Lie to Me: Abusing the Mobile Content Sharing Service for Fun and Profit

Guosheng Xu
Siyi Li
Beijing University of Posts and Telecommunications
Beijing, China

Hao Zhou
The Hong Kong Polytechnic University
Hong Kong, China

Shucen Liu
Beijing University of Posts and Telecommunications
Beijing, China

Yutian Tang
ShanghaiTech University
Shanghai, China

Li Li
Monash University
Monash University, Australia

Xiapu Luo
The Hong Kong Polytechnic University
Hong Kong, China

Xusheng Xiao
Case Western Reserve University
Cleveland, United States

Guoai Xu
Beijing University of Posts and Telecommunications
Beijing, China

Haoyu Wang
Huazhong University of Science and Technology
Wuhan, China

ABSTRACT

Online content sharing is a widely used feature in Android apps. In this paper, we observe a new Fake-Share attack that adversaries can abuse existing content sharing services to manipulate the displayed source of shared content to bypass the content review of targeted Online Social Apps (OSAs) and induce users to click on the shared fraudulent content. We show that seven popular content-sharing services (including WeChat, AliPay, and KakaoTalk) are vulnerable to such an attack. To detect this kind of attack and explore whether adversaries have leveraged it in the wild, we propose DeFash, a multi-granularity detection tool including static analysis and dynamic verification. The extensive in-the-lab and in-the-wild experiments demonstrate that DeFash is effective in detecting such attacks. We have identified 51 real-world apps involved in Fake-Share attacks. We have further harvested over 24K Sharing Identification Information (SIIs) that can be abused by attackers. It is hence urgent for our community to take actions to detect and mitigate this kind of attack.

CCS CONCEPTS

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

KEYWORDS

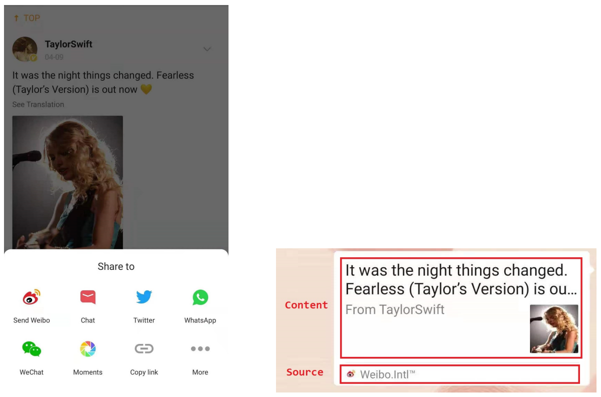
OSAs, Content Sharing, Fake-Share Attack, Data-flow Analysis, Secret Leakage

ACM Reference Format:
Guosheng Xu, Siyi Li, Hao Zhou, Shucen Liu, Yutian Tang, Li Li, Xiapu Luo, Xusheng Xiao, Guoai Xu, and Haoyu Wang. 2022. Lie to Me: Abusing the Mobile Content Sharing Service for Fun and Profit. In *Proceedings of the ACM Web Conference 2022 (WWW '22), April 25–29, 2022, Virtual Event, Lyon, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3485447.3512151

1 INTRODUCTION

Online content sharing has become a commonly used feature in social platforms [1, 3, 39], where people would share the content of interest (usually a website link) with friends, and their friends can get this content by clicking on the link. To facilitate content sharing, many websites or mobile apps provide the one-click content sharing function so that users can click the share button to share the content of interest without describing the content using their own language or manually copying the content.

Content Sharing in Android Apps. Figure 1(a) illustrates an example. After the user clicks the share button in an app, the corresponding interface will pop up for users to select the Online Social Apps (**OSAs** for short) that they want to share, such as WeChat [14], Twitter [10] and WhatsApp [9]. In the past, Many developers implemented one-click content sharing in their apps by directly using Android's Intent mechanism to open the target OSA. In this way, the content that the user wants to share is embedded in the Intent as text. After jumping to the target OSA, the content will be automatically filled in its text input box, and users can click the send button to send the content. While it simplifies the work for developers to implement the content sharing function, it provides little support for customizing the descriptions of the content since it is not much different from manually copying the content and sending it. Therefore, most OSAs provide the Software Development Kit for Sharing (hereafter referred to as **Share-SDK**) for third-party app developers to use. After obtaining the Sharing Identification Information (**SII** for short) provided by the third-party app and the content to be shared, it will generate a content description card as


TOP
TaylorSwift
It was the night things changed. Fearless (Taylor's Version) is out now
See Translation
Share to
Send Weibo
Chat
Twitter
WhatsApp
WeChat
Moments
Copy link
More
Content
It was the night things changed. Fearless (Taylor's Version) is ou...
From TaylorSwift
Source
Weibo.Intl™

(a) sharing button (b) Description Card

Figure 1: The general process of content sharing.

shown in Figure 1(b). This type of description card usually displays the *source app* of the shared content at the bottom. For example, if it is the content shared from `Weibo` to `WeChat`, the source will be displayed as `Weibo`. This method eliminates the need for users to click the send button manually and can allow users to get more information about what is shared.

Fake-Share Attack. While this approach can provide customized third-party ap description cards, upon closer investigation, we found that the source app presented in the card may not be reliable. *Most Share-SDKs only verify that the app identifier (*`APPID`*) sent by the app is correct but do not verify whether the app is sending its own app identifier or stealing another app's identifier.* Due to the verification being implemented within the Share-SDKs, adversaries can easily exploit this type of vulnerabilities. We devise a new type of attack to bypass the target OSA review and display fake sources for the shared content, and refer to this type of attack as the **Fake-Share Attack**. Our preliminary study suggests that numerous popular Share-SDKs (including Wechat [14], QQ [11], DingTalk [6], Yixin [16], Alipay [5], KakaoTalk [12] and Weibo [15]) are vulnerable to this type of attack (**see Section 3**). Users are more likely to trust messages from sources they know than messages from unknown sources, so adversaries can easily disguise the source of the content shared from their malicious apps as the content shared from a reputable app such as `TikTok` or `Weibo`. Similar to phishing websites and emails, fake-share attacks can be used to achieve serious security consequences such as delivering drive-by-download malware, malicious advertising, financial scams, etc.

