Demystifying Cookie Sharing Risks in WebView-based Mobile App-in-app Ecosystems

Miao Zhang*✉1, Shenao Wang*†2, Guilin Zheng[1], Yanjie Zhao†2, Haoyu Wang†✉2
[1] Beijing University of Posts and Telecommunications, China
[2] †Huazhong University of Science and Technology, China

***Abstract*—Mini-programs, an emerging mobile application paradigm within super-apps, offer a seamless and installation-free experience. However, the adoption of the web-view component has disrupted their isolation mechanisms, exposing new attack surfaces and vulnerabilities. In this paper, we introduce a novel vulnerability called Cross Mini-program Cookie Sharing (CMCS), which arises from the shared web-view environment across mini-programs. This vulnerability allows unauthorized data exchange across mini-programs by enabling one mini-program to access cookies set by another within the same web-view context, violating isolation principles. As a preliminary step, we analyzed the web-view mechanisms of four major platforms, including WeChat, AliPay, TikTok, and Baidu, and found that all of them are affected by CMCS vulnerabilities. These findings were responsibly disclosed and acknowledged with two CVEs. Furthermore, we demonstrate the collusion attack enabled by CMCS, where privileged mini-programs exfiltrate sensitive user data via cookies accessible to unprivileged mini-programs. To measure the impact of collusion attacks enabled by CMCS vulnerabilities in the wild, we developed MICOSCAN, a static analysis tool that detects mini-programs affected by CMCS vulnerabilities. MICOSCAN employs *web-view context modeling* to identify clusters of mini-programs sharing the same web-view domain and *cross-webview data flow analysis* to detect sensitive data transmissions to/from web-views. Using MICOSCAN, we conducted a large-scale analysis of 351,483 mini-programs, identifying 45,448 clusters sharing web-view domains, 7,965 instances of privileged data transmission, and 9,877 mini-programs vulnerable to collusion attacks. Our findings highlight the widespread prevalence and significant security risks posed by CMCS vulnerabilities, underscoring the urgent need for improved isolation mechanisms in mini-program ecosystems.**

I. INTRODUCTION

A new mobile application paradigm, referred to as "app-in-app"[1] or mini-programs[2], [3], [4], has gained significant traction and popularity over the past few years. Under this paradigm, a host app or super-app runs a suite of sub-apps as in-app components through a JavaScript engine. This compact paradigm offers users a seamless and installation-free experience, eschewing the traditional process of downloading standalone mobile applications [5], [6]. Within the super-app environment, users can access a diverse range of functionalities, such as shopping, dining, and transportation services, without leaving the host application. This integrated and cohesive experience enhances user engagement and stickiness. As a result, mini-programs have rapidly emerged as a popular and lightweight alternative to conventional mobile application development [7], [8]. Currently, the most prominent and representative super-app, WeChat, boasts an ecosystem of over 3.5 million mini-programs, attracting over 600 million active users on a daily basis [9].

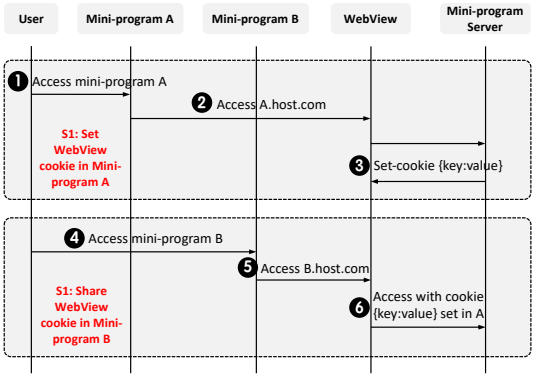

Fig. 1: Cross Mini-program Cookie Sharing via Web-View.

As super-apps increasingly resemble operating systems [10], they are tasked with providing resource management [11], [12], permission control [5], [13], and security mechanisms [14], [15], [16] for the hosted mini-programs. Similar to traditional Android applications [17], mini-programs employ isolation mechanisms [18], [10], [19] to ensure security and prevent unauthorized data sharing. This isolation model aims to create distinct sandboxes for each mini-program, restricting their access to system resources and user data based on predefined permissions and policies. However, the introduction of the web-view component [20] has disrupted the isolation model, introducing new attack surfaces and potential vulnerabilities. When mini-programs leverage the web-view component to render web content or access web resources, they inherit the cookies associated with that particular web-view instance. This inheritance mechanism, while intended for legitimate purposes such as maintaining user sessions, inadvertently opens a channel for unauthorized data exchange between mini-programs.

