Stealthy Trackers: Uncovering Permission-less Fingerprinting in WeChat Miniapps

Yuqing Yang
The Ohio State University
Columbus, OH, USA
yang.5656@osu.edu

Zhiqiang Lin
The Ohio State University
Columbus, OH, USA
zlin@cse.ohio-state.edu

ABSTRACT

Miniapps have become an integral part of super apps like WeChat, enabling third-party developers to offer diverse services such as ride-hailing, shopping, and gaming. While miniapps provide a lightweight, convenient development model with access to rich APIs, they also introduce new privacy concerns. Specifically, miniapps often collect sensitive information including device identifiers and social network metadata via super app APIs, raising the risk of user fingerprinting. Unlike traditional web and mobile platforms, fingerprinting in miniapps has not yet been systematically studied, despite growing concerns and user complaints. In this paper, we present the first large-scale study on fingerprinting within miniapps. We begin by analyzing real-world miniapps to extract fingerprinting patterns, and then introduce Fingerprint-Finder, a detection tool that identifies clusters of permission-less, fingerprintable data. Applying Fingerprint-Finder to a dataset of over 4.03 million miniapps, we identify 1,310 cases of fingerprinting behavior, which are later clustered into 285 families. We further identify that basic, benchmark, and screen information are commonly used for performing fingerprinting users. Besides, canvas fingerprinting techniques are also popular. These codes come from not only miniapp templates and reused components, but also from third-party libraries providing business analytical services. Our findings reveal the breadth and sophistication of fingerprinting practices in miniapps, and we publicly release our dataset to foster further research. This work offers critical insights for building effective defenses and shaping future privacy regulations in the miniapp ecosystem.

ACM Reference Format:

Yuqing Yang and Zhiqiang Lin. 2025. Stealthy Trackers: Uncovering Permission-less Fingerprinting in WeChat Miniapps. In *Proceedings of the 2025 Workshop on Security and Privacy of AI-Empowered Mobile Super Apps (SaTS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3733824.3764871

1 INTRODUCTION

With the emergence of miniapps, giant app vendors such as WeChat has transformed into super apps. These super apps offer a wide range of essential services tied to users' daily lives, such as ride-hailing, food delivery, online shopping, and gaming, provided by various third-party developers via miniapp integration. For these service provides, miniapps enable them to reach a huge amount of users with minimal development cost by leveraging the feature-rich functions from super apps, which brings additional channels to acquire new users and facilitate convenient shopping experience.

However, these services commonly involve sensitive information. For instance, automatic login via phone number becomes a common practice among miniapps, and user information is often collected by large business vendors to deliver personalized advertisements. As a means of extending the services provided via web applications and mobile apps, miniapps have fallen into a gray area where user fingerprinting becomes concerning, just as the traditional web and mobile research domain. With the convenient APIs provided by the super app platforms, accessing these types of information becomes increasingly easy, with just a few invocation to super app APIs, developers can acquire a wide range of information ranging from device information to social network details.

Despite that the super apps enforce permission mechanisms on critical information such as users' phone numbers, there are still a number of data remain overlooked. However, these "permission-less" data, once collected by miniapps, may still generate enough information for tracking a unique user or device. As reported by recent studies [1, 2, 5–8, 12, 16, 21], information such as screen information, system software version, installed fonts, and even images drawn on canvas have been commonly used by web applications to facilitate user fingerprinting. As a result, these information, if collected without proper regulation, can cause serious user privacy issues.

Compared with traditional domains such as web security and android security, the fingerprinting issue in miniapps have not yet been previously studied. Although there has been reports and complaints from users about miniapps attempting to fingerprint users for customized recommendations, it remains unclear why and how the fingerprinting is made possible, given the additional restrictions such as data isolation enforced by super apps. In response to this, this paper presents the first study on fingerprinting in the context of miniapps. To do so, in this paper, we first generate research insights of detecting fingerprinting in miniapps based on motivating examples collected from real-world miniapps. Then, we develop Fingerprint-Finder, a tool to identify and attribute clustered collections of permission-less fingerprintable information.

We apply Fingerprint-Finder and additional semantic constraints to filter miniapps involving fingerprinting among over 4.03 million miniapps, resulting in 1,310 miniapps out of 1,367 miniapp cases, which are clustered into 285 families. We further examined these cases, and identified three types of information (basic, benchmark, and screen information) commonly used for device fingerprinting, together with 2 miniapp template platforms, 2 WeChat

SaTS '25, October 13–17, 2025, Taipei, Taiwan

ACM ISBN 979-8-4007-1912-7/2025/10
https://doi.org/10.1145/3733824.3764871

miniapp components, and 4 analytical libraries that are involved in these fingerprinting behaviors. We are releasing our findings to the public, and we believe that our first study in this field generate valuable insights for future works on comprehensive fingerprinting detection, so as to safeguard the overall privacy of the entire miniapp ecosystem.

