Uncovering and Exploiting Hidden APIs in Mobile Super Apps

Chao Wang
The Ohio State University

Yue Zhang*
The Ohio State University

Zhiqiang Lin
The Ohio State University

ABSTRACT

Mobile applications, particularly those from social media platforms such as WeChat and TikTok, are evolving into "super apps" that offer a wide range of services such as instant messaging and media sharing, e-commerce, e-learning, and e-government. These super apps often provide APIs for developers to create "miniapps" that run within the super app. These APIs should have been thoroughly scrutinized for security. Unfortunately, we find that many of them are undocumented and unsecured, potentially allowing miniapps to bypass restrictions and gain higher privileged access. To systematically identify these hidden APIs before they are exploited by attackers, we have developed a tool APIScope with both static analysis and dynamic analysis, where static analysis is used to recognize hidden undocumented APIs, and dynamic analysis is used to confirm whether the identified APIs can be invoked by an unprivileged 3rd-party miniapps. We have applied APIScope to five popular super apps (i.e., WeChat, WeCom, Baidu, QQ, and Tiktok) and found that all of them contain hidden APIs, many of which can be exploited due to missing security checks. We have also quantified the hidden APIs that may have security implications by verifying if they have access to resources protected by Android permissions. Furthermore, we demonstrate the potential security hazards by presenting various attack scenarios, including unauthorized access to any web pages, downloading and installing malicious software, and stealing sensitive information. We have reported our findings to the relevant vendors, some of whom have patched the vulnerabilities and rewarded us with bug bounties.

CCS CONCEPTS

• **Security and privacy** → **Web application security**; **Mobile and wireless security**.

KEYWORDS

Hidden APIs, Superapp Security, Miniapp Security

ACM Reference Format:
Chao Wang, Yue Zhang, and Zhiqiang Lin. 2023. Uncovering and Exploiting Hidden APIs in Mobile Super Apps. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3616676

1 INTRODUCTION

Over the past a few years, we have witnessed a rapid growth of the miniapp paradigm [29], in which a mobile super app (e.g., WeChat [6] and TikTok [5]) provides a seamless runtime environment for a miniapp, a web-app alike small application, for enhanced user experience (e.g., install-less) and stickiness with the super app (e.g., a user can access almost all the daily services without leaving it). Today, more than 4.3 million miniapps [7] have been developed in WeChat (a super app with 1.2 billion monthly active users [1]), surpassing the total number of Android apps in Google Play (which has about 2.7 million as of November 2022 [2]). These miniapps offer a variety of daily services from transportation (e.g., ride hailing), e-commerce (e.g., online shopping), e-learning, e-government (e.g., pandemic control and contact tracing), mobile gaming, to entertainment (e.g., short-form user videos), and so on. They are developed by both the 1st-party (i.e., the one who makes the super app platform), as well as the 3rd-party (i.e., developers who create additional software based on the platform provided by the 1st-party).

Obviously, since both the 1st-part and the 3rd-party miniapps are all built on top of the APIs provided by the super app platform, they would have used the same set of the APIs. However, by performing a manual analysis, we discovered discrepancies in the APIs used by these miniapps. For instance, privileged APIs like `openUrl` are present in the 1st-party miniapps like Tencent Doc [4], which has more than 200 million online consumers. `openUrl` can open arbitrary URLs, but the 3rd-party miniapps cannot use `openUrl` and must use the `wx.request` API to ensure that the URLs are checked by WeChat to prevent the loading of malicious content. Moreover, not all APIs are equally mentioned in the official documentation. The Chinese version of the development documentation comprises 975 APIs [8], while the English version has only 570 APIs [9]. Additionally, none of the privileged APIs, such as `openUrl` are ever referenced in the official documentation, regardless of the language. Thus, there may be undocumented APIs in the super app platforms (at least in WeChat). Such undocumented APIs may pose security risks. For example, they may have a higher level of privilege, as they are designed exclusively for use by the 1st-party apps. In order to ensure security, super apps should implement proper access controls for these privileged APIs, such as allowing access solely through an approved list for the 1st-party miniapps. Otherwise, they may be a weak spot for unauthorized access by the 3rd-party miniapps.

Although our manual analysis with the host app and its 1st-party miniapp implementation has yielded surprising findings, it is certainly not scalable nor complete. Meanwhile, given the fact that so many super apps are available today, it will be extremely helpful if we can have a tool to identify all of the hidden APIs if that is possible from their implementations. Also, since privileged APIs without

*This author is now with Drexel University.

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11…$15.00
https://doi.org/10.1145/3576915.3616676

any checks can be easily exploited by malicious miniapps, we must inform the super app vendors to patch the missing or misplaced checks. Motivated by these pressing needs, in this paper, we present APIScope, a binary analysis tool combined with both static and dynamic analysis to systematically scrutinize hidden APIs, which are undocumented, from super app implementations.

Multiple challenges must be addressed while developing APIScope. Particularly, several programming languages have been used to implement a super app at various layers (e.g., JavaScript at the miniapp layer, C/C++ at the JavaScript runtime layer, and Java at the service abstraction layer provided by the host app), and consequently it is challenging to recognize how APIs across these different languages and interfaces are invoked. Second, after identifying an undocumented API, it is also challenging to classify whether it is an API that can be invoked by third-party miniapps. Fortunately, we have addressed these challenges and successfully implemented APIScope. There are two key components inside APIScope: *Static API Recognition* and *Dynamic API Classification*. At a high level, it takes a super app binary as well as its list of public APIs as input, and identifies the hidden APIs based on the invariants of the functions and interfaces from the public APIs in the super apps using *Static API Recognition*. Next, it dynamically executes the identified APIs to confirm whether they are true APIs, and further classifies them into checked and unchecked ones based on whether it can only be invoked by the 1st-party miniapps using *Dynamic API Classification*.

We have tested APIScope with five popular super apps: WeChat, WeCom, Baidu, QQ, and TikTok. Our evaluation results show that all the tested super apps contained hidden APIs. Interestingly, our study found hidden APIs in different categories, with some super apps having more hidden APIs than documented ones. For example, the API category of Payment of WeChat contains 28 hidden APIs, which is significantly more than its documented ones (i.e., only one). We also measure the usage of hidden APIs in both 1st party miniapps and 3rd party miniapps. We found that the use of undocumented APIs is common among both the 1st-party miniapps and the 3rd-party miniapps regardless of their category.

It is evident that not all hidden APIs may pose security risks when misused. Therefore, our objective was to dive into the security implications of hidden APIs. Specifically, we focused on the hidden APIs that lack security checks but can access sensitive Android OS resources. To achieve this, we proposed the use of dynamic analysis techniques. Our dynamic analysis approach involves identifying APIs that call native APIs, which can access sensitive resources. We achieved this by hooking APIs that access sensitive resources and monitoring their use by unchecked and undocumented APIs. After conducting our investigation, we found that WeChat has 39 hidden unchecked APIs (7.77%) that invoke Android APIs protected by permissions. Similarly, WeCom has 40 (6.75%), Baidu has 8 (7.61%), Tiktok has 32 (26.23%), and QQ has 38 (12.88%) such APIs, which can have security risks.

To further validate our findings, we conducted several attack case studies by developing a number of malicious miniapp using these hidden APIs. Specifically, in WeChat, we developed a malicious mini-app to exploit the hidden `private_openUrl` API to access arbitrary malicious content without detection by the super apps. Additionally, by using the `installDownloadTask` hidden API, we developed a mini-app that can download and install harmful Android apps surreptitiously. Malicious apps have the capability to pilfer a user's sensitive information. Our demonstration reveals the utilization of hidden APIs such as `captureScreen`, which enables malicious miniapps to steal screenshots, `getLocalPhoneNumber`, which permits theft of the user's phone number, and `searchContacts`, which facilitates the theft of the user's contact information.

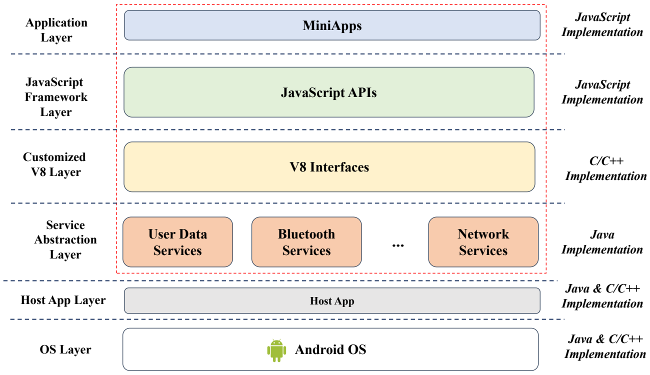


Figure 1: Architecture of Super App Runtime in Android

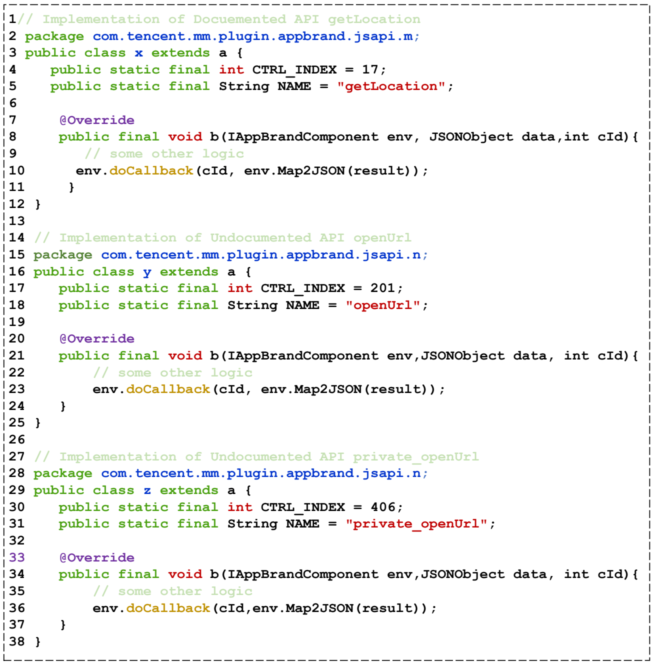
Contributions. We make the following contributions:

- We are the first to discover that super apps may provide hidden, i.e., undocumented, APIs (for the 1st-party miniapps), and those hidden APIs that do not have permission checks can be exploited by the 3rd-party miniapps for privileged accesses.
- We propose APIScope to systematically identify and classify the hidden APIs in super apps, with two novel techniques to statically recognize the APIs and dynamically execute and classify them.
- We implement APIScope, and evaluate it with 5 super apps and find all of them containing hidden APIs, some of which can be exploited by malicious 3rd-party miniapps. We have made the responsible disclosure to their vendors, and received bug bounties from some of them.

2 BACKGROUND

Miniapps are programs that run on top of host apps instead of directly on the operating system. Host apps have to function like an operating system and provide resources (e.g., location, phone numbers, addresses, and social network information) to miniapps through APIs. Mobile super apps are organized in a layered architecture, with each layer focusing on different aspects like portability, security, and convenience, but working together to support miniapp execution within host apps, as shown in Figure 1:

- **Mini-Application Layer**, which is the top layer of a super-app runtime. All miniapps, including the 1st-party and 3rd-party miniapps, are located in this layer. To prevent one miniapp from accessing resources of other miniapps, the host app creates an isolated process for each miniapp. If privileged access is given to the 1st-party miniapps, it must be controlled and checked to prevent the 3rd-party miniapps from using them. Typically, miniapps are implemented using JavaScript [29].



```
1// Implementation of Docuemented API getLocation
2 package com.tencent.mm.plugin.appbrand.jsapi.m;
3 public class x extends a {
4    public static final int CTRL_INDEX = 17;
5    public static final String NAME = "getLocation";
6
7     @Override
8     public final void b(IAppBrandComponent env, JSONObject data,int cId){
9         // some other logic
10      env.doCallback(cId, env.Map2JSON(result));
11     }
12 }
13
14 // Implementation of Undocumented API openUrl
15 package com.tencent.mm.plugin.appbrand.jsapi.n;
16 public class y extends a {
17    public static final int CTRL_INDEX = 201;
18    public static final String NAME = "openUrl";
19
20    @Override
21    public final void b(IAppBrandComponent env,JSONObject data, int cId){
22        // some other logic
23        env.doCallback(cId, env.Map2JSON(result));
24    }
25 }
26
27 // Implementation of Undocumented API private_openUrl
28 package com.tencent.mm.plugin.appbrand.jsapi.n;
29 public class z extends a {
30    public static final int CTRL_INDEX = 406;
31    public static final String NAME = "private_openUrl";
32
33    @Override
34    public final void b(IAppBrandComponent env,JSONObject data, int cId){
35        // some other logic
36        env.doCallback(cId,env.Map2JSON(result));
37    }
38 }
```

Figure 2: APIs implementations of WeChat.

