No Source Code? No Problem! Demystifying and Detecting Mask Apps in iOS

Yijun Zhao*
Institute of Information Engineering, Chinese Academy of Sciences
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
zhaoyijun@iie.ac.cn

Lingjing Yu*
Institute of Information Engineering, Chinese Academy of Sciences
Beijing, China
yulingjing@iie.ac.cn

Yong Sun
Institute of Information Engineering, Chinese Academy of Sciences
Beijing, China
sunyong@iie.ac.cn

Qingyun Liu
Institute of Information Engineering, Chinese Academy of Sciences
Beijing, China
liuqingyun@iie.ac.cn

Bo Luo
Department of Electrical Engineering and Computer Science
The University of Kansas
Lawrence, KS, USA

ABSTRACT

The rise of malicious mobile applications poses a significant threat to users and app stores. While iOS apps have generally been considered more secure due to strict review processes and limited distribution avenues, developers have found ways to evade scrutiny by disguising malicious apps as benign "Mask Apps". Mask Apps activate hidden functionalities after the user downloads or with a trigger event. The malicious and potentially illegal hidden function poses significant risks, including privacy breaches, security vulnerabilities, and harm to legitimate businesses. However, existing defenses are ineffective against Mask Apps developed in web or hybrid models. In this paper, we propose Mask-Catcher, an automated approach that uses four filtering mechanisms to detect Mask Apps. Mask-Catcher leverages inconsistencies between app descriptions and user reviews, inter-app recommendation relationships, and code similarities to discover and identify Mask Apps. Experimental results show that Mask-Catcher achieves high recall and precision when applied to real-world datasets from the Apple App Store.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

Mobile app security, Mask Apps, automated detection

ACM Reference Format:
Yijun Zhao, Lingjing Yu, Yong Sun, Qingyun Liu, and Bo Luo. 2024. No Source Code? No Problem! Demystifying and Detecting Mask Apps in iOS. In *32nd IEEE/ACM International Conference on Program Comprehension (ICPC '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3643916.3644419

*Both authors contributed equally to this research.

ICPC '24, April 15–16, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0586-1/24/04.
https://doi.org/10.1145/3643916.3644419

1 INTRODUCTION

With the rapid growth of smartphone apps, the presence of malicious apps also increases correspondingly. In response, app markets have implemented a range of countermeasures aiming at curbing such malicious applications. For instance, according to Apple's App Store Transparency Report in 2022 [7], out of 6,101,913 applications submitted for review, a substantial 679,694 applications were rejected by Apple's screening process. In contrast to Android applications, which could be easily distributed through third-party platforms to bypass Google's security review, the pathways to distribute iOS apps besides the official App Store are associated with significant intricacies and costs. In particular, publishing apps on TestFlight entails a less extensive review process, but the apps are only valid for 90 days. iOS Super Signing offers another avenue for app distribution without undergoing a formal review. However, developers are charged based on the number of app installations, resulting in a substantial financial burden to achieve widespread distribution. Moreover, users tend to exhibit a strong preference for acquiring apps exclusively from the App Store. This collective trend in user behavior, the rigorous scrutiny of applications, and the elevated cost of third-party distribution methods play a pivotal role in fortifying the overall security of the iOS application ecosystem.

In response to the stringent security protections of the App Store, malicious developers employ a strategy that involves attaching a "mask" to malicious applications (referred to as "*Mask Apps*" hereafter), thereby concealing them as benign apps. This approach allows these malicious applications to escape Apple's official scrutiny and make their way to the iOS App Store. The Mask Apps typically feature very simple user interfaces (UIs) to provide benign functionalities that are somewhat naive, such as a straightforward game, so that they could quickly pass the review process for inclusion in the App Store. The hidden functionalities and the corresponding concealed UIs are only unveiled after users install the app and "activate" the hidden UIs/functions, using methods that are pre-configured by the developer, e.g., search for a specific text string.

In practice, a considerable number of users exhibit a preference for the Mask Apps, primarily because the concealer features often align with their specific needs, such as gambling or accessing pirated content. These users learn the activation methods for the hidden functionalities through various grey channels, including online forums and chat groups. This situation may initially appear to be a "mutually beneficial collaboration", where developers and users collude to circumvent iOS App Store regulations in order to fulfill their respective needs. However, beyond the apparent engagement in distributing potentially illegal content, this arrangement introduces additional risks. To remain profitable, the developers of Mask Apps frequently incorporate grey market advertisements or, in some cases, even malware into such Mask Apps. Users, in their pursuit of immediate gratification, tend to overlook potential risks. This oversight can lead to privacy breaches, financial fraud, device compromise, and other security concerns.

Lee et al. identified a type of Mask Apps for the first time in their measurement study on iOS apps with hidden crowdturfing UIs[19]. These apps display "innocent-looking UIs" during iOS App Store review process and convert to a hidden, potentially harmful illicit UI with crowdturfing content after they are published. They employ binary analysis to identify the conditionally triggered hidden UIs, and then further examine UI layout and content to confirm the malicious crowdturfing functionalities. Lee et al. further extended the work to detect all types of malicious hidden behaviors in iOS apps (denoted as *Chameleon Apps*) [20]. However, Chameleon-Hunter [20] only works for Native Apps, while most Mask Apps are now developed with hybrid mode (please refer to Section 2 for details). Meanwhile, the reverse engineering and analysis of the binaries are very time-consuming (29.11 seconds/app in [20]), hence, the scalability of Chameleon-Hunter is a concern.

In this paper, we present Mask-Catcher, which is designed to automatically and efficiently identify Mask Apps.Mask-Catcher constitutes a two-step filtering mechanism. In the *suspicious app discovery module*, it utilizes a series of highly efficient, high recall, and relatively low precision filters, which exploit the discrepancies between app descriptions and user reviews and the inter-app recommendation relationships to identify a pool of candidate Mask Apps. In the *Mask App identification module*, it employs code similarity analysis among Mask App families to confirm the Mask Apps. To evaluate the efficacy of Mask-Catcher, we first explored side channels to discover a labeled dataset of 180 Mask Apps. Mask-Catcher was evaluated and demonstrated a recall of 99.12% and a precision of 100%. Furthermore, we applied Mask-Catcher to an unlabeled dataset collected over six months from the iOS App Store. Mask-Catcher further identified 31 new Mask Apps, underscoring the real-world effectiveness of Mask-Catcher.

The contributions of this paper are three-fold: (1) we present an analysis of the three distinct development models of the Mask Apps and a measurement study of the state-of-the-art Mask Apps. Our analysis underscores the inadequacy of relying solely on native code examination for the detection of Mask Apps. (2) We are the first to identify the discrepancies between the app descriptions and the app reviews, as well as the inconsistencies in the app recommendations for these Mask Apps. We argue that such discrepancies could be employed for efficient filtering of potential Mask Apps. And (3) we introduce Mask-Catcher, a relatively lightweight tool that efficiently and accurately detects Mask Apps. Unlike previous methodologies, Mask-Catcher can successfully identify Mask Apps developed using all three development methods. Finally, we share the source code of Mastk-Catcher at https://github.com/Junzy71/Code_of_Mask_App, and the dataset of known mask apps at https://github.com/Junzy71/Dataset_of_Mask_Apps.

The rest of this paper is organized as follows: we introduce app development models, define and examine the Mask Apps, and illustrate the discovery process of 180 Mask Apps in Section 2. We present the technical details of Mask-Catcher in Section 3, followed by experiments and discussions in Section 4. We summarize the literature in Section 5 and conclude the paper in Section 6.

2 MASK APPS: DEFINITION AND DISCOVERY

2.1 Preliminaries: Three Models of Mobile App Development

Mobile apps are developed in three models: native, web, and hybrid.

The native model is the most conventional app development approach. Apps are developed with the official languages (Swift and Objective-C for iOS, and Java for Android), libraries, and tools. They are capable of directly accessing the hardware of the phone, e.g., camera, microphone, messaging, and geolocations. Native apps usually provide better stability, performance, and interactive user experiences. However, they require higher development costs, substantial storage spaces, and frequent updates. They cannot achieve cross-platform compatibility due to the divergence in official programming languages across different platforms. Consequently, developers must develop native apps individually for each platform, which introduces challenges in compatibility and higher costs.