In short, this paper makes the following contribution:

- **First study in miniapp user fingerprinting.** In this research, we present the first study of user fingerprinting in miniapps. Through automatic detection and miniapp clustering, we identify 3 types of commonly-used device fingerprinting data and how canvas fingerprinting is performed, through case study of real-world miniapps.
- **Identification of fingerprinting approaches and participants.** We found popular libraries and components commonly integrated by miniapp developers to perform user fingerprinting with access to permission-less data, and these findings are not limited to common ways to perform device fingerprinting and canvas fingerprinting, but also include evasive obfuscation and anti-fraud functionalities.
- **Publicized dataset to enable future research.** We will release our findings and datasets to the public so as to enable future research in a more comprehensive yet automated detection to safeguard the privacy of miniapp users.

2 BACKGROUND

2.1 Miniapp

Miniapps are a novel form of service integration that combines web technologies such as Markup Languages, Stylesheets, and JavaScript, into mobile apps, allowing developers to reuse the mobile app's core components to reduce development cost. These minaipps typically relies on the mobile app called super app, such as WeChat and Alipay. By utilizing these modules and user data provided by super apps, miniapps can provide a seamless user experience without requiring explicit installation or registration.

The exposed core modules offer miniapps great functionality. For instance, a miniapp can invoke `requestPayment()` to invoke the built-in payment component, `showShareMenu()` to access more information if a user shared the miniapp to a group chat, `getSystemInfo()` to obtain a set of device information, or `getPhoneNumber()` to obtain encrypted user phone number if the developer meets certain qualification requirement. The ability to reuse these functionalities while keeping the miniapp packages light-weight, as these complex components are reused instead of writing hundreds of lines of code.

2.2 What is Fingerprinting

In the era of mobile computing, device fingerprinting is commonly used to identify and track devices of a user based on device information, such as software versions, screen information, and hardware details. Compared with other tracking approaches such as cookies, fingerprinting is often covert but raises privacy issues, as they are collected without notifying the users, and thus users are often not aware of whether they are being tracked.

While fingerprinting is usually not as explicit as other explicit tracking approaches, it is still common and vital for business vendors as a form to enable security enhancement (e.g., fraud and anomaly detection, and monetization (e.g., advertisement and recommendation personalization). To achieve these, vendors may collect multiple sets of information, which can be categorized into the following categories.

Passive fingerprinting. A common approach to stealthily fingerprint users is to collect side-channel information that, when combined together, can uniquely identify a user-associated browser, device, or identity.

- **Device-based fingerprinting.** Device-based information is commonly used for fingerprinting. For web apps, when sending network requests, various metadata such as user agent is attached for the back-end server to identify the software version and system information, which includes browser type, version, and operating system information [6]. On top of these user agent information, additional information such as IP addresses, time zone, and list of installed fonts can be collected to increase the uniqueness of collected data [8]. Consequently, these information can be used for tracking a user's device.
- **Behavior-based fingerprinting.** On top of the information that directly represent partial device information, another line of work focuses on collecting data about a user's behavior of using a device. For instance, users' screen touching behaviors can be used to identify a user silently [5], and similar works on implicit continuous authentication based on motion sensor can be seamlessly applied to implement such user tracking [10, 12].

Active fingerprinting. Although fingerprinting generally do not involve active prompts informing users that they are being tracked, vendors attempting to fingerprint users may still actively perform certain activities that, although not necessarily appear to be relevant, can still collect device- or user- specific information.

- **Rendering-based fingerprinting.** As each devices' screen and hardware vary, applications can render certain contents in a canvas or using webGL to fingerprint a user by generating hash values [1]. For instance, the canvas can be used to display texts and pictures. However, the screen configuration for each device may still vary, even though the canvas is created in the same size and the fonts display the same content with the same font sizes. As such, if a screenshot is generated based on these contents, the hash value of these screenshots vary among each device. As such, the canvas can be used to fingerprint user devices.
- **Audio-based fingerprinting.** On top of the visual information, audio-related information can be used to determine a user, as the acoustic environment where each device is settle in are different. By playing certain audio and listening to the

echos from the built-in microphone, a website may be able to infer unique devices based on these information by just playing a sound as simple as a single triangle wave [7].

While these fingerprinting techniques may or may not require permission from the system to acquire necessary information, the vague implications of the fingerprinting behavior makes it hard for users to perceive. As such, these discussed approaches have been widely adopted by malicious website, especially phishing pages. According to a recent study on 1.7 million phishing pages [16], over 73% of phishing pages adopt at least three fingerprinting functions, with close to a quarter (24.6%) adopting more complex active fingerprinting, i.e., canvas, WebGL, and Audio-based fingerprinting.

2.3 The Miniapp Permission System

As resources made accessible to miniapps are not restricted to those managed by the underlying system, miniapps have to comply with an additional layer of permission system built by each super apps. As illustrated in recent research on miniapp security [26], major super apps manage differently on what type of data is subject to additional layer of permission. In general, the protection status of the accessible data falls into three categories:

Cat. I: protected by super apps. The most sensitive data is protected by an additional layer of permission mechanism where users will see an additional prompted dialog showing that the miniapp is accessing the specific type of data. These data include access to location data, camera, audio recorder, as well as user information that is registered to the super app.

Cat. II: inherited from underlying system. The data that is "less" sensitive may not be protected by the additional layer of permission system built by super apps, but may still be protected as the underlying system requires users to grant the permission. For example, although super apps such as Baidu do not explicitly require users to grant access to Bluetooth, the underlying Android system may still prompt the user to grant permission when Bluetooth is invoked if Baidu has not yet obtained the permission from Android.

