and S.-M. Hu. Staying secure and unprepared: understanding and mitigating the security risks of apple zeroconf. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 655–674. IEEE, 2016.

[12] bang590. Jspatch: bridging objective-c and javascript using the objective-c runtime. https://github.com/bang590/JSPatch, May 2015.

[13] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[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 *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 357–376. IEEE, 2016.

[15] G. Cheng. *7 Winning Strategies For Trading Forex: Real and actionable techniques for profiting from the currency markets*. Harriman House Limited, 2007.

[16] Code4App. Code4app: Looking for ios chameleon app developer. http://www.code4app.com/thread-14820-1-1.html, Sep. 2017.

[17] coding mart. Recruitement for ios chameleon app developer. https://mart.coding.net/project/11325, Nov. 2017.

[18] Z. Deng, B. Saltaformaggio, X. Zhang, and D. Xu. iris: Vetting private api abuse in ios applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 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] Freelancer. Freelancer: looking for developer for lottery chameleon app. https://www.freelancer.com/projects/php/app-edt-15321896/, Apr. 2017.

[25] Freelancer. We need to do a universal application on ios, and then display our url through the interface. https://www.freelancer.com/projects/iphone/need-universal-application-ios-then/, Apr. 2017.

[26] B. J. Frey and D. Dueck. Clustering by passing messages between data points. *science*, 315(5814):972–976, 2007.

[27] fxsjy. Jieba chinese text segmentation. https://github.com/fxsjy/jieba, Jul. 2013.

[28] Google. Developer policy center. https://play.google.com/about/developer-content-policy/#!?modal_active=none, Dec. 2017.

[29] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1025–1035. ACM, 2014.

[30] T. S. N. L. P. Group. Stanford word segmenter. https://nlp.stanford.edu/software/segmenter.shtml, May 2006.

[31] hellclq. ios app: Happy english sentences 8k. https://github.com/helloclq/HappyEnglishSentences8000, Aug. 2013.

[32] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang. Supor: Precise and scalable sensitive user input detection for android apps. In *USENIX Security Symposium*, pages 977–992, 2015.

[33] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1036–1046. ACM, 2014.

[34] A. Inc. itunes preview (app store). https://itunes.apple.com/genre/ios/id36?mt=8, Jul. 2008.

[35] P. King. Estmusicplayer. https://github.com/Aufree/ESTMusicPlayer, Nov. 2015.

[36] S. Mori, H. Nishida, and H. Yamada. *Optical character recognition*. John Wiley & Sons, Inc., 1999.

[37] M. Motoyama, D. McCoy, K. Levchenko, S. Savage, and G. M. Voelker. Dirty jobs: The role of freelance labor in web service abuse. In *Proceedings of the 20th USENIX conference on Security*, pages 14–14. USENIX Association, 2011.

[38] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang. Uipicker: User-input privacy identification in mobile applications. In *USENIX Security Symposium*, pages 993–1008, 2015.

[39] T. C. P. N. NETWORK. Apple removes 45,000 apps in china. http://www.asiaone.com/digital/apple-removes-45000-apps-china, Jun. 2017.

[40] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. Whyper: Towards automating risk assessment of mobile applications. In *USENIX Security Symposium*, pages 527–542, 2013.

[41] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen. Autocog: Measuring the description-to-permission fidelity in android applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1354–1365. ACM, 2014.

[42] D. Quesada. ios interface builder utility. https://github.com/davidquesada/ibtool.

[43] SimonLo. Hulumusic. https://github.com/SimonLo/HuluMusic, Apr. 2017.

[44] J. Song, S. Lee, and J. Kim. Crowdtarget: Target-based detection of crowdturfing in online social networks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 793–804. ACM, 2015.

[45] G. Stringhini, G. Wang, M. Egele, C. Kruegel, G. Vigna, H. Zheng, and B. Y. Zhao. Follow the green: growth and dynamics in twitter follower markets. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 163–176. ACM, 2013.

[46] N. Su, Y. Liu, Z. Li, Y. Liu, M. Zhang, and S. Ma. Detecting crowdturfing add to favorites activities in online shopping. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 1673–1682. International World Wide Web Conferences Steering Committee, 2018.