Web apps are developed using web technologies such as HTML5, JavaScript, and CSS, enabling them to be seamlessly accessed through web browsers on all platforms. Web apps have gained popularity among developers due to their low development costs, short development cycles, and ease of maintenance. Nonetheless, web apps come with inherent limitations attributed to the nature of web technology. They rely on browser support for both display and user interaction, they require consistent internet connectivity, and they cannot be included or advertised in the app marketplace.

Hybrid apps blend the strengths of both native and web apps. Similar to web apps, hybrid apps leverage HTML and CSS to design user interfaces, establish core functionalities, and manage user interactions. These components are then rendered via a *webview* to ensure cross-platform versatility. Furthermore, they tap into native device features, e.g., camera and GPS, via the integration of APIs facilitated by a JavaScript Bridge (JSBridge). Hybrid app development has evolved into the mainstream of app creation, as it enjoys the advantages of web-based app development but still has access to native device capabilities.

2.2 An Introduction to Mask Apps

We define a Mask App as a mobile application that exhibits the following characteristics:

1. The app successfully navigates through the official review process, gaining approval for publication in the App Store.

2. In the official App Store description, it claims benign functions.

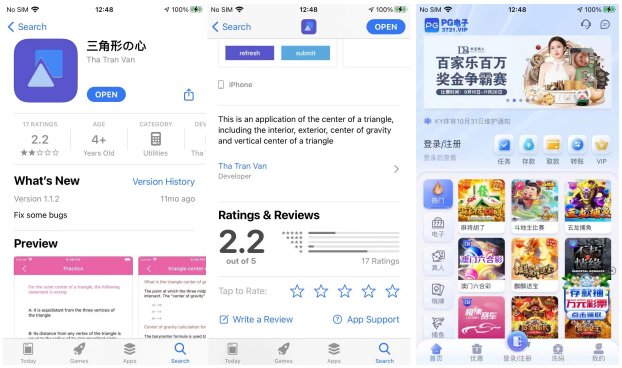

Figure 1: An Example of a Mask App. Left and middle: the "Center of Triangle" app, which appears to be benign. Right: the app transforms into "3721.VIP", a gambling app.

3. The app includes a concealed activation mechanism, which is used to trigger its hidden functionalities. These mechanisms can include actions like clicking on specific locations, searching for specific text strings, or simply restarting the app.
4. Upon activation, the app transforms to unveil a new user interface and expose its hidden functions, which often involve activities that are either malicious, prohibited by the app marketplace's policies, or even illegal. In essence, the app would not pass the review process, if these hidden functions were disclosed during the process.

In theory, the Mask Apps may appear in any platform or app marketplace. However, all the Mask Apps discovered in this project, as well as the Chameleon Apps discovered in [20], are iOS apps in the Apple App Store. This could be explained by the fact that Android devices allow third-party apps to be installed through alternative Android markets or using apk files downloaded from any website. Therefore, the adversaries do not need to spend a lot of effort to deliver their apps through the official Play Store.

An example of a typical Mask App is shown in Figure 1. The app, named "Center of Triangle", appears to have an appropriate icon, a concise description, and all necessary declarations such as developer, version, age rating (4+), and even permission requests. It claims to provide simple functions to compute the centroid, circumcenter, incenter, and orthocenter of triangles. Consequently, it was placed in the "Utilities" category of the App Store. However, once it is installed on the user's device, it transforms into "3721.VIP" upon the first launch. As shown in the figure, 3721.VIP is a gambling app, which is illegal and not allowed on Apple's App Store in China.

2.3 Mask App Discovery

The Mask Apps always appear to be benign on the App Store, however, they obviously do not intend to attract users with their benign functions. Therefore, the developers still need channels to promote the hidden functionalities and attract interested users to download the Mask Apps. To collect a dataset of Mask Apps, we exploited various channels including covert websites and chat groups that are utilized by developers to distribute the App Store links to the Mask Apps and the activation methods. These channels are typically not accessible through search engines.

Our initial sources for collecting Mask Apps were the iOS Resource Sites [23]. The managers of these sites regularly collected and posted data regarding Mask Apps, including the activation methods for the hidden functions. Their underlying motivation for this practice was to leverage these resources to entice users to pay for the premium content on their websites. Notably, the Mask Apps featured on these platforms were primarily providers of pirated movies, music, and e-books. This focus was due to the perception that promoting pornographic and gambling apps carried a higher risk that outweighed the potential benefits.

We followed the iOS Resource Sites to collect the first batch of 97 Mask Apps. We expanded the collection by exploring the websites and advertisements contained within these apps. The developers' official websites featured in the Mask Apps emerged as the most reliable and stable source for the Mask Apps. The developers often first release new Mask Apps or re-produced Mask Apps on their official websites. They may proactively notify users of new apps through in-app announcements, especially when they fear that the current app may be banned by the App Store.

The advertisements within the Mask Apps served as a significant source of new Mask Apps. Notably, such ads often promote Mask Apps with pornography and gambling functions, which are often deemed illegal or at least inappropriate on public platforms. While such in-app advertisements generate additional revenue for the Mask App developers, they also helped us collect samples of gambling and pornography-related Mask Apps.

After obtaining these apps and learning their activation methods, we manually verified them with the participation of four authors (two students and two faculty). In the verification process, we confirm their activation and distinct hidden functions. The activation process and functional distinctions of Mask Apps are quite evident, leading to a high level of verification confidence.

Over ten months, we successfully collected 180 Mask App samples and activated their hidden features. We share metadata of all the mask apps at https://github.com/Junzy71/Dataset_of_Mask_Apps. As of the submission of this manuscript, only three of them remain accessible in the iOS App Store. To facilitate future research, we have stored all metadata and binary files of apps, to be shared with the research community.

2.4 Mask Apps: Features and Measurements

For all 180 apps in our dataset, we track their existence in the App Store on a daily basis and report their lifespan statistics in Figure 2. 5% of the Mask Apps remain on the App Store for less than a week, and 36% of them vanish in a month. These Mask Apps are non-compliant with the regulations of the Apple App Store or even prohibited by law. Hence, they are often quickly removed from the App Store when they are detected or reported.

Next, through our analysis of the collected mask apps, we observed that the majority of them were developed using the hybrid mode. This development model is consistent with their short lifespan: once a Mask App is removed by the App Store, its developer needs to promptly create and publish a new app with the same concealed functionalities to sustain profitability. Hybrid apps, with their advantages of lower cost and shorter development cycles, align well with their needs.

During the design phase of the project, we closely monitored iosre.com (an iOS technical forum where discussions related to mask

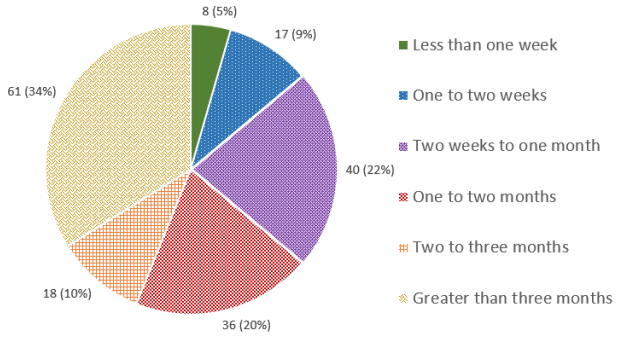

Figure 2: Lifespan of Mask Apps: 5% of Mask Apps remain in the App Store for less than a week, 36% vanish in a month.

apps can be found through search [3]) and www.freelancer.com (a platform for recruiting developers and project discussions [2]). We have sporadically seen messages on mask apps or hidden functionalities on these forums. We have engaged with 6 developers who posted messages on the development of Mask Apps or hidden functionalities. We also contacted 5 very active app developing service providers each on taobao.com and zbj.com, who advertised for mask app development. We asked about the basic mechanism they use to develop these apps. We asked about the basic mechanism they use to develop these apps. They suggested that we provide H5 links for the hidden functions and opt for a hybrid development model, citing the convenience and cost-effectiveness of this approach.

In the development of Mask Apps, developers generate H5 pages for hidden, malicious functions. They then quickly produce distinct UIs (the "masks") to conceal these H5 pages using development frameworks, resulting in different Mask Apps. These "masks" are typically simple pages with minimal functionalities and interactions, often comprising H5 pages or native UIs with few components.