To detect Fake-Share Attack and explore whether adversaries have exploited it in the wild, we present to the community a prototype tool called *DeFash*, which can detect *Fake-Share Apps* that exploit the Share-SDKs to perform Fake-Share Attacks on the apps (**See Section 4**). DeFash can strike an excellent balance between performance and accuracy in identifying Fake-Share attacks and consists of three main components: static data stream analysis, SII match detection, and dynamic SII capture. Experimental results, either through in-the-lab experiments with carefully crafted benchmark apps or in-the-wild experiments containing real-world Android apps, show that our approach effectively detects Fake-Share apps (**See Section 5**). Specifically, in our experiments, we have identified 51 apps that target WeChat for fake content sharing. All of them are claimed to be "money-making apps" that induce users to share content with spam ads and censored images (e.g., porn and gambling images). Besides, we have harvested over 24,000 SIIs from Android apps, including many popular apps, which adversaries can abuse to perform Fake-Share attacks.

In summary, this paper makes the following major contributions:

- We identify and demonstrate the feasibility of *Fake-Share Attack*, a novel type of attack to tamper with the shared source display in the shared content description card. To the best of our knowledge, this paper is the first work that reveals this kind of attack. We show that numerous popular Share-SDKs are vulnerable to Fake-Share Attack.
- We propose *DeFash*, a multi-granularity detection tool that can identify Fake-Share Attack. We craft a benchmark and evaluate the performance of DeFash on it. Experiment results suggest that DeFash can achieve 100% accuracy. Then we used DeFash to scan 1,785 apps that contain sharing functions and found 51 real-world fraudulent apps that perform Fake-Share Attacks on the shared contents.
- We analyzed over 73,014 Android apps from the Huawei app market and collected more than 24,000 SIIs that can be leveraged to implement Fake-Share Attack, which means that adversaries can easily abuse the identities of many popular apps. It reveals the severity of this kind of attack.

2 CONTENT SHARING IN MOBILE APPS

2.1 Content Sharing without Share-SDK

Content sharing is a process of communication between two apps in the Android system. A simple way for content sharing is to construct an Intent to open target OSA directly. This method directly uses Android's cross-app launch function `StartActivity`, which is suitable for all apps. However, it only implements a simple jump and text auto-fill function but does not display the source and other descriptions of the shared content.

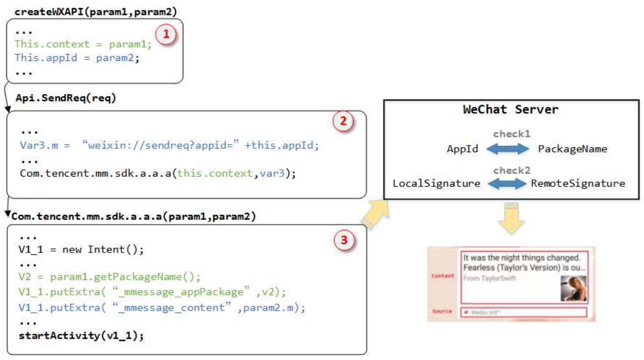
2.2 Content Sharing with Share-SDK

To achieve a better user experience, many OSAs will provide a shared content display in the form of a *description card*, as shown in Figure 1(b). The card body will display the title, thumbnail, and other information of the shared content, and the *source app* will be displayed at the bottom of the card. This form allows users to obtain more information about the shared content to decide whether to click the link. Moreover, many third-party apps (especially news apps) also hope that users can share the contents from their apps to OSAs using the description card because it can better promote their apps through the shared source display function. In order to implement sharing function in the form of a description card, developers need to access the Share-SDK provided by the OSA and share content according to the process given in the official document. Take WeChat SDK as an example, as illustrated in Code Snippet 1, the app first needs to register itself to Share-SDK (line 1-line 2) and then collects specific information about the content to be shared to construct a sharing package (line 4-line 11) and finally invokes the `sendReq` function in Share-SDK to share. If the `sendReq` function is executed correctly, the system will automatically jump to the WeChat contact (friend) selection interface and send the shared content in the form of a description card. Otherwise, it will stay at the original interface, or a pop-up window will remind the user of the sharing error.

```
shareapi = WXAPIFactory.createWXAPI(this, APP_ID);
shareapi.registerApp(APP_ID);
......
WXWebpageObject page = new WXWebpageObject();
page.webpageUrl = shareUrl;
WXMediaMessage message = new WXMediaMessage(page);
message.title = shareTitle;
message.description = shareContent;
message.thumbData = null;
SendMessageToWX.Req localReq = new SendMessageToWX.Req();
localReq.message = message;
......
shareapi.sendReq(localReq);
```

Code Snippet 1. Content sharing with Share-SDK

Other OSA platforms adopt a similar sharing process. For example, KaKaoTalk is a famous Korean social networking platform. Developers first need to submit their apps to the KaKaoTalk developer platform for reviewing and then get an app unique identifier APPID. If the developer wants to access the content sharing function, they need to store this unique identifier in the `.xml` file in a given format and wait for Share-SDK to take it. When the user clicks the share button, similar to WeChat, the app needs to create a new object `FeedTemplate` and package the content related informations to be shared into it, and then pass this `FeedTemplate` to the `LinkClient.instance.defaultTemplate` method for content sharing. Our extensive investigation suggests that most of the Share-SDKs work similarly.



```
createWXAPI(param1,param2)
...
This.context = param1;
This.appId = param2;
...

Api.SendReq(req)
...
Var3.m =  "weixin://sendreq?appid=" +this.appId;
...
Com.tencent.mm.sdk.a.a.a(this.context,var3);

Com.tencent.mm.sdk.a.a.a(param1,param2)
...
V1_1 = new Intent();
...
V2 = param1.getPackageName();
V1_1.putExtra( "_mmessage_appPackage" ,v2);
V1_1.putExtra( "_mmessage_content" ,param2.m);
...
startActivity(v1_1);
```

Figure 2: Content sharing with WeChat Share-SDK

3 ABUSING THE SHARE-SDK

3.1 Potential Risks in Share-SDKs

As shown in Figure 1(b), description cards usually automatically display the *source app* of the shared content, which has become the primary basis for users to determine whether the shared links are reliable or not because content from reliable apps is usually harmless. However, *is the source of the shared content shown on the description card always credible?* Taking WeChat as an example, we first analyze the operation of Share-SDK during the entire content sharing process and display it in Figure 2.

① When the developer calls `createWXAPI` and creates a `shareapi` object with `APPID` and current Context as parameters, the member variables `context` and `APPID` of `shareapi` will be initialized to these two parameters.

