One Size Does Not Fit All: Uncovering and Exploiting Cross Platform Discrepant APIs in WeChat

Chao Wang, Yue Zhang, and Zhiqiang Lin, *The Ohio State University*

https://www.usenix.org/conference/usenixsecurity23/presentation/wang-chao

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

One Size Does Not Fit All: Uncovering and Exploiting Cross Platform Discrepant APIs in WeChat

Chao Wang
The Ohio State University
wang.15147@osu.edu

Yue Zhang
The Ohio State University
zhang.12047@osu.edu

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

Abstract

The past few years have witnessed a boom of mobile super apps, which are the apps offering multiple services such as e-commerce, e-learning, and e-government via miniapps executed inside. While originally designed for mobile platforms, super apps such as WeChat have also been made available on desktop platforms such as Windows. However, when running on desktop platforms, WeChat experiences differences in some behaviors, which presents opportunities for attacks (e.g., platform fingerprinting attacks). This paper thus aims to systematically identify the potential discrepancies in the APIs of WeChat across platforms and demonstrate how these differences can be exploited by remote attackers or local malicious miniapps. To this end, we present APIDIFF, an automatic tool that generates test cases for each API and identifies execution discrepancies. With APIDIFF, we have identified three sets of discrepant APIs that exhibit existence (109), permission (17), and output (22) discrepancies across platforms and devices, and provided concrete examples of their exploitation. We have responsibly disclosed these vulnerabilities to *Tencent* and received bug bounties for our findings. These vulnerabilities were ranked as high-severity and some have already been patched.

1 Introduction

A super app is an app that allows its users to access multiple services such as online shopping, ride-hailing, and instant messaging, from a single app. Today, there are many popular mobile super apps, including China's WeChat, TikTok, and AliPay, India's Paytm, Singapore's Grab, Indonesia's GoTo, Vietnam's Zalo, and South Korea's Kakao [8]. Increasingly, Whatsapp in India and Brazil is also becoming super apps [46]. Among these mobile super apps, WeChat has most of the users (e.g., 1.2 billion monthly users [29]), and it also offers almost all the services from such as booking a doctor's appointment to even filing for a divorce [4].

Certainly, it is impossible for a single super-app company to provide all the daily services. Therefore, super apps such as WeChat and also WeCom (the enterprise version of WeChat) have provided APIs for 3rd-party developers to develop the apps running inside the super apps. This is often called the miniapp or app-in-the-app paradigm [39]. Today, it is estimated that there are more than 4.3 million miniapps [26] running in WeChat. Through offering the miniapps, it creates a win-win situation for both super app providers and 3rd-party developers in that more services are provided (leading to more users using the platform), and a single miniapp can be executed atop multiple platforms (e.g., both Android and iOS) using a single programming language (e.g., JavaScript) rather than different languages such as Objective-C, Java, and C/C++ for different platforms.

Super apps were originally designed for mobile platforms such as Android and iOS, but we have seen an increasing number of them, such as WeChat, WeCom, and Alipay, supporting desktop platforms such as Windows. Among these, WeChat is more aggressive and allows all of their miniapps to be executed on desktops as well. However, when executed on these platforms, the miniapps may exhibit different behaviors due to differences in the implementation of the APIs, such as differences in platform-specific protections. For instance, WeChat running on Windows allows miniapps to access the microphone (via API `wx.record`) without the user's consent, leaving the door open for malware to stealthily record audio and compromising the user's privacy. Therefore, it is imperative to systematically identify all potential platform-discrepant APIs in WeChat (as it is currently the only platform that supports miniapps on desktop platforms) and provide concrete demonstrations of how they can be exploited.

To this end, this paper presents APIDIFF, an automatic tool that identifies discrepancies across different platforms and devices. A key challenge lies in automatically generating the API test cases, which APIDIFF solves through a domain-guided brute-force approach. Particularly, using APIDIFF, we have discovered three categories of discrepancies in WeChat: (*i*) API existence, where the API is present on some platforms but not others; (*ii*) API permission, where the API requires certain permissions on some platforms but not others; and (*iii*) API output, where the same API produces different results on different platforms even with the same inputs.

With these uncovered discrepant APIs, we then develop concrete attacks. Since there are three different categories of discrepancies, we correspondingly demonstrate three different attacks. In particular, we demonstrate: (*i*) Attacks caused by API existence discrepancies, where some platforms lack security-focused APIs, allowing attackers to exploit such absence for malicious purposes. For example, we notice that Android platforms use stronger protections for some devices such as Bluetooth when compared with Windows and iOS, where an Android device enforces authentication with the protections from APIs but the other platforms do not, leading to Man-in-the-Middle (MitM) attacks. (*ii*) Attacks caused by API access control discrepancies, where some platforms lack the necessary checks for APIs when accessing sensitive resources (e.g., locations, cameras, and audio recorders), leading to potential privacy breaches by malicious miniapps. (*iii*) Attacks caused by API output discrepancies, where APIs in some platforms expose fingerprinting vulnerabilities, leading to attacks similar to browser fingerprinting attacks [45, 34, 38]. In such scenarios, a malicious miniapp can leverage the signatures generated from specific API outputs to uniquely identify a device.

Contributions. We make the following contributions:

- **Novel Findings (§3).** We are the first to discover that super apps running on top of different platforms and devices may have a variety of discrepancies during execution, some of which can lead to security vulnerabilities.
- **Empirical Evaluation (§4 and §5).** We have developed an open source, automatic tool APIDIFF[1] to systematically uncover discrepant APIs across different platforms and devices. We have tested our tool with WeChat on three platforms: Android, iOS, and Windows, and identified 109 APIs with existence discrepancies, 17 APIs with permission discrepancies, and 22 APIs with output discrepancies across platforms.
- **New Attacks (§6).** We develop three categories of attacks by exploiting these discrepancies. We show that these attacks can cause severe security and privacy consequences to massive super apps users as well as miniapp developers.

2 Background

2.1 The Evolution of Mobile Super Apps

During the desktop era, web apps were the dominant type of applications, easily accessible from any platform by typing their URLs in a browser. However, the open nature of the web also led to a proliferation of malicious web apps, such as phishing [19]. In the mobile era, users switched to downloading vetted mobile apps from app stores such as the Google Play Store and Apple App Store. While these mobile apps underwent centralized vetting, they also had limitations such as the limited storage and the need for downloads and installations. In addition, developers often had to create separate versions of their apps for different operating systems.

[1]The source code of APIDIFF is available at `github.com/OSUSecLab/APIDiff`.

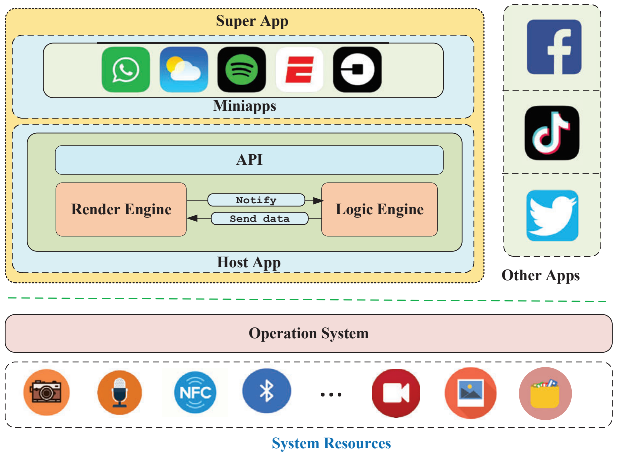


Figure 1: A typical architecture of mobile super apps

Having recognized the benefits and limitations of both web and mobile apps, some social network apps (such as WeChat [26] and TikTok [33]) began to expand their services and eventually became "super apps". Some of these super apps, such as WeChat, even started to provide APIs in 2017 for third-party developers to create miniapps that run within the super app. These miniapps offered a more enhanced user experience and increased user engagement with the super app, by relying on web technologies, such as JavaScript, and integrating the capabilities of native apps and offering the advantages of both types of apps, such as cross-platform support and the ability to run without installation. It is important to note that while the miniapps run within the super app, the super app itself is also considered a native app. Although Miniapps have evolved from web apps and native mobile apps, they are different from both.

Table 1 lists popular super apps from around the world, ranked by monthly active users. It can be seen that these super apps belong to different categories and offer a wide range of services, from business to social networking. Although most support multiple platforms and allow the execution of mini-apps, only WeChat and also WeCom support the execution of mini-apps on desktop computers.

2.2 The Architecture of Super Apps

A high-level overview of the architecture of super apps with miniapp support is presented in Figure 1. We can see that the super app hosts all the miniapps (which can be directly fetched from the miniapp market that is hosted in the back-end of the super app), and provides the miniapps running environment in a sandbox environment. To offer native-app alike experiences, the super app also provides a

Super App	Category	Monthly Users	Country	Services										Platform			Miniapp		
				Business	Education	Communication	Finance	Food Delivery	Games	Lifestyle	Ride-hailing	Shopping	Social	Android	iOS	Windows	Android	iOS	Windows
WeChat [29]	Social	1,200 million +	China	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Tiktok [20]	Social	1,000 million +	China	✗	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗
Alipay [51]	Finance	730 million +	China	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗
Snapchat [50]	Social	347 million +	U.S.	✗	✗	✓	✗	✗	✓	✓	✗	✗	✓	✓	✓	✗	✓	✓	✗
WeCom [31]	Business	180 million +	China	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Paytm [51]	Finance	150 million +	India	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓	✓	✗
Go-Jek [27]	Finance	100 million +	Indonesia	✓	✗	✗	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✗
Zalo [63]	Social	52 million +	Vietnam	✓	✗	✓	✗	✗	✓	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗
Kakao [51]	Social	45 million +	South Korea	✗	✗	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✗	✗	✗
Grab [51]	Delivery	25 million +	Singapore	✗	✗	✗	✓	✓	✗	✓	✓	✓	✗	✓	✓	✗	✓	✓	✗

Table 1: Comparison with popular super apps. "Platform" represents the super apps can run atop the specific platform, and "miniapp" represents the super app have miniapps enabled.

set of in-app APIs for miniapps to access various system resources (e.g., Bluetooth, NFC, microphone, and camera). While most APIs are similar across platforms, some differences exist, such as mobile platforms supporting multiple sensors but not desktop Windows lack of such support. For example, NFC APIs are only supported on Android, and APIs for the gyroscope, compass, and accelerometer are only available on mobile platforms. Also, similar to web apps, to enable the cross-platform capabilities, the miniapps are typically programmed in both script (e.g., JavaScript [35]) and markup languages (e.g., WXML [32], which is an HTML-like language defined by *Tencent* for UI design in WeChat), and they rely on the logic engine of the super app to execute the script and the rendering engine to display the webview components.