It is important to note that the code for the hidden functions in Mask Apps developed in web or hybrid modes does not appear in the binary or UI layout files. Depending on the developer's choice, this code may be stored as HTML or JavaScript files in a separate local folder or on a remote server. Consequently, when we adopted the detection mechanism in Chameleon-Hunter [20] to analyze the apps in our dataset, only 28 of the 180 Mask Apps contained binary code with hidden functions in the IPA files or hidden UI elements with hidden functions in their UI layout files. This implies that the other 152 apps were *not* developed in native mode, which is why the binary code and UI analysis approach in Chameleon-Hunter cannot detect these Mask Apps. The authors of Chameleon-Hunter have also acknowledged this limitation in [20].

Finally, we observed that the client-side native code for Mask Apps with similar hidden functions demonstrates a significant degree of similarity. This similarity can be attributed to their utilization of the same development framework and/or code architecture, where the hidden functionalities constitute only a small fraction of the overall codebase. We were able to classify the Mask App samples into distinct *Mask families* and employ code similarity assessments to determine if an app bears a resemblance to a specific Mask family. This method is applicable to apps developed in native, web, and hybrid modes.

3 MASK-CATCHER: MASK APP DETECTION

3.1 Challenges, Design Rationale, and Solution Overview

The identification of Mask Apps on iOS faces several key challenges: (1) iOS does not provide access to the source code of applications, making it difficult to directly analyze the code to uncover hidden functionalities. Despite the possibility of conducting binary analysis or decompilation, these methods do not provide as much information as the source code would. (2) The emergence of web and hybrid apps has presented a new trend in Mask App development. Unlike native apps, many of these web and hybrid apps include concealed functions that are not directly embedded in the app's source code. These hidden functions are typically displayed through web interfaces or other external mechanisms rather than being explicitly coded within the app itself. Consequently, traditional approaches relying solely on code inspection become ineffective in identifying Mask Apps developed using web or hybrid techniques. (3) There is a significant number of apps in the App Store (~1.8 million in Nov 2023), while many Mask Apps often have very short lifespans. An effective detector must be highly efficient and scalable.

To tackle these challenges, we have observed two key features that inspired the design of the proposed approach:

1. Advertising Reviews posted by the Developer. We observed that app developers may provide subtle "hints" to users regarding the hidden functionalities. They achieve this by posting reviews since these user-generated reviews are not strictly monitored by Apple. By exploring app reviews, we can identify patterns and clues that deviate from the app's declared purpose. For instance, the reviews of a simple game (Mask App) may mention "abundant video resources." This approach becomes particularly relevant when we notice reviews that are posted soon after the app's release, as they are likely to have been generated by the app's producers.

2. Users' Discussions in the Reviews. We also found that some users also post reviews that inadvertently "reveal" the hidden functionality of the Mask App. These reviews tend to discuss features or functions that are completely unrelated to the app's officially declared purpose. For example, app *com.carte.feflmlsy* claims to be an app to design, manage, and share business cards, but we can see that it has user reviews such as "The videos can't play," and "you have to watch ads before every episode." These sentences should theoretically be used to review movies. By analyzing such reviews, we can gather insights into the app's actual capabilities.

3. App Store Recommendations. Additionally, the recommendations provided by the App Store itself can implicitly reveal the actual functionality of an app, e.g., the recommended apps for an emoji app (Mask App) may have several video player apps in the list. These recommendations, which are generated based on various factors, including user behavior, app characteristics, and potentially app source code (Apple has access to the code), can often provide valuable hints about the app's hidden functions. By considering these aspects, our proposed approach leverages app reviews and App Store recommendations to uncover the true nature of Mask Apps, bypassing the limitations posed by source code unavailability and the complexities associated with web and hybrid app development methods.

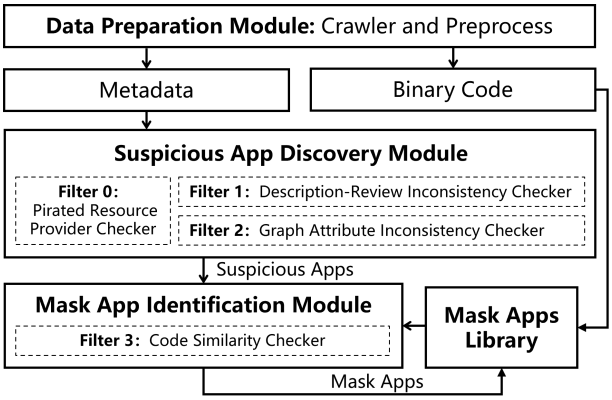

Figure 3: Overview of Mask-Catcher.

An overview of Mask-Catcher is illustrated in Figure 3, which includes a data preparation module, a Mask Apps Library, and four filters to identify Mask Apps. In Mask-Catcher, we first employ methods with high efficiency, high recall, but lower precision to filter a large number of apps and discover only a small number of suspicious apps. Then the static analysis method with higher accuracy and higher cost is used to eventually identify Mask Apps.

In the data preparation module, we employ a crawler tool to acquire app metadata from the App Store. If an app is identified as a suspicious Mask App by Filter 0 or Filter 2, we proceed to crawl the IPA file of this app from iTunes. Subsequently, we decrypt the file and extract the binary file from it. The data preparation module also analyzes the collected Mask App samples and categorizes them into various Mask families based on their concealed functionalities and UIs. They are subsequently integrated into the Mask Apps Library.

The suspicious app discovery module consists of three filters. *Filter 0* serves as a complementary component to Filter 1 and Filter 2 in detecting Mask Apps that closely align their hidden functions with their claimed function, e.g. a pirated movie app (Mask App) that claims to be a viewer of legitimate movies. They specifically target apps in categories such as movies, comics, music, and books. *Filter 1* extracts the claimed function from the app descriptions and the hidden function from user reviews. It then identifies candidate apps with significant inconsistencies between these two sets of functions. *Filter 2* employs the identified candidate apps as seeds to establish a graph convolutional network using inter-app recommendation relationships. Suspicious apps are detected by evaluating the consistencies in the attributes of the graph nodes. This filter is especially effective for apps that lack user reviews as well.

Finally, the Mask App identification module (Filter 3) evaluates the code similarity of suspicious apps. If a suspicious app exhibits high similarity with any Mask family, it is identified as a Mask App and is subsequently incorporated into the Mask Apps Library.

3.2 Data Collection and Preparation

We developed a custom crawler to collect a comprehensive dataset of apps available in the App Store. The crawler randomly selects 10 apps from each category of the App Store as seeds. It follows the inter-app recommendations to crawl additional apps. Only apps that were added or updated within the past six months are retained for further examination since the Mask Apps are quickly removed from the App Store as we discovered in Section 2.3. For the selected apps, we collect their most recent 500 reviews and other metadata, such as app descriptions and inter-app recommendations.

We eliminate information from the app descriptions and user reviews that are not related to the app's functionalities. For example, some developers include subscription service information, official account details on other platforms, or email addresses and telephone numbers in the app description. We manually developed rules to filter out sentences containing this redundant information, thereby retaining only those sentences relevant to the app's functionalities. Meanwhile, a substantial proportion of user reviews in the app's review section typically consist of compliments or criticisms that express emotions without directly relating to the app's functionalities, such as "it's great,""recommended," or "can't open it at all." We also employ manually crafted regular expressions to eliminate these uninformative user reviews.

We further tokenized the descriptions and reviews and removed any stopwords. In line with the specific characteristics of the descriptions and user reviews, we have deliberately constructed both a reserved word list and a list of approximately 100 stopwords, which are specifically compiled for app reviews.

Compared to binary code analysis, metadata in text format is notably faster to collect, process, and analyze. This efficiency is a key reason why Mask-Catcher is capable of rapidly discovering and identifying Mask Apps at a large scale.

3.3 Filter 0: Pirated Resource Provider

The suspicious app discovery module is primarily designed to detect suspicious Mask Apps by analyzing the inconsistencies between their claimed functionalities and their hidden functionalities. However, we have observed that a small portion of Mask Apps intentionally conceal their actual functionalities by closely aligning them with their claimed functionalities. For instance, an app that primarily offers pirated movies may claim to provide legitimate subscription service to movies. In such cases, the consistence-based filters are unlikely to identify inconsistencies between app reviews and app descriptions, or inconsistencies with app recommendations. To address this issue, we introduce Filter 0 as a complementary component to flag this small amount of apps to be directly fed to the Mask App Identification module for further analysis.