② In `sendreq`, the sharing environment would be checked first, e.g., whether WeChat is installed on this device. Then, it uses the member variable `APPID` to get a Uniform Resource Identifier `Var3.m`.

③ In `com.tencent.mm.sdk.a.a.a`[1], it creates an Intent object and uses the package name of the current app as the value of the "_mmessage_appPackage" key, and the Uri containing the APPID as the value of the "_mmessage_content" key. After filling in other necessary data, it uses this Intent as a parameter of `startActivity` to jump to WeChat.

We read the developer documentation of WeChat [14] and found that after WeChat received and parsed the particular format Intent, it performed the following two-step authentication based on the `APPID` and `PackageName` in the Intent. It is necessary to verify that the app has been submitted on Wechat open platform and has the qualification to share the content to WeChat.

- **Check1:** WeChat server will first query the `APPID` in the open platform, that is, to confirm that the `APPID` is legal and the corresponding app has passed the review of the open platform. Then it will query the package name of the app corresponding to this `APPID`, and check whether the package name matches the `PackageName` in the Intent.
- **Check2:** The server will query the app's signature from the app information submitted to the open platform and then compare it with the one obtained from the local device to ensure that these two signatures are consistent.

When the two-step verification has been passed, WeChat will use the `AppName` corresponding to the `APPID` on the open platform as the source of content sharing and display it in the description card. We found that even though these two verification schemes took place on the WeChat server, its verification entirely relies on the `PackageName` and `APPID` passed initially by the current app through the Intent. Therefore, **if we send a fake but matching `PackageName` and `APPID` to the WeChat server, The WeChat server cannot verify the true identity of the current app.**

3.2 Fake-Share Attack

Based on our previous analysis, we found that the verification of WeChat relies on two key parameters (i.e., `PackageName` and `APPID`), which we call them **Sharing Identification Information (SII)**. As the SII is constructed locally, we come up with a possible attack scenario: *if we modify the SII of the fraudulent app itself to the SII of other popular apps, and then we can modify the source of content sharing displayed in the description card, which is the **Fake-Share Attack** proposed in this paper.* Through this attack, *fake sharing source* can be shown on the description card of target OSA to cheat the message receiver. Therefore, a malicious app developer may use this attack method to modify the source of the link shared from the malicious app as another trusted app, which will trick OSA users into clicking the Shared link (maybe a malicious link).

[1]This is an obfuscated method name as the sample app (including its embedded WeChat Share-SDK) has been obfuscated by its developers.

When this type of apps share content to OSA, we refer to the source displayed in the description card as the ***victim app***. We will show how this attack is performed in detail (Section 3.3) and demonstrate the feasibility of performing this kind of attack (Section 3.4).

3.3 Performing Fake-Share Attack

We next propose two methods to perform the Fake-Share Attack.

```
api = WXAPIFactory.createWXAPI(new ContextWrapper(
      MainActivity.this){
      @Override
      public String getPackageName(){
          return "com.tencent.mobileqq";
      }
},"wxf0a80d0ac2e82aa7");
```

Code Snippet 2. Overriding the getPackageName

3.3.1 Overriding the GetPackageName Method. As shown in Figure 2-③, when constructing the Intent, it first uses the method `Context.getPackageName` to get the package name of the application. Therefore, one way to implement the Fake-Share attack is to override the `Context.GetPackageName` method so that the Share-SDK gets a fake `PackageName` (the package name of the victim application) from the Context instead of the current package name. Thus, we first create a proxy Class of type `ContextWrapper` for the original Context Class, as shown in line 1 of Code Snippet 2. Then we override the `getPackageName` method in the proxy Class to return the package name of the victim app, e.g., "com.tencent.mobileqq" (see line 4 in Code Snippet 2). Finally, we register this proxy Class and the APPID of the victim app into the Share-SDK.

After initializing the Share-SDK according to the steps mentioned above, the malicious app can construct the content sharing packet and call the sharing interface provided by the Share-SDK. Till now, Fake-Share Attack can be successfully implemented. This type of method presupposes that the Share-SDK uses the method `Context.GetPackgeName` to get the package name of the current application and register it. Since this method does not require modification of the code in the Share-SDK, it can easily bypass the built-in Share-SDK defense methods (for example, some SDKs use integrity checking to check if the SDK has been tampered with).

```
1 Intent localIntent = new Intent();
2 localIntent.setClassName("com.tencent.mm","com.tencent.mm
      .plugin.base.stub.WXEntryActivity");
3 ......
4 localIntent.putExtra("_mmessage_appPackage",
      fake_packagename);
5 localIntent.putExtra("_mmessage_content", "weixin://
      sendreq?appid=" + fake_appid);
6 localActivity.startActivity(localIntent);
```

Code Snippet 3. Construct a malicious Intent explicitely

3.3.2 Construct Malicious Intent Explicitly. The essence of sharing via Share-SDK is to build an `Intent` in the current app in a predefined format that can be parsed by the target OSA. After the target OSA parses and implements the necessary authentication, the information is displayed to the user in a description card. Based on this observation, we devise a more general attack method. First, we analyze the `Intent` passed to the target OSA via the Share-SDK and parse the format of the Intent, and then manually construct the Intent-based on this specific format. Our research shows that we can easily obtain their SIIs from a number of popular apps and use these SIIs to perform Fake-Share Attack (see Section 5.3). Code Snippet 3 shows the process of constructing malicious Intent. As shown in lines 4-5, we need to rewrite the values of the keys "_mmessage_appPackage" and "_mmessage_content" in the Intent to the PackageName and APPID of the victim's app, and then use this Intent as the parameter of `startActivity` to jump to the WeChat.

Table 1: The Identified OSAs with Vulnerable Share-SDKs.

	Overriding GetPackageName	Constructing Fake Intent
QQ	√	√
WeChat	√	√
DingTalk	√	√
Yixin	√	√
Alipay	√	√
KakaoTalk		√
Weibo		√

3.3.3 Attack implementation. We have created a lightweight demo app to show the effectiveness of the proposed Fake-Share Attack. First, we analyze the execution process of the target Share-SDK and then parse the Intent sent to the target OSA through `StartActivity`, and finally modify the Intent indirectly (by overriding the method `getPackageName`) or directly (by constructing a malicious `Intent` explicitly). In this way, the target OSA will parse the fake content sharing source from the malicious `Intent` and display it. The proposed Fake-Share Attack can successfully bypass the open platform app review, as the malicious app does not even need to have its own `APPID` to implement this attack.