- **JavaScript Framework Layer** provides APIs for resource accesses and management, which are consumed by miniapps in the Application Layer. These APIs allow miniapps to access resources (such as location-based services) and manage UI elements (such as opening a new UI window). The JavaScript Framework Layer is also implemented using JavaScript.
- **Customized V8 Layer**, which provides support for native C/C++ libraries such as WebGL to power the execution of miniapps. It also acts as a bridge between the JavaScript Framework layer and lower-layers. When miniapps call APIs such as wx.getLocation, the Framework layer sends the API name and parameters to the Customized V8 layer, which then passes the request to the underlying layers. This layer is usually implemented using C/C++.
- **Service Abstraction Layer**, which provides an interface to access services from either the super apps (e.g., user account information) or the underlying OS (e.g., Bluetooth, location-based services). In the case of the wx.getLocation API, this layer communicates with the host app using IPC to invoke the Java API getSystemService(LOCATION_SERVICE) to retrieve the current location. This layer is implemented using a combination of Java and C/C++ code for the Android platform.

3 MOTIVATION AND PROBLEM STATEMENT

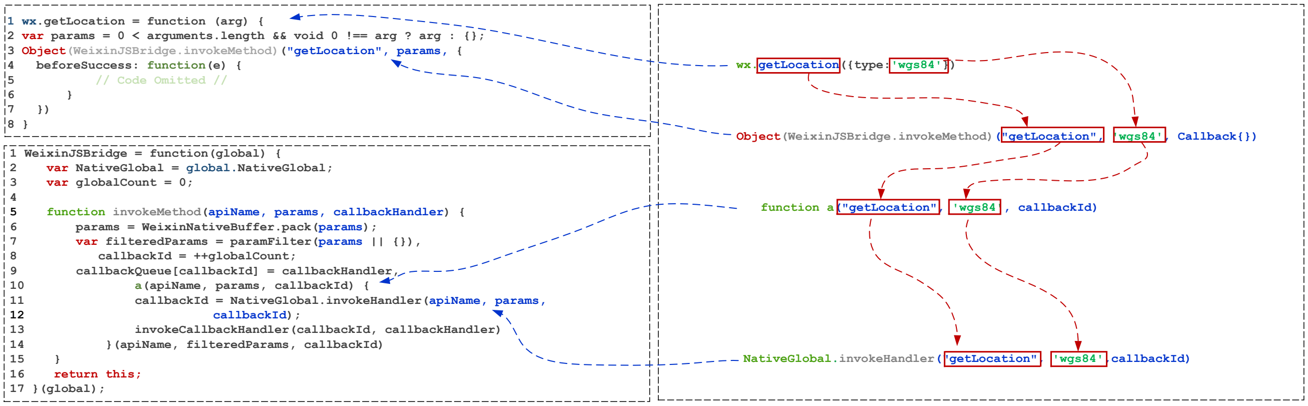
3.1 Key Observations

As alluded earlier, when manually inspecting the implementation of some of the 1st-party miniapps offered by WeChat, we found that other than the public APIs that all the miniapps can access without restrictions, the 1st-party miniapp Tencent Doc actually uses some undocumented APIs (e.g., openUrl for opening arbitrary URLs). Moreover, the designers of WeChat do not make the APIs available to be public (their documentation does not even mention openUrl), and have placed security checks to prevent openUrl from being accessed by arbitrary miniapps. For example, whenever a 3rd-party miniapp attempts to invoke openUrl, WeChat will throw an insufficient permission exception (i.e., "fail: no permission") and terminate its execution. The use of openUrl in the 1st-party Tencent Doc miniapp prompted us to investigate the possibility of other hidden APIs offered by WeChat without proper security checks. This inspired us to explore the feasibility of identifying and exploiting these APIs, but we faced two challenges: (i) identifying the hidden APIs and (ii) properly invoking them to test for potential vulnerabilities. Through further exploration, we made two key observations to address these challenges.

Observation-I: Undocumented API Recognition. By manually inspecting the implementation of WeChat, we found that multiple suspicious undocumented functions are co-located with their documented APIs. That is, those functions and the public APIs are located in the same super app packages, and their implementations look similar to that of the documented APIs (e.g., they have similar function signature, similar parameter type and return value type). We start by inferring whether those functions are indeed undocumented APIs, since intuitively the public APIs and undocumented APIs are APIs, and the developers would have followed the same practice to implement them. Without surprise, we found the implementation of openUrl, which confirms our observation. In Figure 2, we show 3 API implementations of WeChat. Although the code is highly obfuscated (where the names of the classes and methods are replaced with meaningless letters, such as "a","b"), we still can observe some invariants: WeChat's public API getLocation (line 1–13) and its undocumented API openUrl (line 14–25) both have the same parameter types and return types, as well as the same superclass (i.e., class b). As such, we can use these invariants (e.g., the superclass of the API, the parameters of the API) collected from the public APIs to search for possible undocumented APIs. For instance, as shown in Figure 2, we identified another function private_openUrl (lines 28–38) that has the same function signature, which is very likely an undocumented API.

Observation-II: Undocumented API Invocation. Although there may be undocumented APIs (e.g., private_openUrl) provided by WeChat, we have to find a way to invoke them (if they are indeed APIs). Interestingly, when we directly invoke undocumented APIs such as private_openUrl in a miniapp, we obtain an error, "fail: not supported", which is different from the error we observed when invoking openUrl with "fail: no permission". As such, we infer that the accessibility of the API private_openUrl is not the same as that of openUrl (since the observed error messages are different), and there may be a way to invoke it. As such, we further inspected the normal invocation of the documented APIs, and seek to obtain insights from the process.

To be more precise, as described in §2, the JavaScript Framework Layer acquires the invocation request during a regular API call and transfers it to the lower layers via the interfaces exposed by the Customized V8 Layer. In Figure 3, we provide a code snippet illustrating the API invocation chain of WeChat, where the invocation request for the getLocation API (line 3 in the top-left frame) is



```
1 wx.getLocation = function (arg) {
2 var params = 0 < arguments.length && void 0 !== arg ? arg : {};
3 Object(WeixinJSBridge.invokeMethod)("getLocation", params, {
4   beforeSuccess: function(e) {
5             // Code Omitted //
6         }
7   })
8 }
```

```
1 WeixinJSBridge = function(global) {
2    var NativeGlobal = global.NativeGlobal;
3    var globalCount = 0;
4
5    function invokeMethod(apiName, params, callbackHandler) {
6        params = WeixinNativeBuffer.pack(params);
7        var filteredParams = paramFilter(params || {}),
8           callbackId = ++globalCount;
9        callbackQueue[callbackId] = callbackHandler,
10               a(apiName, params, callbackId) {
11               callbackId = NativeGlobal.invokeHandler(apiName, params,
12                          callbackId);
13               invokeCallbackHandler(callbackId, callbackHandler)
14           }(apiName, filteredParams, callbackId)
15    }
16    return this;
17 }(global);
```

```
wx.getLocation({type:'wgs84'})

Object(WeixinJSBridge.invokeMethod)("getLocation", 'wgs84', Callback{})

function a("getLocation", 'wgs84', callbackId)

NativeGlobal.invokeHandler("getLocation", 'wgs84',callbackId)
```

Figure 3: An Example of WeChat API Invocation At JavaScript Framework Layer.

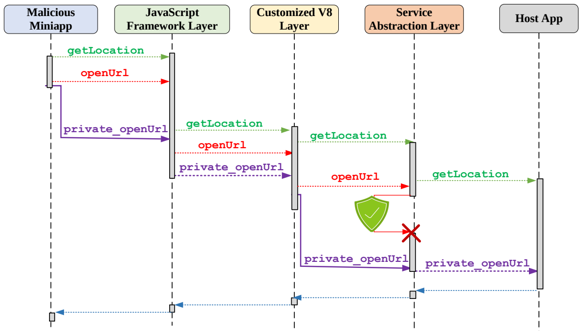


Figure 4: The Workflow of API invocations. Public API invocation `getLocation` (green line); Checked Undocumented API `openUrl` (red line); Unchecked Undocumented API `private_openUrl` (purple line).

eventually passed to the `NativeGlobal.invokeHandler` function (line 11 in the bottom-left frame), which in turn conveys the API invocation request to the underlying layers. Notably, the `NativeGlobal.invokeHandler` function receives three inputs: the API name (e.g., `getLocation`), the API parameters, and a callback function ID (which enables the API to manage the asynchronous call).

Given that `NativeGlobal.invokeHandler` can deliver the normal invocation request to the underlying layers, we conclude that it also has the capabilities to deliver undocumented API invocation requests. Therefore, we feed the API name `private_openUrl` and its parameter (which is a URL) to the interface and let it pass the API name and the URL to the underlying layers. Interestingly, we find that the underlying layers handle the passed API name and the parameter as normal API invocations and further pass the invocation requests to the host apps. As shown in Figure 4, while WeChat restricts the undocumented APIs to be accessed by mini-apps, unfortunately we find that not all undocumented APIs are protected through security checks. In particular, WeChat has enforced the security check for the undocumented API `openUrl`, but it does not add the security checks for the undocumented API `private_openUrl`, which has the exact same functionalities as `openUrl`. Also, the API name and parameters are not obfuscated since they have to be passed to lower layers.

3.2 Problem Statement and Scope

Since our manual investigation has revealed that there are indeed hidden APIs in the super app platform and some of them can be exploited, the goal of this work is to develop techniques to uncover them. More specifically, we need to recognize the hidden APIs based on how documented APIs are implemented and executed, and meanwhile test them to determine whether they can be invoked by the 3rd-party miniapps to bypass security restrictions (or those APIs themselves may have vulnerabilities). Please note that we do not consider all those 3rd-party invocable APIs as exploitable, since whether an API is exploitable depends on the functionalities of the APIs (e.g., the API implements privileged operations).

Also, since there are multiple super apps available today, ideally, we would like to develop generic techniques to cover them all. However, our observation is heavily based on the miniapp run-time architecture presented in Figure 1. Therefore, the super apps that do not follow this architecture, e.g., do not use V8 engine to execute their miniapp code, will be out of our scope. Finally, because of the convenience and also our expertise, we focus on the super apps running on Android platform, though in theory our approach should also work for the iOS platform.

3.3 Threat Model

As previously discussed, our objective is to develop techniques for detecting hidden APIs that lack security checks before a malicious app exploits them. In this context, the attacker is a malware that has been installed on the user's mobile device. We will not delve into the details of how this malware can be installed, as we believe it is practical to assume that super apps are not aware of such types of malware until we report our findings to them. It is worth noting that previous research on super apps has also made similar assumptions [24]. Undocumented APIs refer to functions or APIs that are not included in the official documentation, regardless of whether it is in English or Chinese. An attacker could acquire knowledge about the existence of these hidden APIs by reverse engineering the super app client or by reading technical blogs on the internet. Specifically, undocumented APIs may have access to sensitive resources that are safeguarded by Android OS. If an attacker exploits these APIs, they can launch attacks against the victim users.

4 CHALLENGES AND INSIGHTS

(I) Challenges in API Recognition. The first step of our APIScope is to identify undocumented APIs when given a host app. Intuitively, it sounds trivial, since when given an API, we could compare it with the APIs released on the official documentation to decide whether it is documented or not. However, it is challenging to determine whether an internal function or an interface is an API. For instance, there are 3,702 functions and interfaces implemented in JavaScript, not to mention those implemented in 92 native C/C++ libraries, and 56,492 Java classes in WeChat's latest version. Note that we do not have to consider the functions at lower-layer's implementations (i.e., any layer below the JavaScript framework), since the hidden APIs are not exposed at these layers. Obviously, we cannot directly treat all these functions as APIs.

Also, although for a specific implementation of host apps (e.g., WeChat), simple pattern matching approaches can be applied to recognize APIs. For example, when implementing the callbacks of the APIs, WeChat uses `android.webkit.ValueCallback` at the Service Abstraction layer to handle all the callback results. From the callbacks, we can locate the corresponding APIs and extract patterns to pinpoint the rest APIs. However, there are multiple super apps, each of which could have different implementations. For example, unlike the implementation of WeChat, TikTok uses `com.he.jsbinding.JsContext.ScopeCallback` at the Service Abstraction layer to handle the callback results of their APIs, and the pattern for WeChat will fail when dealing with TikTok. Moreover, such a pattern-matching approach requires recognizing callbacks first, which may be challenging due to the code obfuscation. As discussed in §3.1, the miniapp is executed on top of the super apps (e.g., Android apps), which is often heavily obfuscated. It is hard to recognize callbacks statically unless we fully understand the obfuscated code, and as such, we need a more obfuscation-resilient approach instead of simple pattern matching.