In this paper, we introduce a new type of vulnerability, called **Cross Mini-program Cookie Sharing (CMCS)**. As

*Both authors contributed equally to this research.

✉Miao Zhang (zhangmiao@bupt.edu.cn) and Haoyu Wang (haoyuwang@hust.edu.cn) are the corresponding authors.

†The full name of the affiliation is Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00142

illustrated in Figure 1, the vulnerability arises from the shared web-view environment among mini-programs. When mini-program A accesses a web resource (e.g., `A.host.com`) through the web-view component, it sets cookies within the current context (❶-❸). Subsequently, when mini-program B accesses a different web resource (e.g., `B.host.com`) through the same web-view instance, it inherits and can access the cookies previously set by mini-program A (❹-❻). This unintended sharing of cookies across mini-programs within the same web-view context violates the isolation principles and enables unauthorized data exchange. As the preliminary analysis in our study, we investigated the web-view mechanisms of four major mini-program platforms, including WeChat, AliPay, TikTok, and Baidu, to assess their susceptibility to CMCS vulnerabilities. Our findings reveal that all four platforms are affected due to their shared web-view environments. We responsibly disclosed these vulnerabilities, and our efforts were acknowledged with 2 CVEs.

Specifically, based on the current implementations of web-view cookies, we demonstrate a type of attack enabled by CMCS, referred to as *Collusion Attacks*. *Collusion Attacks* involve a privileged mini-program obtaining sensitive user data by abusing its granted permissions. The privileged mini-program then transmits this sensitive data to the web-view and stores it in the web-view cookies. These cookies can subsequently be accessed by an unprivileged, colluding mini-program, effectively allowing the unauthorized sharing of privileged data that the latter should not have access to.

To measure the potential impact of collusion attacks enabled CMCS vulnerabilities in the wild, we have developed MiCoScan, a static analysis tool designed to detect vulnerable mini-programs. MiCoScan employs a three-step approach: First, MiCoScan performs *web-view context modeling* by examining the URLs accessed through web-view components across different mini-programs. Mini-programs that interact with the same web-view domain are grouped into clusters, as these groupings represent contexts where cookies may be shared, thereby exposing them to potential vulnerabilities. Next, MiCoScan introduces *cross-webview data flow analysis* to inspect the bi-directional communication channels between mini-programs and their associated web-view components. It detects instances where privileged data, such as user contacts, location, or other sensitive information, is transmitted from mini-programs to the web-view environment. Such transmissions increase the likelihood of privileged data being stored in web-view cookies, where it becomes accessible to other mini-programs within the same context. Finally, by integrating web-view context modeling with privileged data transmission inspection, MiCoScan effectively identifies mini-programs susceptible to CMCS vulnerabilities.

Based on MiCoScan, we conducted a large-scale, systematic analysis of 351,483 real-world mini-programs, offering the first comprehensive measurement of CMCS vulnerabilities in the wild. Our investigation identified 45,448 unique clusters of mini-programs accessing the same web-view domain. Further examination revealed 7,849 instances of mini-programs transmitting privileged data (such as contacts, location, and authentication tokens) to web-views and 116 instances of mini-programs receiving privileged data from web-views, indicating bi-directional data flows that compromise isolation. Through systematic analysis of the associated cookies, we identified 9,877 mini-programs vulnerable to collusion attacks, highlighting the widespread impact of this issue.

Contributions. The contributions are outlined as follows:

- **Novel Attack Vectors (§ III).** We are the first to systematically study the isolation mechanisms in mini-programs. Our research uncovered the CMCS vulnerability in four major platforms, including WeChat, AliPay, TikTok, and Baidu, which enables unauthorized data sharing across mini-programs. These findings have been responsibly disclosed and acknowledged with 2 CVEs.
- **Practical Detection (§ IV).** We present MiCoScan, a static analysis tool that detects potential CMCS vulnerabilities through. The web-view context modeling identifies clusters sharing the same web-view domain as potential candidates. The cross-webview data flow analysis examines bi-directional communication channels to uncover sensitive data transmission.
- **Large-scale Measurement (§ VI).** We conduct a large-scale study of 351,483 mini-programs, revealing the widespread prevalence of the CMCS vulnerability. Our analysis identifies 45,448 clusters of mini-programs sharing web-view domains and 7,965 instances of privileged data transmission to/from web-views, leading to the discovery of 9,877 vulnerable mini-programs.