Cat. III: not protected. In addition to data fetched from the underlying system, super apps may still provide miniapps with sensitive data users gave to the super app vendors. Although a part of these data such as user information is protected under category I, there still exists data that is not protected by certain platforms. For example, while Douyin (Chinese version of Tiktok) enforces additional permission for clipboard data, a majority of platforms do not enforce additional protection, allowing miniapps to access these data without prompting the users of the usage.

3 OVERVIEW

While many sensitive information is protected by the permission mechanism, there are still ways to fingerprint users with a combination of seemingly-less-sensitive data, such as device information, screen information, and the users' environments. As such, it creates a risk of permission-less fingerprinting, which is much more concerning because users being fingerprinted will not receive notifications nor see prompted dialogues about permission.

However, it is still non-trivial to detect such cases. As the first effort in the realm of miniapp fingerprinting, the first step of this research is to identify real-world cases performing permission-less fingerprinting practices, which requires systematic analysis and evaluation. To do so, in the following of this paper, we first analyze and list all the APIs with the potential to track users without having to obtain user permission, and then generate research insight of detection with a real-world example. Based on this insight, we develop detection tools based on the research insights to study these miniapps in the rest of the paper.

3.1 Permission-less fingerprinting

As discussed in this paper, while privacy data is protected by permission system, there are still APIs that can enable various user fingerprinting approaches that do not require users to authorize. As such, these APIs, when invoked to collect user related information, can enable permission-less fingerprinting, which is stealth and not visible to users, casusing severe potential privacy risks. To systematically identify these threats, we performed a comprehensive analysis on the APIs provided to WeChat miniapps, and identified 20 APIs that can be used to perform the four fingerprinting techniques as discussed in subsection 2.2.

As illustrated in Table 1, among the 19 APIs, 12 APIs are adopted whereas 7 are not. These APIs can be used in either device fingerprinting, audio fingerprinting, render-related fingerprinting and behavior-related fingerprinting. For instance, there are 11 APIs that can be used to collect device information such as screen width and device version. On the other hand, rendering-based fingerprinting can be performed by creating a canvas, draw certain texts and convert the screenshot into URL, from which a hash value of the screenshot can be generated to fingerprinting a device. However, as we focus on permission-less fingerprinting in this paper, we remove the 4 APIs that require specific permissions such as bluetooth and record (access to microphone) permission. Meanwhile, we did not include the behavior-related APIs because they involve collecting sensor information in the background. Unfortunately, during our preliminary experiment, WeChat reduces the sample rate and timeframe of sensors when they are in the background, making the collected information inaccurate for fingerprinting. As such, we consider the 12 APIs as listed in the table.

3.2 Motivating example

While we have summarized the APIs that can be utilized to collect informatino to fingerprint users, the real cases may be far more complex than simple sequential invocation, which makes it challenging for detection. As shown in Figure 1, a miniapp is incorporating a third-party script to collect information of user devices. However, instead of immediately invoking the APIs, the script first defines the methods (APIs) to be invoked, together with the parameters in the return values that will be used. Also, while the API getSystemInfo returns 33 parameters, only 11 are used.

Then, the actual collection behavior is triggered later in the main function, which is invoked when the miniapp is launched. These collected information are then constructed into a JSON object, which is further encoded, hashed, and attached to corresponding web

Type	API Name	Permission	Adopted
Device	wx.getSystemInfoSync	-	✓
	wx.getSystemInfoAsync	-	✓
	wx.getSystemInfo	-	✓
	wx.getWindowInfo	-	✓
	wx.getDeviceInfo	-	✓
	wx.getDeviceBenchmarkInfo	-	✓
	wx.getAppBaseInfo	-	✓
	wx.getBluetoothDevices	bluetooth	
	wx.getConnectedWifi	location	
	wx.getNetworkType	-	✓
	wx.getSystemSetting	-	✓
Audio	wx.startRecord	record	
	wx.joinVoIPChat	record	
Render	wx.canvasToTempFilePath	-	✓
	wx.canvasGetImageData	-	✓
	toDataURL	-	✓
Behavior	wx.startAccelerometer	-	
	wx.startCompass	-	
	wx.startGyroscope	-	

Table 1: List of fingerprinting-capable APIs, permission requirement, and whether adopted as permission-less fingerprinting

```
var e, t = getApp({
  allowDefault: !0
}).globalData.wxCookie, ...,
s = [{
    method: wx.getScreenBrightness,
    infos: [ [ "screenBrightness", "value" ] ]
  },{
    method: wx.getSystemInfo,
    infos: [ "brand", "model", "screenWidth", ... ]
  },{
    method: wx.getNetworkType,
    infos: [ "networkType" ]
} ],
a = t.getCookie("shshshfpa"),
r = t.getCookie("shshshfpb")...
```

Figure 1: An example of scripts collecting side-channel information for user tracking

requests. As such, the script can identify user device based on the system information, screen brightness, and network information. Interestingly, this module belongs to an online shopping miniapp, which involves subpages from the giant vendor JingDong. As such, it is possible that the behavior is deliberately made stealth to collect business data while avoiding regulation issues.