Insights. We notice that there exist some invariants such as the method signatures of public APIs and their superclasses in the API implementations, as illustrated in §3.1 based on super app WeChat (e.g., every API has the same superclass `a`, though this name is obfuscated; every public API must contain the name of the API for the references by the miniapps, and this cannot be obfuscated but can be easily recognized). As such, we can first extract these API invariants based on these public API implementations, from which to recognize the rest of the APIs. This process can be automated since it is easy to identify these API invariants when the implementation of public APIs is provided.

(II) Challenges in API Classification. Once we have identified all these hidden APIs, we still need to further classify them into different categories and determine whether they are invocable (when there is no security check). It will be very challenging if we only use static analysis to decide this, and thus we need to rely on dynamic analysis to dynamically invoke them. However, to invoke a hidden API, we still need to recognize the interface that can communicate with the underlying layers. Although we have already known that the interface communicates with the underlying layers takes the API name as its inputs (as described in §3.1), it is still challenging to know whether this interface accepts the API name as its input before we actually execute it (due to the obfuscated JavaScript code). Meanwhile, although multiple dynamic tools are available for JavaScript, they cannot be applied to our case directly due to the highly customized JavaScript framework implementations. For example, most JavaScript analysis tools (e.g., Jalangi2 [27]) are designed for traditional web browsers. They cannot run with the super apps since the offered APIs are different. Moreover, most of these tools need to instrument the testing instances, which involves the modification of the testing instances. In our case, the testing instances are the miniapps (not web applications), which usually have integrity checks and cannot be modified easily.

Insights. To invoke the API for its behavior classification, we need to find the interface, e.g., `NativeGlobal.invokeHandler` as shown in Figure 3. Interestingly, to identify this interface, we can monitor how a public API is executed, e.g., how it is invoked (its name, parameters), and when it is passed between the boundary of the layers. More specifically, we notice that we can use function trace analysis to identify interfaces such as `NativeGlobal.invokeHandler`, since the API execution starts from the invocation, and ends at the interface boundary. By tracing all of the function executions with their parameters and then identifying them based on the use of the API name, which is passed as parameters, we can automatically identify the interface, which is typically the last invocation point in the JavaScript layer. With the identified invocation point, we can then feed it with different API names and invoke them to classify further (e.g., whether they can be invoked by the 3rd-party miniapps).

5 APISCOPE

As shown in Figure 5, our developed APIScope consists of two phases of analysis—static analysis first and then dynamic analysis, with the following two key components:

- **Static API Recognition (§5.1).** This component takes the binary code of super apps (i.e., APKs) and the list of the official APIs in the documentation as input, and produces the undocumented APIs as output. At a high level, it first decompiles the APKs by Soot [3], automatically extracts the invariants based on the public APIs, and then uses the invariants to recognize the hidden APIs from the implementations of super apps.
- **Dynamic API Classification (§5.2).** This component takes the hidden APIs as input, and classifies them into three different categories: unchecked hidden APIs (exploitable by 3rd miniapps), checked APIs (available to only the 1st-party miniapps), and non-APIs, as the final output. At a high level, it first uses the Test Case Generator to produce two types of test cases: one is for API invocation identification executed by a lightweight tracing engine for the monitored execution, and the other is for API classification. With these test cases, APIScope eventually identifies the interfaces as well as the categories of the APIs.

5.1 Static API Recognition

To recognize APIs, APIScope first needs to extract the invariants based on the decompiled code of public APIs. With the invariants, it then recognizes the hidden APIs. Therefore, it is a two-step process. In the following, we describe these two steps in greater details.

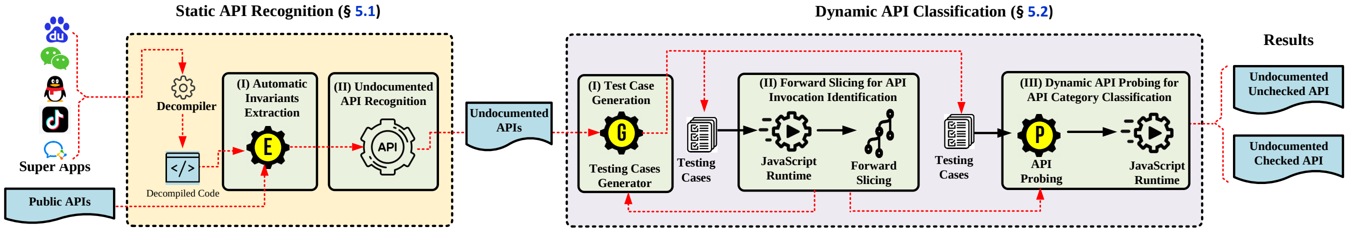

Static API Recognition (§ 5.1)
Dynamic API Classification (§ 5.2)
Results
Super Apps
Public APIs
Decompiler
Decompiled Code
(I) Automatic Invariants Extraction
E
(II) Undocumented API Recognition
API
Undocumented APIs
(I) Test Case Generation
G
Testing Cases Generator
Testing Cases
(II) Forward Slicing for API Invocation Identification
JavaScript Runtime
Forward Slicing
Testing Cases
(III) Dynamic API Probing for API Category Classification
P
API Probing
JavaScript Runtime
Undocumented Unchecked API
Undocumented Checked API

Figure 5: APIScope Architecture

Step-I: Automatic Invariants Extraction. APIScope first needs to extract the invariants based on the decompiled code of the public APIs. from the implementations of the super apps. In particular, when given an API, APIScope will aggressively identify as many invariants as possible from the implementation, and these invariants include: (i) the method signatures (e.g., the return type, the number of the parameters, and parameter types); (ii) the superclass; (iii) the super packages (e.g., in super app Baidu `com.baidu.swan.apps` is the super package of `com.baidu.swan.apps.scheme.actions.f` as shown in Figure 6), and (iv) their callers. Again, they are invariants because they will not be changed in the API implementation (both public and undocumented) for a specific super app, though the specific content for the invariant may be changed across super apps. For instance, in the superclass invariant of APIs, in WeChat, when comparing any two implementations of the provided APIs (e.g., `getLocation` and `private_openUrl`), we can easily recognize that they are both extended from the superclass `a`, as shown in Figure 2; similarly, the superclass of APIs provided by Baidu is extended from the same superclass `aa`, as shown in Figure 6.

Step-II: Undocumented API Recognition. With the invariants, APIScope then recognizes the undocumented APIs. In particular, it iterates each of the function implementations again, by matching the invariants extracted; if it matches with all the invariants as in the public APIs and it has not been added in the undocumented set yet, this function's implementation is an undocumented API. That is, we have used quite restrictive patterns that need to exist in all public API implementations for a particular super app, and a function must contain all of these invariants in order to be considered an undocumented API.

5.2 Dynamic API Classification

With the identified undocumented APIs, next we need to invoke each of them to decide whether they can be exploited by attackers based on the error messages obtained while executing the corresponding test cases for each of the API. This is a three-step process, starting from test case generation, followed by API invocation identification using function trace analysis, and finally the API classification through dynamic API probing.

Step-I: Test Case Generation. In this step, we use our test case generator to produce test cases. The test cases are the JavaScript code snippets that contain the APIs to be invoked (with their parameters configured). For example, `wx.getLocation({type: "wgs84"})` is a test case for testing API `wx.getLocation` (how to invoke such test cases will be described in API invocation identification). There are two types of test cases: one for API invocation identification and the other for API classification. The goal of API invocation identification is to execute the documented API, and use the function trace analysis to identify the invocation point. Therefore, we only need to generate a few test cases (which are the test cases of documented APIs). However, in API classification, which invokes the undocumented APIs and categorizes them based on their outputs, we need to produce at least one valid test case for each undocumented API (to obtain the outputs). In particular, since each API may accept one or multiple parameters, to produce a valid test case, we have to identify all the types (e.g., `Integer`, `Boolean`) of the parameters, through which we can further feed each API a list of parameter instances in the right order (e.g., `testAPI(true, 1234)`):

```
1 // Docuemented API Implementation of Baidu
2 package com.baidu.swan.apps.scheme.actions.f;
3 public class a extends aa {
4    public a (e context) {
5        super(context, "/swanAPI/getLocation");
6    }
7
8    @Override
9    public boolean a (Context c, Scheme s, CallbackHandler cb,  SwanApp a){
10        // some other logic
11    }
12 }
13
14 // Unocuemented API Implementation of Baidu
15 package com.baidu.swan.apps.impl.account.a;
16 public class f extends aa {
17    public f (e context) {
18        super(context, "/swanAPI/getBDUSS");
19    }
20
21    @Override
22    public boolean a (Context c, Scheme s, CallbackHandler cb, SwanApp a){
23        // some other logic
24    }
25 }
```

Figure 6: APIs implementations of Baidu. Note that lines 1 – 12 contain a documented API, and lines 14 – 25 contain an undocumented API.

- **Parameter Type Extraction.** While APIScope could identify the types of parameters through documentation analysis, such an approach cannot identify the types of parameters for undocumented APIs. Therefore, we need a more reliable approach to ensure that we can extract parameter types for both documented and undocumented APIs. Our idea is to analyze the implementations of the APIs, since we have already identified the implementations for both documented and undocumented APIs as described in §5.1. For instance, in WeChat's implementation, we notice that the types of the parameters of an API can be recognized by inspecting the methods invoked by JSON instances, e.g., in the implementation of `getLocation`, we can notice that a JSON object invokes method `optString("paramname", paramvalue)`, which indicates that `getLocation` has a "`paramname`" parameter with type `String`. Similarly, if the API accepts a Boolean value as its parameter, there will be a method `optBoolean("paramname", paramvalue)` in its implementation.

- **Parameter Instance Generation.** The parameters must be instantiated before being fed into the APIs. We used a pre-defined template-based approach to instantiate the parameters. At a high level, the template specified the appropriate values with different types that can be used to produce the parameters (e.g., "1" and "0" are used when the "type" of the parameter is of type "number", and "test" was used when the "type" of the parameter is of type "string"). For instance, WeChat API `showToast` (which shows a message to the user) has two parameters `title` and `duration`, with types string and number, respectively. As such, we produced an instance with the predefined template, where `title` is set to "test" and `duration` is set to "1". Using such a template method, we successfully instantiated all the parameters.
- **Parameter Order Permutation.** Although we have instantiated the parameters, we still do not know the orders of those parameters for the undocumented APIs, as the parameters in the Service Abstraction layer are all encapsulated in JSON objects. Therefore, we have to properly order the parameters, and we use a brute-force approach. For example, `true` and `1234` are two parameters of `testAPI`, which could have two possible combinations: `testAPI(true, 1234))` and `testAPI(1234, true)`. We just assume that all those combinations are valid and invoke them one-by-one (the invalid ones will be filtered out during the API classification, which will be described later). Given that one API can accept no more than 4 parameters (which results in 24 combinations), according to our static analysis with the code, we believe such a brute-force approach is acceptable.