Mini-apps can access both system resources (e.g., Bluetooth, NFC) and user data (e.g., phone number, billing address). To access these resources, mini-apps need permission from users on mobile platforms. Users receive a permission request dialog box that explains why the mini-app needs access to a specific resource, and users can grant or deny permission. Once permission is granted, it is recorded for future use, so users don't have to grant access again. Please note that in this permission model [7], some permissions are inherited from the Android system, such as the Bluetooth permission, while others are not, like the `scope.werun` permission, which is required when miniapps attempt to obtain `WeRun` data (the user's daily steps) from WeChat. When a user grants permission for WeChat to access a specific type of resource, it does not mean that WeChat can automatically grant all access to mini-apps. Therefore, WeChat will prompt the user again to confirm whether they would like to grant data access to a specific mini-app. For instance, if the user grants WeChat access to Bluetooth resources when a mini-app wants to access Bluetooth through WeChat, WeChat will ask the user a second time to ensure that the user wants to give Bluetooth access to that specific miniapp.

2.3 The Comparison

Regarding the differences between different apps and their corresponding host platforms, Table 2 provides a summary. We can notice that there could be six differences when comparing miniapps with traditional native apps and also web apps. First, similar to web apps, miniapps are installation-free, and users do not need to install them. Instead, the users can directly access those miniapps from the super apps' markets. Second, unlike native apps, miniapps cannot be automatically launched. Instead, users must click on the icon of a miniapp to launch it. The super app will keep track of the user's preferences, including their favorite miniapps. These preferences will be synced across all devices owned by the user, although the preferred miniapps will not be automatically downloaded on all devices. Third, although most miniapps running on the mobile version of the super app cannot run in the background for longer than 5 minutes [53], some miniapps can by requesting special approval from super app vendors. It is important to note that this restriction only applies to mobile super apps, and miniapps running on the desktop version of the super app can run in the background without any time limit. Fourth, similar to mobile apps, miniapps do not have mandatory back-ends. Fifth, super apps create the runtime environment for the miniapps, allowing the developer only to implement one miniapp, and run it everywhere (e.g., in both Android and iOS). Finally, before the miniapps open to the public, they must be vetted by the super apps providers to protect end users from using any malicious miniapp in the market.

3 Overview

Key Observations and Goals. Miniapps that run on various platforms may encounter different behaviors due to platform differences such as permissions and threat models. The ob-

Hosts & Supported Apps	Mobile OS (Native Apps)	Web Browsers (Web Apps)	Super Apps (Miniapps)
End Users			
Install-free?	✗	✓	✓
Auto-Started?	✓	✗	✗
Running at Background?	✓	✗	✓
Developers			
Mandatory Backend?	✗	✓	✗
Cross-platforms?	✗	✓	✓
Centralized Vetting?	✓	✗	✓

Table 2: Comparison between different hosts and their supported apps

jective of this study is to systematically identify these discrepancies at the API level for miniapps and examine their security implications. *This is because the knowledge gained from testing a single platform does not necessarily apply to other platforms.* At a high level, there are at least three categories of API discrepancies that could result in security issues. In the following, we dive into the details of the observed root causes and the objectives we aim to accomplish:

- **API Existence Discrepancies.** *The root cause is the inconsistency in the implementation of security-focused APIs across platforms.* In particular, some APIs are only available on certain platforms, such as most sensor APIs (e.g., accelerometer, compass, and gyroscope) which are only supported on mobile platforms such as Android and iOS. While most of those API differences may just cause compatibility issues, some can lead to security attacks. An example of such API is `wx.makeBluetoothPair`, which initiates authentication for Bluetooth devices. Its absence on platforms such as iOS makes the Bluetooth device unable to differentiate between trusted and untrusted devices, leading to Man-in-the-Middle attacks.
- **API Permission Discrepancies.** *The fundamental reason behind this issue is that certain APIs are created to enable access to sensitive resources, but the safeguards to protect sensitive resources are not applied consistently across platforms.* Specifically, miniapps often need to request user authorization to access sensitive resources such as audio, video, and contacts. However, we observed that the Windows platform (note that WeChat for Windows is not downloaded from the Microsoft Store because the store version lacks support for miniapps) does not have any permission protection when it comes to accessing these resources. This means that miniapps on the Windows platform can potentially access sensitive resources without requiring any user consent, which could put user privacy and security at risk.
- **API Output Discrepancies.** *The root cause of this issue is the varying amounts of device information and configuration exposed through super app APIs across different platforms, leading to the possibility of fingerprinting attacks.* Specifically, it is well known that attackers can use the browser or mobile APIs to identify specific users for targeted advertising [45, 34, 38]. Similarly, WeChat also has APIs that are vulnerable to fingerprinting attacks. We refer to these types of APIs as "fingerprintable" in the rest of this paper. Interestingly, we notice that for a specific API, it can be fingerprintable in some platforms, but not others. Fingerprintable APIs may also vary in their accuracy across platforms. For example, the `getScreenBrightness` API returns varying levels of output precision on different platforms. The Windows version provides precision to two decimal places, whereas the Android version offers 16 decimal places, allowing for more accurate differentiation and fingerprinting of users and devices on the Android platform.

Scope. While we could study all the super apps available on the market (as listed in Table 1), we focus exclusively on WeChat for four reasons. First, WeChat has the largest number of users (with 1.2 billion monthly active users), and any security vulnerability in this super app can have a striking impact. Second, WeChat pioneered the concept of miniapp paradigm, and so far it has more than 4.3 million miniapps, which is way more than any other platform (e.g., Alipay has about 1 million miniapps [55], and Snapchat has only 62 miniapps). Third, WeChat and WeCom are the only two super apps that support the execution of miniapps on three platforms, namely Windows, Android, and iOS. While miniapps can run on macOS by clicking on a download link, there is no public portal for users to search and access them, unlike on iOS, Android, and Windows. We believe that the miniapp environment on macOS is still in development. Meanwhile, we conducted tests on macOS and found that the results were the same to those on iOS. We attribute this to the adoption of the same sandboxing techniques and runtime on both platforms (iOS and macOS). Therefore, we have excluded macOS from our analysis to avoid duplication and only report our findings on iOS, Android, and Windows. Fourth, while we can also analyze WeCom, we find that it uses the same framework to manage and execute miniapps, sharing much of the same code base, as confirmed by both our binary code inspection and Tencent's official documents [30], which mention that miniapp developers do not even have to change their code at all to make their miniapps compatible with WeCom. Therefore, we just need to focus on WeChat for proof of concept.

Assumptions. Our goal is to find the miniapp APIs that can be exploited by malicious miniapps due to the discrepant behaviors. As such, we make a few assumptions for the malicious miniapps. First, we assume that the attacker is able to distribute the malicious miniapps onto the user's mobile, as in the study by Lu *et al.* [39]. This is also reasonable, since miniapps are install-less, and to launch a miniapp is just one-

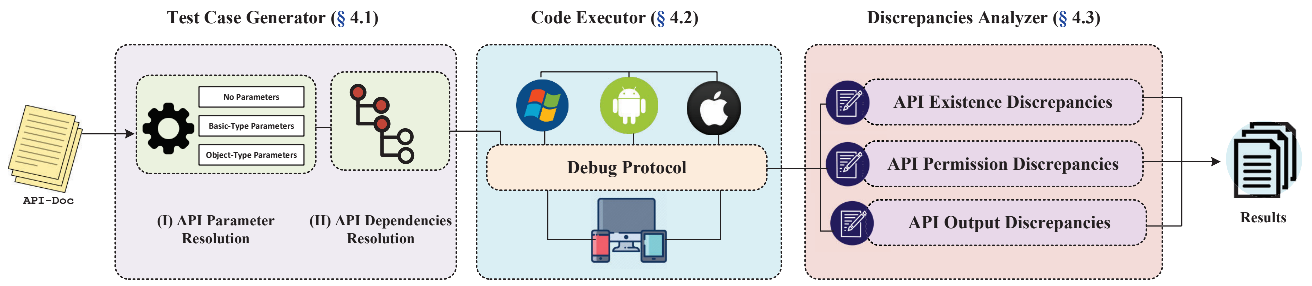

Test Case Generator (§ 4.1)
Code Executor (§ 4.2)
Discrepancies Analyzer (§ 4.3)
No Parameters
Basic-Type Parameters
Object-Type Parameters
API-Doc
(I) API Parameter Resolution
(II) API Dependencies Resolution
Debug Protocol
API Existence Discrepancies
API Permission Discrepancies
API Output Discrepancies
Results

Figure 2: The Architecture of APIDIFF

click away from the users. Second, we assume the trustworthiness of the code from the OS and the super apps, since we do not believe that they have any malicious intentions (otherwise, they can trivially launch any layer below attacks to attack the miniapps). Finally, we also assume the trustworthiness of the super apps' backends or miniapps' backends, since they are typically out of reach of the attackers.

4 APIDIFF Design

This section presents the detailed design of APIDIFF. As illustrated in Figure 2, APIDIFF consists of three key components.