3.3 Methodology

Based on the motivating example, in this paper, we perform a preliminary analysis on the fingerprinting miniapps. It is worth noting that there has not yet been a dataset for miniapps confirmed to involve fingerprinting. As the first line of study, this paper specifically focuses on the miniapps with two criteria to minimize false positives of the results. First, the miniapp invokes APIs capable of performing permission-less fingerprinting, as listed in Table 1. Such invocation includes complex cases where listing of APIs and invocation of APIs are separated, such as the example shown in Figure 1. Second, to narrow down the scope of sampling for focused analysis, the miniapp has to display semantic characteristics explicitly related to fingerprinting. In this paper's case, we search for keyword "fingerprint" to reduce the scope. The results are then manually examined to ensure the correctness of the result. We hope that our results serves as the start point to facilitate future works of more comprehensive detection in the entire miniapp community.

4 IDENTIFYING FINGERPRINTING

During our analysis of motivating examples, we observed that despite that the collection of user fingerprintable data can be complex and the declaration and actual collection can be separated, the developers commonly list the collection APIs in batch. For instance, the API getSystemInfo and the parameters listed in the infos variable, including brand, model, screenWidth, are declared back-to-back within a single script, as shown in Figure 1. Similarly, recent research [2, 21] identified that fingerprinting scripts commonly access data in batch within a few scripts, enabling us to identify miniapps performing fingerprinting by counting the occurrences of access to fingerprintable data within a single script. Based on this observation, we develop the **M**iniapp **U**ser **F**ingerprinting **F**eature **I**dentification A**n**alyzer (MUFFIn), a light-weight multi-thread analysis tool to identify permission-less fingerprinting among miniapps, based on JavaScript analysis framework JAW [11].

Semantic Filtering. While we could apply the analysis directly on the total dataset of miniapps, as the first study in fingerprinting of miniapps, this research aims to prioritize the correctness of research insights and findings, which means that the results need to be examined by the researchers. As such, we reduce the total sampling size, and specifically focus on the miniapps that involve keywords of "fingerprint" in the script file. While we admit that this keyword is narrow, but it serves as an important step to facilitate future detection techniques that can be applied to all miniapps.

Dependency Graph Generation. The first step of the analysis is to generate the dependency graph for each script in the miniapp. During this process, JAW will analyze the Abstract Syntax Tree of the miniapp and to generate two separate fies: the nodes and the relationships. More specifically, the node files record node type (e.g., CallExpression or MemberExpression), location, value, and whether it is an AST node or a CFG node. Then, the relationship file records records data flow edges generated from the original code, including source and target node ID, relation type and arguments attached. As such, we can perform backward tracing on nodes.

Invocation Identification. The next step is to identify invocation of APIs and parameters of interest. To do so, we further analyzed the APIs listed in Table 1 and developed a mapping between APIs and the parameters in their return values that can be used for permission-less fingerprinting, spanning 112 parameters among 12 APIs. To find the starting point, we search through the node AST nodes to find invocations to interested APIs. If found, these APIs are added to the starting point.

Then, for each starting point node, we perform backward tracing to resolve the name of the API or the parameter. This may involve three conditions:

- **Function invocation.** If the type of the edge is invocation, it means that we identified a fingerprintable API, such as

Algorithm 1: Identify and generate data of fingerprinting API/param within a page

Input: *path_to_miniapp*, Analyzer, API_f, $param_f$
Output: pagedata

```
nodes,edges ← Analzer.generate(path_to_miniapp)
G ← generateGraph(nodes,edges)
to_analyze ← find_nodes_of_interest(G, API_f, param_f)
foreach (id, kind, value) in to_analyze do
    if kind = 'invoc' and backward_trace(G, id, kind, value) then
        pagedata.append({type: 'invoc', context: find_successor(G, id), value: value+"()"})
    end
    else if kind = 'param' and value in param_allowlist and backward_trace(G, id, kind, value) then
        pagedata.append({type: 'param', context: find_successor(G, id), value: value})
    end
    else if kind = 'elem' then
        declared ← array_graph_to_list(G, element_relationship, id, param_allowlist)
        if declared ≠ None then
            pagedata.append({type: 'elem', context: 'ArrayExpression', value: declared})
        end
    end
end
return pagedata
```

getSystemInfo. As such, we record the name of the API along with its context AST node type.

- **Parameter access.** If the type of the edge is parameter, we identify the access to a member of an object returned by fingerprinting API. This usually happen when a script invokes a fingerprinting API first, assigns the return value to a variable (e.g., `a`), and then access the member directly from the variable, e.g., `a.screenHeight`. As such, we record the parameter name and its context AST node.
- **Array element.** If the type is element, it hits the special case we identified, where the developers declare parameters to access in batch, in the form of an array. As illustrated in Figure 1, the script accesses the `screenBrightness` and `value` for the API `getScreenBrightness()`. If this happens, we resolve the values declared in the list separately, which will be introduced in the next paragraph.