II. Background

A. Mini-programs

Architecture of Mini-programs. A mini-program typically comprises two integral components: the front end, executing within the host program, and the back end, operating on the server to provide essential services. Similar to how Android necessitates an APK for application usage, the host program must acquire the `WXAPKG` package before launching the mini-program. As illustrated in Figure 2, this package contains crucial resources for the mini-program, including `WXML` files, `WXSS` files, and `JavaScript` files. `WXML` files, akin to `HTML`, define the UI interface structure, while `WXSS` files, resembling `CSS`, dictate the UI's style. `JavaScript` files are responsible for implementing the interactive logic, managing user interactions, handling network requests, and more. Additionally, the package may include various resource files, such as images.

Web-View Components. The web-view component is provided by WeChat for developers to introduce HTML5 (H5 for short) pages. It contains four attributes: `src`, `bindmessage`, `bindload`, and `binderror`. The `src` attribute points to the embedded URL of web-view pages, which needs to be configured in the business domain name of the mini-program's backend. The remaining three attributes serve as event handlers: The `bindmessage` attribute is used to bind

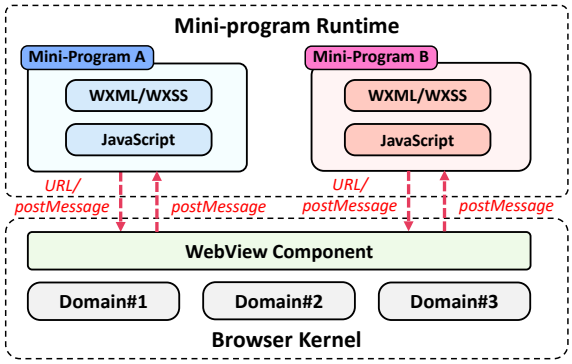

Fig. 2: Architecture of Mini-programs.

an event handler for receiving messages sent from the embedded web-view page to the mini-program, while `bindload` and `binderror` are callback functions triggered when the web page loads successfully or fails to load.

Bi-direction Communication. One of the key features of the web-view component is its ability to facilitate bi-directional communication between the mini-program and the embedded web-view page, as illustrated in Figure 2. This communication is primarily managed through the `postMessage` API and the `bindmessage` attribute.

- **C#1: Mini-program→Web-View.** Two prevalent methods are transmitting information through URL parameters and using the `postMessage` API: 1) One straightforward approach involves attaching the required data as query parameters to the URL that the Web-View component loads. When the web-view loads the URL, JavaScript within the web page can extract these parameters. 2) A more sophisticated and flexible approach is to use the `postMessage` API. This involves creating a web-view context using `createWebViewContext` and then invoking the `postMessage` method to send messages. The web-view page can listen for these messages using the `onMessage` event listener.
- **C#2: Web-View→Mini-program.** The `postMessage` also allows the web-view page to send messages to the mini-program. On the mini-program side, the `bindmessage` attribute is used to define an event handler that listens for these messages. When the web-view page sends a message, the event handler in the mini-program is triggered, allowing the mini-program to process the message content and respond accordingly.

B. Security Model and Isolation

The security model of mini-programs is a fundamental aspect that ensures the integrity, privacy, and stability of the host application and its contained mini-programs. This model is built upon a robust isolation mechanism that segregates each mini-program into an independent operational environment. This segregation prevents unauthorized access and interference between mini-programs and protects the host application from potential security breaches. The specific implementation of this technology includes:

- **Sandbox Technology.** Sandbox technology [18] confines the operations of a mini-program within a controllable security sandbox. This restriction limits the mini-program's interface calling permissions, allowing access only to interfaces provided by the host application. This containment ensures that mini-programs can only interact with permitted system components, enhancing security.
- **Independent Processes.** Each mini-program in the host application operates as a separate process [19]. During execution, each mini-program functions as an individual process, ensuring that any abnormalities or crashes within one mini-program do not impact the operation of others. This process isolation enhances the stability and reliability of the entire system.
- **Resource Isolation.** The host application allocates an independent resource directory for each mini-program [18]. Resources saved by each mini-program are stored in isolated directories, ensuring they are not shared with or accessible to other mini-programs. This resource isolation maintains the independence of each mini-program's data.

C. Cookie Mechanism

In the context of mini-programs and web-view components, the cookie mechanism is crucial for ensuring seamless user experiences and consistent data management.