- **Test Case Generator (§4.1).** For each API, APIDIFF needs to create a test case, with the corresponding parameters properly initialized. If an API's execution depends on other APIs, APIDIFF also needs to resolve the dependencies with the right order of executing them. Additionally, to have high coverage of API execution, it also needs to mutate the values of the parameters when necessary.
- **Code Executor (§4.2).** To exercise the discrepancies across platforms (i.e., Windows, Android, and iOS), APIDIFF must execute the test cases on these platforms, to produce the corresponding outputs, from which to identify discrepancies that may have security concerns.
- **Discrepancies Analyzer (§4.3).** To identify the discrepant APIs, APIDIFF uses differential analysis with a set of predefined policies to inspect the error code and return values, which are the ones observed by miniapps, of the tested APIs.

Note that manual efforts were only required at the beginning to investigate the workflow, including investigating the possible error codes that may be observed for each API. After the tool is built, no further manual analysis is required. This is because we use function `evaluate` to execute the tool-generated JavaScript testing code for all the APIs and collect the outputs. Discrepancies are then identified based on the returned error code and outputs. With the discovered API discrepancies, we then demonstrate how to exploit them for malicious purposes. Since the attacks will highly depend on the semantics of the APIs, we will not be able to construct the attacks automatically. Instead, we develop a few case studies to demonstrate the consequences of the discrepancies and the impact of the corresponding attacks (§6).

4.1 Test Case Generator

The goal of our test case generator is to produce a code snippet that contains the API to be tested. However, this is challenging since we need to feed the testing API with the right parameters to trigger the discrepant behaviors. That is, we need to properly understand the parameters and initialize with the desired seed values. Meanwhile, since some APIs may have dependencies on the execution of other APIs, we have to properly resolve the order the API executions.

Generating values for API parameters. Intuitively, we would have to go through all the documentation to understand the semantics for each API in order to provide the right values for its parameters, which is tedious and time consuming. Interestingly, we notice that the documentation contains well-formatted descriptions for each parameter and its type. Therefore, we propose first extracting the parameter type from the documentation and then initializing them based on the domain knowledge.

The rationale for this approach is that our objective is to initiate API execution to identify discrepant APIs based on their input and output. Therefore, initializing APIs with seed values and mutating them can allow us to uncover the three types of discrepant APIs we aim to identify, without exploring all their branches. For instance, to determine whether an API exists or not, an error message is thrown regardless of its parameters. Second, permission checks are often performed before executing an API, and we can thus easily detect permission discrepancies without providing the correct values. Finally, fingerprintable APIs are mostly used to obtain hardware information, as long as we have initialized the environment correctly.

Type	Value	Example
Any	`null`	`console.log(<any>)`
Array	Recursively Resolved	`wx.foo([1,2,3,4])`
Boolean	`true`	`fs.statSync("", true)`
Function	`(e) => {e}`	`success: (e) => {e}`
Null	`null`	N/A
Number	`1`	`fs.writeSync(1)`
Object	Recursively Resolved	`{fail: (e) => {e}}`
String	`test`	`wx.getStorageSync("test")`
Undefined	`undefined`	N/A

Table 3: Values Used for Basic Type Parameters

- **APIs without Parameters.** In total, there are 231 APIs that do not accept any parameters. They are often used to obtain the configurations of the execution environment. For example, API `wx.getSystemSetting()` returns the system settings of the devices (e.g., whether the device has Bluetooth or Wi-Fi enabled). These APIs can be easily recognized in the documentation.
- **APIs with Basic Type Parameters.** Miniapps are developed using JavaScript. Consequently, the parameters of miniapp APIs use a variety of basic types such as `String`, `Boolean`, `Number`, and `Array` defined in standard JavaScript, to pass information for the execution. For example, `wx.getStorageSync(string key)` fetches data indexed by `key`, and `wx.removeStorageSync(string key)` deletes the corresponding data. In total, there are 107 APIs that accept only the basic type parameters. We initialize these parameters with the default seed values (See Table 3).
- **APIs with Composite Type Parameters.** The majority of the APIs (in total 693) take parameters in composite types. For example, `getLocation(Object)`, which returns the user's actual location, accepts an object as its parameter, and this object has multiple fields, and their types are just basic types (e.g., `String`, `Boolean`) or a callback function of either `success`, `fail`, or `complete`. Fortunately, since we already have the pre-cooked values for the basic types, we just initialize these basic types with the corresponding seed values.
- **APIs with a Callback Function Parameter**. Based on our analysis of the documentation, we found that in total there are three types of callback functions, which are needed by 364 APIs. These three callbacks are `success` (which specifies the follow-up actions when the API is invoked successfully), `fail` (which specifies the follow-up actions when the API invocation is failed) and `complete` (which specifies the follow-up actions regardless of the success or failure of the API). We just created an implementation of these three callbacks by just displaying a message showing they are called.

We now discuss the mutation policy we employed. As depicted in Table 3, there are various types of parameters used in our experiment, some of which are limited and

```
const uploadTask = wx.uploadFile({
  url: 'http://example.weixin.qq.com/upload',
  filePath: tempFilePaths[0],
  name: 'file',
  formData:{     'user': 'test'   },
  success (res){
    const data = res.data
    //do something
  }})
uploadTask.onProgressUpdate((res) => {
  console.log('uploaded:', res.progress)})
uploadTask.abort()
```

Figure 3: Code Example for API `uploadFile` in WeChat **Official Document**

enumerable, such as Boolean parameters. We enumerated these parameters to collect a wider range of outputs. For instance, consider the method `getLocation(type, isHighAccuracy, highAccuracyExpireTime)`. The second parameter, `isHighAccuracy`, is a Boolean value that can take on the values of either true or false. In our experiment, we tested this parameter with both true and false values. Similarly, some parameters can only take on certain string values. For example, the first parameter of `getLocation` is a string that can have the value of either "wg84" (indicating that the API will return GPS location) or "gcj02" (indicating that the API will use cellular or other wireless networks to retrieve location data). These variables are also enumerable, and we obtained the possible candidate values (such as wg84 and gcj02) by analyzing the documentation.

However, the third parameter of `getLocation`, `highAccuracyExpireTime`, is a numerical value that specifies a time window in which the API should return the output. This parameter is not enumerable, so we mutated the numerical values to collect more outputs. To mutate numerical values, we used a range of numbers from -1 to 100, and for non-enumerable strings (The string list we used is a set of built-in payloads in Burp Suite, which can be found in [2]), we utilized a list of strings. Obviously, we were unable to enumerate all possible numerical values and strings. This is our limitation. Notwithstanding this, our confidence in the dependability of our findings persists, since our objective was to pinpoint inconsistencies in existence, permission, and output, which can typically be uncovered irrespective of code coverage. For instance, `highAccuracyExpireTime` specifies the allowed time for the API to return the output. As long as we can obtain the output from the API, the specific value of `highAccuracyExpireTime` becomes irrelevant.

Resolving the execution order of dependent APIs. It is worth noting that many APIs actually have dependencies, meaning that some APIs must be executed first (we denote them dominating APIs), before the execution of others (we denote them dominated APIs). For example, in Figure 3, `onProgressUpdate` requires `uploadFile` to be execute first. As such, we need to resolve the API dependencies. An intuitive approach is to use regular expression with patterns such as `start*` (e.g., `startHEC`), `get*` (e.g., `getNFCAdapter`), `create*` (e.g., `createUDPSocket`) to identify the dominating APIs. However, there could be multiple dominating APIs that do not start with these patterns, as shown in Case II and III in Table 4), where the execution of the last API depends on all of its predecessors. Then, another intuitive approach would be to exhaustively enumerate all possible combinations. However, given that we have 1,031 APIs in total, theoretically this brute-force approach would require `factorial`(1031) combinations (we need to try all of their permutations since there could be more than three or four layer of dependencies ahead of the dominated API, as shown in Case II and III). Certainly, we have to optimize this approach further.

To effectively identify and order the APIs, we opted for a category-guided brute-force approach. A key observation is that the dominating APIs are usually those APIs that initialize the hardware and environments for that specific category. Examples of those dominating APIs include `startHCE`, which initializes NFC, `wx.getFileSystemManager`, which initializes the file system manager, and `wx.openBluetoothAdapter`, which initializes the Bluetooth devices. As such, if an API is the dominating API for a specific dominated API, it will likely be the dominating API for those in the same category. For example, `startHCE` is the dominating API for `sendHCEMessage`, it is also the dominating API for `wx.onHCEMessage` and `wx.getHCEState`. As such, we determined that we could effectively reduce the number of possible combinations by categorizing the APIs and enumerating them in order within each category. Taking the `NFC` category as an example, there are only 6 APIs in total, and we only need to try `factorial`(6) (i.e., 6!) at most to exercise them in the right order. Eventually, we decide to use this category guided brute-force approach to generate the desired execution orders.

4.2 Code Executor

Our APIDIFF takes the code fragments (which essentially is a program) that contains the API to be tested, and executes the programs in order to collect the results. To this end, an intuitive approach is to compile the miniapp that contains the API for testing and then execute the produced miniapps to observe the outputs. However, since this approach requires developers to produce multiple miniapps when testing multiple APIs, it is time consuming. Another possible approach is to use dynamic code execution to load and

Case #	API Sequence
I	**`wx.startHEC`** `wx.sendHECMessage`
II	**`wx.createBLEPeripheralServer`** **`BLEPeripheralServer.addService`** `BLEPeripheralServer.removeService`
III	**`wx.createInterstitialAd`** **`InterstitialAd.load`** **`InterstitialAd.onLoad`** **`InterstitialAd.show`** `InterstitialAd.destory`
IV	**`wx.createUDPSocket`** `UDPSocket.connect`

Table 4: Examples of API Sequence (The bold font are the dominating APIs)

execute the code dynamically. Unfortunately, WeChat has disabled this feature due to security concerns [53].

Our approach, however, does not rely on compile-and-then-execute process or JavaScript dynamic code execution. Instead, we notice that the development tool can directly update the code of the miniapp when running on top of the testing platforms, and as such, we reverse engineered and customized the debug protocol of WeChat to allow the API to execute directly on the targeted platform. Specifically, we found that whenever the miniapps execute the code through the debug protocol, they pass their JavaScript to be executed to the IDE, which then invokes an internal function called `evaluate` to ultimately execute the JavaScript code and return the output. Therefore, we simply feed the code into `evaluate` directly and log the output if there is any, including error codes, for post-mortem analysis.