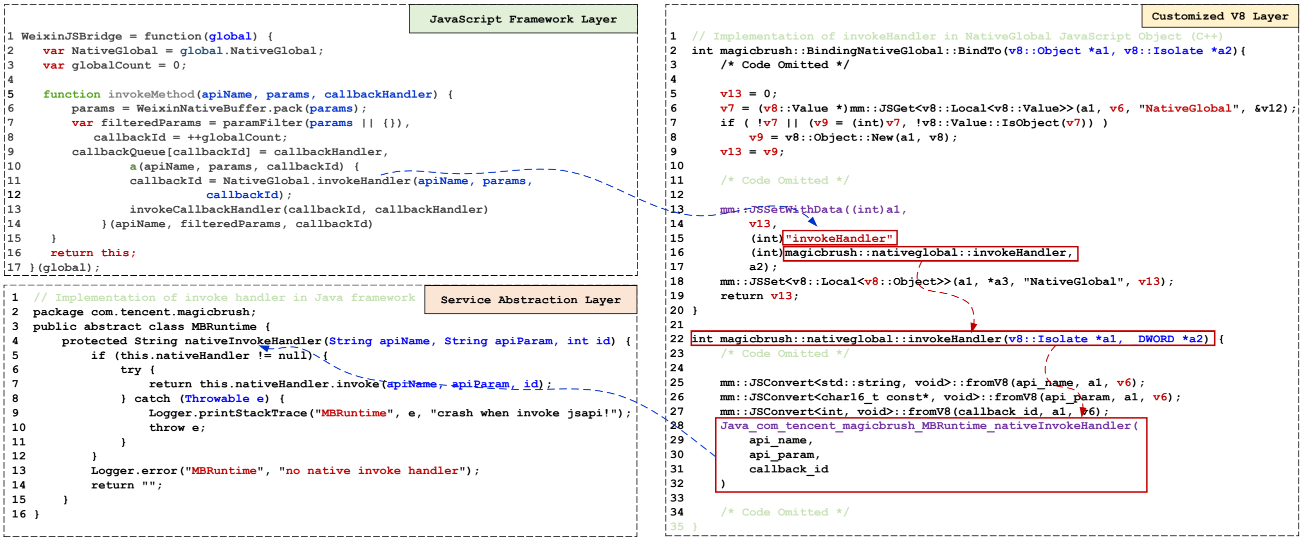
Specifically, we would like to clarify certain technical details. First, during our dynamic analysis, we only explore a limited range of inputs. This is because dynamic tracing does not require a broad range of input to expose hidden APIs. Additionally, the test case generation is sufficient for testing security checks, such as whether the hidden API is protected by security checks. In other words, as long as valid inputs are provided to the API, our tool can trigger the API if there are no security checks. If there are security checks, we can observe errors. Our objective is not to enumerate all possible inputs, as we are not fuzzing the actual hidden API. Second, hidden APIs may require complex parameter types, such as JSON-objects. These complex parameter types are combinations of other basic parameter types (e.g., integer, string), and can be recursively derived until they become primitive types. For instance, an object may contain a string, an integer, and a boolean. We can simply inflate each parameter based on its respective parameter type. As APIs implemented in the Service Abstraction Layer lack states or context, it is unnecessary to determine their execution state within this layer. Our testing process involves providing our tool with a code snippet containing the API to be tested, which is sufficient for our purposes. The JavaScript Framework Layer handles most of the checks, so the API invocation is checked before its order or dependency state is resolved.

Step-II: API Invocation Identification. Next, APIScope needs to execute the generated test cases on top of our customized V8 engine to identify how the documented API is invoked, so that it can later similarly invoke the undocumented ones. Intuitively, when we test a specific API, we need to compile and produce a testing miniapp that contains the API for our test. However, this approach is not scaled and can slow down our testing performance. Interestingly, we notice that we can let the V8 engine directly inject the JavaScript code into the JavaScript Framework Layer (the V8 engine has a function named `script`, which accepts JavaScript code as input, and injects the code for the JavaScript Framework Layer to execute). Since the JavaScript code is injected into the JavaScript Framework layer, the super apps will handle the code as they handle the code in a regular miniapp.

Also, in most cases, V8 Engine has a built-in Profiler, but the super apps do not directly expose any interfaces for developers to use. Meanwhile, although it is true that different platforms may customize the V8 Engine to enable their desired functionalities, they will not intentionally remove the built-in Profiler since it is also helpful for their own debugging purposes. Therefore, as long as we can find a way to invoke Profiler, we will be able to collect the traces. Fortunately, we can use Frida [15], an Android hooking tool, to dynamically instrument the V8 Engine to invoke `startProfiling` of Profiler and let it start profiling, and collect the function traces of documented API execution.

With the collected function traces, we then present how to find the desired interface using function trace analysis, a standard technique widely used in program analysis. As discussed in §3.1, API invocation is a complicated process involving multiple layers. Fortunately, the Profiler only runs inside the JavaScript Framework layer, and we can just monitor the function traces produced at this layer since we aim to identify how to invoke an API from the JavaScript layer. In particular, our analysis starts from the API of our interests (e.g., `wx.getLocation`), identifies all the functions involved based on the dependencies of parameter and API names, and eventually identifies the last invocation function, e.g., `NativeGlobal.invokeHandler` (see Figure 3), which is the desired interface we aim to discover. Specifically, the dependencies are indeed the chained relationship, and we actually build such dependencies based on the parameters that are fed into the functions (we can monitor the changes of parameters of the functions). For example, when we execute `wx.getLocation`, we will observe a function named `NativeGlobal.invokeHandler` that takes a parameter named `getLocation` as its inputs. Therefore, we know that `wx.getLocation` and `NativeGlobal.invokeHandler` have dependencies.

To provide a detailed explanation of how our trace analysis works, we will utilize an example that features the implementations of API invocations across three layers, namely the JavaScript Framework layer, the Customized V8 layer, and the Service Abstraction layer. The process begins with the JavaScript Framework layer, which initiates the API invocation by calling `NativeGlobal.invokeHandler`. This invocation is then handed over to the Customized V8 layer, which is responsible for handling it. As shown in Figure 7, this step is represented line 10 of the JavaScript Framework layer's implementation. Next, the Customized V8 layer extracts critical information from the API invocation, including the API name, its parameters, and any corresponding callbacks. This information is obtained from lines 28–32 of the Customized V8 layer's implementation. The Customized V8 layer then proceeds to invoke



```
1 WeixinJSBridge = function(global) {
2    var NativeGlobal = global.NativeGlobal;
3    var globalCount = 0;
4
5    function invokeMethod(apiName, params, callbackHandler) {
6        params = WeixinNativeBuffer.pack(params);
7        var filteredParams = paramFilter(params || {}),
8           callbackId = ++globalCount;
9        callbackQueue[callbackId] = callbackHandler,
10               a(apiName, params, callbackId) {
11               callbackId = NativeGlobal.invokeHandler(apiName, params,
12                          callbackId);
13               invokeCallbackHandler(callbackId, callbackHandler)
14           }(apiName, filteredParams, callbackId)
15    }
16    return this;
17 }(global);
```

```
1  // Implementation of invoke handler in Java framework
2  package com.tencent.magicbrush;
3  public abstract class MBRuntime {
4      protected String nativeInvokeHandler(String apiName, String apiParam, int id) {
5          if (this.nativeHandler != null) {
6              try {
7                  return this.nativeHandler.invoke(apiName, apiParam, id);
8              } catch (Throwable e) {
9                  Logger.printStackTrace("MBRuntime", e, "crash when invoke jsapi!");
10                 throw e;
11             }
12         }
13         Logger.error("MBRuntime", "no native invoke handler");
14         return "";
15     }
16 }
```

```
1  // Implementation of invokeHandler in NativeGlobal JavaScript Object (C++)
2  int magicbrush::BindingNativeGlobal::BindTo(v8::Object *a1, v8::Isolate *a2){
3      /* Code Omitted */
4
5      v13 = 0;
6      v7 = (v8::Value *)mm::JSGet<v8::Local<v8::Value>>(a1, v6, "NativeGlobal", &v12);
7      if ( !v7 || (v9 = (int)v7, !v8::Value::IsObject(v7)) )
8          v9 = v8::Object::New(a1, v8);
9      v13 = v9;
10
11     /* Code Omitted */
12
13     mm::JSSetWithData((int)a1,
14         v13,
15         (int)"invokeHandler"
16         (int)magicbrush::nativeglobal::invokeHandler,
17         a2);
18     mm::JSSet<v8::Local<v8::Object>>(a1, *a3, "NativeGlobal", v13);
19     return v13;
20 }
21
22 int magicbrush::nativeglobal::invokeHandler(v8::Isolate *a1,  DWORD *a2) {
23     /* Code Omitted */
24
25     mm::JSConvert<std::string, void>::fromV8(api_name, a1, v6);
26     mm::JSConvert<char16_t const*, void>::fromV8(api_param, a1, v6);
27     mm::JSConvert<int, void>::fromV8(callback_id, a1, v6);
28     Java_com_tencent_magicbrush_MBRuntime_nativeInvokeHandler(
29         api_name,
30         api_param,
31         callback_id
32     )
33
34     /* Code Omitted */
35 }
```

Figure 7: The implementations of API invocations across three layers (WeChat)

the relevant APIs at the Service Abstraction Layer through the use of the Java Native Interface (JNI) [21]. Finally, during the API invocations at the Service Abstraction layer (line 4), this layer may need to communicate with the Customized V8 layer for additional operations, such as performing permission checks if the API requires them. We have omitted this code for the sake of brevity. In summary, our trace analysis provides insight into the entire process of API invocations across the three layers of the system. We track the flow of control and collect data on API names, parameters, and callbacks to enable a more comprehensive analysis of the system's behavior.

Step-III: Dynamic Probing for API Category Classification. With the identified interfaces of how to invoke a public API, we then use it to similarly invoke undocumented APIs, by first generating the corresponding test cases, and then injecting the JavaScript code using the script function into the V8 engine, as described earlier. When executing a particular test case, there could be three types of outcomes: the tested "API" is a checked API (when invoked, a permission denial will be observed based on the standard error messages), the tested "API" is an unchecked API (which can be invoked successfully), the tested "API" is not an API. As such, we can use the following strategies to identify them.

- **Unchecked APIs.** Similar to the public APIs, the unchecked undocumented APIs can be invoked without requiring additional permissions. As such, we first deliver a public API invocation request, such as `getLocation`, and record the feedback of the host app. For example, WeChat and Baidu will not print any errors when the invocation request gets approved, and we then use this as a signature to see whether an invocation request is successfully executed.
- **Checked APIs.** The checked APIs are the APIs that are protected by security checks, which can only be invoked by their 1st-party miniapps. In the event of a security check failure, the super apps will generate error messages notifying the user of insufficient permissions. This exception applies to all APIs within various super apps, albeit with minor variations in the error messages displayed. For example, when the 3rd-party mini-apps attempt to invoke a checked API of WeChat, the host app will throw an error message "`fail: no permission`". For WeCom, the error message becomes "`fail: access denied`". Therefore, we use keywords such as "`fail`", "`no permission`" and "`access denied`" to match and decide whether the invocation request gets denied. If so, it is a checked API.
- **Non-APIs.** Theoretically, APIScope may have false positives, and as such, our tool may mistakenly recognize some non-APIs. Therefore, we need to filter them out. To that end, we first create an invalid request and then send it to the host app to see the feedback. For example, if we initiate an invalid request and send it to WeChat, WeChat will reject the invocation request and throw an error message "`fail: not supported`". Then, such an error message is used as a signature to match the non-APIs.

As an example, in the case of WeChat, if we attempt to use the API `openUrl`, the super app will generate an error message stating "`fail: no permission`". This error message implies that the API is a checked hidden API. On the other hand, if we use the API `private_openUrl`, the super app will handle the invocation request as a regular request without displaying any error message. As a result, we can conclude that this API is an unchecked hidden API.

6 EVALUATION

6.1 Experiment Setup

The Tested Host Apps. Today, there are quite a number of super apps that support the execution of miniapps. Although we wish to test all of them, eventually we selected five of them, as shown in Table 1, and these include WeChat, WeCom and QQ from Tencent Holdings Ltd., Baidu from Baidu Inc., and TikTok from ByteDance Ltd. We excluded other super apps such as Alipay and Snapchat particularly because they do not build on the V8 engine (making our

Name	Vendor	Version	V8	Date	Installs	1st-party miniapp being tested?
Baidu	Baidu	12.21	7.6	08/13/2021	5,000,000+	✓
QQ	Tencent	8.8	7.2	10/05/2021	10,000,000+	✓
TikTok	ByteDance	17.9	7.2	10/19/2021	1,000,000,000+	✗
WeChat	Tencent	8.0	8.0	07/21/2021	100,000,000+	✓
WeCom	Tencent	3.1	8.0	09/14/2021	100,000+	✓

Table 1: Summary of the Tested Super Apps

tool unsuitable for them at this moment). Also, to study the security issues of the tested super apps correspondingly, we registered an account in each platform, downloaded their development tools and SDKs, built miniapps by following their official documents, and inspected their code. Among them, Baidu has a relatively closed ecosystem, where only the enterprise developers are allowed to register as their developers. However, they allow individuals to apply for trial accounts to use their development tools to develop miniapps, and therefore, we tested Baidu using their trial accounts.

The Tested Miniapps. We believe it is important to measure the usage of undocumented APIs in the 1st-party and 3rd-party miniapps for two reasons. First, understanding how the 1st-party miniapps use these APIs can help us comprehend the entire ecosystem. Second, if the 3rd-party developers know about these APIs, they may use them, which can lead to security issues if these APIs have access to sensitive resources. To analyze the usage of undocumented APIs in 1st-party miniapps, we searched for interfaces provided by host apps and collected 236 miniapps from WeChat and WeCom, 340 miniapps from Baidu, and 24 miniapps from QQ. We could not find information about the 1st-party miniapps of TikTok, so we did not report their API usage. We could not scan all 3rd-party miniapps because there is no public dataset or crawlers available. Therefore, we can only measure the usage of hidden APIs among the 3rd-party miniapps within the WeChat ecosystem. We collected 267, 359 miniapps using Mini-Crawler [37] within 3 weeks.