Nested Declaration Processing. If a script separates declaration and access of fingerprintable parameter, a list containing parameters of interest will be declared. However, such list may be nested. For example, in Figure 1, the screenBrightness and value are declared in a nested list. Nevertheless, Fingerprint-Finder needs to output a sequential list of function invocation, access to parameter, and elements declared. As such, we need to flatten the nested declarations. To do so, we identify each of these ArrayExpression, and traverse the child nodes in a depth-first manner. If we encounter an array when traversing through the list element, we first resolve the contents in this sub-list, and then continue the traversal. As such, we generate a sequential list of elements declared in the list.

5 RESULTS

5.1 Execution

We executed our experiment on 4.03 million miniapps in two rounds. In the first round, we perform semantic matching of the miniapps related to fingerprinting by obtaining the package file, reading the binary file and looking for any keywords matching fingerprinting, case insensitive. This results in 1,367 miniapps. Then, in the second round, we apply our MUFFIn analyzer on the filtered list of miniapps. The entire process utilizes a server with 16 Xeon 4314 CPU core and 64 GB memory, which takes around one and half a month to finish. MUFFIn processes each miniapps for 85.96 seonds on average, identifying 1,310 miniapps that actually invoke fingerprint-related APIs. We sampled the 57 miniapps and found that part of the reason for these cases not related to fingerprintable API is because they implement unlock with user fingerprint (i.e., the actual finger unlock), which is related to user biometric unlocking instead of user fingerprinting. As such, MUFFIn is able to filter out these irrelevant cases. In the rest of the paper, we will discuss in details how we group the miniapps based on similar scripts and sample for identifying real-world cases to understand the ecosystem.

5.2 Families

Upon analyzing the miniapps, we first identify a list for each scripts found to be involved in user fingerprinting APIs. Then, we generate a hash based on the list of script paths. This procedure yields 285 families in total, whose size ranges from 110 to 1. As shown in Table 2, we show the top 10 popular families involved with fingerprinting. We also categorized the data that can be used for fingerprinting user into 9 categories: authorized permission list, basic information about device, benchmarking ratings, canvas-fingerprintable data, enabled peripheral features, hardware information, network information, screen information, and user settings.

We discover that permission authorization list, enabled features and hardware information is not accessed by any of the popular families, but a majority of popular families access 100% of the total basic information and network information. Also all popular families access from 43% to 86% subtypes out of the total accessible types of screen information, and more than half of these cluster access benchmark information. As such, we observe that basic and screen information is commonly used to perform user fingerprinting. On top of that, we are surprised to find that 8 out of 10 popular families utilize canvas information to fingerprint user device.

5.3 Case studies

To further understand the ecosystem, we perform additional case studies based on the families we generated. More specifically, we

Family #	Family Size	Fingerprintable Data									# Related Pages
		Authorization	Basic	Benchmark	Canvas	Enabled Features	Hardware	Network	Screen	User Setting	
1	110	0.00%	100.00%	100.00%	100.00%	0.00%	0.00%	100.00%	85.71%	50.00%	10
2	75	0.00%	100.00%	100.00%	100.00%	0.00%	0.00%	100.00%	85.71%	50.00%	26
3	64	0.00%	100.00%	100.00%	100.00%	0.00%	0.00%	100.00%	85.71%	50.00%	25
4	62	0.00%	100.00%	100.00%	100.00%	0.00%	0.00%	100.00%	85.71%	50.00%	27
5	51	0.00%	25.00%	0.00%	100.00%	0.00%	0.00%	100.00%	42.86%	12.50%	7
6	45	0.00%	100.00%	0.00%	100.00%	0.00%	0.00%	100.00%	85.71%	50.00%	19
7	38	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%	0.00%	12.50%	1
8	36	0.00%	100.00%	100.00%	100.00%	0.00%	0.00%	100.00%	85.71%	50.00%	28
9	25	0.00%	100.00%	100.00%	100.00%	0.00%	0.00%	100.00%	85.71%	50.00%	26
10	25	0.00%	100.00%	0.00%	0.00%	0.00%	0.00%	100.00%	42.86%	37.50%	12

Table 2: Top 10 families by size

```
1  <!--fmSDK/fm.min-1.0.3.4.js-->
2  wx.getSystemInfoSync().SDKVersion, "1.9.9") >= 0 &&
3  i = setInterval(function() {
4    ...
5    r = wx.createCanvasContext(h, this);
6    // Draw rectangle
7    r.rect(0, 0, 10, 10), r.textBaseline = "alphabetic",
8    r.fillStyle = "#f60", r.fillRect(32, 1, 62, 20),
9    // Print text
10   r.font = "18pt Arial",
11   r.fillText("Cwwm aa fjorddbank", 4, 45),
12   r.fillStyle = "rgb(255,0,255)",
13   // Draw arc
14   r.beginPath(), r.arc(16, 16, 16, 0, 2 * Math.PI, !0),
     ↪ r.closePath(),
15   ...
16   r.draw(!1, function() {
17   wx.canvasToTempFilePath({
18   canvasId: h, x: 0, y: 0, ...
19   fileType: "png",
20   success: function(n) {
21     n.tempFilePath;
22     try {
23       a.drawt = new Date().getTime();
24       var i = wx.getFileSystemManager()
25       .readFileSync(n.tempFilePath, "base64");
26       d = c.hexMD5(i), a.md5t = new Date().getTime();
27     } catch (t) {...}
28   },...});
29   });
30 , 100}
```