Next, we need to run the test cases on multiple platforms (e.g., Android, iOS, and Windows). A straightforward approach is to test each API on each platform sequentially, but this would be time-consuming. Instead, we designed a simple client-server (CS) mode in which a single server maintains a task queue containing the APIs to be tested, and collects and records the outputs. Multiple clients, each of which executes the code on a specific testing platform, fetch the tasks from the queue and submit the outputs to the server. This approach allows the testing code to be deployed on Android, iOS, and Windows simultaneously, enabling the test results of the APIs to be collected in parallel. To collect results for both cross-platform and cross-device testing, we run the test cases on six devices: two Windows, two Android, and two iOS. This way, we have results for different platforms and devices for the same platform.

Having prepared for the test cases and the devices, we also have another important problem to solve, namely how to configure the permissions for each testing API to detect the missing permissions in the testing platform. An intuitive approach is not to assign any permission for the testing API, and then wait for permission request dialogue to determine whether a particular request requires certain permission. However, this approach would require parsing the dialog window, understanding the output, and meanwhile clicking the confirmation button on the screen. Also, note that if there is no clicking within 5 minutes window, the testing miniapp will abort and an error message will be logged. If the permission is configured, then there will be no runtime permission request, and the miniapp will be executed without any interruption. Therefore, based on these observations, we propose a simpler approach of directly running the testing miniapps (when testing the specific API) with two sets of permission configurations without parsing and clicking any permission request window: one configuration has all permission enabled (the miniapp will be executed quietly), and the other has all permission disabled (an error message will be logged if it requires a specific permission). By doing so, we just parse the logged error message to determine whether a testing API requires certain permissions.

4.3 Discrepancies Analyzer

With the collected testing results for each API across different platforms and devices, we then identify the discrepant APIs that exhibit the three types of discrepancies we aim to identify. In the following, we explain the specific policies we used to detect these APIs.

(I) API Existence Discrepancies. Discrepancies of this kind can be caused by the missing implementations of the APIs on the corresponding platforms. To determine whether an API has been implemented, we then simply inspect the logged messages on the tested platform: if error code "`not supported`" is observed for this particular API, then this API is classified into non-existing API for this platform.

An API with existence discrepancies is identified if the API on one platform is executed successfully, and the API on another one throws errors "`not supported`".

(II) API Permission Discrepancies. Discrepancies of this kind are caused by the missing permission protections on the testing platform. To determine whether an API has been implemented on a specific platform, we again inspect the logged messages on the tested platform: if error code "`permission errors`" is observed for this particular API, then this API requires permission. If it does not require permission in other platform(s), then this API is classified into permission discrepant API.

An API with permission discrepancies is identified if the API execution does not throw any "`permission errors`" in this platform but in others.

(III) API Output Discrepancies. Discrepancies of this kind are caused due to the discrepancies in device-specific outputs and platform-specific outputs. As such, we particularly inspect the output of the APIs for different platforms as well as different devices. Whenever we notice any differences in the output, this API is classified into output discrepant API.

An API with output discrepancies is identified if the API's outputs are platform and/or device specific.

5 Evaluation

5.1 Experiment Setup

To study the security issues and impact of our targeted super app WeChat, we have registered several user accounts, downloaded the corresponding miniapp development tools and SDKs, followed their official documents to build miniapps (e.g., some of the attacks need to be launched by malicious miniapps). In addition, some results from the experiments require us to reverse engineer WeChat, and therefore, we used JEB [5] and IDA Pro [23] to inspect the decompiled code statically and programmed Frida [25] scripts to dynamically verify our findings. Again, we run our ApiDiff on six devices, two Windows-11 desktops, and four smartphones (two with Android-13, and two with iOS-16).

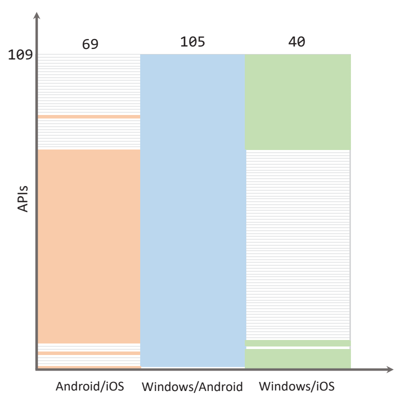
	Existence Discrepancies			Permission Discrepancies			Output Discrepancies		
	⊞	🤖		⊞	🤖		⊞	🤖	
⊞		105	40		16	17		8	12
🤖	105		69	16		1	8		14
	40	69		17	1		12	14	

Table 5: Summary of API Discrepancies

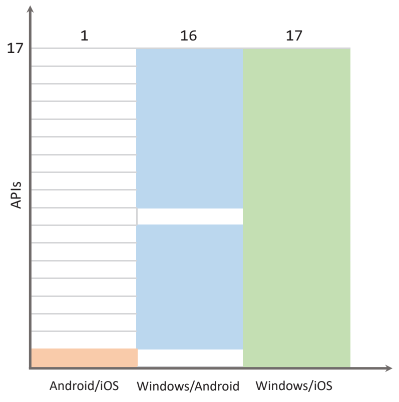
5.2 Experiment Results

Among the tested miniapp APIs, ApiDiff has identified three sets of APIs that exhibit existence (109), permission (17), and output (22) discrepancies across platforms and devices, which may be exploited by attackers to launch various attacks. As shown in Table 5, there are 105 APIs that have existence discrepancies between Windows and Android, 40 APIs between Windows and iOS, 69 APIs between Android and iOS. Meanwhile, there are 16 APIs that have permission discrepancies between Windows and Android, 17 APIs between Windows and iOS, and only one API (i.e., Bluetooth API) between Android and iOS. Finally, there are 8 APIs that have output discrepancies between Windows and Android, 12 APIs between iOS and Windows, and 14 APIs between Android and iOS.

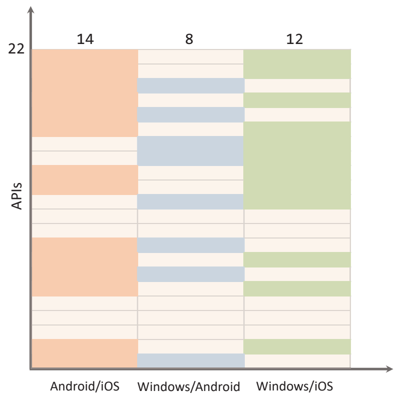
We present three figures in Figure 4 to illustrate the three types of discrepancies (existence, permission, and output)



(a) Existence Discrepancies



(b) Permission Discrepancies



(c) Output Discrepancies

Figure 4: Discrepancies Summary. Particularly, each row represents one API. Since output discrepancies exist not only across platforms but also across devices, we have highlighted all the rows in pink.

between different platforms and devices. Each row represents a specific API, and each column indicates whether there are discrepancies between the compared platforms. We color the cells to indicate discrepancies and leave them blank otherwise. For instance, there are 109 APIs with existence discrepancies, with 69 APIs exhibiting existence discrepancies between Android and iOS, 105 APIs between Windows and Android, and 40 APIs between Windows and iOS. Meanwhile, since output discrepancies exist not only across platforms but also across devices, we have highlighted all the rows (i.e., APIs) that exhibit output discrepancies across devices in pink. Next, we zoom in the results to obtain a few insights regarding the discrepancies in each category.

Results of API Existence Discrepancies. The API existence discrepancies may allow attackers to mount attacks against the device with weaker protections. In particular, we have identified 109 APIs of this kind, and these APIs fall into 32 categories, as shown in Table 6. Note that there are overlaps among those APIs, and therefore, the total number is less than the sum of discrepant APIs while comparing different platforms. For example, while Android has 69 APIs that are different from those on iOS, almost all of them are also included in Windows and iOS, resulting in a total of 109 APIs. We notice that Android and Windows are significantly different with the existence of the APIs, followed by Android and iOS. Windows and iOS share the most similar. Among all the APIs that have existence discrepancies, we notice that (i) most of those discrepancies are because of the support of specific hardware. For example, Windows does not support NFC and Gyroscope at all. (ii) Many of the discrepancies are caused by the system implementation discrepancies. For example, both Android and iOS support the accessibility services by nature, but Windows does not. As such, Windows does not have the accessibility services APIs for the miniapps. (iii) There are also some of the discrepancies that are caused by the super app's implementation discrepancies. For example, the crypto random number generation API is not implemented on Windows, but implemented on Android and iOS.

Results of API Permission Discrepancies. We have identified 17 APIs that have permission discrepancies, and those APIs fall into 10 categories including location. Again, we notice that Windows and iOS are mostly diversified, while Android and iOS are quite similar. The experiment results show that WeChat in Windows does not request any permissions from users to access privacy-sensitive resources (as shown in Table 7), and we can thus simply create a malicious miniapp to exploit this vulnerability. For instance, with a malicious miniapp, an attacker can directly open the microphones quietly without the user's awareness. We will demonstrate concretely how to exploit permission discrepancies in §6.2.

Results of API Output Discrepancies. In our experiment, we closed the super app first and then re-launched it again to run the experiment a second time. This was done to ensure that any observed differences were not caused by the running environment. As a result, we have identified 22 APIs that have output discrepancies, falling into 8 categories including UI, media, and device. Specifically, a fingerprintable API is defined by its unique and stable properties. The uniqueness of an API refers to its ability to generate a unique identifier for a user. The stability of the API is measured by its consistency over time, allowing it to be effectively used for tracking purposes. The uniqueness of an API is determined by running the API on multiple devices (6 devices in our experiment), with the APIs that produce different outputs considered unique. As shown in Table 8, among all the APIs with output discrepancies, 22 of them have the uniqueness feature. Among them, we further evaluated their stability.