We utilize regular expressions on app descriptions to identify apps from the following categories: movies, comics, music, and books (categories that may have pirated content). Among these categories, we manually verify and exclude the top-ranked apps from reputable developers or companies. Next, we identify apps within these categories that had a significant number of user reviews containing specific content indicating the provision of abundant resources, e.g., "lots of book sources," "you can watch movies by just searching", or "you can watch VIP content for free." The rich resource offering would typically appear only in top apps with significant investments. It is quite suspicious for a small vendor to receive such reviews. Furthermore, we observed that some users even asked questions in their reviews such as "how to activate?" or "What's the activation code?" These inquiries further strengthen the suspicion that an app might be a Mask App. If an app from the

specified categories exhibits multiple such reviews, we label it as suspicious and directly send it to Filter 3 for identification.

3.4 Filter 1: Suspicious App Discovery Based on Description-Review Inconsistency

For a Mask App, there is often a disparity between the functionalities claimed in the app description and those mentioned in user reviews. For a benign app, the claimed and the hidden functionalities should be consistent. However, in the case of a Mask App, the claimed functionalities specified in the description serve as a disguise, while the functionalities discussed in user reviews are its true/hidden functionalities. This discrepancy provides a basis for discovering suspicious Mask Apps. Filter 1 leverages this observation to identify suspicious apps quickly and cost-effectively, thereby reducing the volume of data that needs to be processed by the subsequent, more resource-intensive modules.

We employ *Latent Dirichlet Allocation* (LDA) to extract topics from app descriptions and user reviews. LDA condenses a predetermined number of topics, each consisting of a topic probability and a set of associated topic words. Also, each topic word has a topic-word probability which represents how likely the the word belongs to this topic. We apply LDA on full app descriptions to identify 25 topics, which matches the number of categories in the App Store. We extract topic words with a topic-word probability exceeding 0.005 within each topic, constructing a topic word list based on these selections. Simultaneously, we include words that describe app functionalities in this topic words list (e.g. "notes", "copies" and "fonts"), with the objective of encompassing common functionalities that may not be covered within the training corpus. This adaptation allows Mask-Catcher to be flexible and applicable in various scenarios. Subsequently, descriptions are processed in accordance with the topic words list, retaining only the topic words present in the list as the claimed functionalities of the apps.

Given the inherent differences in the hidden functionalities of apps even within the same category, we opt for an individualized approach for user reviews. For each app, we employ LDA with a set number of topics, which in this case is 5. The hidden functionalities of the app are determined as the topic words whose sum of topic-word probabilities surpasses 0.005 among all topics with topic probabilities exceeding 0.01. Any remaining words are then excluded from the user reviews.

Upon functionality extraction, the original descriptions are transformed into a set of words representing the claimed functionalities, while the original user reviews are represented as another set of words corresponding to the hidden functionalities.

Next, *Word2vec* is utilized to map the representative words of claimed and hidden functionalities to two sets of 64-dimensional vectors (V_d and V_a). The cosine similarity of word pairs is calculated as: $Sim_{ij} = (v_{di} \cdot v_{aj})/(\|v_{di}\|\|v_{aj}\|)$, where v_{di} represents the i-th word vector in V_d and v_{aj} represents the j-th word vector in V_a. We then employ three methods to calculate the cosine similarities between these two sets of vectors as follows:

(1) Average Similarity Sim_{avg}: we calculate the cosine similarities of all the word pairs from claimed and hidden functionalities. The average similarity, Sim_{avg}, reflects the overall similarity between the claimed and hidden functionalities.
(2) Similar-words similarity Sim_{sw}: we calculate the cosine similarities of all word pairs and rank them in descending order, and then take the average of the top 10% as Sim_{sw}. This similarity focuses on similar word pairs and therefore reduces the number of false positives caused by the neglect of small but highly similar word pairs.
(3) High-frequency-words similarity Sim_{hfw}: we calculate the cosine similarities of all word pairs, and rank them in descending order according to the frequency of word pairs. We take the average of the top 10% of them as Sim_{hfw}. This similarity pays more attention to the words that frequently occur in the claimed and hidden functionalities, which reduces false positives due to low-frequency word pairs.

Finally, we set thresholds for each of the three similarities at 0.3, 0.3, and 0.6, and we consider an app to be benign if it has more than two similarities above the thresholds.

Following this filtering process, a significant portion of benign apps are eliminated, resulting in a small number of candidate Mask Apps progressing to Filter 2 for subsequent investigation.

3.5 Filter 2: Suspicious App Discovery Based on Recommendation Relationship Graph

Apple's App Store employs a recommendation system between apps, often referred to as "You Might Also Like." While the detailed mechanism behind the recommendations is not released, it is expected that it is based on user behaviors (e.g., frequently downloaded by users with similar preferences) and functionalities (since Apple has the code). Leveraging this inter-app recommendation relationship among Mask Apps, Filter 2 utilizes the candidate apps identified by Filter 1 and apps without user reviews as seeds to construct a recommendation relationship graph. Filter 2 attempts to discover the true identities (i.e., functions) of the apps through graph analysis. For instance, when an app declares itself as a utility app but many of its recommendations fall in the category of movies, the app is very likely to have a hidden function as a (pirated) movie app, thus it is suspicious. Meanwhile, Filter 2 may also introduce additional Mask Apps that were not included in the seed set since Mask Apps are often recommended by other Mask Apps.

A recommendation relationship graph is an undirected graph $G = \{N, E, C_d, C_a\}$, where the set of nodes N denotes the Apps and the set of edges E represents the recommendation relationships between apps. If App A's recommendation list includes App B, then an edge is drawn between node A and node B. Each node has two attributes: the claimed categories set (C_d) and the hidden categories set (C_a). The set of categories can have values within a range of 25 categories, which we redefine based on the clustering results of the *K-Means* algorithm applied to the descriptions. The complete redefined categories can be found in [1]. The reason for this redefinition is that the categories of apps in the App Store are typically chosen by developers, and the functionalities of many apps can span multiple categories, e.g., a children's song app mostly is categorized into "Kids", but it also belongs to Music. The claimed categories set of each app is obtained through two mechanisms: (1) directly extracted from App Store Categories; (2) mined from app descriptions and app names using Random Forest.

We train a three-layer graph convolutional network (GCN) to discover the *hidden categories set* of nodes. The feature vectors of nodes are initialized to the hidden functionalities vectors calculated in subsection 3.4, denoted X. The layer-wise propagation rule of GCN is defined as follows:

$$H^0 = X$$
$$H^1 = ReLU(AH^0W^0 + B^0)$$
$$H^2 = ReLU(AH^1W^1 + B^1)$$
$$H^3 = softmax(H^2W^2 + B^2)$$

where A is the adjacency matrix of the recommendation graph with added self-connections. W is the weight matrix for each layer, B is the bias. The activations $ReLU$ and $Softmax$ are defined as:

$$ReLU(x_i) = max(0, x_i)$$
$$Softmax(x_i) = exp(x_i)/\sum_j exp(x_j)$$

For training, we used a cross-entropy loss function, defined as:

$$L(Y, Lable) = -\sum_i lable_i logy_i$$

The predicted result of the GCN for the i-th node is denoted as y_i, and the label for the i-th node is denoted as $label_i$. Nodes for apps in the unlabeled dataset are initialized with the claimed categories.

The model generates a category-probability vector representing the probability that apps belong to each category. If the probability that an app belongs to a category is higher than a predetermined threshold, the category is added to its hidden categories set. Note that the significantly smaller number of Mask Apps compared to benign apps leads to the data imbalance issue. Therefore, we do not use GCN to directly determine whether an app is a Mask App. Instead, we use GCN to predict the set of hidden categories.

Filter 2 proceeds to detect suspicious apps by examining the inconsistencies between the claimed and hidden category sets. If there is no commonality between the claimed and hidden categories for a node, the corresponding app is deemed suspicious. Note that Filter 2 can discover suspicious Mask Apps without user reviews, solely based on inter-app recommendation relationships. This highlights the versatility and effectiveness of the approach.