Figure 2: An example of canvas fingerprinting. The case draws shapes on the canvas, and writes the canvas to a local file, which is used to generate MD5 hash and timestamp.

would like to analyze both the most popular families and the least popular families, so as to provide a thorough understanding between the most common practices and the more fragmented practices. As such, based on the clustering results, we first sample one miniapp out of each top-30 popular families, and then sample 20 miniapps out of each low-popularity families, thus forming a dataset consisting of 50 miniapps. Upon examination, we observed that there is a predominant adoption of a few miniapp libraries when performing user fingerprinting, most of which are for reporting business analytics to the vendors. Meanwhile, there are additional behaviors, such as antifraud, canvas fingerprinting to augment the accuracy of the fingerprinting results. In the end, we even discovered cases where miniapps actively attempt to obfuscate the operations to canvas, potentially trying to escape from the regulation from the platforms.

Device fingerprinting. During our investigation, we found that device-based fingerprinting is common among miniapps, which is mainly used for reporting user-related analytics to the vendors providing miniapp services. For example, the motivating example we displayed in this paper comes from a widely-adopted template called the JingDong Miniprogram Open Platform, which involves 16 out of 30 most popular cases we have sampled. This component set allows developers to utilize the marketing and transaction chain from JingDong and deploy their own businesses in WeChat ecosystem, which can significantly reduce the cost of development. Similarly, Youzan [28], a open platform providing Platform-as-a-service (PaaS) for shop owners to deploy their own miniapp stores by customizing templates, is also commonly used in the most popular families.

On top of that, we identify multiple active libraries that are commonly used for user analytics. For instance, in the JingDong template, we identified a library named `Kepler` [3], which is used to perform data agnostic. Other than that, we found 3 families using `ald-stat` [17], 2 family using `sentry` [18], and 3 families using components from the WeChat miniapp store which incorporates similar logics. For the case in Youzan, the fingerprinting logics are bundled in a file named `vendor.js`, which is a common practice for miniapp developers to integrate external libraries. Other than that, for the 20 fragmented miniapp families, we found 2 additional families using `raven` [4] to perform such fingerprinting. As such, we conclude that the fingerprinting behaviors are commonly invoked in the form of external libraries instead of being implemented by individual developers, potentially due to the complexity of developing fingerprinting functionalities.

Canvas fingerprinting. On top of the common cases where miniapps utilize libraries to collect information to facilitate device fingerprinting, we found that miniapps commonly perform canvas fingerprinting by displaying contents on the user's devices and export the files or convert the canvas image to URLs, which is accessible in the future. To our surprise, we even found a case that explicitly save the rendered canvas and generate md5 hash, which is a typical pattern for canvas fingerprinting. As illustrated in Figure 2, the code in fmSDK creates a canvas in line 5, which draws two rectangles, a line of text, and an arc. Then, it generated the canvas into a temporary file in line 22 as a png image. After that, the file is read into memory, and a md5 hash together with the timestamp is attached to the file. As such, the miniapp is able to fingerprint a unique device based on the hash value generated from the canvas information.

```
1  <!--matters/barlas.js-->
2  var t =
   ↪ wx.getFileSystemManager().readFileSync("/txt/barlas.txt",
   ↪ "utf-8")
3  var n = JSON.parse(t)
4  s = function(e, t) {
5      return n[e -= 0];
6  }
7  g.addEventListener(window, s(28), n), g[s(26)](window, s(29),
   ↪ n), function(t, n, i) {
8    ...
9    getCanvasFingerprint: function() {
10     var e = document.createElement(s(24)),
11     t = e.getContext("2d");
12     return ...,
13     t[s(139)] = s(144),
       ↪ t[s(143)]("https://www.talkingdata.com", 4, 17),
       ↪ e[s(145)]();
14     //
15     t["fillStyle"] = rgba(102,204,0,0.7),
       ↪ t["fillText"]("https://www.talkingdata.com", 4, 17),
       ↪ e["toDataURL"]();
16   }
17 }
18 <!--matters/barlas.txt-->
19 [... "savnac", ...        // 24th canvas
20 "elytSllif", ...          // 139th fillStyle
21 "txeTllif", ...           // 143th fillText
22 "LRUataDot",              // 145th toDataURL ...]
```

Figure 3: A case that obfuscates the canvas fingerprinting behavior with reversed texts. Due to space limit, only important strings are shown from line 18.

Self-implemented Obfuscation. In our sampled miniapps, it is surprising that there are miniapps that deeply obfuscate the canvas fingerprinting behavior by hiding reverse-ordered code inside text files, potentially attempting to evade the vetting or regulation from the platform. For example, as shown in the case in Figure 3, the miniapp involves two scripts: the `barlas.js` and the `barlas.txt`. When performing API detection and semantic search of fingerprint, neither of the two files involve related keywords, and thus the miniapp will remain undetected. However, it encapsulates a function in line 4, which extracts the corresponding text from the file in a reversed manner. Meanwhile, it edits the prototype of the object `t` to invoke canvas-related APIs. For instance, the code in line 14 actually edits the `fillStyle` and `fillText`, and then invoke the API `wx.toDataURL()`, a critical API to facilitate canvas fingerprinting.