The Testing Environment. We performed our static analysis on one laptop, which has 6 cores, Intel Core i7-10850H (4.90 GHz) CPUs and 64 GB RAM, and our dynamic analysis on a Google Pixel 4 running Android 11 and a Google Pixel 2 running Android 9, since we particularly focused on the Android version of miniapps.

6.2 Effectiveness

The effectiveness evaluation aims to quantify how APIScope uncovered the hidden APIs in terms of the specific numbers for the involved analysis (which is presented in Table 2), and their qualities (i.e., whether there are any false positives). It is worth noting that the manually created cases are indeed rare. For example, for Baidu, we automatically created 423 test cases, and created another 56 test cases manually, so the manual efforts are around 11%, i.e., 56/(56+423) = 0.11. Other super apps even have a lower amount of manual efforts than Baidu (e.g., WeCom has 2.9 % manual efforts).

Specifically, the effectiveness of our static analysis is measured by the identification of API invariants, the number of identified API candidates (i.e., the functions that are very likely to be APIs). However, whether those API candidates are really APIs are determined in dynamic API classification. For the API invariants, while we have listed four invariants in §5.1, not all of them will exist in all super apps (e.g., Baidu and QQ do not have caller invariant), as shown in Table 2. That is why APIScope aggressively identifies as many invariants as possible. With these invariants, it sufficiently recognizes the undocumented APIs even though some of them do not exist in other super apps. During static API recognition, APIScope recognized in total 1,829 API candidates for these super apps. Among them, WeCom contains the most hidden API candidates (683), followed by WeChat (containing 575 API candidates). Tiktok has fewer API candidates (i.e., 124 API candidates), likely due to its smallest LoC compared to other super apps.

The effectiveness of dynamic analysis is measured by the number of traced functions during API invocation identification and the number of test cases used during API classification. Among the test cases, we also quantify the number of automatically generated test cases and manually created test cases. We can see that most of the test cases are automatically generated by our test case generation algorithm, and the number of automatically generated test cases is greater than the number of API candidates due to the parameter order permutation (as discussed in §5.2). With our dynamic classification for the identified APIs, APIScope detected a large number of hidden APIs, many of which are unchecked (as reported in Table 2). WeChat has more APIs (590 public APIs, 502 undocumented unchecked APIs, and 65 undocumented checked APIs) than the other super apps. However, TikTok has a relatively small number of APIs (383 public APIs, 120 undocumented unchecked APIs, and 2 undocumented checked APIs). With respect to the percentage of undocumented unchecked and checked APIs, WeCom has the most undocumented unchecked APIs (46.3%) and undocumented checked APIs (6.4%).

Correctness of Our Result. We quantify whether there are any false positives or false positives for the identified hidden APIs. First, a false positive here means that the identified API is not hidden, or is not an API. By design, APIScope will not have false positives for two reasons: (1) the invariants we extracted have very strict patterns (they have to exist among all public APIs and all of them have to be present in the undocumented APIs), and (2) our dynamic probing for API classification can filter out those non-APIs, which eliminate potential false positives. Nevertheless, we still thoroughly scrutinized each API identified for WeChat by conducting a manual check to ensure that there were no false positives. In other words, we made sure that the tool did not mistakenly classify non-APIs as APIs. Thanks to our design, we did not come across any false positives during our examination. Second, with respect to false negatives (i.e., "true" hidden API is missed by APIScope), we note that theoretically APIScope could have false negatives, for instance, if our invariants are too strong. However, we will not be able to quantify this, since we do not have the ground truth, unless we can manually examine each line of code. Therefore, we leave this to future work.

Categories of the Identified APIs. With the identified APIs, we can then obtain some insights with them, such as which category contains more hidden APIs. To this end, we manually walked through each API, and categorize them based on the categories of the documented ones, to classify the undocumented (i.e., hidden) APIs. This result is presented in Table 3. Interestingly, we found that most of the categories contain undocumented unchecked APIs. In particular, for some of the super apps (e.g., WeChat), their undocumented unchecked APIs can be even more than the documented

Name	Input			Static Analysis					Dynamic Analysis			Output		
	#Size (MBs)	# of LoC	# of Public API	API Invariants				# of Hidden API Candidates	Invocation Identification (# of Traced Functions)	API Classification (# of Test Cases)		# of Checked API	# of Unchecked API	# of Non API
				Method Signature	Super Class	Super Package	Callers			# of Auto Generated	# of Manually Created			
Baidu	123.6	2,005,003	464	✓	✓	✓	✗	143	30	423	56	25	113	5
QQ	138.6	1,557,805	506	✓	✓	✓	✗	304	43	1,083	61	6	295	3
TikTok	6.2	718,395	383	✓	✓	✓	✓	124	37	352	53	2	122	0
WeChat	199.2	1,609,650	590	✓	✓	✓	✓	575	28	2,184	66	65	502	8
WeCom	224.8	1,067,273	606	✓	✓	✓	✓	683	31	2,315	70	82	593	8

Table 2: Effectiveness of APIScope with the tested super apps. The terms “Signature”, “Super Class”, “Super Package”, and “Callers” have consistent meanings with those defined in §5.1.

Available APIs		WeChat						WeCom						Baidu						TikTok						QQ					
		D	%	UU	%	UC	%	D	%	UU	%	UC	%	D	%	UU	%	UC	%	D	%	UU	%	UC	%	D	%	UU	%	UC	%
Base	Basic	5	71.4	2	28.6	-	0.0	6	66.7	3	33.3	-	0.0	8	72.7	2	18.2	1	9.1	7	63.6	4	36.4	-	0.0	3	100.0	-	0.0	-	0.0
	App	13	39.4	14	42.4	6	18.2	13	37.1	16	45.7	6	17.1	8	42.1	10	52.6	1	5.3	6	50.0	6	50.0	-	0.0	9	34.6	17	65.4	-	0.0
	Debug	15	88.2	2	11.8	-	0.0	15	88.2	2	11.8	-	0.0	1	3.3	28	93.3	1	3.3	-	0.0	-	0.0	-	0.0	20	100.0	-	0.0	-	0.0
	Misc	10	58.8	7	41.2	-	0.0	10	55.6	8	44.4	-	0.0	9	100.0	-	0.0	-	0.0	10	52.6	9	47.4	-	0.0	9	100.0	-	0.0	-	0.0
UI	Interaction	6	46.2	7	53.8	-	0.0	6	46.2	7	53.8	-	0.0	7	41.2	10	58.8	-	0.0	9	81.8	2	18.2	-	0.0	6	40.0	9	60.0	-	0.0
	Navigation	4	44.4	5	55.6	-	0.0	4	40.0	6	60.0	-	0.0	4	100.0	-	0.0	-	0.0	5	100.0	-	0.0	-	0.0	4	33.3	8	66.7	-	0.0
	Animation	32	100.0	-	0.0	-	0.0	32	100.0	-	0.0	-	0.0	21	95.5	1	4.5	-	0.0	1	100.0	-	0.0	-	0.0	31	100.0	-	0.0	-	0.0
	WebView	-	0.0	22	95.7	1	4.3	-	0.0	24	96.0	1	4.0	-	0.0	3	75.0	1	25.0	-	0.0	3	100.0	-	0.0	-	0.0	16	100.0	-	0.0
	Misc	20	27.0	54	73.0	-	0.0	20	25.6	58	74.4	-	0.0	37	77.1	11	22.9	-	0.0	14	73.7	5	26.3	-	0.0	18	42.9	24	57.1	-	0.0
Network	Request	5	55.6	4	44.4	-	0.0	5	55.6	4	44.4	-	0.0	2	66.7	1	33.3	-	0.0	6	60.0	4	40.0	-	0.0	4	66.7	2	33.3	-	0.0
	Download	7	24.1	21	72.4	1	3.4	7	23.3	22	73.3	1	3.3	11	100.0	-	0.0	-	0.0	-	0.0	4	100.0	-	0.0	6	60.0	4	40.0	-	0.0
	Upload	7	50.0	5	35.7	2	14.3	7	46.7	6	40.0	2	13.3	6	100.0	-	0.0	-	0.0	-	0.0	4	100.0	-	0.0	6	75.0	2	25.0	-	0.0
	Websocket	14	93.3	1	6.7	-	0.0	14	93.3	1	6.7	-	0.0	13	100.0	-	0.0	-	0.0	7	77.8	2	22.2	-	0.0	13	86.7	2	13.3	-	0.0
	Misc	23	88.5	3	11.5	-	0.0	23	85.2	4	14.8	-	0.0	-	0.0	-	0.0	-	0.0	-	0.0	-	0.0	-	0.0	10	55.6	8	44.4	-	0.0
Storage		10	66.7	5	33.3	-	0.0	10	66.7	5	33.3	-	0.0	10	100.0	-	0.0	-	0.0	10	90.9	1	9.1	-	0.0	10	83.3	2	16.7	-	0.0
Media	Map	8	14.3	48	85.7	-	0.0	8	14.3	48	85.7	-	0.0	7	100.0	-	0.0	-	0.0	6	100.0	-	0.0	-	0.0	9	36.0	16	64.0	-	0.0
	Image	6	60.0	4	40.0	-	0.0	6	60.0	4	40.0	-	0.0	6	85.7	1	14.3	-	0.0	5	83.3	1	16.7	-	0.0	6	60.0	4	40.0	-	0.0
	Video	14	35.0	26	65.0	-	0.0	14	31.8	30	68.2	-	0.0	19	95.0	1	5.0	-	0.0	8	80.0	2	20.0	-	0.0	14	63.6	8	36.4	-	0.0
	Audio	64	84.2	9	11.8	3	3.9	64	79.0	14	17.3	3	3.7	44	100.0	-	0.0	-	0.0	44	81.5	10	18.5	-	0.0	61	85.9	10	14.1	-	0.0
	Live	26	46.4	30	53.6	-	0.0	26	39.4	40	60.6	-	0.0	8	100.0	-	0.0	-	0.0	19	100.0	-	0.0	-	0.0	23	57.5	17	42.5	-	0.0
	Recorder	16	84.2	3	15.8	-	0.0	16	84.2	3	15.8	-	0.0	12	100.0	-	0.0	-	0.0	11	91.7	1	8.3	-	0.0	15	88.2	2	11.8	-	0.0
	Camera	9	60.0	6	40.0	-	0.0	9	52.9	8	47.1	-	0.0	9	50.0	9	50.0	-	0.0	20	95.2	1	4.8	-	0.0	4	36.4	7	63.6	-	0.0
	Misc	12	75.0	3	18.8	1	6.3	12	75.0	3	18.8	1	6.3	18	100.0	-	0.0	-	0.0	-	0.0	-	0.0	-	0.0	6	100.0	-	0.0	-	0.0
Location		3	42.9	4	57.1	-	0.0	3	42.9	4	57.1	-	0.0	7	100.0	-	0.0	-	0.0	3	100.0	-	0.0	-	0.0	3	100.0	-	0.0	-	0.0
Share		4	33.3	7	58.3	1	8.3	4	16.7	19	79.2	1	4.2	3	100.0	-	0.0	-	0.0	5	71.4	2	28.6	-	0.0	5	35.7	9	64.3	-	0.0
Canvas		60	74.1	21	25.9	-	0.0	60	74.1	21	25.9	-	0.0	46	92.0	4	8.0	-	0.0	49	98.0	1	2.0	-	0.0	48	92.3	4	7.7	-	0.0
File		39	97.5	1	2.5	-	0.0	39	92.9	3	7.1	-	0.0	35	100.0	-	0.0	-	0.0	34	97.1	1	2.9	-	0.0	37	97.4	1	2.6	-	0.0
Open API	Login	2	100.0	-	0.0	-	0.0	5	83.3	1	16.7	-	0.0	3	42.9	1	14.3	3	42.9	2	100.0	-	0.0	-	0.0	2	100.0	-	0.0	-	0.0
	Navigate	2	33.3	2	33.3	2	33.3	2	22.2	5	55.6	2	22.2	3	100.0	-	0.0	-	0.0	7	100.0	-	0.0	-	0.0	2	50.0	1	25.0	1	25.0
	User Info	2	16.7	7	58.3	3	25.0	5	23.8	13	61.9	3	14.3	1	10.0	6	60.0	3	30.0	2	13.3	13	86.7	-	0.0	2	28.6	4	57.1	1	14.3
	Payment	1	3.4	13	44.8	15	51.7	1	3.2	15	48.4	15	48.4	1	50.0	-	0.0	1	50.0	1	33.3	1	33.3	1	33.3	2	22.2	7	77.8	-	0.0
	Bio-Auth	3	27.3	3	27.3	5	45.5	3	21.4	6	42.9	5	35.7	-	0.0	-	0.0	-	0.0	-	0.0	1	100.0	-	0.0	3	100.0	-	0.0	-	0.0
	Enterprise	-	0.0	1	100.0	-	0.0	5	17.9	6	21.4	17	60.7	-	0.0	-	0.0	-	0.0	-	0.0	-	0.0	-	0.0	-	0.0	-	0.0	-	0.0
	Misc	14	19.4	42	58.3	16	22.2	14	16.7	54	64.3	16	19.0	16	57.1	2	7.1	10	35.7	25	55.6	20	44.4	-	0.0	12	13.0	78	84.8	2	2.2
Device	Wi-Fi	9	100.0	-	0.0	-	0.0	9	100.0	-	0.0	-	0.0	10	100.0	-	0.0	-	0.0	4	100.0	-	0.0	-	0.0	9	100.0	-	0.0	-	0.0
	Bluetooth	18	60.0	11	36.7	1	3.3	18	58.1	12	38.7	1	3.2	-	0.0	-	0.0	-	0.0	-	0.0	-	0.0	-	0.0	18	100.0	-	0.0	-	0.0
	Contact	1	10.0	5	50.0	4	40.0	1	9.1	6	54.5	4	36.4	1	33.3	2	66.7	-	0.0	-	0.0	-	0.0	-	0.0	1	25.0	2	50.0	1	25.0
	NFC	5	26.3	14	73.7	-	0.0	9	39.1	14	60.9	-	0.0	-	0.0	-	0.0	-	0.0	-	0.0	-	0.0	-	0.0	5	100.0	-	0.0	-	0.0
	Screen	4	36.4	6	54.5	1	9.1	4	36.4	6	54.5	1	9.1	3	100.0	-	0.0	-	0.0	9	100.0	-	0.0	-	0.0	4	100.0	-	0.0	-	0.0
	Phone	1	4.3	21	91.3	1	4.3	1	4.3	21	91.3	1	4.3	1	100.0	-	0.0	-	0.0	1	100.0	-	0.0	-	0.0	1	50.0	1	50.0	-	0.0
	Misc	28	63.6	15	34.1	1	2.3	28	59.6	18	38.3	1	2.1	21	80.8	5	19.2	-	0.0	16	69.6	7	30.4	-	0.0	28	82.4	6	17.6	-	0.0
AI	CV	19	100.0	-	0.0	-	0.0	19	100.0	-	0.0	-	0.0	18	90.0	2	10.0	-	0.0	-	0.0	-	0.0	-	0.0	-	0.0	-	0.0	-	0.0
	Misc	-	0.0	-	0.0	-	0.0	-	0.0	1	100.0	-	0.0	11	100.0	-	0.0	-	0.0	7	100.0	-	0.0	-	0.0	-	0.0	-	0.0	-	0.0
AD		19	95.0	1	5.0	-	0.0	19	95.0	1	5.0	-	0.0	9	64.3	4	28.6	1	7.1	13	61.9	8	38.1	-	0.0	3	25.0	9	75.0	-	0.0
Uncategorized		30	38.5	47	60.3	1	1.3	30	36.6	51	62.2	1	1.2	15	53.6	10	35.7	3	10.7	17	68.0	7	28.0	1	4.0	34	68.0	15	30.0	1	2.0
All		590	51.0	502	43.4	65	5.6	606	47.3	593	46.3	82	6.4	464	77.1	113	18.8	25	4.2	383	75.8	120	23.8	2	0.4	506	62.7	295	36.6	6	0.7