3.6 Filter 3: Mask Apps Identification Based on Code Similarity

The hidden functionalities of the Mask Apps often deviate from the expectations outlined in the App Store's guidelines and, in many cases, may even be unlawful. App Stores routinely review apps using methods like user reports or random inspections, Mask Apps often do not last long in the App Store. Hence, developers of Mask Apps need to quickly create and publish new versions to sustain profitability. Since the claimed functionalities (the "masks") of Mask Apps are not their primary focus but need to be replaced and updated frequently, developers often use simple user interfaces and basic functionalities developed with the same development framework, which make up a small portion of the app's codebase. The true/hidden functionalities of a Mask App can be re-published multiple times by the same developer or by different developer accounts. Consequently, multiple Mask Apps may have similar or even identical hidden functionalities, despite having different "masks". This similarity in development process functionalities leads to similar code blocks of these apps being alike. A suspicious app can be identified as a Mask App if its code is similar to that of previously verified Mask Apps.

Due to the closed source nature of iOS, precise reverse engineering of iOS apps can be challenging. To preserve information and enhance accuracy, we identify Mask App directly by the the assembly code. We employ BinDiff [12] of IDA Pro to compare the similarities and discrepancies between two files in terms of functions, calls, basic blocks, jumps, and instructions. It then calculates the similarity of the two files and provides a confidence score.

The data preparation module constructs a library using the collected Mask Apps and categorizes them into 22 families based on their concealed functionalities and user interfaces. We compute the code similarity between Mask Apps within each family. We then chose the Mask App in each family that exhibits the highest similarity to other members of the family as the representative.

For each suspicious app identified in Filter 0 and Filter 2, we obtain a similarity vector representing the likeness of the app to each of the Mask family representatives. We then utilize a random forest classifier to categorize these app similarity vectors and determine whether the app is a Mask App or not.

The method may not always accurately identify newly emerging Mask Apps with hidden functionalities and UIs that differ significantly from the samples in the Mask App library, which is a natural limitation of the approach. In our future work, we plan to continuously collect Mask Apps from various sources such as news, forums, and other channels to expand the library and enhance the effectiveness of Mask-Catcher.

4 EXPERIMENT RESULTS

4.1 Data Collection and Metrics

We collected metadata of 70,678 apps from all 25 categories in the App Store using the approach described in Section 3. Out of these apps, 43,868 had user reviews. 3,545 apps were identified as suspicious by Filter 0 and Filter 2 in Mask-Catcher, and their IPA files were also crawled through ipatool [5]. We decrypted these IPA files using frida [6] on an iPhone 7 (14.0.0) and extracted the binary files. We selected apps that ranked within the top 100 of the App Store's rankings for a continuous period of more than 90 days, and labeled the top 1,500 as benign. In order to accurately assess the effectiveness of each filter of Mask-Catcher, we used the dataset consisting of the 1500 benign apps and the manually collected 180 Mask Apps mentioned in Section 2 as our labeled dataset. We have made our dataset publicly available on Github: https://github.com/Junzy71/Dataset_of_Mask_Apps.

We evaluate the four filters using $Recall(R)$ and $Precision(P)$, and also considered the $FilterRate(FR)$ particularly for Filters 0, 1, and 2. This is because the suspicious app discovery module is designed to efficiently filter out the majority of benign apps.

$$R = \frac{TP}{TP + FN} \qquad P = \frac{TP}{TP + FP} \qquad FR = \frac{TN}{TN + FP}$$

where, TP (true positive) denotes number of Mask apps labeled as suspicious (or Mask Apps for Filter 3), FN (false negative) denotes Mask apps labeled as benign, TN and FP denotes benign apps labeled as benign and suspicious (or Mask Apps for Filter 3), respectively.

Table 1: Performance of Mask-Catcher on Labeled Dataset

	TP	FN	TN	FP	R	FR	P
Filter 0	2	0	349	4	100.00%	98.87%	33.33%
Filter 1	161	1	1432	56	99.38%	96.24%	74.19%
Filter 2	179*	1	390	129	99.44%	75.14%	58.12%
Filter 3**	180.4	0.6	133	0	99.67%	-	100.00%
Mask-Catcher	180.4	1.6	1624	0	99.12%	-	100.00%

*: These 179 apps contain 161 TP samples from Filter 1 that are confirmed by Filter 2, plus 15 additional apps that do not have any review (thus not processed by Filter 1), plus 3 apps newly added through graph analysis, one of which was misclassified by Filter 1.
**: Cross-validation is used for Filter 3, and the metric here is the average of 10 validations. These 181 apps (TP+FN) contain 179 TP samples from Filter 2, plus 2 additional apps from Filter 0.

4.2 Performance of Mask-Catcher

4.2.1 Performance on labeled dataset. We evaluate Mask-Catcher on the labeled Mask Apps and benign apps to assess the effectiveness of each filter and to compare its overall performance with that of Chameleon-Hunter.

We first evaluate Filter 0 on the entire labeled dataset and identified 6 apps (4 benign apps and 2 Mask Apps) as suspicious. These 6 apps were directly forwarded to Filter 3 for further evaluation, bypassing the intermediate stages of Filters 1 and Filter 2.

In the remaining apps, 162 Mask Apps and 1488 benign apps had user reviews. We evaluate Filter 1 with these apps as Filter 1 relies on user reviews to identify suspicious apps. The labeled apps without user reviews will be handled by Filter 2. As shown in Table 1, Filter 1 successfully identified 161 labeled Mask Apps and 56 labeled benign apps as candidate Mask Apps. This indicated a recall of 99.38% and a filter rate of 96.24%. The precision, which is less important in this step, is 74.19%. In particular, filter 1 effectively handles 73 labeled Mask Apps with fewer than 20 reviews, showcasing its effectiveness with sparse user data. We examined the app *com.lovely.photo*, which was misclassified by Filter 1, and found that it has similar claimed functionalities (video editing) and hidden functionalities (video playback) thus it is difficult for Filter 1 to determine whether it has a "Mask" only through the description and user reviews.

Filter 2 utilized the 217 candidate apps discovered by Filter 1 (161 were labeled Mask App and 56 were labeled benign apps), along with 24 labeled apps without user reviews, as seeds to construct a recommendation relationship graph with an inter-app recommendation depth of 1. The constructed graph consisted of 699 nodes and 1,799 edges, with 14 nodes being isolated and directly classified as suspicious apps. Ultimately, Filter 2 identified 182 seed apps (176 Mask Apps and 5 benign apps) as suspicious. Additionally, 126 apps in the graph that were not part of the labeled dataset were also labeled as suspicious. After manual verification, 2 out of these 126 newly discovered suspicious apps were real Mask Apps. In total, Filter 2 achieved a 99.44% recall, a 58.12% precision and a 75.14% filtering rate.

For the 16 Mask Apps without user reviews, Filter 2 labeled 15 as suspicious. The misclassified app *com.yskdt.tools*, whose claimed and hidden functionalities are both movie-related, and therefore the inter-app recommended apps also belong to the Entertainment category, resulting in mis-identification in Filter 2. Meanwhile, the app *com.lovely.photo*, which was misclassified by Filter 1, was included in the recommending lists of other Mask Apps and therefore was re-discovered by Filter 2 and labeled as suspicious, as shown in Table 2. This demonstrates that Filter 2's ability to extract additional information from app recommendation relationships played a key role in identifying the app's true identity. Filter 2's enhanced capability to leverage these inter-app relationships contributes to a more accurate and comprehensive detection of Mask Apps.

It is important to note that our prioritization in the development of suspicious app discovery module (including Filters 0, 1 and 2) is to maximize the recall, while accepting a relatively high filtering rate and a lower precision. This approach is chosen to ensure that a higher number of Mask Apps could be fed to Filter 3 for further identification, while also ensuring the efficiency of the Mask-Catcher by filtering out benign apps before they reach Filter 3.

All suspicious Mask Apps were evaluated by Filter 3 to make a final determination of whether they were indeed Mask Apps. To assess the performance of Filter 3, we employed cross-validation. In each iteration, 60% of the labeled apps were randomly chosen as the training dataset for training the random forest classifier. The remaining 40% of the labeled apps were then used as the testing dataset. This process was repeated ten times.