Cookies in Mini-Programs. Unlike traditional web applications, mini-programs operate within a controlled environment managed by the host application. The cookie mechanism in mini-programs is tailored to this unique context, emphasizing isolation and security. Each mini-program operates within its own sandboxed environment. This means that cookies set by one mini-program are stored in a separate storage space that is inaccessible to other mini-programs. In addition to isolating cookies between mini-programs, there is also a clear separation between cookies used by mini-programs and those used by the host application. This ensures that cookies set by mini-programs do not interfere with the host's data and vice versa.

Cookies in Web-View Components. Web-view components serve as embedded browser engines within mini-programs, enabling the display of web content. The cookie mechanism in web-view components mirrors that of traditional web browsers. Contrary to the isolation seen in many aspects of mini-program architecture, the cookies managed within web-view components and the host application are not isolated. Instead, they share the same embedded browser instance, leading to shared cookie storage (which we will further discuss in § III-B). This shared environment introduces a series of privacy and security risks that must be carefully managed[1].

III. The CMCS Attacks

While the host application offers various methods to isolate mini-programs, the introduction of the web-view component

[1]Our primary focus in this study will be on the cookies used within web-view components unless otherwise specified.

disrupts the original security model. Notably, we observed that when external web pages within the web-view components of two distinct mini-programs belong to the same domain, these two web pages, belonging to different mini-programs, can share cookies during runtime. Now we present the CMCS attacks, whereby an attacker can exploit this vulnerability to facilitate the collusion attacks.

A. Threat Model and Scope

Assumption. As illustrated in Figure 1, cross mini-program cookie sharing involves multiple parties, including the host application (e.g., WeChat), and both the cookie-setting mini-program and the cookie-receiving mini-program, as well as their respective front-ends and back-ends. Our study of CMCS attacks is based on several key assumptions. First, we assume that the host application permits multiple mini-programs to embed web-view components, which can load and render external webpages. This setup is critical for enabling the interaction between different mini-programs via shared web-view components. Second, we assume that these web-view components do not enforce strict isolation regarding cookie storage between different mini-programs, particularly when the webpages belong to the same domain[2]. This lack of isolation allows cookies to be shared across mini-programs, creating a potential vector for attacks. The validity of these assumptions will be substantiated in § III-B.

The Attacker's Capability. Given these assumptions, the attacker possesses specific capabilities primarily within the scope of the mini-program ecosystem. The attacker can develop and distribute a malicious mini-program within the host application's ecosystem. This malicious mini-program is designed to load a webpage from a domain under the attacker's control. The attacker's main capability lies in their control over the content and behavior of the webpage hosted on their domain. This capability includes the ability to read, write, and manipulate cookies within the web-view component of the malicious mini-program. Consequently, the attacker can leverage shared cookies to track user activity across different mini-programs, thereby breaching user privacy and enabling cross-mini-program tracking. The scope of our threat model is confined to the interaction between mini-programs and web-view components within a single host application. We focus on the security implications of cookie sharing, particularly how it can lead to privacy breaches and unauthorized tracking of users. This threat model does not encompass other forms of inter-process communication or potential vulnerabilities outside the realm of web-view component interactions.

Scope. The app-in-app ecosystem has gained significant attention in recent years. To illustrate the existence and applicability of cookie sharing across different super-app platforms, we conducted a preliminary proof-of-concept study on four major platforms: WeChat, Alipay, Baidu, and TikTok. This initial analysis focused on evaluating the web-view mechanisms of these platforms to confirm their susceptibility to CMCS vulnerabilities. Building on these findings, our subsequent measurement study specifically targets WeChat. As a pioneer in the mini-program paradigm, WeChat hosts over 3.5 million mini-programs [9], making it an ideal candidate for large-scale analysis to evaluate the real-world impact of CMCS vulnerabilities. It is important to note that while our measurement is confined to WeChat, the standardized architecture [3] of mini-program ecosystems ensures that our proposed methods are scalable and transferable to other platforms.

[2]It is worth noting that the Same-Origin Policy (SOP) is not a strong security assumption in this context, as many mini-programs are developed based on frameworks or hosted by third-parties, making the presence of third-party domains within mini-programs highly common.