API Category		Total	Platforms					
			🤖 🍎	%	⊞ 🤖	%	⊞ 🍎	%
Device	Accessibility	1	-	-	1	100.00	1	100.00
	Bluetooth	13	-	-	5	38.46	5	38.46
	BLE	15	-	-	3	20.00	3	20.00
	RandomNumber	1	-	-	1	100.00	1	100.00
	Gyroscope	3	-	-	1	33.33	1	33.33
	iBeacon	8	-	-	3	37.50	3	37.50
	Keyboard	4	-	-	1	25.00	1	25.00
	Motion	4	-	-	2	50.00	2	50.00
	Scan	1	-	-	1	100.00	1	100.00
	Screen	9	-	-	1	11.11	1	11.11
	Vibrate	2	-	-	1	50.00	1	50.00
	Wi-Fi	13	-	-	1	7.69	1	7.69
File		8	1	12.50	1	12.50	1	12.50
Location		12	-	-	2	16.67	2	16.67
Media	Audio	9	-	-	3	33.33	3	33.33
	Video	7	-	-	2	28.57	2	28.57
	VoIP	16	-	-	3	18.75	3	18.75
mDNS		10	-	-	1	10.00	1	10.00
NFC	IsoDep	7	7	100.00	7	100.00	-	-
	MifareClassic	6	6	100.00	6	100.00	-	-
	MifareUltralight	6	6	100.00	6	100.00	-	-
	Ndef	7	7	100.00	7	100.00	-	-
	NfcA	8	8	100.00	8	100.00	-	-
	NFCAdapter	13	13	100.00	13	100.00	-	-
	NfcB	6	6	100.00	6	100.00	-	-
	NfcF	6	6	100.00	6	100.00	-	-
	NfcV	6	6	100.00	6	100.00	-	-
OpenAPI		5	1	20.00	3	60.00	2	40.00
Storage		4	-	-	1	25.00	1	25.00
System		12	1	8.33	1	8.33	1	8.33
UI	Interaction	8	-	-	2	25.00	2	25.00
	Sticky	1	1	100.00	-	-	1	100.00

Table 6: Summary of APIs w.r.t Existence Discrepancies. The API categories are defined by Tencent [53], and we only list the categories that have discrepancies.

An API is deemed stable when it produces consistent outputs on the same device over multiple runs (3 times in our experiment). For all the APIs with uniqueness, 13 of them were found to be stable, which become fingerprintable APIs (as highlighted in the pink color in Table 8).

We then further grouped them into two categories: Input-Oriented Fingerprintable (IOF) APIs are those that accept inputs and generate outputs that are uniquely identifiable, while Fingerprintable (OOF) APIs do not accept input but produce outputs that are fingerprintable. For example, IOF API `getLocalIPAddress` takes an object specifying callback functions as input and produces a fingerprintable IP address as output. OOF API `getSystemInfo` does not take any inputs and directly returns fingerprintable system information. Specifically, for input-based IOF fingerprintable APIs, they may require specific parameters to be passed. We test them using the simple mutation method described in the paper, but we cannot guarantee the completeness of such APIs. However, this issue does not apply to OOF APIs.

At a high level, there are 9 OOF APIs and 4 IOF APIs. We found discrepancies across platforms, meaning that APIs that

APIs	Permission Scope	Mobile				PC	
		🤖		🍎		⊞	
		A	P	A	P	A	P
`getLocation`	**`userLocation`**	✓	✓	✓	✓	✓	✗
`chooseLocation`		✓	✓	✓	✓	✓	✗
`startLocationUpdate`		✓	✓	✓	✓	✓	✗
`SLUBackground*`	**`userLocationBackground`**	✓	✓	✓	✓	✗	-
`startRecord`	**`record`**	✓	✓	✓	✓	✓	✗
`joinVoIPChat`		✓	✓	✓	✓	✗	-
`RecorderManager.start`		✓	✓	✓	✓	✓	✗
`createCameraContext`	**`camera`**	✓	✓	✓	✓	✓	✗
`createVKSession`		✓	✓	✓	✓	✗	-
`openBluetoothAdapter`	**`bluetooth`**	✗	-	✓	✓	✗	-
`BLEPeripheralServer`		✓	✓	✓	✓	✗	-
`saveImageToPhotosAlbum`	**`writePhotosAlbum`**	✓	✓	✓	✓	✓	✗
`saveVideoToPhotosAlbum`		✓	✓	✓	✓	✓	✗
`addPhoneContact`	**`addPhoneContact`**	✓	✓	✓	✓	✗	-
`addPhoneRepeatCalendar`	**`addPhoneCalendar`**	✓	✓	✓	✓	✗	-
`addPhoneCalendar`		✓	✓	✓	✓	✗	-
`getWeRunData`	**`werun`**	✓	✓	✓	✓	✗	-

Table 7: Summary of the API permission discrepancies across platforms. "A" means "available" and "P" means "permission protected". "`SLUBackground`" is short for `startLocationUpdateBackground`.

work for fingerprinting on one platform may not work on another. It is worth noting that the precision of an API's output can have an impact on the effectiveness of fingerprinting attacks. APIs that provide more precise information can potentially fingerprint more users. For example, on Windows, the output precision of `getScreenBrightness` is limited to two decimal places, whereas on Android, it offers up to 16 decimal places, allowing for a greater differentiation and fingerprinting of users and devices.Considering that brightness values range from 0 to 1, the API output for Android has only two decimal places, thereby restricting the maximum number of identifiable users to 10^2. However, on Windows, with an output precision of 16 decimal places, the maximum number of identifiable users increases significantly to 10^{16}. For instance, suppose a user sets their screen brightness to `15%`. While the API output value on Windows may be 0.15, on Android it can be as precise as 0.1512452067894578, which can enable the identification of more users due to the higher level of precision.

We would also like to emphasize that some APIs may produce different outputs over time, as users may change their settings or devices may be relocated to a different location. For instance, `getLocalIPAddress` API returns the IP address of the device being tested, and its output may vary if the device is connected to a different network. Similarly, the `getSystemInfo` API contains some information that is specific to the device's settings (e.g., whether Bluetooth is turned on), and its output may also change if the user changes their settings. However, we will not consider these settings or changes in our analysis, as in traditional fingerprinting, the features used for fingerprinting, such as font size and timezone, may also change over time.

6 Exploiting the API Discrepancies

6.1 Exploiting API Existence Discrepancies

(A1) Fake Peripheral Attacks against Centrals. We have detected existence discrepancies in the Bluetooth pairing API. Note that Bluetooth resorts to pairing for its security, where the two Bluetooth-enabled devices to negotiate a communication key. In particular, `wx.makeBluetoothPair` is an API that can be invoked by the miniapps to initiate the pairing for the devices. Although WeChat has designed `wx.makeBluetoothPair` for Bluetooth pairing, this API is only available on Android, making iOS devices vulnerable to Bluetooth device impersonation.

We first demonstrate how fake peripheral attacks can be launched against centrals. Assuming that a miniapp running on the iOS central is a controller for a peripheral, the attacker waits for a moment when the central and peripheral (e.g., smart blood pressure monitor) are disconnected but intends to initiate a new session. This moment can be easily identified, since BLE communication transmits all traffic in plain text over the air and is sniffable before the connection is established. The attacker collects the identifier of the blood pressure monitor (i.e., its Bluetooth MAC address [65]) that goes over the air and impersonates the blood pressure monitor to establish a connection with the victim's smartphone. In a regular scenario, a smartphone is supposed to connect only one peripheral at a time, and therefore, the smartphone is currently not available for the blood pressure monitor. When the miniapp on the smartphone is launched and the victim smartphone attempts to start encryption using the key that is negotiated with the blood pressure monitor, the attacker responds with a `PIN_OR_KEY_MISSING` error. This is a widely used trick to attack Bluetooth devices [67, 57]. According to the current practice, Android OS will delete the key when it receives the error code. At this point, the peripheral currently is the attacker and will not initiate the pairing process for communication security, and for miniapp platforms, they do not provide APIs for the victim miniapp to start the pairing either. As a result, the communication continues in plaintext. Moreover, there is no way for the miniapp to know whether the link is secure. The attacker can now wait for the pairing request from the central and select plaintext to communicate with the central. However, since the miniapp running on iOS does not have API `wx.makeBluetoothPair` to authenticate the fake peripheral, the communication proceeds normally as usual without notifying users. This could cause serious consequences for miniapps running on iOS devices. For example, the attacker may inject false blood pressure measurements, misguiding doctors. We have discovered that the function makeBluetoothPair is widely utilized by many government system mini-applications. One such example is the water-meter reading system, which may leverage these mini-apps to monitor household water usage. Unfortunately, these mini-apps are vulnerable to attackers who may manipulate the measurement readings. Our investigation also revealed that some major Chinese companies, such as Meituan (a prominent food delivery company [6]) and DiDi (a ride-hailing service [3]), employ Bluetooth functionality and are thus susceptible to these attacks.

Practicality. This type of attack is highly practical due to three reasons. First, low-cost devices such as Bluetooth sniffers (e.g., 20 dollars [10]) and smartphones can be easily obtained to carry out the attack. Second, all traffic in BLE communication is transmitted in plain text before the connection is established. This enables the attacker to obtain the MAC address needed for impersonation and to determine the ideal moment for the attack. Third, during the attack, the user of the victim smartphone will not receive any notifications or warnings. This is because the API `wx.isBluetoothDevicePaired`, which is used to check the pairing status between two devices, is not available for iOS devices. As a result, the attack can be deployed stealthily and without the user's knowledge.

Defense. To defend against A1, developers may need to utilize cryptographic techniques for authentication, as relying solely on pairing may not be sufficient to defend against a fake peripheral injecting messages into miniapps. While these cryptographic techniques are standard, negotiating a key between the IoT device and the smartphone can be challenging. A potential solution is to involve the user in entering the same password on both devices when they are first connected, allowing the devices to derive the same cryptographic key.