The evaluation results demonstrated that Filter 3 accurately identified all 133 benign apps. However, in six out of ten repetitions, it misclassified one Mask App, specifically the "XinYee.Com" app, as benign. This misclassification occurred because the family to which this app belonged had only one sample in our Mask App Library, whereas most other families had 5 to 10 different samples. When this app was not included in the training dataset, the classifier failed to learn the features of the family, leading to misclassification in Filter 3. In summary, Filter 3 achieved an average recall of 99.67% and 100% precision.

An additional noteworthy observation is that a considerable number of the newly suspicious apps discovered by Filter 2 have names that are strikingly similar to the Mask Apps we have collected. Moreover, the user reviews associated with these apps predominantly contain phrases such as "Fake" and "Can't activate." This pattern suggests that the developers behind these apps are attempting to deceive users by posing as legitimate Mask Apps. However, through the utilization of our detection method, these apps were correctly identified as not being Mask Apps. By distinguishing these deceptive apps from authentic ones, our approach demonstrates the robustness and proficiency in identifying Mask Apps and mitigating the risks posed by such fraudulent applications.

4.2.2 Performance Comparison with Chameleon-Hunter. To compare Mask-Catcher with a state-of-the-art Mask App detector, we applied the mechanism of Chameleon-Hunter on our labeled dataset, and the results are shown in Figure 4. Mask-Catcher had a better performance with 99.12% recall and 100% precision, while Chameleon-Hunter had the performance with 15.38% recall and 75.68% precision. This is because almost all Mask Apps in this dataset are developed in Web or Hybrid mode, while Chameleon-Hunter only analyzes the native UIs of the apps, and thus cannot achieve better performance.

4.2.3 Performance on unlabeled dataset. We applied Mask-Catcher to all collected apps that were unlabeled, which included 69,178

Table 2: Result Examples of Mask-Catcher on Labeled Dataset

Bundle ID	Category	Claimed Functionality	Hidden Functionality	Filter 0	Filter 1	Filter 2	Filter 3
cn.onedayapp.onedayNovelRedesign	Utilities	local reader	pirated book reader	✓	-	-	✓
com.fymjtv	Entertainment	drama community	pirated movies	✓	-	-	✓
com.lovely.photo	Utilities	video editing	pirated movies	-	×	✓	✓
com.yskdt.tools	Utilities	movie knowledge quiz	pirated movies	-	-	×	-
com.mahuaFunAgeAdver.summerrain	Utilities	cycling record	pirated movies	-	-	new	✓
com.eightfive.zhizuoqrcode	Utilities	QR code generation	pirated movies	-	-	new	✓

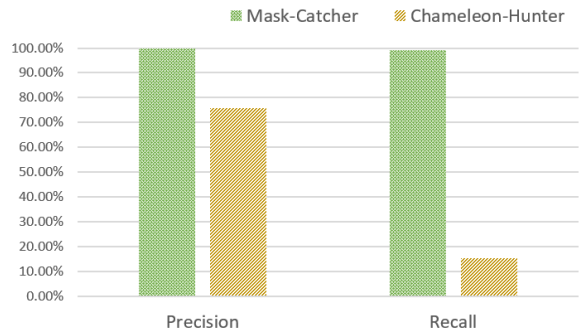

Figure 4: Performance comparison of Mask-Catcher and Chameleon-Hunter on Labeled Dataset.

apps, 42,376 of which had user reviews. Filter 0 labeled 399 apps as suspicious. Filter 1 labeled 6,870 apps as suspicious, and they were used by Filter 2 as a seed to construct the recommendation relationship graph along with 26,802 apps without user reviews. The graph consisted of 112,954 nodes and 407,233 edges, of which 92 nodes were isolated. Eventually, 1,217 apps in the seeds were continued to be labeled as suspicious and 1,615 additional apps were labeled as suspicious. In the Mask App identification module, 31 suspicious apps, which were not included in the set of 180 known Mask Apps, were identified as Mask Apps. Examples of the new Mask Apps are shown in Table 3. The complete table can be found in [1]. The majority of these newly identified Mask Apps had emerged only after August 1st, 2023.

In manual verification, we successfully activated the hidden functions of 28 apps, which were found to be substantially different from their claimed functionalities. However, there were 3 apps for which we could not find the activation methods through conventional means. Therefore, we manually examined their metadata for further insights. The user reviews for the app *com.llc.weeingcarManager* fell into three main categories. Firstly, there were reviews containing only the phrase "cute girl", which was a popular activation code in the labeled Mask Apps. Secondly, there were queries regarding app activation, such as "Why can't I activate?" and "What's the activation code?". Lastly, there were responses to the above queries, providing instructions such as "Enter the code word in the text box feedback". These findings strongly suggest that the apps in question are indeed Mask Apps, as the activation method was explicitly mentioned in the user reviews. However, it appears that the app developers have subsequently changed their activation methods.

We also examined the Mask Apps discovered from the unlabeled dataset using Chameleon-Hunter, which successfully identified 6 out of the 31 Mask Apps. Chameleon-Hunter's performance is limited, primarily because hybrid apps are dominant in Mask App development due to their convenience and cost-effectiveness.

Finally, as shown in [20] and in our observations, the Mask Apps are extremely sparse in the App Store. It is impractical to manually validate all apps labeled as benign by Mask-Catcher. Hence, we randomly selected 200 apps from benign apps labeled by Mask-Catcher and manually validated these apps using the following methods: (1) inspecting their metadata, including descriptions, user reviews, authors, etc., (2) examining the user interface, and (3) searching in online communities and developer forums. The process was very time-consuming, which again confirms that it is impractical to manually identify the Mask Apps. Eventually, we did not find any hint of Mask App behaviors in these 200 samples.

4.2.4 Time Consumption of Mask-Catcher and Chameleon-Hunter. We measured the time consumption of each filter using 14 processes (same setup as Chameleon-Hunter [20]). As shown in Table 4, the computational cost for Filters 0 to 3 increased as expected. Notably, the time consumption for the metadata-based suspicious app discovery module (Filters 0 to 2) was considerably lower than the code-similarity-based Mask App identification module (Filter 3). This observation highlights the effectiveness of the progressive filtering approach in Mask-Catcher, leading to a significant improvement in overall efficiency.

In total, Mask-Catcher spent approximately 117.5 hours on data processing and the four filters, resulting in an average processing time of 6.11 seconds per app. This is notably faster compared to the 29.11 seconds per app achieved by Chameleon-Hunter [20].

4.3 Discussions

4.3.1 Attack Against Mask-Catcher. Mask-Catcher relies on app metadata, including app descriptions, user reviews, and inter-app recommendations in discovering and identifying Mask Apps. A knowledgeable attacker may deliberately manipulate the metadata in an attempt to compromise the effectiveness of Mask-Catcher. However, it is indeed challenging for attackers to interfere with inter-app recommendations as they are primarily governed by Apple's proprietary algorithms. The recommendations are influenced by numerous factors, including user patterns, app categories, and more. Influencing app recommendations would require significant resources and costs beyond what Mask App developers may be willing to invest. Regarding user reviews, developers of Mask Apps typically want users to use their apps, and they cannot control the users' feedback on the hidden functionality of the app, which can be captured by Mask-Catcher.

Table 3: Newly Discovered Mask Apps by Mask-Catcher from Unlabeled Dataset

Bundle ID	Category	Claimed Functionality	Hidden Functionality	Lifespan	verified
duoduo.video.generator	Utilities	video and photo editing	pirated movies	2022-06-01 to 2023-10-21(208 days)	×
com.tristan.aichat	Lifestyle	AI assistant	pirated music	2023-06-06 to 2023-10-17(133 days)	×
com.llc.weeingcarManager	Utilities	wedding car rental	pirated movies	2023-07-03 to 2023-08-27(55 days)	×
com.crz.beishuChallenge	Games	calculation game	pirated movies	2023-09-04 to 2023-10-10 (36 days)	✓
com.learnsindhialphabets	Education	Sindhi alphabets learning	pornographic video	2023-09-14 to 2023-10-07 (23 days)	✓
com.yefad.FACEGIJU	Lifestyle	emoticon stickers	pornographic video	2023-09-20 to 2023-10-09 (19 days)	✓
com.ineya.ROLINXUE	Entertainment	emoticon stickers	pornographic video	2023-09-20 to 2023-10-09 (19 days)	✓
com.xtxgj.tools	health & Fitness	hypoglycemia record	pirated movies	2023-09-25 to 2023-10-17 (22 days)	✓

Table 4: Time consumption of each filter.