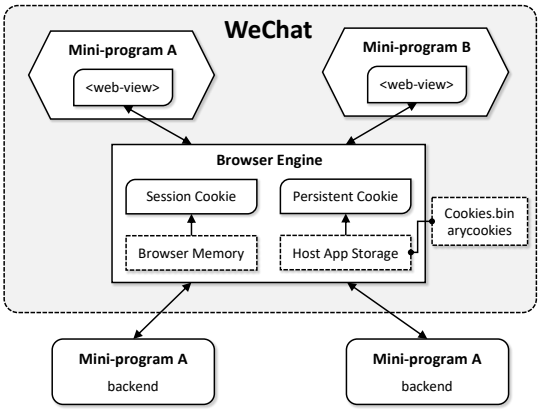

Fig. 3: Design Pitfalls and Root Cause of CMCS.

B. The Design Pitfalls and Root Cause

In this section, we report our preliminary analysis of cookie management within app-in-app ecosystems. Our study reveals that all four major ecosystems, including WeChat, Alipay, TikTok, and Baidu, exhibit CMCS vulnerabilities.

Design Pitfalls of WeChat. We used Safari and the iOS version of WeChat for remote debugging of web pages within web-view components. Our observations revealed two types of cookies: (1) persistent cookies (with explicit expiration times denoted by the `Expires` field); and (2) session cookies (with the `Expires` field set to the session, indicating they last for the session duration). Through analysis of captured network packets, we discovered that both types of cookies are shared across different mini-programs' web-view components, suggesting that these web-view components share the same browser kernel. After exiting a web-view component and closing the mini-program, session connections do not immediately disconnect. Consequently, when opening another mini-program with an external web page, session cookies are shared. As illustrated in Figure 3, we identified the `Cookies.binarycookies` file located under `'/Library/Cookies/'` in the WeChat directory of iOS, which contains cookies generated by the web-view components of different mini-programs and is stored in a shared public directory rather than isolated storage. This confirms that cookies are shared across the web-view components of different mini-programs, highlighting a violation

TABLE I: CMCS in Four Major Mini-program Ecosystems.

Ecosystem	Cookie Storage	CVE
WeChat [21]	Cookies.binarycookies	CVE-2024-40433
AliPay [22]	Cookies.binarycookies	pending
TikTok [23]	Cookies.binarycookies	CVE-2025-25436
Baidu [24]	Cookies.binarycookies	pending

of the intended isolation principles. Similarly, on Android, we observed the same behavior with web-view components. The cookies associated with external web-view are stored at `app_xweb_data/xweb_xxx/profile/Cookies` in the WeChat directory. This further confirms that web-view components across different mini-programs on both iOS and Android share the same underlying cookie storage mechanism.

CMCS in Other Mini-program Ecosystems. As illustrated in Table I, we also analyzed the iOS and Android versions of Alipay [22], TikTok [23], and Baidu [24]. We found that, like WeChat [21], the cookies generated by the external web pages of the web-view components of their mini programs are stored in the corresponding `/Library/Cookies/Cookies.binarycookies` path in the application directory, and during the access to the mini-program, we found the same situation that appeared in WeChat, that is, some cookies of the domain name in the web-view will be saved In the `Cookies.binarycookies` file, that is to say, the same cookie sharing will also occur in these mini-program ecosystem. Based on these findings, we responsibly disclosed the CMCS vulnerabilities to the affected platforms. As shown in Table I, our disclosures have been acknowledged with two CVEs.

C. Attack Scenario

Collusion Attack. One potential attack scenario is the collusion between mini-programs that leverage cookie sharing to share privileged data. In such scenario, one mini-program A can maliciously exploit shared cookies to access privileged data initially intended for another mini-program B, thereby circumventing established permission controls. We explain the mechanism of such an attack in Figure 4, using a hypothetical example involving two mini-programs, A and B, within a platform like WeChat. Firstly, consider that a user grants mini-program A the necessary permissions to access privileged data (❶❷). Mini-program A retrieves this data (❸) and saves it within the web-view's cookie (❹❺). This storage action is performed without the user's explicit awareness that such sensitive data is being saved in a shared, less secure manner. Subsequently, the user accesses mini-program B, which does not have the same permissions as mini-program A to access the privileged data directly. However, because mini-program B operates within the same web-view environment, it can exploit the shared cookie mechanism. When mini-program B requests the same web-view, it can read the cookies stored by mini-program A. This allows mini-program B to retrieve the privileged data without the user's explicit consent or knowledge. This process effectively bypasses the intended security model,