3.4 Feasibility of the Attack

We next show whether popular Share-SDKs are vulnerable to this kind of attack. We found that Fake-Share Attack can be implemented on most OSAs that provide the functionality of sharing content in the form of description card and showing the source of sharing because most of these Share-SDKs need to construct SII in the local app and further send it to the target OSA. The specific investigation step is as follows. Firstly, we manually collected OSAs that include the content sharing functions and identified 35 commonly used OSAs. Secondly, we read the official documents of these 35 OSAs and flag OSAs with Share-SDK released. Then we manually analyze the operating mechanism of these Share-SDKs and implement Fake-Share Attack on these OSAs using the two kinds as mentioned above of attack methods. Finally, we apply the aforementioned Fake-Share Attacks to those OSAs and experimentally confirm that, as highlighted in Table 1, seven OSAs could be successfully exploited. All of them can be abused by constructing the `malicious Intent`, and five of them could be attacked by simply overriding the `GetPackageName`. This kind of vulnerability opens new attack surfaces for scammers and adversaries.

4 DEFASH: A HYBRID APPROACH FOR IDENTIFYING FAKE-SHARE ATTACK

We proposed **DeFash** (<u>De</u>tection of <u>Fa</u>ke-<u>Sh</u>are Attack), a hybrid approach that aims accurate detection of Fake-Share Attack. To achieve the balance between accuracy and performance, Defash is designed as a two-phase detection tool, which incorporates the coarse-grained and fine-grained detection phases. The coarse-grained detection uses a hybrid approach that combines static data-flow analysis and dynamic SII matching for identifying all suspicious applications that may conduct fake sharing attacks. In the fine-grained detection phase, DeFash will automatically run the suspicious app to trigger the content sharing feature and capture the SII used for sharing to confirm whether the suspicious app has implemented Fake-Share Attack.

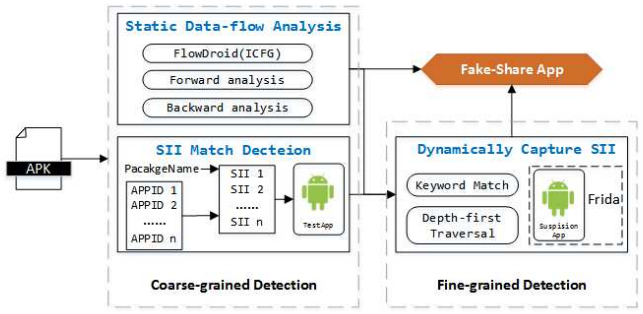


Figure 3: The working process of DeFash.

Figure 3 illustrates the detailed working process of DeFash, which is mainly made up of three modules: **1) Static Data-flow Analysis**, **2) SII Matching Detection** and **3) Dynamic SII Capturing**. We will depict each module in the following.

4.1 Static Data-flow Analysis

If a malicious app pretends to be a popular app for content sharing via Fake-Share Attack, it will inevitably use the SII of the famous (victim) app to directly or indirectly construct a malicious Intent (see Section 3.3). To perform data-flow analysis on the entire app, we use Soot [13], a widely used Java code analysis tool. It compiles Java code into an intermediate representation (i.e., Jimple) and builds an intra-procedure control flow graph based on it. To further enable inter-procedure analysis, we take advantage of FlowDroid [17], a widely used static analysis framework to generate a precise Inter-process Control Flow Graph (ICFG). Therefore, we obtain the function call relationships in the program based on the ICFG generated by FlowDroid and perform data flow analysis based on Soot. Specifically, our static data-flow analysis includes backward data-flow analysis (see section 4.1.1) and forward-backward data-flow analysis (see section 4.1.2).

4.1.1 Backward data-flow analysis: from the Intent back to the data source. By inheriting the `BackwardFlowAnalysis` Class in Soot, we can automatically maintain the InSet and OutSet of each statement in a given method backward after designing the state transfer function. Taking WeChat ShareSDK as an example, we first scan all methods in ICFG and find all statements related to the construction of SII during content sharing through the keywords "_mmessage_appPackage" and "_mmessage_content". We set the *use* domain variables of these statements to be live variables and traversed the statements backward for live variable analysis.

After the backward data-flow analysis within the program, we can find the statement where the live variable died and consider it

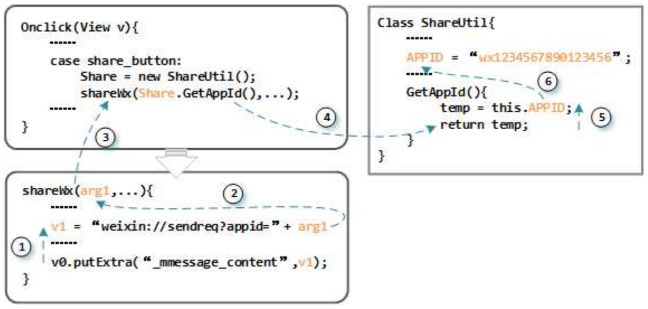

```
Onclick(View v){
    ------
    case share_button:
        Share = new ShareUtil();
        shareWx(Share.GetAppId(),...);
    ------
}
```
```
Class ShareUtil{
    ------
    APPID = "wx1234567890123456";
    ------
    GetAppId(){
        temp = this.APPID;
        return temp;
    }
}
```
```
shareWx(arg1,...){
    ------
    v1 = "weixin://sendreq?appid=" + arg1
    ------
    v0.putExtra( "_mmessage_content" ,v1);
}
```

Figure 4: An example of backward data flow analysis.

initialized in this statement and then process the inter-procedural data flow according to the following rules so that we can perform the inter-procedural data-flow analysis:

- If it is a parameter assignment statement, we will identify all callers of the current method in ICFG and set this parameter as a live variable to continue to perform backward data-flow analysis in callers.
- If it is an invoke statement with no parameters, we will search for all return statements in the callee and set the variables in these statements as live variables for backward data-flow analysis in the callee.
- If it is a member variable assignment statement, we will search for the initial value of this variable in the initialization methods `<init>` and `<clinit>` of the class to which this field variable belongs.

Figure 4 shows an example. We start with the method `putExtra`, perform the inter-procedure backward data-flow analysis in `shareWX`, and find the variable to be initialized with the parameter *arg1*. Then, we find the caller `Onclick`, and continue to trace to the `GetAppId` method in step 4. After the member variable assignment statement is analyzed in steps 5 and 6, we get the value of APPID in the initialization field of `ShareUtil` Class.