Table 3: Categories of Documented and Undocumented APIs. “D” means documented APIs; “UU” means undocumented unchecked APIs; “UC” means undocumented checked APIs.

APIs in some of the categories (e.g., the API category Payment has 28 undocumented APIs, which is way more than their documented APIs). Finally, we found that some well-documented APIs of a specific super app may not be open to the public in other super apps. For example, `getUserInfo` is an undocumented API of Baidu, while WeChat has the same API with the same functionalities, which is publicly accessible. Finally, since APIScope is a systematic and mostly automated tool, it can inspect API changes based on previous versions of the super app implementations as long as we can obtain both their APKs and documentation.

Category	WeChat			WeCom			Baidu			QQ		
	# U	# App	%	# U	# App	%	# U	# App	%	# U	# App	%
Business	14	49	28.6	16	49	32.7	21	38	55.3	1	3	33.3
Education	6	26	23.1	7	26	26.9	5	16	31.3	-	3	0.0
E-learning	5	9	55.6	5	9	55.6	12	33	36.4	-	1	0.0
Entertainment	9	17	52.9	9	17	52.9	29	75	38.7	2	2	100.0
Finance	1	1	100.0	1	1	100.0	21	23	91.3	-	-	0.0
Food	-	-	0.0	-	-	0.0	-	5	0.0	-	-	0.0
Games	18	36	50.0	18	36	50.0	-	-	0.0	-	-	0.0
Government	2	7	28.6	2	7	28.6	3	8	37.5	1	1	100.0
Health	2	7	28.6	2	7	28.6	1	5	20.0	-	1	0.0
Job	-	1	0.0	-	1	0.0	-	-	0.0	-	-	0.0
Lifestyle	2	5	40.0	2	5	40.0	3	15	20.0	-	1	0.0
Photo	3	7	42.9	3	7	42.9	-	-	0.0	-	-	0.0
Shopping	1	1	100.0	1	1	100.0	-	2	0.0	-	-	0.0
Social	4	8	50.0	4	8	50.0	1	4	25.0	-	1	0.0
Sports	-	-	0.0	-	-	0.0	-	1	0.0	-	-	0.0
Tool	15	55	27.3	15	55	27.3	16	47	34.0	4	8	50.0
Traffic	3	5	60.0	3	5	60.0	4	10	40.0	-	1	0.0
Travelling	2	2	100.0	2	2	100.0	1	56	1.8	1	2	50.0
Uncategorized	-	-	0.0	-	-	0.0	1	2	50.0	-	-	0.0
Total	87	236	36.9	90	236	38.1	118	340	34.7	9	24	37.5

Table 4: The 1st party miniapps that have used the undocumented APIs. The first column indicates the number of the 1st-party mini-apps using undocumented APIs, and the second column represents the total number of the 1st-party mini-apps. We calculate the percentage of mini-apps by using the first column divided by the second.

Usage of Hidden APIs (Among the 1st-party Miniapps). We obtained many 1st-party miniapps and classified them into categories based on their meta-data. From the data in Table 4, we found that the use of undocumented APIs is common among the 1st-party miniapps regardless of their category. WeCom had the highest percentage of the 1st-party miniapps using undocumented APIs at 38.1%, followed by QQ at 37.5%, WeChat at 36.9%, and Baidu at 34.7%. We also observed that the 1st-party miniapps in the Traveling, Shopping, and Finance categories were more likely to use undocumented APIs, and these APIs were often related to payment. For example, many miniapps in these categories would use the undocumented API `verifyPaymentPassword` to verify payment passwords.

Next, we sought to understand the most popular undocumented APIs and how often they are used by the 1st-party miniapps. We grouped the APIs by name and counted the number of miniapps that used each API. This information is presented in table Table 5. We found that 7 undocumented APIs provided by Baidu were used by their 1st-party miniapps, 34 undocumented APIs provided by WeChat were used by their 1st-party miniapps (only 19 of which are listed in Table 5 due to space constraints), 43 undocumented APIs provided by WeCom were used by their 1st-party miniapps (again, only those used by more than two miniapps are shown), and 14 undocumented APIs provided by QQ were used by their 1st-party miniapps.

Finally, we present whether there are any missing security checks for these undocumented APIs from our API classification result in the last column of Table 5. We found that 3 out of 7 (42.9%) APIs used by Baidu's 1st-party miniapps do not have security checks and can be invoked and exploited by its 3rd-party miniapps; 16 of 34 (47.06%) APIs of WeChat; 22 of 43 (51.16%) APIs of WeCom; and 12 of 14 (85.7%) APIs of QQ can be exploited by their 3rd-party miniapps. We also noticed that different vendors have different security restrictions on their undocumented APIs. For example, WeChat and WeCom place security checks on their undocumented APIs that are related to payment (`wx.requestVirtualPayment`), authentication (`wx.startFacialRecognitionVerify`) and access to resources (`wx.openUrl`).

	API Name	Category	# App	% *App	w/ Check
Baidu	swan.button	Interaction	104	88.14	✗
	swan.login	Login	31	26.27	✓
	swan.postMessage	Uncategorized	8	6.78	✗
	swan.getBDUSS	User Info	4	3.39	✓
	swan.getCommonSysInfo	System	3	2.54	✓
	swan.getUserInfo	User Info	3	2.54	✗
	swan.getChannelID	Uncategorized	2	1.69	✓
WeChat	wx.hideNavigationBar	Bar	28	32.18	✗
	wx.requestSubscribeMessage	Subscribe	25	28.74	✗
	wx.showNavigationBar	Bar	23	26.44	✗
	wx.requestVirtualPayment	Payment	11	12.64	✓
	wx.openUrl	Misc	8	9.20	✓
	wx.hideHomeButton	Interaction	8	9.20	✗
	wx.enterContact	Contact	5	5.75	✓
	wx.drawCanvas	Canvas	5	5.75	✗
	wx.setPageOrientation	Misc	4	4.60	✗
	wx.operateWXData	Misc	4	4.60	✗
	wx.getBackgroundFetchData	Misc	3	3.45	✗
	wx.setBackgroundFetchToken	Misc	3	3.45	✗
	wx.startFacialRecognitionVerify	Bio-Auth	3	3.45	✓
	wx.checkIsSupportFacialRecognition	Bio-Auth	2	2.30	✓
	wx.navigateBackApplication	Navigate	2	2.30	✗
	wx.navigateBackNative	Navigate	2	2.30	✓
	wx.onDeviceOrientationChange	Device	2	2.30	✗
	wx.openBusinessView	View	2	2.30	✗
	wx.verifyPaymentPassword	Payment	2	2.30	✓
WeCom	wx.hideNavigationBar	Bar	28	31.11	✗
	wx.requestSubscribeMessage	Subscribe	25	27.78	✗
	wx.showNavigationBar	Bar	23	25.56	✗
	wx.requestVirtualPayment	Payment	11	12.22	✓
	wx.openUrl	Misc	8	8.89	✓
	wx.hideHomeButton	Interaction	8	8.89	✗
	wx.enterContact	Contact	5	5.56	✓
	wx.drawCanvas	Canvas	5	5.56	✗
	wx.setPageOrientation	Misc	4	4.44	✗
	wx.operateWXData	Misc	4	4.44	✗
	wx.getBackgroundFetchData	Misc	3	3.33	✗
	wx.setBackgroundFetchToken	Misc	3	3.33	✗
	wx.startFacialRecognitionVerify	Bio-Auth	3	3.33	✓
	wx.checkIsSupportFacialRecognition	Bio-Auth	2	2.22	✓
	wx.navigateBackApplication	Navigate	2	2.22	✗
	wx.navigateBackNative	Navigate	2	2.22	✓
	wx.openBusinessView	Misc	2	2.22	✗
	wx.qy.chooseAttach	File	2	2.22	✓
	wx.qy.chooseWxworkContact	Enterprise	2	2.22	✓
	wx.qy.chooseWxworkVisibleRange	Enterprise	2	2.22	✓
	wx.qy.openWechatWebviewUrl	WebView	2	2.22	✗
	wx.qy.postNotification	System	2	2.22	✓
	wx.qy.showUserProfile	User Info	2	2.22	✓
	wx.qy.wwLog	Uncategorized	2	2.22	✗
	wx.qy.wwOpenUrlScheme	Uncategorized	2	2.22	✓
	wx.verifyPaymentPassword	Payment	2	2.22	✓
QQ	qq.openUrl	Misc	4	44.44	✗
	qq.addRecentColorSign	UI	3	33.33	✗
	qq.exitMiniProgram	App	2	22.22	✗
	qq.getGroupInfo	User Info	2	22.22	✗
	qq.getGroupInfoExtra	User Info	2	22.22	✗
	qq.getPerformance	System	1	11.11	✗
	qq.getQua	Uncategorized	1	11.11	✗
	qq.getUserInfoExtra	User Info	1	11.11	✗
	qq.invokeNativePlugin	System	1	11.11	✓
	qq.notifyNative	System	1	11.11	✗
	qq.openScheme	Misc	1	11.11	✓
	qq.requestMidasPayment	Payment	1	11.11	✗
	qq.toggleSecureWindow	UI	1	11.11	✗
	qq.wnsRequest	App	1	11.11	✗

Table 5: The popular hidden APIs invoked by the 1st-party miniapps.