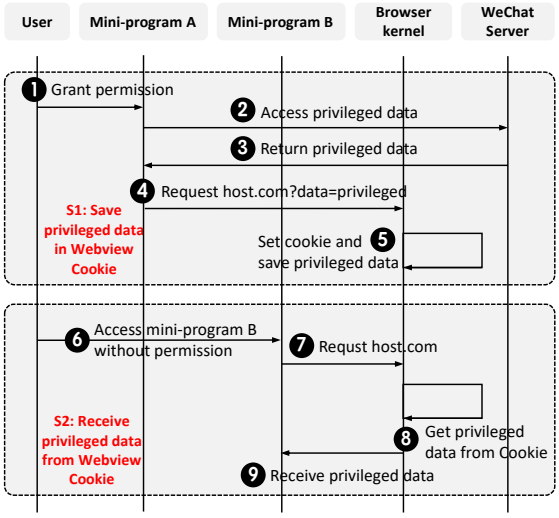

Fig. 4: Collusion Attack via CMCS.

which relies on explicit permissions. The shared web-view component acts as an unintended bridge, enabling data leakage from a privileged context to an unprivileged one.

D. A Running Example

The provided code snippets in Figure 5 demonstrate a collusive attack that enables cross mini-program sharing of sensitive user data through cookie sharing within the web-view component. The attack involves two mini-programs, Sender#A and Receiver#B, and a shared web-view component Web-View#C. The goal is to transmit the user's phone number from Sender#A to Receiver#B, even though Receiver#B is not authorized to access this sensitive information directly.

Sender#A. In `index.wxml`, Sender#A embeds a web-view component and binds its `src` attribute to the `host` variable, which is set to a URL containing the user's phone number `args.detail.userInfo.phonenumber`.

Web-View#C In `index.html`, upon receiving the URL containing the user's phone number, Web-View#C extracts the phone number from the URL query parameter (`this.$route.query.number`). If no cookie is present, Web-View#C creates a new cookie with the name `phone` and the value set to the extracted phone number `value`. If a cookie already exists, Web-View#C retrieves the phone number from the existing cookie (`document.cookie.phone`). Web-View#C then sends a `postMessage` back to the mini-program, containing the phone number.

Receiver#B In `index.wxml`, Receiver#B also embeds a web-view component and binds its `bindmessage` attribute to the `handleMsg` function. In `index.js`, the `handleMsg` function listens for `event` received from the web-view component. If the received `event` contains a phone number (`event.detail.phone`), Receiver#B logs it, thereby accessing sensitive user data from Sender#A via CMCS.

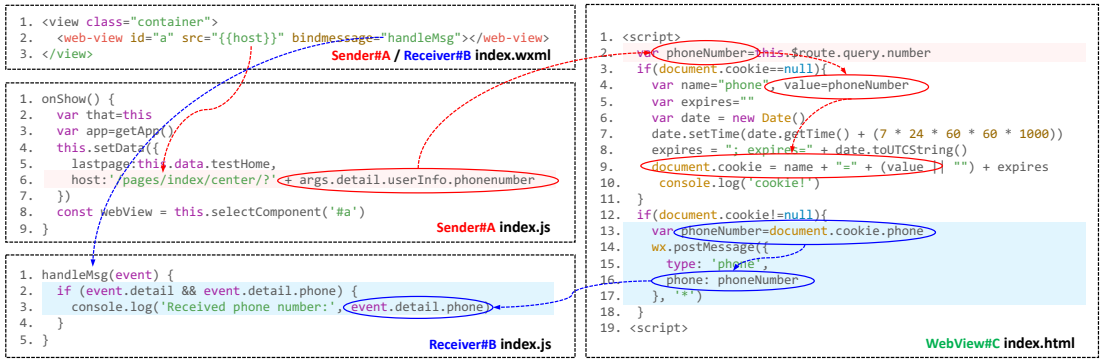

Fig. 5: A Running Example Code Snippet of Collusion Attack via CMCS.

IV. Design of MiCoScan

CMCS occurs especially when mini-programs operate under the same domain. This risk is highlighted by the potential for privileged data to be transmitted to the web-view and subsequently stored in cookies. So we need to identify the web-view domain context and privileged data flow. In this section, we introduce MiCoScan, which identifies web-view contexts and employs cross web-view data flow analysis to pinpoint the transmission of privileged data during bidirectional communication.

A. Step I: Web-View Context Modeling

The first step in detecting CMCS vulnerabilities is to accurately identify the domain context of each web-view instance within the mini-programs. This process involves resolving the two-way binding syntax used in mini-programs to determine the web-view `URL`. As illustrated in Figure 6, there are two primary methods for binding `URL` to the `src` attribute: *direct binding* and *double binding*.