4.1.2 Forward-backward data-flow analysis: identifying the matching pair of APPID and PackageName. Since `APPIDs` used in ShareSDKs generally follow a pre-defined format, we can find these `APPIDs` from the bytecode according to this format and propose a forward-backward data-flow analysis method. First, we scan the bytecode to find the statements contains APPID in a pre-defined format[2]. We use these statements for forward data-flow analysis to find all reachable `StartActivity`, and then use each `StartActivity` as a starting point for backward data-flow analysis as described in Section 4.1.1 to find the matching PackageName. If both APPID and PackageName are stored in the bytecode, we can find the paired APPID and PackageName in this way.

We process return statements and invoke statements reachable by sensitive variables (APPID in this case) for inter-procedural data-flow analysis according to the following rules:

- If it is an invoke statement, we will mark its sensitive parameters based on the variables in the InSet and use the sensitive parameters as the starting point to perform forward data-flow analysis on the callee.

[2]This format is "wx + 16 lowercase and numeric characters" for Wechat ShareSDK.

- If it is a return statement, we will find all the callers of the current method and the calling statement in ICFG and use the *defs* of the calling statement as the starting point for forward data-flow analysis.

After static data-flow analysis, we could flag whether a app is a benign app, a confirmed Fake-Share app, or a suspicious app. If the `PackageName` used for content sharing is not obtained through the original `GetPackageName` during the backward data-flow analysis, we will mark this app as suspicious. Further, if we can find the data flow that involves malicious SII (i,e, SII that does not belong to the current app) through the forward-backward analysis, we would flag this app as a Fake-Share app. Otherwise, we consider this app to be a benign app. Suspicious apps will be further analyzed using the fine-grained analysis module of DeFash.

4.2 SII Matching detection

SII usually consists of verification information of the app itself, such as the APPID that can uniquely identify the app. It is reasonable to assume that if an app's code or resource files contain SII from other apps, the current app may use that for malicious purposes (e.g., Fake-Share Attack). We will further perform SII Matching detection for these apps, a hybrid method to automatically flag these suspicious apps. Specifically, this detection can be divided into the following steps. Firstly, we scan each app's code and resource files based on regular expression matching to obtain all the APPIDs contained in the app. Secondly, we combine each APPID contained in the app and the app's package name to form a pair of SII. Note that if more than one APPID (e.g., *n* APPIDs) is found in the app, we will compose *n* pairs of SIIs and mark them as temporary SIIs. Finally, we develop a test app, which can implement content sharing using temporary SII and record whether the sharing is successful. Due to the authentication of OSAs, only the matching PackageName and APPID can pass the authentication. We record the results of each sharing to determine whether a temporary SII is correct. If an app contains incorrect SII, we will flag it as a suspicious app. The identified suspicious apps will be sent to the following fine-grained analysis for confirmation.

4.3 Dynamic SII Capturing

The purpose of fine-grained detection is to determine whether the suspicious app implements Fake-Share Attack, which can remove the possible false positives flagged in the previous steps. We design an automation testing scheme to automatically explore the sharing buttons in the app for triggering content sharing and then monitor whether the SII of the victim application is used in a fake sharing attack. This process consists of two vital techniques: 1) Automate Android app testing and 2) Dynamic instrumentation.

Automate Android app testing. To automatically explore and click on share buttons, we use a depth-first search strategy. We will keep exploring the clickable buttons on the activity until the depth reaches a specified maximum, or we cannot continue exploring and then return to the previous activity. We use DroidBot [7] to capture the activity view and generate simulated click or slide actions. We propose a keyword-based priority queue traversal method to traverse the sharing operation activity effectively and quickly to trigger. In detail, we have specified some particular keywords for content sharing like "share" and "friend". Then we set the traversal priority of the views containing these keywords to the highest.

Dynamic instrumentation We use Frida [8] to monitor the entire app runtime. Once the share operation is triggered, it captures the SII used for sharing and checks if it matches the current shared app. If it does not match, it indicates that the current app uses the SII of another app (victim app) for content sharing, i.e., the current app has implemented Fake-Share Attack.

5 EVALUATION

Our evaluation is driven by the following research questions (RQs):

RQ1 *How does DeFash perform in the lab?* No previous work is aware of the Fake-Share attack, and no available benchmark can be used to evaluate the effectiveness of DeFash. Thus, we have to craft our own benchmark for evaluation.

RQ2 *Whether the adversaries have leveraged the Fake-Share attack and how does DeFash perform in the wild?* We want to explore real-world Fake-Share attacks and characterize them.

RQ3 *How feasible is it to perform Fake-Share attacks?* We want to investigate it from the adversaries' perspective to see whether we can harvest the SIIs of popular apps and leverage them to perform Fake-Share attacks.

5.1 RQ1: How does DeFash perform in the lab?

5.1.1 Craft the benchmark. In the absence of established benchmarks in this research direction, we propose to create a benchmark of Fake-Share attacks based on app repackaging. Real-world apps have implemented different content sharing methods via connecting with the Share-SDKs. Thus we first collect 9 benign content sharing apps from the market and repackage them to create Fake-Share apps. The WeChat open platform has reviewed these nine apps we selected, and we confirm that they use their own SII for content sharing. Specifically, our app repackaging process is as follows: 1) First, we use Apktool [2] to extract the smali code from the `.apk` file, and then manually analyze the code related to content sharing and the storage location of the `APPID` used. 2) Second, we change the execution mode of the app by modifying the smali code. Specifically, we modify the original APPID to the APPID of the victim app. For `PackageName`, we override the `GetPackageName` method in the incoming Context or directly Modify the parameters used by `Putextra` when constructing the Intent according to the two methods proposed in Section 3.3. For the original app, these two methods will form corresponding copies respectively. 3) Finally, we repackage the app and get the Fake-Share App. Since we used two methods to change the `PackageName` in the previous step, a benign app can be repackaged to generate two different Fake-Share apps. Thus, we can build a benchmark with a total of 27 apps, including 18 Fake-Share apps and nine benign apps.

5.1.2 Result. We next report the accuracy and runtime performance of DeFash on the created benchmark.

Accuracy of DeFash. Using the static data-flow analysis and SII Matching detection, DeFash flags 22 suspicious apps in the coarse-grained phase. Then, through fine-grained dynamic analysis, DeFash can detect all 18 Fake-Share apps with 100% accuracy. For the four benign apps that were falsely reported in the coarse-grained stage, we further manually analyze them to pinpoint the reasons.