(A2) Fake Central Attacks against Peripherals. Miniapps can also operate as software-defined peripherals (SDPs), allowing developers to add services with API `addService` for other native apps or miniapps to utilize. Since these services may contain sensitive information, the Bluetooth Specification provides security levels [13] that devices can customize to safeguard their services. For instance, if two devices are paired, the security level of the connection meets the "encryption" level. At this point, the central device can access services configured with "encryption". However, if the two devices are not paired, the security level is lower, and the central device cannot access the services that necessitate an encrypted link. We found that WeChat has implemented security levels for Android devices but not for iOS. The absence of a stronger security level makes miniapps on iOS devices vulnerable to MitM attacks, which can result in unauthorized data access.

In this attack, we assume that an Android phone runs a peripheral miniapp and makes the phone to be a SDP, which provides services. Another device, which attempts to connect the peripheral, is denoted as the victim central. We assume that the attacker is in the range of the two communicat-

APIs				Mobile						Desktop		
				🤖						⊞		
Name	Category	Type	Precision	A	S	U	A	S	U	A	S	U
`createAudioContext`	Media	⇥	✗	✓	✗	✓	✓	✗	✓	✓	✗	✓
`createBufferURL`	Storage	⇥	✗	✓	✗	✓	✓	✗	✓	✓	✗	✓
`createCameraContext`	Media	↦	✗	✓	✗	✓	✓	✗	✓	✓	✗	✓
`createCanvasContext`	Canvas	⇥	✗	✓	✗	✓	✓	✗	✓	✓	✗	✓
`createIntersectionObserver`	WXML	⇥	✗	✓	✗	✓	✓	✗	✓	✓	✗	✓
`createLivePusherContext`	Media	↦	✗	✓	✗	✓	✓	✗	✓	✓	✗	✓
`createOffscreenCanvas`	Canvas	⇥	✗	✓	✗	✓	✓	✗	✓	✓	✗	✓
`createSelectorQuery`	WXML	↦	✗	✓	✗	✓	✓	✗	✓	✓	✗	✓
`createWebAudioContext`	Media	↦	✗	✓	✗	✓	✓	✗	✓	✓	✗	✓
`getAccountInfoSync`	OpenAPI	↦	✗	✓	✓	✗	✓	✓	✓	✓	✓	✗
`getAppAuthorizeSetting`	Base	↦	✗	✓	✓	✓	✓	✓	✓	✓	✓	✗
`getAppBaseInfo`	Base	↦	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
`getDeviceInfo`	Base	↦	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
`getLocalIPAddress`	Device	⇥	✗	✓	✓	✓	✓	✓	✗	✓	✓	✗
`getMenuButtonBoundingClientRect`	UI	↦	✗	✓	✓	✗	✓	✓	✓	✓	✓	✗
`getPerformance`	Base	↦	✗	✓	✓	✓	✓	✓	✗	✓	✓	✗
`getScreenBrightness`	Device	⇥	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓
`getSystemInfo`	Base	⇥	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`getSystemInfoAsync`	Base	⇥	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`getSystemInfoSync`	Base	↦	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`getSystemSetting`	Base	↦	✗	✓	✓	✗	✓	✓	✓	✓	✓	✗
`getWindowInfo`	Base	↦	✗	✓	✓	✗	✓	✓	✓	✓	✓	✓

Table 8: Summary of APIs w.r.t Output Discrepancies. ⇥ represents Input-Oriented and ↦ represents Output-Oriented. A means "Available", S means "Stable" and U means "Unique". An API is fingerprintable if there is at least one platform in which it is available (A check), stable (S check), and unique (U check). The fingerprintable APIs have been distinctly marked in pink for easy identification.

ing devices, and selects the moment when the two communicating devices intend to communicate. First, the attacker pretends to be the victim central and connects to the victim SDP. Then, the attacker initiates a service access request to the peripheral. When receiving the request, the peripheral should have the capabilities to check the status of the link to either approve or reject the request. For example, for a miniapp running atop Android, the miniapp can set the security level to `readEncryptionRequired` through `addService`, which will guide the device to only accept the request when the link is encrypted. However, there is no protection on the iOS device, and the attacker can access all the services (e.g., the services can be the user's contact list) on the peripheral without any protection. Indeed, we have found that SDP services are commonly utilized by numerous TV controllers, and smart home devices (such as weight scales, smart lights, and smart locks), which are sourced from different vendors such as Alibaba Group [1] and Zhanzhibao [9].

Practicality. This type of attack is also highly practical for three reasons. First, since the SDP provides services that are accessible to all other devices or apps, there is no additional information required (e.g., MAC address) for the attacker to impersonate a victim central device. Second, the attacker can easily use his or her smartphone to connect to the SDP and consume its services without needing any other devices. Third, the attack can be carried out quietly since SDPs do not send notifications to the user when other devices or apps access their services.

Defense. Since the root cause of A2 is the lack of adequate security levels for the services SDP, developers can address this issue by implementing different levels of keys based on the authentication method used by the two devices. Additionally, they can track the connection status to ensure that other miniapps or native apps can only access the services when a specific security level is met. For instance, the SDP can detect whether a request is coming from a device that doesn't share any keys with the SDP, or shares a common key produced by cryptographic techniques (i.e., encryption level).

6.2 Exploiting API Permission Discrepancies

(A3) Information Collection Attack. APIs for accessing sensitive resources lack consistent safeguards across platforms, enabling attackers to steal privacy-sensitive information in Windows based on available resource categories. Specifically, miniapps on Windows can access resources even when in the background, allowing a malicious app to run unnoticed for an extended period. In contrast, miniapps on Android or iOS can only access system resources when running in the foreground, and if left in the background for more than 5 minutes, WeChat on Android and iOS will terminate their execution to free up resources. This gives a malicious miniapp on Windows more capability to collect user data surreptitiously. To confirm these observations, we developed a miniapp with all the identified resources access APIs to intentionally collect the resources, and it runs as expected. We assume that the malicious miniapp can be installed onto the user's device (e.g., a Windows device). In the following, we show how the permission discrepant APIs in Windows can be exploited.

- **User Tracking Attacks:** A malicious miniapp can track a user's location by invoking API `getLocation`, allowing the attacker to know the user's whereabouts over time.
- **Conversation Eavesdropping Attacks:** A miniapp can activate the user's microphone (using `wx.startRecord`) to eavesdrop on conversations, recording and uploading the audio files to its back-end.
- **Stealthy Photo and Video Capture:** A miniapp can use the user's camera without their knowledge via `wx.createCameraContext`, and take photos and videos using `CameraContext.takePhoto` and `CameraContext.record`, respectively.
- **User Information Stealing:** A miniapp can extract sensitive user information, such as gender, username, and addresses, by invoking `getUserInfo`, which should have been adequately protected to prevent abuse.

Practicality. This attack is highly practical for three reasons. First, since no permission is required to access sensitive resources, the user will not be notified when a malicious miniapp accesses them. The malicious app can disguise its true intentions by providing seemingly harmless services, such as a game app, and then records the user's audio and location to send the collected sensitive information to a remote server. Second, the malicious miniapp on Windows can run in the background without being terminated by super apps, allowing the attacker to monitor the victim for an extended period. Finally, these miniapps are difficult to detect because when they run on mobile platforms, permissions are actually required. Therefore, from the super app's perspective, it is challenging to determine whether the miniapp is intended to run on mobile devices or on Windows.

Defense. To address the issue of permission discrepancies, Tencent must first implement missing permission checks on resource access APIs for the Windows platform. Currently, Tencent is working on fixing these issues. For instance, we observed that WeChat on Windows no longer supports accessing user locations. In addition to considering the threat models of different OSs, designers must also consider the threat models of their services. As discussed, users trust and install WeChat on Windows, and they do not expect installless miniapps to access their resources without authorization. Web browsers have already implemented some best practices to address this issue. For instance, regardless of the OS, web apps running on Chrome must explicitly request permission to use the camera when needed [44]. WeChat needs to have its own permission system imposed on miniapps, independent of the underlying OS, just like Chrome (on all platforms) does for web apps.

6.3 Exploiting API Output Discrepancies

(A4) Fingerprinting Attacks. This attack collects unique device information, such as app version, screen resolution, and installed fonts, through fingerprintable APIs. When combined, this information can create a unique fingerprint that can be used to track a user's online behavior across multiple devices and sessions. While there are many APIs that can be used to fingerprint users, we select `getSystemInfo` to demonstrate the attack given that it can collect up to 27 types of device-specific information (please refer to Table 9 in our Appendix-§A for the types of collected information). Specifically, `getSystemInfo` is an API that can collect various types of information about a device, including its hardware and software specifications, network information, and other system-related data. To use `getSystemInfo` for fingerprinting, a developer can create a miniapp that calls the API and collects the relevant data. We assume that the malicious miniapp is installed onto the user's device.

The workflow of the attack can be described as follows: the malicious miniapp first invokes the `getSystemInfo` to collect a number of device-specific information (e.g., system version, device model). Second, the collected data can be hashed or otherwise processed to create an identifier for the device. By combining this information, it is possible to create a relatively fingerprint for a device, which can be used to track it across different sessions or applications. The identifier will be saved for future reference. Finally, the next time when the attacker would like to fingerprint the device, the attacker invokes `getSystemInfo` again to calculate the identifier. If the identifier matches one of the recorded identifiers, the device is fingerprinted. Fingerprinting attacks can lead to privacy violations and potential abuses, such as targeted advertising or even identity theft.