Interestingly, the script displays an URL, which enable us to dig further more about the provider. This domain belongs to a company called "TalkingData" [20], which provides marketting solutions for business owners. This company has a set of SDKs which involves functionalities to fingerprint users for reporting user analytics, similar to the aforementioned libraries.

Anti-fraud. Despite being commonly used in tracking users, we have found legitimate banks attempting to collect fingerprintable data for anti-fraud functionalities. For instance, we found that a miniapp integrates a file `fingerprint2_x.js` as a library that obtains the system information, screen information, canvas information, etc, to generate fingerprints for a user. This library is invoked by a script called `antiFraud.js`, which is invoked in pages where sensitive operations are involved, such as changing phone numbers, binding new cards, and resetting passwords.

6 DISCUSSIONS

Takeaways. In this paper, we developed Fingerprint-Finder, an analysis tool to identify fingerprinting miniapps based on semantic filtering and static analysis based on API identification. We performed a large-scale analysis and found 1,310 miniapps involved in such behavior, During our case studies, we identify that basic information, benchmarking scores, and screen information are commonly used to fingerprint users. Canvas fingerprinting may also be commonly adopted. In this ecosystem, there are not only miniapp template platforms and components, but also third-party libraries that perform user fingerprinting to provide vendors with business analytic data. There are even cases where individually-implemented libraries attempting to adopt obfuscation to evade through the vetting process.

Limitation and Future Work. In this paper, we strive to provide a preliminary fingerprinting miniapp dataset, thus we limit the scope by strict semantic matching, resulting in a relatively small dataset consisting of 1,310 miniapps. Also, we specifically focus on simple access to fingerprintable data in this research. Future works can improve the scope of the analysis, identify more complex fingerprinting cases, and attribute the fingerprinting behaviors so as to provide more research insight for the community as well as platforms to better protect the privacy of users.

Research Ethics and Open Policy. We practice caution during our experiment to minimalize ethics issues. First, the experiment operates on downloaded miniapp packages, where the analysis does not influence the deployed miniapps. Second, we have reported our findings to the platform, and we are working with these platforms to improve our framework. On top of that, this paper follows common open science practices, and our artifacts will be released to the public once published to facilitate future research.

7 RELATED WORK

User Fingerprinting. The issue of user fingerprinting has been studied over the past decade in mobile and web security domain. For instance, there has been multiple research [1, 2, 16, 21] that summarize the threat of stealth fingerprinting in browsers. Additionally, recent works [5, 10, 12] have shown the capabilities for sensor data to be utilized to infer users who is unlocking or using a mobile device. On top of that, the possibility for audio fingerpinting [7], and how canvas information can be utilized to perform user tracking [1, 9, 13], have been discussed in the past decades.

Miniapp Security. Recent works on miniapp security spans three major direction. On one hand, recent works have been focusing on identifying vulnerabilities exploitable to malicious developers against the super app platform [22, 23, 25, 31] and individual miniapps [19, 26, 29]. Additionally, another line of work focus on identifying approaches enabling malware to sneak through the ecosystem, thereby causing harm to the privacy and security of users [15, 27]. The last line of work focuses on the privacy practice of information sharing between miniapps and the platform [14, 24, 30]. Compared with these existing works, this paper presents the first study to understand how and why miniapps may perform fingerprinting on users that can bypass the permission system built by

the super apps to protect user privacy, which can cause privacy issues on users if misused by minaipp developers.

8 CONCLUSION

Our study presents the first analysis of permission-less user fingerprinting issues among miniapps. We developed FingerprintFinder to detect and cluster miniapps that collect permission-less data for fingerprinting, identifying 1,310 cases in 285 families from a dataset of over 4.03 million miniapps. Our analysis uncovered three common categories of fingerprintable data, widespread use of shared components and libraries, and even obfuscation techniques employed to evade detection. These findings expose critical privacy challenges and underscore the need for more rigorous controls over seemingly innocuous data access. By releasing our dataset and insights, we aim to support future research and platform governance efforts to enhance user privacy in the miniapp landscape.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful feedbacks. This research was supported in part by NSF award 2330264. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the NSF.

REFERENCES

[1] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. The web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 674–689. ACM, 2014.

[2] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. Fpdetective: Dusting the web for fingerprinters. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*, pages 1129–1140, New York, NY, USA, November 2013. Association for Computing Machinery, Inc.

[3] Kepler Analytics. Kepler analytics: Retail sales improvement & network optimisation. https://kepleranalytics.com, 2025. Accessed: 2025-06-12.

[4] Raven Analytics. Raven analytics: Uncover hidden causation, no code required. https://www.ravenanalytics.org, 2025. Accessed: 2025-06-12.

[5] Cheng Bo, Lan Zhang, Xiang-Yang Li, Zhenjiang Huang, and Yu Wang. Silentsense: Silent user identification via dynamics of touch and movement behavioral biometrics. In *Proceedings of the 19th Annual International Conference on Mobile Computing & Networking*, pages 187–190. ACM, 2013.

[6] BrowserScan. Browser fingerprints 101: Useragent, 2023. Accessed: 2025-06-04.