We observe that FlowDroid cannot construct the ICFGs correctly, leading to false positives. Nevertheless, our following fine-grained dynamic analysis can remove these false positives. It suggests the high accuracy of DeFash.

Time consumption. We further evaluate the runtime performance of DeFash. Note that, in the dynamic analysis stage, we limit the testing time of DeFash to 10 minutes. On average, it takes 292 seconds for DeFash to analyze each app. The most time-consuming part is the dynamic app automation (211 seconds on average), as we need to continuously generate click operations to explore the sharing button on the UI in a depth-first search. Nevertheless, the static data flow analysis and SII matching detection is quick (67 seconds and 9 seconds, respectively), which could help filter most irrelevant apps and improve the overall performance. It meets our goal to achieve the balance between accuracy and scalability.

Answer to RQ1: *DeFash can detect Fake-Share Attack with 100% accuracy in the benchmark we created. In terms of detection performance, the average time consumed by coarse-grained detection is only 76 seconds, while the time consumed by fine-grained detection is 211 seconds on average.*

5.2 RQ2: Fake-Share attack in the wild

5.2.1 Method. Next, we want to investigate whether Fake-Share Attack is presented in the wild. Thus, we first use keyword matching to identify apps with content sharing functions from the app market. We set a number of keywords (e.g., "news" and "share") and crawled apps that contain these keywords in app name or description. We flag these apps as the candidate's apps to have content sharing functions. Second, We take advantage of Libradar++ [30, 38], a widely used Android third-party library detection tool, to screen out all apps that contain Share-SDKs. In addition, beyond the Share-SDK, developers can also directly construct an Intent for content sharing (see Section 3.3), so we further analyze the decompiled code and resource files to identify apps that contain Sharing APPID. Finally, we use DeFash to detect whether these apps have implemented Fake-Share Attack.

Note that our dataset is from the Huawei Android app market, and most of the content-sharing apps have embedded Wechat Share-SDK. Thus, our exploration in this paper is mainly focused on WeChat Share-SDK. Nevertheless, as we detailed explained and demonstrated in Section 3.4, this type of attack can be generalized to other OSA Share-SDKs, and our approach can be adopted directly.

5.2.2 Result. In the first step, we crawled a total of 1,785 apps from the Huawei App market. In the second step, taking advantage of LibRadar++, we can identify 1,379 apps containing WeChat third-party libraries, and we further detect 968 apps containing Wechat APPID, resulting in a total of 1,493 apps. Finally, we use DeFash to analyze these apps. Through coarse-grained detection, we get 342 suspicious apps. After fine-grained detection, we finally got 51 Fake-Share apps, whose fake sharing behaviors can be confirmed.

5.2.3 Analysis. We next analyze these 51 Fake-Share apps in detail.

What kinds of apps are they? We install these apps on smartphones and manually analyze them. We observe that all of them are "money-making apps" that induce users for content sharing. They induce users to share an article in the app to OSAs, and promise that users can earn some money reward based on the amount of link clicks for each article shared. We further investigate these links that we are encouraged to share and observe that *most of the links contained a large number of ads, pornographic images, or videos.* Furthermore, we obtain the two links (before and after sharing) and compare their content. We find that the article before sharing is ad-free, but the app will automatically add extra ads or censored images (e.g., porn and gambling images) to the link after sharing. We consider that the purpose of this behavior is to mislead users with an ad-free article, making them believe that this article is harmless and thus more willing to share it.

Table 2: SIIs of the top-10 victim apps.

APPID	PackageName	count	downloads(M)
wx64f9cf5b17af074d	com.tencent.mtt	24	10830
wx020a535dccd46c11	com.UCMobile	22	6928
wx27a43222a6bf2931	com.baidu.searchbox	21	10670
wx50d801314d9eb858	com.ss.android.article.news	21	11046
wxc2ff198ba4a63f06	com.ijinshan.browser_fast	16	146
wxf0a80d0ac2e82aa7	com.tencent.mobileqq	15	6365
wx2fab8a9063c8c6d0	com.qiyi.video	13	5973
wx8b777d060608ec99	com.duokan.reader	13	43
wx299208e619de7026	com.sina.weibo	12	9707
wx60d9d5c44ca9386e	com.qihoo.browser	10	202

Which apps are the targets (victims)? We have observed 22 kinds of fake SIIs exploited by these 51 fraudulent apps. Table 2 lists the top-10 most frequently occurring APPIDs stored in Fake-Share App, the corresponding package names, and their number of downloads in the Huawei app market (which can indicate their popularity). We found that all these victim apps are popular apps with very high downloads in the market. Even the lowest one "com.duokan.reader" has gained 43 Million downloads, while the highest "com.ss.android.article.news" reaches 11,046 Million downloads. It suggests that many popular apps have been exploited to (indirectly) involved in the attacks, and their reputation would be damaged in this way.

Answer to RQ2: *Adversaries have already performed Fake-Share attacks in the wild. We identify 51 Fake-Share apps in the wild, and all of them are claimed to be "money-making apps" that induce users to share content with spam ads or censored images. All the target (victim) apps are quite popular in the market with a large number of downloads.*

5.3 RQ3: How feasible is it to perform Fake-Share attacks in the wild?

5.3.1 Method. We next investigate the Fake-Share Attack from the perspective of the adversaries – *whether we can easily get the SIIs of popular apps to implement the attack.* To this end, we first extract all available `APPIDs` in each app based on regular expression matching, and then we combine these `APPIDs` with the package name of the app to form a pair of temporary SIIs and try to share. This is similar to the SII Matching Detection phase of DeFash. The difference is that we will record all successfully shared SIIs in the form of `<PacakgeName, APPID>`, as adversaries can abuse them to implement Fake-Share Attack directly.

5.3.2 Result. We crawled a total number of 73,014 popular Android apps from the Huawei app market. Through static analysis, we found that 27,708 apps contain WeChat APPID in their bytecode. Through SII matching, we obtained 24,515 available SIIs, *all of them can be used by us to implement the Fake-Share Attack*. For the top-20 apps in the Huawei app market, we have identified SIIs for 17 of them. It suggests that SII leakage is a general issue that should be raised awareness of our community.