- **Direct Binding** involves specifying the `URL` directly within the `src` attribute of the web-view component. This method is straightforward and does not involve any dynamic data binding or JavaScript logic. When the user triggers the web-view component, the web-view component will jump directly to the URL pointed to. For the direct data binding, we can easily extract the target `URL` of the web page from the web-view component.
- **Double Binding**, also known as two-way binding, involves dynamically binding the `src` attribute to a JavaScript variable. This method allows the `URL` to be determined at runtime based on the application state or external data sources. Although the URL declaration is in the web-view component, the definition and assignment of data are in the JavaScript code. As shown in Figure 6, although the variable `url` in the web-view component is bound in `page.wxml`, its actual declaration is in `page.js`, and the data is assigned through `baseUrl` of `globalData`. So we can get the domain name of the `URL` based on data flow analysis of JavaScript and WXML files.

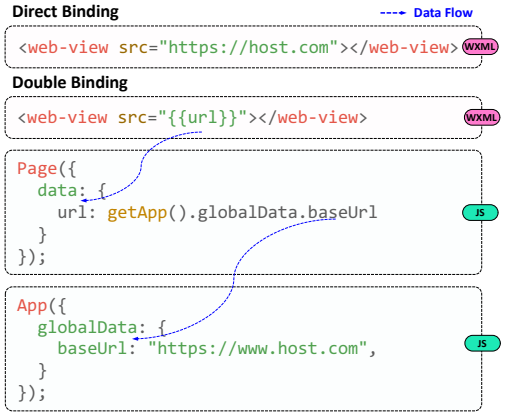

Fig. 6: Data Binding Mechanism in Web-View Domain.

To resolve the target `URL` of the web-view component, the first step is to identify the bound variables in WXML, and then the next step is to trace their origins and transformations within the JavaScript code. This involves following the variable from its declaration to its final assignment, capturing all intermediate states and modifications. Although the data flow and value analysis follow the general flow of JavaScript program analysis, special handling is required for global variables in `Page` instances and `App` instances due to their unique assignment and referencing syntax supported by the host app. Traditional program analysis frameworks often struggle to support these nuances. In mini-programs, variables bound in WXML are typically located in the `data` object, which serves as a state storage akin to global variables within the current `Page` instance. The data object's properties can be dynamically updated via `this.setData` method, and referenced using `this.data.prop`. Additionally, a global `App` instance exists, encompassing the `globalData` object, which can be accessed and modified across any page using `getApp().globalData.prop`. Consequently, during data flow and value analysis, we need to parse these referencing and assignment rules to maintain a global variable table for the page of the mini-program, thereby accurately

obtaining the values of variables bound in WXML.

B. Step II: Cross Web-View Data Flow Analysis

Another critical aspect of CSCS is the potential transmission of privileged data. To address this, we propose analyzing the data transmission during the bi-directional communication between the mini-program and the web-view. As described in § II-A, there are two channels for communication from the mini-program to the web-view: URL parameter concatenation and `postMessage` invocations. The communication from the web-view to the mini-program can only occur through `postMessage` calls, and the mini-program receives via the `bindMessage` handler. Since the web-view logic executes on the browser engine, we can only examine the reception of privileged data from the web-view via the `bindMessage` handler. The data transmission using URL parameter concatenation can be analyzed based on Step I, so we focus on examining privileged data sent from `postMessage` invocations and received from `bindMessage` handler in this step.

C#1: Mini-program→Web-View. To scrutinize the potential privileged data transmission through `postMessage` calls, we employ a two-pronged approach: identifying `postMessage` invocations and tracing the data flow of messages. The first step involves locating all instances of `postMessage` calls, which is achieved by traversing the Abstract Syntax Tree (AST) and identifying function calls with the name `postMessage`. Additionally, we locate invocations of `createWebViewContext`, as this function is a prerequisite for `postMessage` communication with the web-view. Once the `postMessage` call sites are identified, we trace the data flow of the messages being sent. This involves backward data flow analysis to determine the origins and transformations of the message content. Particular attention is paid to data sources that may contain privileged information, such as user credentials, device identifiers, or sensitive application data.

C#2: Web-View→Mini-program. Complementing the analysis of `postMessage` data transmission, we also examine the reception of data from the web-view through `bindMessage` handlers within the miniprogram. The analysis begins by identifying the callback functions bound to `bindMessage` within the WXML files. These callbacks serve as event handlers for messages received from the web-view. Subsequently, we leverage AST parsing to locate the corresponding callback functions within the JavaScript code. Inside these functions, we identify instances where the received data is accessed through the `event.detail` property, which contains the message content sent from the web-view. Once the access points of `event.detail` are identified, we trace the data flow of the received messages within the mini-program code. This involves forward data flow analysis to determine how the received data is processed, stored, and propagated throughout the application.