Practicality. This attack is quite practical as the attack can be conducted stealthily. WeChat miniapps have the ability to access user ID information, such as phone numbers (i.e.,`getPhonenumber`) and profiles (i.e.,`getUserInfo`). However, accessing this information requires specific permissions granted by the user. In contrast, fingerprintable APIs are more practical for tracking users since they do not require explicit permission from the user. Interestingly, we have already identified some malicious miniapps that used those fingerprintable APIs to track the users. For example, Figure 5 shows a code snippet of miniapp "`wx58f310cf31f0d423`", which generates an identifier for devices with the same type and settings based on specific device properties, including `brand`, `model`, `pixelRatio`, `screenWidth`, `screenHeight`, `system`, `platform`. All those information is collected by invoking `getSystemInfo`, which is one of our identified fingerprintable APIs. Particularly, the hash function md5 (line 8) is used to provide a simple and efficient way to compute an identifier for devices with the same type and settings. As shown in Table 9 of Appendix-§A, `getSystemInfo` can identify certain classes of devices based on the platform information it returns (such as Android, Windows, iOS, or Mac). Indeed, `getSystemInfo` returns not only system information (e.g., `platform` and `brand`) but also user-specific settings information such as font size and language, which can increase the fingerprinting capabilities.

It is also surprising to notice that even for this single API, some returned values differ across different platforms. While most of the returned values are supported by both Android and iOS, some values are only available in Android or iOS. For instance, `benchmarkLevel` indicates the hardware condition of a specific device and is defined by Tencent, ranging from -2 to 50. The higher the value, the better the device's performance. Therefore, an extra value is available for fingerprinting Android devices. Another example is `getScreenBrightness`. As discussed, in contrast to Windows, the `getScreenBrightness` function on Android provides a greater output precision of up to 16 decimal places (instead of two), which enables a more detailed differentiation and fingerprinting of both users and devices. These findings further highlight the close relationship between cross-platform nature and our results.

Cross-platform information can be leveraged to enhance the existing fingerprinting techniques. current fingerprinting techniques typically do not take cross-platform differences into account, or simply use platform information as one feature among others to improve their performance. However, in a cross-platform context, an attacker can leverage knowledge about which APIs are effective on which platforms to enhance their fingerprinting accuracy. For example, if malware detects that the running environment is Android, it can use benchmarkLevel as a feature to produce a fingerprint (as this API does not work on iOS). Similarly, if the malware detects that the running environment is Android, it can use the 16 decimal place output of getScreenBrightness to fingerprint more users.

Defense. Fingerprinting attacks are difficult to defend against because they use unique device properties that are hard to hide without affecting legitimate functionality. In the web domain, browser extensions and tools can generate false fingerprint data to mask actual device or browser properties. Similarly, as the host of miniapps, WeChat could adopt a similar strategy. If a miniapp requests device-specific information too frequently or in an unexpected way (e.g., collecting multiple device-specific information at the same time), the super app can return randomized data to prevent tracking. It can also alert the user if it detects such behaviors to inform them of potential tracking attempts.

```
1   function a(t) {
2       var o = [ "brand", "model", "pixelRatio",
     ↪  "screenWidth", "screenHeight", "system",
     ↪  "platform" ];
3       // t = [{key: "brand", value: "samsung"}, {key:
     ↪  "model", value: "SM-S901U1"}, {key: "pixelRatio",
     ↪  value: 3}, {key: "screenWidth", value: "360"},
     ↪  {key: "screenHeight", value: "765"}, {key:
     ↪  "system", value: "Android 13"}, {key: "platform",
     ↪  value: "android"}] (from wx.getSystemInfo)
4       var n = t.reduce(function(e, t) {
5           return o.indexOf(t.key) > -1 ? e + t.value +
     ↪  "," : e + "";
6       }, "");
7       // n = "samsung,SM-S901U1,3,360,765,Android
     ↪  13,android"
8       _ = f.hex_md5(n.substring(0, n.length - 1)),
     ↪  l.setCookie({
9           data: {
10              shshshfp: {
11                  value: _,
12                  maxAge: 3153e3
13              }
14          }
15      });
16  }
```

Figure 5: Code snippet of a real world miniapp used fingerprintable API

7 Discussion

Lessons Learned. Our findings differ from existing works (e.g., lack of permissions, fingerprinting) due to security issues arising from differences in API implementation across various platforms. As a super app, WeChat should ensure uniform implementation or protection of APIs for security purposes. Although some platforms have implemented security functions and are aware of the problem, their designers may not be aware of the risk (e.g., missing permissions on essential resources). Despite differing threat models for underlying systems, WeChat should assume responsibility for ensuring consistent security guarantees.

Ethics and Responsible Disclosure. We followed community practice by analyzing and launching attacks in a controlled environment using our own accounts and machines. We developed attack code and malware to demonstrate the attacks, but kept them private to prevent harm to users, miniapp developers, and platform providers. We reported our findings to Tencent and they acknowledged them by awarding us bug bounties. Tencent's security engineers have actively worked with us over the past year, meeting online multiple times to discuss vulnerabilities and corresponding fixes, some of which have already been applied. For example, for (A3), Tencent has taken action on the Windows version by completely removing sensitive information access APIs: invoking `wx.getLocation` results in a message stating that the API is not supported.

8 Related Work

Super App Security. Super apps, despite their popularity and convenience, are not immune to security vulnerabilities. Lu *et al.* [39] identified a resource management flaw that allows attackers to acquire sensitive data without requiring permissions and highlighted the potential for phishing attacks. Meanwhile, Zhang *et al.* [64] discovered identity confusion vulnerabilities that could lead to severe consequences, including malware installation. While our study also focuses on super app security, we specifically address how to automatically detect and exploit discrepancies in API execution across different platforms. Zhang *et al.* developed MiniCrawler [66], a tool for analyzing security practices in miniapps. It focuses on aspects such as obfuscation usage and security-related API invocations. Yang *et al.* [61] conducted research on the security check vulnerabilities that are missing in cross-miniapp channels of miniapps found on popular platforms such as WeChat and Baidu. Wang *et al.* [16] presented TaintMini, a comprehensive framework aimed at detecting collusion attacks and data leakages among miniapps through taint analysis. Zhang *et al.* [68] examined the misuse and consequences of cryptographic keys within miniapps, shedding light on the potential risks associated with improper key handling. Wang *et al.* [17] revealed undisclosed APIs in super apps, drawing attention to their exploitable nature. Additionally, they systematically identified inconsistencies and derived vulnerabilities in WeChat APIs across different platforms, offering a comprehensive overview of potential weaknesses.

Super apps evolved from web browsers, and previous research on browser security, such as web extensions [59, 49, 18, 56], is closely related to our study. For instance, SABRE [21] tracks in-browser information flow to detect malicious browser extensions that leak sensitive information, while Hulk [36] dynamically detects malicious browser extensions by monitoring their execution. Expector [59] identifies browser extensions that involve advertisements and detects malicious ones. However, unlike these works, our study is the first to uncover discrepancies in super apps that can be exploited for attacks.

Cross Platform App Studies. There are also efforts studying the platform differences on app's development, security and privacy. For instance, Han *et al.* [28] investigated both Android and iOS apps and examine their difference in the usage of their security sensitive APIs. Dhillon *et al.* [22] developed a framework for evaluating cross-platform development tools. Xanthopoulos *et al.* [58] studied cross-platform mobile app development approaches. Different from those works, we studied how the implementation discrepancies affect the security of super apps.

Fingerprinting Attacks. Browser fingerprinting attacks have been widely discussed in recent years, where the configuration information and hardware information of the user devices is exposed through JavaScript APIs [15, 41, 54, 40, 41, 14] or HTTP headers [43, 42, 37]. Those attacks can also be used for various malicious purposes such as breaking reCAPTCHA [48, 11] and web authentication [47, 62]. Meanwhile, the topic of detecting fingerprinting attacks on Android has been extensively discussed in the literature, utilizing both static techniques [24, 12] and dynamic approaches [60, 52]. When compared with those attacks, our work is different from at least two aspects: First, our work explores the attack surface in a novel domain, where the miniapps run atop super apps (not the web apps running atop browsers). Second, in addition to the fingerprinting attacks, we also identified a few types of novel attacks such as info leaks.

9 Conclusion

In this paper, we have shown that there are API discrepancies for super apps when executed in different platforms and devices, and such discrepancies can be exploited for various malicious purposes such as spying and even fingerprinting users. To automatically uncover these APIs, we have developed APIDIFF, a tool that is able to automatically generate and execute the test cases and identify the discrepancies. We have tested APIDIFF with WeChat, and it has found 109 APIs with existence discrepancies, 17 APIs with permission discrepancies, and 22 APIs with output discrepancies across platforms. We have disclosed the vulnerabilities to WeChat, and some of the vulnerabilities have been patched.

Acknowledgment

We thank our shepherd as well as the anonymous reviewers for their insightful feedbacks. 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] "Alibaba group," https://en.wikipedia.org/wiki/Alibaba_Group.

[2] "Burb suite fuzz payloads," https://github.com/1N3/IntruderPayloads/blob/master/FuzzLists/full_fuzz.txt.

[3] "Didi," https://en.wikipedia.org/wiki/DiDi.

[4] "How facebook, apple, google copied china's wechat messaging app," https://exbulletin.com/tech/274959/.

[5] "Jeb," https://www.pnfsoftware.com/jeb/android.

[6] "Meituan," https://en.wikipedia.org/wiki/Meituan.

[7] "Scope list of wechat," https://developers.weixin.qq.com/miniprogram/en/dev/framework/open-ability/authorize.html.

[8] "What is a super app and why haven't they gone global?" https://www.cnbc.com/video/2021/07/16/what-is-a-super-app-and-why-havent-they-gone-global.html.

[9] "Zhanzhibao," http://zhanzhibaoqijiandian.mall.etsstar.com/.

[10] Adafruit , "Adafruit sniffer," https://learn.adafruit.com/introducing-the-adafruit-bluefruit-le-sniffer/.

[11] I. Akrout, A. Feriani, and M. Akrout, "Hacking google recaptcha v3 using reinforcement learning," *arXiv preprint arXiv:1903.01003*, 2019.

[12] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[13] S. Bluetooth, "Bluetooth core specification version 5.1," *Specification of the Bluetooth System*, 2019.

[14] K. Boda, Á. M. Földes, G. G. Gulyás, and S. Imre, "User tracking on the web via cross-browser fingerprinting," in *Nordic conference on secure it systems*. Springer, 2011, pp. 31–46.