Answer to RQ3: *It is easy to extract SIIs from Android apps. We have harvested SIIs from over 24,000 popular apps, which can be abused by attackers to perform Fake-Share attacks. It shows the weakness of SII protection in existing apps.*

6 DISCUSSION

6.1 Mitigating the Issue

We discuss how to mitigate Fake-Share Attack from the perspectives of content sharing service providers and app developers.

Content sharing service providers. OSA needs to re-examine the security of the released Share-SDK. The fundamental factor leading to Fake-Share Attack is that the verification server completely trusts the SII sent from third-party apps. Therefore, we proposed two feasible solutions to mitigating the Fake-Share Attack. 1) The first solution is that the description card of the shared content uses the information content obtained after parsing the shared link (e.g., the title and the main domain of the page), i.e., it no longer trusts any information transferred from third-party applications except the link to be shared, just as Facebook or Twitter do, whose description cards show not the source application but the main domain of the shared link. 2) The second solution is to design a complete authentication mechanism. When the OSA server receives the pending SII from a third-party app during the content sharing process, an additional authentication process needs to be designed to ensure that the SII belongs to the current third-party app. For example, the message should be signed with a secret key known only by the app to which the SII belongs so that the server can verify it. This, however, might rely on system-level support.

Android app developers. We show that the SIIs of many popular apps can be stolen by simple static analysis, so benign app developers should store and use their own SIIs in a secure manner. Unfortunately, many Share-SDK developer documents do not make such suggestions for developers. An effective way to alleviate SII leakage is to store these SIIs in the app's private server and obtain them through the network when needed. In order to prevent attackers from capturing such information by capturing packets, it is often a good practice to encrypt the traffic.

6.2 Limitations of DeFash

When DeFash performs static data flow analysis, it only analyzes the Java bytecode. For the case of storing SII in native code or other local sources, although we can track the corresponding data flow, we cannot directly get the value of SII. This is one of the challenges we need to address in the future. In addition, our dynamic analysis module in DeFash only uses a simple keyword priority queue algorithm. Although it is effective for a single type of app, exploring the sharing button for more complex apps takes more time. We hope to optimize the exploration strategy to reduce the time consumption of triggering sharing in the future. Furthermore, although we show that seven OSAs are vulnerable to the Fake-Share attack, our exploration only identified real-world attacks targeting WeChat Share-SDK due to the dataset limitation. Nevertheless, we have demonstrated that it is a general issue in the ecosystem, and our proposed method can be applied directly to detect such attacks.

7 RELATED WORK

Secret Leakage Detection CredMiner [47] can programmatically identify and recover developer credentials unsafely embedded in Android apps which is similar to our work for mining SII from other apps. Yang et al. [42] also found that there is a secret key leakage when developers implementations third-party in-app payment. Sinha et al. [34] expose the possibility of malicious users stealing API keys embedded in public code repositories, and Meli's work [31] on detecting privacy leakage of GitHub repositories further suggests that developers' secrets are not well protected. In addition, other work on developer privacy mining includes HARVESTER [33], CredScan [4], StringHound [22], ASTANA [20], etc.

Malicious/Gray Behaviors Analysis The Fake-Share Attack proposed in this paper can be used for spam advertising, spreading malicious links, and other malicious purposes. A large number of papers in our community were focused on Mobile malware detection [24–28, 32, 35–37, 40, 41, 43–46]. Besides, there are currently many studies on these apps that contain fraudulent behaviors [18, 19, 21, 29]. FraudDroid [21] analyses apps dynamically to collect runtime network traffics, which are then leveraged to check against a set of heuristic-based rules for identifying fraudulent ad behaviors. In addition, researchers have also discovered some new apps that use social engineering methods to perform scams, such as the fraudulent dating apps [23].

8 CONCLUSION

In this paper, we uncover a novel attack that against online social apps named Fake-Share Attack. It can bypass online social apps' review of third-party apps, share content (usually links) to online social apps, and display fake sources in the description card to induce users to click. We have designed DeFash, a multi-granularity tool to detect this type of attack. Extensive experiments have shown that DeFash is effective in detecting Fake-Share apps. By applying DeFash to the app market, we have identified 51 Fake-Share apps and over 24,000 SIIs of popular apps that adversaries can abuse. Our observations in this paper suggest that our community should invest more effort into detecting and mitigating this kind of attack.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (grants No.62072046, 61873069, 62102042), Hong Kong RGC Project (No.PolyU15224121), Shanghai Pujiang Program (No.21PJ1410700), the NSF (grants CCF-2046953 and CNS-2028748), the ARC Discovery Early Career Researcher Award (DECRA) project DE200100016 and a Discovery project DP200100020. Guosheng Xu and Siyi Li are co-first authors. Guoai Xu and Haoyu Wang (haoyuwang@hust.edu.cn) are co-corresponding authors.

REFERENCES

[1] 2017. Content sharing: what content people share and why. https://www.i-scoop.eu/content-sharing-content-people-share/. (2017).

[2] 2020. ApkTool:A tool for reverse engineering Android apk files. https://ibotpeaches.github.io/Apktool/. (2020).

[3] 2020. Content sharing and storytelling: why and how people share content. https://www.i-scoop.eu/content-marketing/content-sharing-storytelling/. (2020).

[4] 2020. Getting started with Credential Scanner (CredScan). https://secdevtools.azurewebsites.net/helpcredscan.html. (2020).

[5] 2021. Alipay open platform documentation: third-party application. https://opendocs.alipay.com/isv. (2021).

[6] 2021. DingTalk Sharing Introduction. https://developers.dingtalk.com/document/mobile-app-guide. (2021).

[7] 2021. DroidBot:a lightweight test input generator for Android. https://github.com/honeynet/droidbot. (2021).

[8] 2021. Frida:Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. https://frida.re/. (2021).

[9] 2021. How to link to whatsapp from a different app. https://faq.whatsapp.com/iphone/how-to-link-to-whatsapp-from-a-different-app. (2021).

[10] 2021. Overview:Twitter Developer. https://developer.twitter.com/en/docs/twitter-for-websites/embedded-tweets/overview. (2021).

[11] 2021. Tencent Open Platform. https://wiki.open.qq.com/wiki/. (2021).

[12] 2021. This document introduces the Messaging API. https://developers.kakao.com/docs/latest/en/message/common. (2021).

[13] 2021. Using Soot? Let us know about it! https://github.com/soot-oss/soot. (2021).