Usage of Hidden APIs (Among the 3rd-party Miniapps). Based on the data presented in Table 6, we have discovered that the utilization of undocumented APIs is widespread among the 3rd-party miniapps, regardless of their category. The percentage of the 3rd-party miniapps employing undocumented APIs is 29.54%. Our observations have further revealed that the 3rd-party miniapps in the

Category	# U	# App	%
Business	8,116	14,887	54.52
E-learning	335	2,088	16.04
Education	2,738	40,410	6.78
Entertainment	1,286	5,258	24.46
Finance	262	1,408	18.61
Food	1,107	6,345	17.45
Games	1,777	4,745	37.45
Government	929	7,808	11.90
Health	795	6,422	12.38
Job	177	4,399	4.02
Lifestyle	11,846	35,371	33.49
Photo	136	1,981	6.87
Shopping	44,629	46,202	96.60
Social	217	5,694	3.81
Sports	312	3,378	9.24
Tool	3,423	72,301	4.73
Traffic	580	6,502	8.92
Travelling	309	2,160	14.31
Total	78,974	267,359	29.54

Table 6: The 3rd party WeChat miniapps that have used the undocumented APIs.

Shopping and Business categories are more inclined to use undocumented APIs, particularly those linked to sensitive operations like payment.

In addition, we conducted an analysis to comprehend the most popular undocumented APIs and the frequency of their usage by the 3rd-party miniapps. We categorized the APIs by name and tallied the number of miniapps that leveraged each API. We have found that 103 undocumented APIs provided by WeChat were utilized by their 3rd-party miniapps. Among these APIs, it is notable that 79 of them lack security checks. As shown in Table 7, we present a summary of undocumented APIs that have been utilized by over 50 mini-apps. It is evident that a majority of these hidden APIs lack proper security measures. To further understand the details, we delved into a selection of them to uncover why the 3rd-party mini-apps have knowledge of them and whether they are being exploited.

Our investigation has yielded some intriguing findings. (i) While some APIs are not publicly documented, Tencent does share them with certain vendors who work closely with them and permit these vendors to request access. An example of such an API is `requestFacetoFacePayment` [25] (which is used by 40,091 miniapps). (ii) There were some concealed APIs that were once freely available for use without any security checks. However, Tencent subsequently banned them. One such API is "`openUrl`" [22]. Interestingly, even though Tencent has banned the usage of this API, a whopping 17,140 miniapps have yet to remove the invocation of this API from their code (obviously, this will not work). This API has already been banned by Tencent prior to our report. (iii) There are still some APIs that remain usable until we notify Tencent of the issue. For example, `captureScreen` (12 miniapps used this API) can be utilized to obtain the user's sensitive information (See §7.2).

7 EXPLOITING UNCHECKED HIDDEN APIS

7.1 Quantifying the Security Risks

Methodology. After quantifying the number of unchecked undocumented APIs, our goal is to gain a better understanding of whether or not these APIs pose any security risks. While it is possible to manually analyze each API individually, it is not very practical or reliable, especially given the vast number of APIs we need to analyze (more than 1,500 APIs). However, our observation is that for an undocumented API to have potential security implications, it must be able to access sensitive information and resources on the Android system (e.g., location, files, and the internet). Therefore, if we find that the hidden API calls a native API, we can conclude that it has the potential to pose security risks. Otherwise, we can proceed to examine the implementation of each method within that hidden API, conducting the process recursively as needed.

API Name	Category	# App	% *App	w/ Check
wx.requestFacetoFacePayment	Payment	40,091	14.98	✓
wx.operateWXData	Misc	21,834	8.16	✗
wx.setPageOrientation	UI	18,499	6.91	✗
wx.enterContact	Contact	17,421	6.51	✓
wx.openUrl	Misc	17,140	6.41	✓
wx.preloadWebview	WebView	15,335	5.73	✓
wx.navigateBackNative	Navigate	13,407	5.01	✓
wx.editTextWithPopForm	Misc	13,390	5.00	✗
wx.openAddressWithLightMode	Address	13,390	5.00	✗
wx.requestPersonalPay	Payment	10,263	3.84	✗
wx.previewMedia	Media	6,635	2.48	✗
wx.drawCanvas	Canvas	6,055	2.26	✗
wx.openBusinessView	Misc	3,800	1.42	✗
wx.onDeviceOrientationChange	Device	1,626	0.61	✗
wx.startFacialRecognitionVerify	Bio-Auth	1,239	0.46	✓
wx.checkIsSupportFacialRecognition	Bio-Auth	669	0.25	✓
wx.notifyBLECharacteristicValueChanged	Bluetooth	603	0.23	✗
wx.getBackgroundFetchData	Misc	498	0.19	✗
wx.setBackgroundFetchToken	Misc	485	0.18	✗
wx.startFacialRecognitionVerifyAndUploadVideo	Bio-Auth	464	0.17	✓
wx.updateApp	Update	448	0.17	✗
wx.openOfflinePayView	UI	324	0.12	✓
wx.sendBizRedPacket	Payment	212	0.08	✓
wx.getVideoInfo	Video	193	0.07	✗
wx.compressVideo	Video	148	0.06	✗
wx.setBLEMTU	Bluetooth	127	0.05	✗
wx.getPhoneNumber	User Info	122	0.05	✗
wx.openVideoEditor	Video	118	0.04	✗
wx.chooseContact	Contact	100	0.04	✗
wx.openChannelsLive	Misc	97	0.04	✗
wx.openAddress	Address	96	0.04	✗
wx.setMenuStyle	Menu	74	0.03	✗

Table 7: The popular hidden APIs invoked by the 3rd-party WeChat miniapps.

However, not all invoked APIs manipulate sensitive resources within the Android system. For example, the `android.graphics` API offers graphics tools that allow developers to draw directly onto the screen. It is evident that invoking these APIs would not result in any security consequences. Therefore, we consider APIs that access resources protected by permissions (such as location, the Internet, and file system) to have security risks. Consequently, we opted to utilize a lightweight dynamic analysis approach to identify such APIs. Specifically, we hook all Android APIs that access sensitive resources, which are typically protected by Android permissions, and invoke unchecked undocumented APIs one by one. By monitoring whether the sensitive resource access APIs are invoked during this process, we can determine whether the undocumented APIs are implemented based on them. Furthermore, we are able to infer whether these APIs posed any security risks. While this approach may not uncover all the APIs since the execution of the hidden APIs may depend on the parameters and may not trigger the underlying security sensitive APIs, it can at least provide a lower-bound.

Results. We categorize the hidden APIs by analyzing the Android APIs that utilize the resources and grouping them accordingly. As shown in Table 8, we have identified 39 APIs (7.77%) in WeChat, 40 APIs (6.75%) in WeCom, 8 APIs (7.08%) in Baidu, 32 APIs (26.67%)

Resource	WeChat		WeCom		Baidu		Tiktok		QQ	
	# UUS	%	# UUS	%	# UUS	%	# UUS	%	# UUS	%
Bluetooth	3	0.59	3	0.51	-	-	-	-	-	-
Camera	1	0.20	1	0.17	-	-	-	-	1	0.34
Location	-	-	-	-	-	-	-	-	1	0.34
Media	5	0.96	5	0.84	-	-	11	9.17	11	3.73
NFC	3	0.59	3	0.51	-	-	-	-	-	-
Network	16	3.19	16	2.70	7	6.19	20	16.67	24	8.14
Package	3	0.59	4	0.67	1	0.88	-	-	1	0.34
Storage	25	4.98	26	4.38	3	2.65	2	1.67	8	2.71
Telephony	-	-	-	-	-	-	1	0.83	-	-
Total	39	7.77	40	6.75	8	7.08	32	26.67	38	12.88

Table 8: The sensitive resources that undocumented unchecked APIs accessed. UUS means undocumented unchecked sensitive APIs. Please note that a single hidden API may have access to multiple types of resources. Therefore, the total number of hidden APIs may not be equal to the sum of all the APIs that have been identified for each individual resource type.

in Tiktok and 38 APIs (12.88%) in QQ that invoke Android APIs that are protected by permissions. It should be noted that WeChat and WeCom have the most APIs that can access sensitive resources, while Baidu has the least number of such APIs. This is likely due to the fact that super apps require more Android permissions. To be more specific, WeChat requires 92 permissions, which is larger than that of Baidu (82). These accessed sensitive resources include camera, location, audio, and Internet. It is important to note that hidden APIs that access sensitive resources do not necessarily mean that they can access them without requiring permission. Specifically, in addition to the resources that are safeguarded by Android permissions, we are also including `SharedPreferences` in our checklist. This is because miniapps may utilize this Android API to store files in the space belonging to the super apps, which could potentially compromise the files of both the super apps and other apps.

Next, our objective is to understand the Android APIs utilized by the undocumented APIs. For this purpose, we count the number of Android APIs invoked by each hidden API of the super app, and classify them based on the names of the corresponding Android API Packages. We exclude the API packages that only be invoked once. It can be observed from Figure 8 that the API most commonly used is `SharedPreferences`. This is reasonable, as many of the APIs involve file operations. The available APIs consist of those dedicated to saving screenshots onto disks, which can be utilized to launch A3. Besides file access APIs, numerous hidden APIs make use of Internet access APIs for different purposes, including payment processing, network resource access, and more. The currently available APIs comprise those responsible for website access, which can be leveraged to trigger A1, APIs created for APK downloading and installation, which can be utilized to launch A2, and APIs for querying contact information, which can be employed to initiate A5. Please note that there are also APIs that access NFC, Camera, and Telephony Manager (which can be used to launch A4). However, since they have only been invoked once, we have excluded them.

7.2 Attack Case Studies

We present a few case studies to demonstrate how we can exploit those hidden unchecked (i.e., unprotected) APIs. For proof of concept, we present five case studies covering from arbitrary webpage access to information theft, as shown in Table 9.

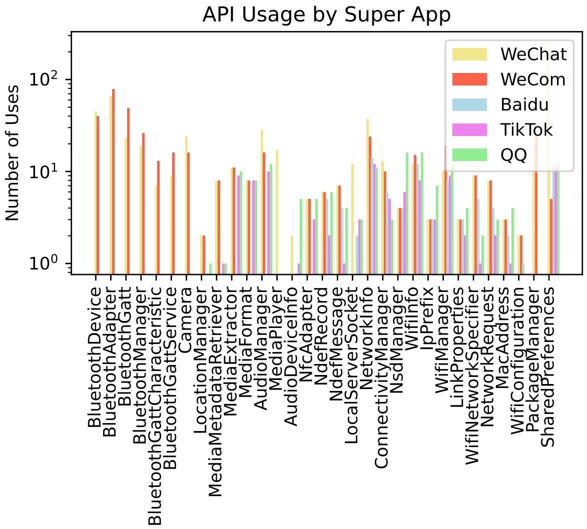


Figure 8: Android APIs used by the hidden APIs from different companies.

Attacks	Targeted Resources	Exploited APIs	Vulnerable Super Apps

A1	Web Resources	`private_openUrl`
		`openUrl`
		`postMessage`
			WeChat,WeCom
			QQ, Baidu

A2	Web Resources	`installDownloadTask`
		`addDownloadTaskStraight`
		`startDownloadAppTask`
		`installApp`
			WeChat,WeCom
			QQ

A3	User information	`captureScreen`	WeChat, WeCom
A4	User phonenumber	`getLocalPhoneNumber`	Tiktok
A5	User contacts	`searchContacts`	WeChat

Table 9: Summary of the attacks we tested

(A1) Arbitrary Web Page Access. We made a malicious miniapp that can open any webpage using the hidden API `private_openUrl`. Super apps usually have an allowlist of approved domains to prevent users from accessing untrusted sources (i.e., miniapps usually utilize the official API `wx.request` to access websites, and any network requests made through this API will be thoroughly vetted), but our malware can bypass these restrictions and navigate to any webpage without being vetted. This vulnerability allows our miniapp to open phishing websites and steal sensitive information, which is more powerful than previous phishing attacks [24]. We were successful in this attack on several super apps but could not test it on TikTok because it does not have the necessary APIs. This vulnerability is a significant security risk for super apps because they have a unique threat model that differs from web browsers. Super apps only allow access to specific domains, unlike web browsers that can access any website. This vulnerability has been confirmed as a high-severity vulnerability by Tencent.

(A2) Malware Download and Installation. We developed a malicious miniapp that can download and install malware using APIs `installDownloadTask` or `addDownloadTaskStraight`. Regular miniapps cannot download or install APK files on a mobile device because they have limited capabilities and can only download certain file types from specific servers. However, by using these APIs, a miniapp can download and install harmful APKs, which can cause significant damage to the user's mobile security and privacy.