[15] D. Cameron, "Apple declares war on browser fingerprinting, the sneaky tactic that tracks you in incognito mode."

[16] W. Chao, Z. Yue, and L. Zhiqiang, "Taintmini: Detecting flow of sensitive data in mini-programs with static taint analysis," in *ICSE*.

[17] W. Chao, Y. Zhang, and Z. Lin, "Uncovering and exploiting hidden apis in mobile super apps," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[18] Q. Chen and A. Kapravelos, "Mystique: Uncovering information leakage from browser extensions," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1687–1700.

[19] H. Choi, B. B. Zhu, and H. Lee, "Detecting malicious web links and identifying their attack types." *WebApps*, vol. 11, no. 11, p. 218, 2011.

[20] B. Dean, "Tiktok user statistics (2022)," https://backlinko.com/tiktok-users.

[21] M. Dhawan and V. Ganapathy, "Analyzing information flow in javascript-based browser extensions," in *2009 Annual Computer Security Applications Conference*. IEEE, 2009, pp. 382–391.

[22] S. Dhillon and Q. H. Mahmoud, "An evaluation framework for cross-platform mobile application development tools," *Software: Practice and Experience*, vol. 45, no. 10, pp. 1331–1357, 2015.

[23] C. Eagle, *The IDA pro book*. no starch press, 2011.

[24] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, pp. 1–29, 2014.

[25] Frida, "Firda–dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers." https://frida.re/docs/android/, 2012.

[26] T. GRAZIANI, "What are wechat mini-programs? a simple introduction - walkthechat," https://walkthechat.com/wechat-mini-programs-simple-introduction/.

[27] G. Group, "Gojek and tokopedia combine to form goto," https://newsroom.gojek.com/gojek/b8zdnwe8rv98aealdlh05i0pf04v4f.

[28] J. Han, Q. Yan, D. Gao, J. Zhou, and R. H. Deng, "Comparing mobile privacy protection through cross-platform applications," 2013.

[29] T. Inc, "55+ wechat statistics - 2022 update," https://99firms.com/blog/wechat-statistics/#gref.

[30] ——, "Wechat official documents - wecom pre-development notes," https://developers.weixin.qq.com/miniprogram/dev/dev_wxwork/.

[31] ——, "Wecom 2022 annual conference," https://work.weixin.qq.com/nl/index/v4Intro.

[32] ——, "WXML," https://developers.weixin.qq.com/miniprogram/en/dev/reference/wxml/, 03 2020.

[33] M. IQBAL, "Tiktok revenue and usage statistics (2020)," https://www.businessofapps.com/data/tik-tok-statistics/, 2020.

[34] U. Iqbal, S. Englehardt, and Z. Shafiq, "Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1143–1161.

[35] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for javascript," in *International Static Analysis Symposium*. Springer, 2009, pp. 238–255.

[36] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson, "Hulk: Eliciting malicious behavior in browser extensions," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 641–654.

[37] P. Laperdrix, W. Rudametkin, and B. Baudry, "Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 878–894.

[38] X. Lin, P. Ilia, S. Solanki, and J. Polakis, "Phish in sheep's clothing: Exploring the authentication pitfalls of browser fingerprinting," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1651–1668.

[39] 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.

[40] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham, "Fingerprinting information in javascript implementations," *Proceedings of W2SP*, vol. 2, no. 11, 2011.

[41] M. Mulazzani, P. Reschl, M. Huber, M. Leithner, S. Schrittwieser, E. Weippl, and F. Wien, "Fast and reliable browser identification with javascript engine fingerprinting," in *Web 2.0 Workshop on Security and Privacy (W2SP)*, vol. 5. Citeseer, 2013, p. 4.

[42] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "Cookieless monster: Exploring the ecosystem of web-based device fingerprinting," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 541–555.

[43] ——, "On the workings and current practices of web-based device fingerprinting," *IEEE Security & Privacy*, vol. 12, no. 3, pp. 28–36, 2014.

[44] K. Paul and S. Mat, "Capturing an image from the user," https://developers.google.com/web/fundamentals/media/capturing-images.

[45] N. Reitinger and M. L. Mazurek, "Ml-cb: Machine learning canvas block," *Proceedings on Privacy Enhancing Technologies*, vol. 2021, no. 3, pp. 453–473, 2021.

[46] R. Rodenbaugh, "A breakdown of whatsapp and facebook's super-app ambitions," https://www.techinasia.com/whatsapp-facebooks-super-app-ambitions.

[47] S. Ruoti, B. Roberts, and K. Seamons, "Authentication melee: A usability analysis of seven web authentication systems," in *Proceedings of the 24th international conference on world wide web*, 2015, pp. 916–926.

[48] S. Sivakorn, J. Polakis, and A. D. Keromytis, "I'm not a human: Breaking the google recaptcha," *Black Hat*, vol. 14, 2016.

[49] D. F. Somé, "Empoweb: Empowering web applications with browser extensions," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 227–245.

[50] Statista, "Tiktok user statistics (2022)," https://www.statista.com/statistics/552671/snapchat-app-dau-region/.

[51] M. Stiltner, "The top 6 super apps in asia – and what they reveal about the global trend," https://www.rapyd.net/blog/the-top-6-super-apps-in-asia-and-what-they-reveal-about-a-global-trend/.

[52] K. Tam, A. Fattori, S. Khan, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors," in *NDSS Symposium 2015*, 2015, pp. 1–15.

[53] Tencent, "WeChat English Documentation," https://developers.weixin.qq.com/miniprogram/en/dev/api/, 06 2020.

[54] R. Upathilake, Y. Li, and A. Matrawy, "A classification of web browser fingerprinting techniques," in *2015 7th International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 2015, pp. 1–5.

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

[56] R. Wang, L. Xing, X. Wang, and S. Chen, "Unauthorized origin crossing on mobile platforms: Threats and mitigation," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 635–646.

[57] J. Wu, Y. Nan, V. Kumar, D. J. Tian, A. Bianchi, M. Payer, and D. Xu, "Blesa: Spoofing attacks against reconnections in bluetooth low energy." in *WOOT@ USENIX Security Symposium*, 2020.

[58] S. Xanthopoulos and S. Xinogalos, "A comparative analysis of cross-platform development approaches for mobile applications," in *Proceedings of the 6th Balkan Conference in Informatics*, 2013, pp. 213–220.

[59] X. Xing, W. Meng, B. Lee, U. Weinsberg, A. Sheth, R. Perdisci, and W. Lee, "Understanding malvertising through ad-injecting browser extensions," in *Proceedings of the 24th international conference on world wide web*, 2015, pp. 1286–1295.

[60] L.-K. Yan and H. Yin, "Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis." in *USENIX security symposium*, 2012, pp. 569–584.

[61] 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.

[62] N. Yıldırım and A. Varol, "Android based mobile application development for web login authentication using fingerprint recognition feature," in *2015 23nd Signal Processing and Communications Applications Conference (SIU)*. IEEE, 2015, pp. 2662–2665.

[63] Zalo, "Zalo: About us," https://zalo.careers/about.

[64] 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.

[65] Y. Zhang and Z. Lin, "When good becomes evil: Tracking bluetooth low energy devices via allowlist-based side channel and its countermeasure," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 3181–3194. [Online]. Available: https://doi.org/10.1145/3548606.3559372

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

[67] Y. Zhang, J. Weng, R. Dey, Y. Jin, Z. Lin, and X. Fu, "Breaking secure pairing of bluetooth low energy using downgrade attacks," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 37–54.

[68] 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.

Property	**Type**	**Description**
`brand`	string	Device brand
`model`	string	Device model
`pixelRatio`	number	Device's pixel ratio
`screenWidth`	number	Screen width in px
`screenHeight`	number	Screen height in px
`windowWidth`	number	Available window width in px
`windowHeight`	number	Available window height in px
`statusBarHeight`	number	Status bar height in px
`language`	string	Language set in WeChat
`version`	string	WeChat version
`system`	string	Operating system and version
`platform`	string	Client platform
`fontSizeSetting`	number	User's font size in px.
`SDKVersion`	string	Base library version for the WeChat app
`benchmarkLevel`	number	The device performance grade (only for Miniapps on **Android**).
`albumAuthorized`	boolean	The switch that allows WeChat to use Photos (only for **iOS**)
`cameraAuthorized`	boolean	The switch that allows WeChat to use the camera
`locationAuthorized`	boolean	The switch that allows WeChat to use the location function
`microphoneAuthorized`	boolean	The switch that allows WeChat to use the microphone
`notificationAuthorized`	boolean	The switch that allows WeChat to send notifications
`notificationAlertAuthorized`	boolean	The switch that allows WeChat to send notifications with reminders (only for **iOS**)
`notificationBadgeAuthorized`	boolean	The switch that allows WeChat to send notifications with flags (only for **iOS**)
`notificationSoundAuthorized`	boolean	The switch that allows WeChat to send notifications with sound (only for **iOS**).
`bluetoothEnabled`	boolean	The system switch for Bluetooth
`locationEnabled`	boolean	The system switch for the GPS function
`wifiEnabled`	boolean	The system switch for Wi-Fi
`safeArea`	Object	Safe area when the screen is in vertical orientation

Table 9: The information can be collected through `getSystemInfo`

A A Fingerprintable API `getSystemInfo`

As shown in Table 9, `getSystemInfo` has the ability to gather 27 types of information, enhancing its effectiveness in fingerprinting users. It is noteworthy that some returned values of this API vary across different platforms. Although most of the returned values are supported by both Android and iOS, a few values are exclusive to either platform. We found some malicious miniapps that tracked users by using this API. For example, miniapp "`wx58f310cf31f0d423`" used the `getSystemInfo` API to collect device properties such as `brand`, `model`, `pixelRatio`, `screenWidth`, `screenHeight`, `system`, and `platform` to generate a device identifier using the md5 hash function (line 8 in Figure 5). This allowed the miniapp to track devices with the same settings and type.