[7] Shekhar Chalise, Hoang Dai Nguyen, and Phani Vadrevu. Your speaker or my snooper? measuring the effectiveness of web audio browser fingerprints. In *Proceedings of the 22nd ACM Internet Measurement Conference*, IMC '22, page 349–357, New York, NY, USA, 2022. Association for Computing Machinery.

[8] Clearcode. Device fingerprinting: What it is and how does it work?, 2016. Accessed: 2025-06-04.

[9] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1388–1401, New York, NY, USA, 2016. Association for Computing Machinery.

[10] Mario Frank, Ralf Biedert, Eugene Ma, Ivan Martinovic, and Dawn Song. Touchalytics: On the applicability of touchscreen input as a behavioral biometric for continuous authentication. *IEEE Transactions on Information Forensics and Security*, 8(1):136–148, 2013.

[11] Sokheil Khodayari. Jaw: A graph-based security analysis framework for client-side javascript, 2025. Accessed: 2025-07-04.

[12] Wei-Han Lee and Ruby B. Lee. Implicit smartphone user authentication with sensors and contextual machine learning. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 297–308, 2017.

[13] Ada Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[14] Shuai Li, Zhemin Yang, Yunteng Yang, Dingyi Liu, and Min Yang. Identifying cross-user privacy leakage in mobile mini-apps at a large scale. *IEEE Transactions on Information Forensics and Security*, 19:3135–3147, 2024.

[15] Haoran Lu, Luyi Xing, Yue Xiao, Yifan Zhang, Xiaojing Liao, XiaoFeng Wang, and Xueqiang 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*, pages 569–585, 2020.

[16] Iskander Sanchez-Rola, Leyla Bilge, Davide Balzarotti, Armin Buescher, and Petros Efstathopoulos. Rods with laser beams: Understanding browser fingerprinting on phishing pages. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4157–4173, Anaheim, CA, August 2023. USENIX Association.

[17] Sentry. aldwx. https://github.com/aldwx/aldSdk, 2025. Accessed: 2025-06-12.

[18] Sentry. Sentry: Application performance monitoring & error tracking software. https://sentry.io, 2025. Accessed: 2025-06-12.

[19] Yizhe Shi, Zhemin Yang, Kangwei Zhong, Guangliang Yang, Yifan Yang, Xiaohan Zhang, and Min Yang. The skeleton keys: A large scale analysis of credential leakage in mini-apps. In *32nd Network and Distributed Systems Security Symposium (NDSS)*, 2025.

[20] TalkingData. Talkingdata - mobile, data, value. https://www.talkingdata.com/, 2025. Accessed: 2025-06-12.

[21] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. Fp-stalker: Tracking browser fingerprint evolutions. In *2018 IEEE Symposium on Security and Privacy*, pages 728–741, San Francisco, CA, USA, May 2018. IEEE.

[22] Chao Wang, Yue Zhang, and Zhiqiang Lin. Uncovering and exploiting hidden apis in mobile super apps, 2023.

[23] Chao Wang, Yue Zhang, and Zhiqiang Lin. Root free attacks: Exploiting mobile platform's super apps from desktop. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '24, pages 830–842, New York, NY, USA, 2024. Association for Computing Machinery.

[24] Yin Wang, Ming Fan, Junfeng Liu, Junjie Tao, Wuxia Jin, Qi Xiong, Yuhao Liu, Qinghua Zheng, and Ting Liu. Do as you say: Consistency detection of data practice in program code and privacy policy in mini-app, 2023.

[25] Yuqing Yang, Chao Wang, Yue Zhang, and Zhiqiang Lin. Sok: Decoding the super app enigma: The security mechanisms, threats, and trade-offs in os-alike apps. *arXiv preprint arXiv:2306.07495*, 2023.

[26] Yuqing Yang, Yue Zhang, and Zhiqiang Lin. Cross miniapp request forgery: Root causes, attacks, and vulnerability detection. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 3079–3092, 2022.

[27] Yuqing Yang, Yue Zhang, and Zhiqiang Lin. Understanding the miniapp malware: Identification, dissection, and characterization. In *32nd Network and Distributed Systems Security Symposium (NDSS)*, 2025.

[28] Youzan. A better business with youzan. https://ir.youzan.com/en, 2025. Accessed: 2025-06-12.

[29] Yue Zhang, Yuqing Yang, and Zhiqiang Lin. Don't leak your keys: Understanding, measuring, and exploiting the appsecret leaks in mini-programs. *arXiv preprint arXiv:2306.08151*, 2023.

[30] Zhibo Zhang, Lei Zhang, Guangliang Yang, Yanjun Chen, Jiahao Xu, and Min Yang. The dark forest: Understanding security risks of cross-party delegated resources in mobile app-in-app ecosystems. *IEEE Transactions on Information Forensics and Security*, 19:5434–5448, 2024.

[31] Zhibo Zhang, Zhangyue Zhang, Keke Lian, Guangliang Yang, Lei Zhang, Yuan Zhang, and Min Yang. Trusteddomain compromise attack in app-in-app ecosystems. In *Proceedings of the 2023 ACM Workshop on Secure and Trustworthy Superapps*, SaTS '23, page 51–57, New York, NY, USA, 2023. Association for Computing Machinery.