[14] 2021. WeChat Android developer documentation: sharing and favorite functions. https://developers.weixin.qq.com/doc/oplatform/Mobile_App/Share_and_Favorites/Android.html. (2021).

[15] 2021. Weibo open platform: mobile application. https://open.weibo.com/development/mobile. (2021).

[16] 2021. Yixin Open Platform: Development Documents. http://open.yixin.im/document. (2021).

[17] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[18] Geumhwan Cho, Junsung Cho, Youngbae Song, and Hyoungshick Kim. 2015. An empirical study of click fraud in mobile advertising networks. In *2015 10th International Conference on Availability, Reliability and Security*. IEEE, 382–388.

[19] Jonathan Crussell, Ryan Stevens, and Hao Chen. 2014. Madfraud: Investigating ad fraud in android applications. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. 123–134.

[20] Martijn de Vos and Johan Pouwelse. 2021. ASTANA: Practical String Deobfuscation for Android Applications Using Program Slicing. *arXiv preprint arXiv:2104.02612* (2021).

[21] 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 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 257–268.

[22] Leonid Glanz, Patrick Müller, Lars Baumgärtner, Michael Reif, Sven Amann, Pauline Anthonysamy, and Mira Mezini. 2020. Hidden in plain sight: Obfuscated strings threatening your privacy. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 694–707.

[23] Yangyu Hu, Haoyu Wang, Yajin Zhou, Yao Guo, Li Li, Bingxuan Luo, and Fangren Xu. 2018. Dating with scambots: Understanding the ecosystem of fraudulent dating applications. *arXiv preprint arXiv:1807.04901* (2018).

[24] Roberto Jordaney, Kumar Sharad, Santanu K Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. 2017. Transcend: Detecting concept drift in malware classification models. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 625–642.

[25] Li Li, Tegawendé F Bissyandé, and Jacques Klein. 2019. Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark. *IEEE Transactions on Software Engineering (TSE)* (2019).

[26] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Haipeng Cai, David Lo, and Yves Le Traon. 2017. Automatically locating malicious packages in piggybacked android apps. In *The 4th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MobileSoft 2017)*.

[27] Jialiu Lin, Shahriyar Amini, Jason I Hong, Norman Sadeh, Janne Lindqvist, and Joy Zhang. 2012. Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing. In *Proceedings of the 2012 ACM conference on ubiquitous computing*. 501–510.

[28] Tianming Liu, Haoyu Wang, Li Li, Guangdong Bai, Yao Guo, and Guoai Xu. 2019. Dapanda: Detecting aggressive push notifications in android apps. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 66–78.

[29] Wei Liu, Yueqian Zhang, Zhou Li, and Haixin Duan. 2016. What you see isn't always what you get: A measurement study of usage fraud on android apps. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*. 23–32.

[30] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. Libradar: Fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th international conference on software engineering companion*. 653–656.

[31] Michael Meli, Matthew R McNiece, and Bradley Reaves. 2019. How Bad Can It Git? Characterizing Secret Leakage in Public GitHub Repositories.. In *NDSS*.

[32] Omid Mirzaei, Guillermo Suarez-Tangil, Jose M de Fuentes, Juan Tapiador, and Gianluca Stringhini. 2019. Andrensemble: Leveraging api ensembles to characterize android malware families. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. 307–314.

[33] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques.. In *NDSS*.

[34] Vibha Singhal Sinha, Diptikalyan Saha, Pankaj Dhoolia, Rohan Padhye, and Senthil Mani. 2015. Detecting and mitigating secret-key leaks in source code repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 396–400.

[35] Haoyu Wang, Jason 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*. 1107–1118.

[36] Haoyu Wang, Yuanchun Li, Yao Guo, Yuvraj Agarwal, and Jason I Hong. 2017. Understanding the purpose of permission use in mobile apps. *ACM Transactions on Information Systems (TOIS)* 35, 4 (2017), 1–40.

[37] Haoyu Wang, Hongxuan Liu, Xusheng Xiao, Guozhu Meng, and Yao Guo. 2019. Characterizing Android app signing issues. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 280–292.

[38] Haoyu Wang, Zhe Liu, Jingyue Liang, Narseo Vallina-Rodriguez, Yao Guo, Li Li, Juan Tapiador, Jingcun Cao, and Guoai Xu. 2018. Beyond google play: A large-scale comparative study of chinese android app markets. In *Proceedings of the Internet Measurement Conference 2018*. 293–307.

[39] Lorraine YC Wong and Jacquelyn Burkell. 2017. Motivations for sharing news on social media. In *Proceedings of the 8th International conference on social media & society*. 1–5.

[40] Shengqu Xi, Shao Yang, Xusheng Xiao, Yuan Yao, Yayuan Xiong, Fengyuan Xu, Haoyu Wang, Peng Gao, Zhuotao Liu, Feng Xu, et al. 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*. 2421–2436.

[41] Ke Xu, Yingjiu Li, Robert Deng, Kai Chen, and Jiayun Xu. 2019. Droidevolver: Self-evolving android malware detection system. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 47–62.

[42] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Hui Liu, Qing Wang, Yueheng Zhang, and Dawu Gu. 2017. Show Me the Money! Finding Flawed Implementations of Third-party In-app Payment in Android Apps.. In *NDSS*.

[43] Xinli Yang, David Lo, Li Li, Xin Xia, Tegawendé F Bissyandé, and Jacques Klein. 2017. Characterizing malicious Android apps by mining topic-specific data flow signatures. *Information and Software Technology* (2017).

[44] Xiaohan Zhang, Yuan Zhang, Ming Zhong, Daizong Ding, Yinzhi Cao, Yukun Zhang, Mi Zhang, and Min Yang. 2020. Enhancing State-of-the-art Classifiers with API Semantics to Detect Evolved Android Malware. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 757–770.

[45] Yanjie Zhao, Li Li, Haoyu Wang, Haipeng Cai, Tegawende Bissyande, Jacques Klein, and John Grundy. 2021. On the Impact of Sample Duplication in Machine Learning based Android Malware Detection. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2021).

[46] Yajin Zhou and Xuxian Jiang. 2012. Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy*. IEEE, 95–109.

[47] Yajin Zhou, Lei Wu, Zhi Wang, and Xuxian Jiang. 2015. Harvesting developer credentials in android apps. In *Proceedings of the 8th ACM conference on security & privacy in wireless and mobile networks*. 1–12.