[47] M. Tabini. How apple is improving mobile app security. https://www.macworld.com/article/2047567/how-apple-is-improving-mobile-app-security.html, SEP 2013.

[48] S. T. M. Toolbox. Stanford word segmenter. https://nlp.stanford.edu/software/tmt/tmt-0.4/, May 2006.

[49] G. Wang, C. Wilson, X. Zhao, Y. Zhu, M. Mohanlal, H. Zheng, and B. Y. Zhao. Serf and turf: crowdturfing for fun and profit. In *Proceedings of the 21st international conference on World Wide Web*, pages 679–688. ACM, 2012.

[50] Wikipedia. Word2vec: a model to produce word embeddings. https://en.wikipedia.org/wiki/Word2vec, Feb. 2018.

[51] witmart. Buy covering ios apps for 30,000 cny. http://www.witmart.com/cn/app-software/jobs/jobid_34788.html, Oct. 2017.

[52] C. Xiao. Pirated ios app store's client successfully evaded apple ios code review. https://researchcenter.paloaltonetworks.com/2016/02/pirated-ios-app-stores-client-successfully-evaded-apple-ios-code-review/, Feb. 2016.

[53] L. Xing, X. Bai, T. Li, X. Wang, K. Chen, X. Liao, S.-M. Hu, and X. Han. Cracking app isolation on apple: Unauthorized cross-app resource access on mac os. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 31–43. ACM, 2015.

9 Appendix

9.1 Performance evaluation of *Cruiser* and *NaiveCruiser*

To understand the performance of *Cruiser*, we measured the time it takes to process all the apps in the unknown set, on our Red Hat server using 14 processes. The breakdowns of the delays observed at each stage (Structure Miner and Semantic Analyzer) are reported in Table 7. As we can see here, on average, 27.4 seconds were spent on each app. The results demonstrate that *Cruiser* scales well and can easily process a large number of iOS apps. Furthermore, we evaluated the performance of *NaiveCruiser* (Table 7). As we can see, in the absence of the conditionally triggered UI detection step to first filter out legitimate VCs, the performance overhead of the Semantic Analyzer became overwhelming: introducing a delay at least 14 times as large as our original approach, which makes it difficult to scale. In addition, we evaluated the performance of app collection. On average, downloading an app took 15 seconds and decrypting it took 10 seconds; however, the time varied greatly depending on the network speed, program sizes and etc. In total collecting and decrypting apps took 3 months.

Table 7: Running time at different stages, where SM means Structure Miner and SA means Semantic Analyzer.

Cruiser	Average time (s/app)	*NaiveCruiser*	Average time (s/app)
SM	18.88	LVCG construction	16.2
SA	8.56	SA	122.95
Total	27.43	Total	139.15

9.2 UI element objects without semantic UI texts

Table 8: UI element objects without semantic UI texts

Pattern type	**UI element object**	**Parent UI element object**[1]
A[3]	NSKey	*[2]
A	UIColor	*
A	UIFont	*
A	UINibKeyValuePair	*
A	NS.rectval	*
A	UIViewContentHuggingPriority	*
A	UIViewContentCompressionResistancePriority	*
A	UIOriginalClassName	*
A	UINibName	*
A	UIDestinationViewControllerIdentifier	*
A	UIActionName	*
A	UISource	*
A	UIDestination	*
A	UIStoryboardIdentifier	*
A	NSLayoutIdentifier	*
B[4]	UIProxiedObjectIdentifier	UIProxyObject
B	UIAction	UIStoryboardUnwindSegueTemplate
B	UIKeyPath	_UIAttributeTraitStorage
B	_UILayoutGuideIdentifier	_UILayoutGuide
B	UIKeyPath	_UIRelationshipTraitStorage
B	runtimeCollectionClassName	UIRuntimeOutletCollectionConnection

[1] Parent UI element object: The parent UI object of UI element object.

[2] *(asterisk): Any Object.

[3] Type A: The string of a UI element object will be removed regardless its parent.

[4] Type B: The string of a UI element object will be removed only if its parent UI element object also matches.