This attack works on both WeChat and WeCom. Finally, although APKs cannot be installed without the user consent, miniApps is running inside the Super Apps, and as long as Super App has the installing permission (which most users will grant because they trust Super Apps), the malicious miniApp can install arbitrary APKs.

(A3) Screenshot-based Information Theft. We made a malicious miniapp that uses the captureScreen to secretly take screenshots and store them without the user's permission. This could be used by attackers to steal sensitive information like passwords and credit card numbers from the user's screen. The consequences of this kind of attack are serious. For example, the attacker could use them to steal the victim's identity and open fake accounts or make illegal purchases. They could also use the screenshots to commit financial fraud by stealing the victim's credit card.

(A4) Phone Number Theft. The malicious miniapps may use getLocalPhoneNumber to illicitly obtain the user's phone numbers. The hidden API is implemented by getLine1Number, which is a built-in feature of the Android SDK intended to provide the phone number associated with the SIM card currently inserted in the device. Nevertheless, access to phone number information from the SIM card may be blocked or restricted by some carriers or manufacturers, thereby rendering this attack unsuccessful in certain cases.

(A5) Contact Information Theft. A miniapp can potentially access sensitive information, such as friend list (including the usernames and WeChat ID) using searchContacts. Our experiments were conducted primarily in 2021, during which we found that this hidden API was still functional based on our raw results. Upon reporting the issue to WeChat, we were informed that another group had already reported the problem to them (CVE-2021-40180 [28]), and that the exploit no longer works on the new version of WeChat.

8 DISCUSSION

Limitations and Future Work. While effective, APIScope has room for improvement. It's prone to false positives and negatives, although none have surfaced during dynamic validation and manual checks. Although currently tested on Android, more work is needed to extend support to other platforms. However, our findings are generally applicable since miniapp codebases tend to be similar. Notably, APIScope is restricted to V8 engine-powered super-apps and isn't compatible with others like Alipay.

Our study uncovered vulnerabilities in hidden APIs, such as installDownloadTask and addDownloadTaskStraight, susceptible to SQL injection attacks. Exploiting these, attackers could manipulate download URLs in super app file tasks. Additionally, dumpHeapSnapshot and HeapProfiler APIs are flawed, misused by our miniapp to write to unauthorized files. Android's efforts to prevent this fall short, endangering essential files like chat histories, undermining security measures of super apps. Our experiment demonstrated overwriting EnMicroMsg.db, an WeChat chat history file. These vulnerabilities could lead to serious consequences, motivating us to create a tool identifying hidden API vulnerabilities (e.g., SQL injection, buffer overflow).

Ethics and Responsible Disclosure. Being an attack work by nature, we must carefully address the ethical concerns. To this end, we have followed the community practice when exploiting the vulnerabilities and demonstrated our attacks. First, for proof of concept, we developed quite a number of malicious miniapps and launched attacks against our own accounts and devices. We have never uploaded our malicious miniapps onto the markets to harm other users. Second, we have disclosed the vulnerabilities and our attacks against WeChat to Tencent in September 2021, and the other four super apps in November 2021. They have all acknowledged and confirmed our findings, and so far among them Tencent (the biggest super app vendor with 1.2 billion monthly users) has confirmed with 4 vulnerabilities, ranked 1 low, 2 medium, and 1 high, and awarded us with bug bounty and fixed them. TikTok has been patched too, but not Baidu at this time of writing.

9 RELATED WORK

Super Apps Security. More and more super apps have started to support the miniapp paradigm. Correspondingly, its security has received increasing attention. For instance, Lu et al. [24] identified multiple flaws in WeChat, and demonstrated how an attacker would be able to launch phishing attacks against mobile users and collect sensitive data from the host apps. Zhang et al. [37] developed a crawler, and understood the super apps by measuring the program practices of the provided miniapps, including how often the miniapp code will be obfuscated. Most recently, Zhang et al. [36] studied the identity confusion in WebView-based super apps, and identified that multiple super apps contain this vulnerability. A new attack named cross-miniapp request forgery (CMRF) [35] was also recently discovered, which exploits the missing checks of miniapp IDs for various attacks. Wang et al. introduced TaintMini, a method for tracing sensitive data flow in mini-programs using a data flow graph [30]. Through APIDiff, they also identified API variations in WeChat across platforms by generating test cases, revealing discrepancies in presence, permissions, and outcomes [31]. Zhang et al. [38] delved into the exploitation of cryptographic keys within miniapps. Baskaran et al. [12] studied how developers' unsafe habits (i.e., hardcoding keys) can lead to mini-app bypassing super-app authentication. Yang et al. [34] explore the super app paradigm, studying security measures, threats (13 security mechanisms, 10 threats), and trade-offs. It reveals violations due to system issues, isolation, and suggests improvements for security and privacy. Differently from those works, our study uncovers the undocumented APIs provided by the super apps and demonstrates how they can be exploited.

Undocumented API Detection and Exploitation. APIScope is the first system to detect and exploit undocumented APIs in mobile super apps like WeChat. Previous work has focused on detecting undocumented APIs in other platforms, such as Android and iOS, or on identifying missing security checks (e.g., [10, 14, 19, 23, 26]). For example, PScout analyzed undocumented APIs in Android [11], and Li et al. showed that there are 17 undocumented Android APIs that are widely accessed by the 3rd-party apps [20]. El-Rewini and Aafer studied access control vulnerabilities caused by residual APIs [16]. In addition, there are ways to invoke undocumented APIs in iOS [17, 32] and detect their abuses [13]. Yang et al. [33] proposed BridgeScope to identify sensitive JavaScript bridge APIs in hybrid apps. Undocumented APIs have also been found in the Java language and exploited by attackers [18]. APIScope builds on this prior work to specifically focus on mobile super-apps. Finding hidden APIs in super apps using traditional techniques is difficult due to the combination of web views, host native apps, and mini app execution environments, along with code scattering and obfuscation. Our new approach monitors parameter propagation to detect API usage, using robust signatures based on super classnames and public methods. We have also created a method for automatic test case generation and API classification.

10 CONCLUSION

In this paper, we have revealed that super apps often contain undocumented and unchecked APIs for their 1st-party mini-apps, which can grant elevated privileges such as APK downloading, arbitrary web view accessing, and sensitive information querying. Unfortunately, these undocumented APIs can be exploited by malicious 3rd-party mini-apps, as they lack security checks. To address this issue, we have designed and implemented APIScope, a tool that can statically identify these undocumented APIs and dynamically verify their exploitability. Through our testing on five popular super apps such as WeChat and TikTok, we have found that all of them contain these types of APIs. Our findings suggest that super app vendors must thoroughly examine and take caution with their privileged APIs to prevent them from being exploited.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful feedback. This research was supported in part by NSF award 2330264. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of NSF.

REFERENCES

[1] "6 powerful wechat statistics you need to know in 2022," https://brewinteractive.com/wechat-statistics/, (Accessed on 08/27/2023).

[2] "Google play store: number of apps 2022 | statista," https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, (Accessed on 08/27/2023).

[3] "Soot:a framework for analyzing and transforming java and android applications," http://soot-oss.github.io/soot/, (Accessed on 08/27/2023).

[4] "Tencent app," https://www.nbd.com.cn/articles/2022-12-01/2576229.html.

[5] "Tiktok - make your day," https://www.tiktok.com/, (Accessed on 08/27/2023).

[6] "Wechat mini programs showcases new capabilities to celebrate its third anniversary," https://www.tencent.com/en-us/articles/2200946.html.

[7] "What are wechat mini-programs? a simple introduction - walkthechat," https://walkthechat.com/wechat-mini-programs-simple-introduction/, (Accessed on 08/27/2023).

[8] "WeChat Chinese Documentation," https://developers.weixin.qq.com/miniprogram/en/dev/api/, 04 2022, (Accessed on 08/27/2023).

[9] "WeChat English Documentation," https://developers.weixin.qq.com/miniprogram/en/dev/api/, 04 2022, (Accessed on 08/27/2023).

[10] M. Alhanahnah, Q. Yan, H. Bagheri, H. Zhou, Y. Tsutano, W. Srisa-An, and X. Luo, "Dina: Detecting hidden android inter-app communication in dynamic loaded code," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 2782–2797, 2020.

[11] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 217–228.

[12] S. Baskaran, L. Zhao, M. Mannan, and A. Youssef, "Measuring the leakage and exploitability of authentication secrets in super-apps: The wechat case," in *26nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2023)*, 2023.

[13] Z. Deng, B. Saltaformaggio, X. Zhang, and D. Xu, "iris: Vetting private api abuse in ios applications," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 44–56.

[14] K. Drakonakis, S. Ioannidis, and J. Polakis, "The cookie hunter: Automated black-box auditing for web authentication and authorization flaws," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1953–1970.

[15] A. Druffel and K. Heid, "Davinci: Android app analysis beyond frida via dynamic system call instrumentation," in *International Conference on Applied Cryptography and Network Security*. Springer, 2020, pp. 473–489.

[16] Z. El-Rewini and Y. Aafer, "Dissecting residual apis in custom android roms," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1598–1611.

[17] J. Han, S. M. Kywe, Q. Yan, F. Bao, R. Deng, D. Gao, Y. Li, and J. Zhou, "Launching generic attacks on ios with approved third-party applications," in *International Conference on Applied Cryptography and Network Security*. Springer, 2013, pp. 272–289.

[18] S. Huang, J. Guo, S. Li, X. Li, Y. Qi, K. Chow, and J. Huang, "Safecheck: safety enhancement of java unsafe api," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 889–899.

[19] S. M. Kywe, Y. Li, K. Petal, and M. Grace, "Attacking android smartphone systems without permissions," in *2016 14th Annual Conference on Privacy, Security and Trust (PST)*. IEEE, 2016.

[20] L. Li, T. F. Bissyandé, Y. Le Traon, and J. Klein, "Accessing inaccessible android apis: An empirical study," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 411–422.

[21] S. Liang, *The Java native interface: programmer's guide and specification*. Addison-Wesley Professional, 1999.

[22] Listen, "How to use "openUrl"?" https://developers.weixin.qq.com/community/develop/article/doc/00000efea1c4785424fc1dd4e51c13.

[23] B. Livshits and J. Jung, "Automatic mediation of {Privacy-Sensitive} resource access in smartphone applications," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 113–130.

[24] H. Lu, L. Xing, Y. Xiao, Y. Zhang, X. Liao, X. Wang, and X. Wang, "Demystifying resource management risks in emerging mobile app-in-app ecosystems," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 569–585.

[25] MayBG, "How to use "requestFacetoFacePayment"?" https://developers.weixin.qq.com/community/develop/doc/000cce1ebd80006b1e8f5185b56800.

[26] X. Pan, X. Wang, Y. Duan, X. Wang, and H. Yin, "Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps." in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2017.

[27] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for javascript," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 488–498.

[28] vuldb, "Cve-2021-40180," https://vuldb.com/?id.205138.

[29] W3C, "Miniapp standardization white paper," https://w3c.github.io/miniapp/white-paper/, 2020.

[30] C. Wang, R. Ko, Y. Zhang, Y. Yang, and Z. Lin, "Taintmini: Detecting flow of sensitive data in mini-programs with static taint analysis," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023.

[31] C. Wang, Y. Zhang, and Z. Lin, "One size does not fit all: Uncovering and exploiting cross platform discrepant apis in wechat," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[32] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee, "Jekyll on ios: When benign apps become evil," in *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, 2013, pp. 559–572.

[33] G. Yang, A. Mendoza, J. Zhang, and G. Gu, "Precisely and scalably vetting javascript bridge in android hybrid apps," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2017, pp. 143–166.

[34] Y. Yang, C. Wang, Y. Zhang, and Z. Lin, "Sok: Decoding the super app enigma: The security mechanisms, threats, and trade-offs in os-alike apps," *arXiv preprint arXiv:2306.07495*, 2023.

[35] Y. Yang, Y. Zhang, and Z. Lin, "Cross miniapp request forgery: Root causes, attacks, and vulnerability detection," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 3079–3092.

[36] L. Zhang, Z. Zhang, A. Liu, Y. Cao, X. Zhang, Y. Chen, Y. Zhang, G. Yang, and M. Yang, "Identity confusion in webview-based mobile app-in-app ecosystems," in *31st {USENIX} Security Symposium ({USENIX} Security 22)*, 2022.

[37] Y. Zhang, B. Turkistani, A. Y. Yang, C. Zuo, and Z. Lin, "A measurement study of wechat mini-apps," in *Abstract Proceedings of the 2021 ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, 2021.

[38] Y. Zhang, Y. Yang, and Z. Lin, "Don't leak your keys: Understanding, measuring, and exploiting the appsecret leaks in mini-programs." in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023.