Module/Filter	Total	Number of apps	Average
Data preparation	40min	69,178	35ms/app
Filter 0	23s	42,376	0.5ms/app
Filter 1	4min	41,977	6ms/app
Filter 2	46min	112,954	24ms/app
Filter 3	116h	3,231	129s/app

The attackers have more control over App descriptions. They can provide descriptions that are closely aligned with the hidden functionality of the app. However, this attack method is only viable for apps that have similar functions as legal apps so that they can declare such legal functions. For example, an app providing pirated movies could pretend to be a movie app. However, if the hidden functionality falls into an illegal category, such as gambling, the description can hardly describe a function that is both "consistent" with gambling and does not trigger alerts to Apple's reviewers. In response, Mask-Catcher employs a specific analysis for this type of Mask Apps, i.e. Filter 0, which is detailed in Section 3.

Regarding the source code, Mask App developers could employ different masks in each iteration of their development process. However, as described in Section 2, the short lifespan of Mask Apps necessitates frequent mask changes. Developing new mask templates for each iteration would result in prohibitively high costs, which do not align with the cost requirements of Mask App developers.

4.3.2 Limitations. The limitations of Mask-Catcher primarily remain in the following two aspects: (1) Accessing metadata, especially collecting reviews, in the app discovery module requires some delay after the app has been published on the App Store. During this time, potentially harmful apps may already have been downloaded and used by users, leading to potential damage. Filter 3 partially mitigates this issue by directly analyzing apps without reviews using more comprehensive code analysis techniques. This filter could be utilized as soon as an app is published in the App Store, thus minimizing the time gap between app publication and analysis, allowing for more timely identification of Mask Apps and reducing the risk to users. In practice, the choice of which filters to use can be adjusted based on the specific circumstances. If there is a need for more immediate detection, the focus can be shifted towards utilizing Filter 3 to prioritize code analysis methods if higher overhead can be accepted. This flexibility allows for a customizable approach that balances efficiency and accuracy while mitigating potential harm to users. (2) While Filter 3 in Mask-Catcher uses code similarity as an identification criterion, it is true that new Mask App families may emerge, which cannot be accurately identified due to the evolving nature of the threat landscape. As new Mask App families emerge with unique code patterns, it may take time for Mask-Catcher to adapt and accurately identify them. It is our future work to add a novelty detector in Filter 3 to identify new app samples with moderate similarity to existing Mask Apps and pass them to human evaluators to confirm. It is also our plan to maintain and expand the Mask App library.

To address this limitation, continuous efforts are required to collect samples of newly discovered Mask App families. By regularly updating the database with these samples, Mask-Catcher can gradually narrow down the identification gap and improve its ability to detect and classify emerging Mask App families.

5 RELATED WORK

In this section, we summarize research works in the literature that are most relevant to this paper: (1) app metadata analysis, (2) identification of suspicious app behaviors, and (3) the detection of disguised apps.

The metadata of apps contains extensive information pertaining to their functionalities. AUTOREB [18] leverages user reviews to analyze whether an app exhibits security and privacy-related behaviors. CHAMP [16] employs a semi-automated approach to construct semantic rules from developer policies across multiple app markets and utilizes the semantic rules to identify malicious app behaviors. [25] detects excessive permission requests based on comparisons with similar apps. PRISharer [11] further categorizes apps based on their metadata and subsequently examines user reviews of similar apps to identify potential permission misuses. KEFE [33] uses a regression model to identify the key functions of apps from the descriptions and reviews. In this paper, we extract the hidden functions of the apps from user reviews and then compare them with the claimed functions in app descriptions to identify potentially suspicious apps.

Several studies have employed feature-based deep learning methods [13] or static and dynamic binary code analysis [8, 9, 21, 22, 30, 37, 39] to identify malicious apps. WHYPER [28] selects three commonly used permissions for protecting sensitive resources and uses NLP methods to identify the reasons behind their usage based

on app descriptions. AutoCog [29] extends WHYPER to 11 permissions, significantly improving recall and precision by considering description-to-permission fidelity. Wang et al. [31] extract features from decompiled code and train classifiers to evaluate two sensitive permissions. Instead of detecting malicious permissions or behaviors, CHABADA [14] directly examines whether an app provides the declared functionalities. Zhang et al. [36] consider the impact of third-party libraries, reducing CHABADA's false positives by 54%. Le et al. [35] simultaneously analyze privacy policies, bytecode, descriptions, and permissions to evaluate app permission requests. IoTProfiler [27] and IoTPrivComp [4] examine data collection and sharing practices in IoT companion apps through static and dynamic analysis, and then evaluate them against the apps' privacy policies. ACODE [32] summarizes the four main factors that contribute to inconsistencies between textual descriptions and the use of privacy-sensitive resources. In addition to metadata, other studies [17, 34, 38] utilize text found in user interfaces to detect suspicious app behaviors. However, these research efforts primarily focus on the Android platform and do not specifically address the detection of Mask Apps in iOS.

Several reports in recent years have highlighted the presence and the risk of Mask Apps [10, 15, 24, 26]. Cruiser [19] made the first attempt to identify iOS apps with hidden crowdturfing UIs, which shed light on the mobile-based crowdturfing ecosystem and the tactics employed by underground developers to evade app review. They further introduced Chameleon-Hunter [20], which uses static analysis to examine the binary files and UI layouts of apps to identify hidden UIs. Features derived from these UIs and metadata are then utilized to ascertain the legality of these hidden UIs.

In comparison with Cruiser [19] and Chameleon-Hunter [20], Mask-Catcher has the following advantages: Mask-Catcher is capable of handling apps developed in native, web, and hybrid models, while Cruiser and Chameleon-Hunter only handles native apps. As we have observed in the recent trend, the vast majority of the Mask App are developed using the hybrid model. (2) The metadata-analysis mechanisms in Mask-Catcher (Filters 0, 1, and 2) efficiently eliminates ∼90% of the benign apps with extremely low overhead (30ms/app). With this filtering mechanism, it becomes practical to continuously monitor and examine all the apps in the App Store.

6 CONCLUSION

In this paper, we propose a novel approach called Mask-Catcher for the discovery and identification of Mask Apps, a type of malicious smartphone apps that claim a set of benign functions to pass the App Store review and transform into a different set of malicious/illegal functions after they are installed on users' devices and triggered by a specific activation activity. Mask-Catcher autonomously collects metadata, including app descriptions, user reviews, and inter-app recommendation relationships, along with the app IPA files from the App Store. It initially detects suspicious apps by analyzing the inconsistency in their claimed functions and the reviews and inter-app recommendations. It then employs highly accurate code similarity checking to identify Mask Apps. This progressive filtering approach significantly improves detection efficiency while maintaining high accuracy.

ACKNOWLEDGMENTS

Yijun Zhao, Lingjing Yu, Yong Sun, and Qingyun Liu were supported by the Scaling Program of the Institute of Information Engineering, CAS (Grant No. E3Z0191101) and the Scaling Program of the Institute of Information Engineering, CAS (Grant No. E3Z0041101).

REFERENCES

[1] 2023. Dataset. https://anonymous.4open.science/r/Dataset_of_Mask_Apps-Anonymize/readme.md.
[2] 2023. Hire Freelancers & Find Freelance Jobs Online | Freelancer. https://www.freelancer.com/.
[3] 2023. Re Forum - Internet people's Roman square. iosre.com.
[4] Javaria Ahmad, Fengjun Li, and Bo Luo. 2022. IoTPrivComp: A Measurement Study of Privacy Compliance in IoT Apps. In *European Symposium on Research in Computer Security*. Springer, 589–609.
[5] Majd Alfhaily. 2021. ipatool. https://github.com/majd/ipatool.
[6] Alone_Monkey. 2021. frida-ios-dump. https://github.com/AloneMonkey/frida-ios-dump.
[7] Apple. 2023. 2022 App Store Transparency Report. https://www.apple.com/legal/more-resources/docs/2022-App-Store-Transparency-Report.pdf.
[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 259–269. https://doi.org/10.1145/2594291.2594299
[9] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Tegawendé F. Bissyandé, Tianming Liu, Guoai Xu, and Jacques Klein. 2018. FraudDroid: automated ad fraud detection for Android apps. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 257–268. https://doi.org/10.1145/3236024.3236045
[10] TechNode Feed. 2023. Porn apps disguised as learning apps on China's iOS App Store. https://technode.com/2023/10/08/porn-apps-disguised-as-learning-apps-on-chinas-ios-app-store/.
[11] Hongcan Gao, Chenkai Guo, Guangdong Bai, Dengrong Huang, Zhen He, Yanfeng Wu, and Jing Xu. 2022. Sharing runtime permission issues for developers based on similar-app review mining. *J. Syst. Softw.* 184 (2022), 111118. https://doi.org/10.1016/j.jss.2021.111118
[12] google. 2023. bindiff. https://github.com/google/bindiff.
[13] M. Gopinath and Sibi Chakkaravarthy Sethuraman. 2023. A comprehensive survey on deep learning based malware detection techniques. *Comput. Sci. Rev.* 47 (2023), 100529. https://doi.org/10.1016/j.cosrev.2022.100529
[14] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. 2014. Checking app behavior against app descriptions. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 1025–1035. https://doi.org/10.1145/2568225.2568276
[15] Răzvan GOSA, Albert ENDRE-LASZLO, Vlad Sebastian CREȚU, Marius TIVADAR, and Silviu STAHIE. 2023. Tens of Thousands of Compromised Android Apps Found by Bitdefender Anomaly Detection Technology. https://www.bitdefender.com/blog/labs/tens-of-thousands-of-compromised-android-apps-found-by-bitdefender-anomaly-detection-technology/.
[16] Yangyu Hu, Haoyu Wang, Tiantong Ji, Xusheng Xiao, Xiapu Luo, Peng Gao, and Yao Guo. 2021. CHAMP: Characterizing Undesired App Behaviors from User Comments based on Market Policies. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 933–945. https://doi.org/10.1109/ICSE43902.2021.00089
[17] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. 2014. AsDroid: detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 1036–1046. https://doi.org/10.1145/2568225.2568301
[18] Deguang Kong, Lei Cen, and Hongxia Jin. 2015. AUTOREB: Automatically Understanding the Review-to-Behavior Fidelity in Android Applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM, 530–541. https://doi.org/10.1145/2810103.2813689
[19] Yeonjoon Lee, Xueqiang Wang, Kwangwuk Lee, Xiaojing Liao, XiaoFeng Wang, Tongxin Li, and Xianghang Mi. 2019. Understanding iOS-based Crowdturfing Through Hidden UI Analysis. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and

Patrick Traynor (Eds.). USENIX Association, 765–781. https://www.usenix.org/conference/usenixsecurity19/presentation/lee

[20] Yeonjoon Lee, Xueqiang Wang, Xiaojing Liao, and XiaoFeng Wang. 2021. Understanding Illicit UI in iOS Apps Through Hidden UI Analysis. *IEEE Trans. Dependable Secur. Comput.* 18, 5 (2021), 2390–2402. https://doi.org/10.1109/TDSC.2019.2950253

[21] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick D. McDaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 280–291. https://doi.org/10.1109/ICSE.2015.48

[22] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: a lightweight UI-guided test input generator for Android. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE Computer Society, 23–26. https://doi.org/10.1109/ICSE-C.2017.8

[23] Mr. Lin. 2023. Movie Watching - Mr. Lin's Apple Toolkit. https://lin.mrlin.vip/index.php?m=home&c=Lists&a=index&tid=76.

[24] Ben Lovejoy. 2018. China accuses Apple of failing to counter pornography, gambling and counterfeit goods. https://9to5mac.com/2018/07/31/apple-china-filtering-banned-content/.

[25] Prashanthi Mallojula, Javaria Ahmad, Fengjun Li, and Bo Luo. 2021. You Are (not) Who Your Peers Are: Identification of Potentially Excessive Permission Requests in Android Apps. In *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 114–121.

[26] Elizabeth Montalbano. 2023. 60K+ Android Apps Have Delivered Adware Undetected for Months. https://www.darkreading.com/application-security/60k-android-apps-adware-undetected-months.

[27] Yuhong Nan, Xueqiang Wang, Luyi Xing, Xiaojing Liao, Ruoyu Wu, Jianliang Wu, Yifan Zhang, and XiaoFeng Wang. 2023. Are You Spying on Me?{Large-Scale} Analysis on {IoT} Data Exposure through Companion Apps. In *32nd USENIX Security Symposium (USENIX Security 23)*. 6665–6682.

[28] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, Samuel T. King (Ed.). USENIX Association, 527–542. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/pandita

[29] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. 2014. AutoCog: Measuring the Description-to-permission Fidelity in Android Applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM, 1354–1365. https://doi.org/10.1145/2660267.2660287

[30] Kimberly Tam, Salahuddin J. Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society. https://www.ndss-symposium.org/ndss2015/copperdroid-automatic-reconstruction-android-malware-behaviors

[31] Haoyu Wang, Jason I. Hong, and Yao Guo. 2015. Using text mining to infer the purpose of permission use in mobile apps. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp 2015, Osaka, Japan, September 7-11, 2015*, Kenji Mase, Marc Langheinrich, Daniel Gatica-Perez, Hans Gellersen, Tanzeem Choudhury, and Koji Yatani (Eds.). ACM, 1107–1118. https://doi.org/10.1145/2750858.2805833

[32] Takuya Watanabe, Mitsuaki Akiyama, Tetsuya Sakai, Hironori Washizaki, and Tatsuya Mori. 2018. Understanding the Inconsistency between Behaviors and Descriptions of Mobile Apps. *IEICE Trans. Inf. Syst.* 101-D, 11 (2018), 2584–2599. https://doi.org/10.1587/transinf.2017ICP0006

[33] Huayao Wu, Wenjun Deng, Xintao Niu, and Changhai Nie. 2021. Identifying Key Features from App User Reviews. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 922–932. https://doi.org/10.1109/ICSE43902.2021.00088

[34] Shengqu Xi, Shao Yang, Xusheng Xiao, Yuan Yao, Yayuan Xiong, Fengyuan Xu, Haoyu Wang, Peng Gao, Zhuotao Liu, Feng Xu, and Jian Lu. 2019. DeepIntent: Deep Icon-Behavior Learning for Detecting Intention-Behavior Discrepancy in Mobile Apps. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 2421–2436. https://doi.org/10.1145/3319535.3363193

[35] Le Yu, Xiapu Luo, Chenxiong Qian, Shuai Wang, and Hareton K. N. Leung. 2018. Enhancing the Description-to-Behavior Fidelity in Android Apps with Privacy Policy. *IEEE Trans. Software Eng.* 44, 9 (2018), 834–854. https://doi.org/10.1109/TSE.2017.2730198

[36] Chengpeng Zhang, Haoyu Wang, Ran Wang, Yao Guo, and Guoai Xu. 2018. Re-checking App Behavior against App Description in the Context of Third-party Libraries. In *The 30th International Conference on Software Engineering and Knowledge Engineering, Hotel Pullman, Redwood City, California, USA, July 1-3, 2018*, Óscar Mortágua Pereira (Ed.). KSI Research Inc. and Knowledge Systems Institute Graduate School, 665–664. https://doi.org/10.18293/SEKE2018-180

[37] Yichi Zhang, Guoxing Chen, Yan Meng, and Haojin Zhu. 2023. Understanding and Identifying Cross-platform UI Framework based Potentially Unwanted Apps. In *IEEE Global Communications Conference, GLOBECOM 2023, Kuala Lumpur, Malaysia, December 4-8, 2023*. IEEE. https://yan4meng.github.io/files/paper_globecom_23_xpua.pdf to be published..

[38] Qingchuan Zhao, Chaoshun Zuo, Brendan Dolan-Gavitt, Giancarlo Pellegrino, and Zhiqiang Lin. 2020. Automatic Uncovering of Hidden Behaviors From Input Validation in Mobile Apps. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1106–1120. https://doi.org/10.1109/SP40000.2020.00072

[39] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. 2012. SmartDroid: an automatic system for revealing UI-based trigger conditions in android applications. In *SPSM'12, Proceedings of the Workshop on Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2012, October 19, 2012, Raleigh, NC, USA*, Ting Yu, William Enck, and Xuxian Jiang (Eds.). ACM, 93–104. https://doi.org/10.1145/2381934.2381950