C. Step III: Collusion Attack Detection

To detect potential collusion attacks enabled by CMCS vulnerabilities, we combine the results from Step I and Step II to identify unauthorized data exchange between mini-programs.

Grouping of Candidates. The first step is to group mini-programs that share the same web-view domain. Using the domain context extracted in Step I, where the target URLs of web-view instances were resolved through static data flow analysis, we identify mini-programs operating within the same web-view domain. Mini-programs sharing the same domain inherently share cookies and other browser states, making them potential candidates for cross-mini-program cookie sharing vulnerabilities. By grouping these mini-programs together, we establish the scope of potential collusion and limit further analysis to interactions within each group.

Analysis of Overlapping Sensitive Data Flows. Once mini-programs are grouped by web-view domain, the next step is to analyze sensitive data flows within each group to detect potential overlaps between the data sent and received by mini-programs. For every pair of mini-programs, we examine the data sent to the web-view by one mini-program and determine whether it can be accessed or processed by another mini-program in the same group. Specifically, we analyze the data sent to the web-view through `postMessage` invocations or URL parameter concatenations using backward data flow analysis. This helps trace the origins of transmitted data to identify whether it includes sensitive information, such as user credentials, device identifiers, or authentication tokens. Similarly, we analyze the data received from the web-view by each mini-program through `bindMessage` handlers using forward data flow analysis. By correlating the data sent by one mini-program with the data received by another, we identify potential paths where sensitive information is exchanged between mini-programs. This pairwise analysis ensures that all possible interactions between mini-programs within the same domain group are systematically explored, enabling the detection of unauthorized data flows that could lead to collusion attacks.

V. Implementation

PoC Implementation. We have implemented Proof-of-Concept (PoC) code following the developer guidelines of super apps. As cookie sharing typically occurs within the same web-view domain, we developed a sender mini-program and a receiver mini-program that can access the same web-view domain. For the collusion attack, we injected the privileged phone number obtained by the sender into a cookie. The non-privileged receiver then successfully retrieved this sensitive data from the cookie, bypassing the app's isolation mechanisms and enabling unauthorized data exposure.

MiCoScan Implementation. We have implemented MiCoScan based on the existing mini-program data flow analysis framework TaintMini [25]. Specifically, we added custom rules to track web-view domains accessed by each mini-program, as cookie sharing occurs within the same domain. We monitored URLs passed to web-view components and mapped mini-programs to their associated domains. Additionally, we incorporated rules to detect potential transmission

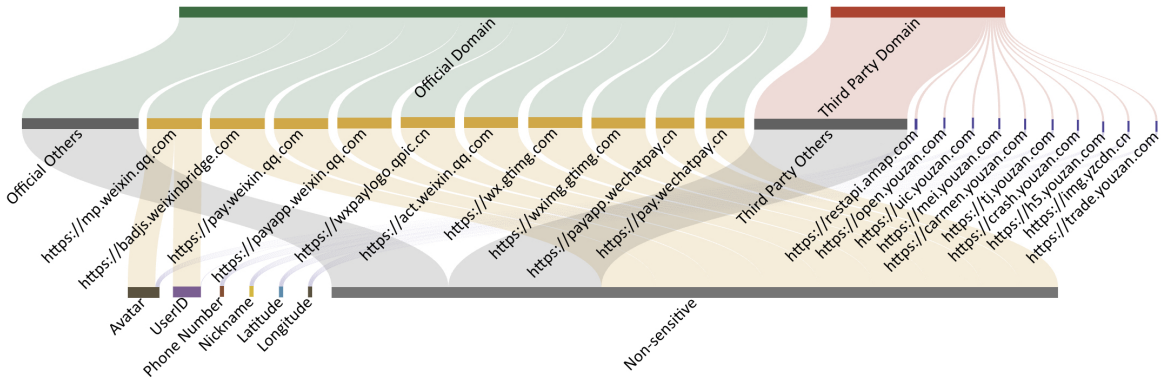

Fig. 7: Privileged Data Exposure Across Web-View Domains in Mini-Programs.

Filtering Methods	Count
Total number of mini-programs	351,483
Filter#1: Containing `<web-view>`	261,538
Filter#2: Web-View domain clustering	45,448
Filter#3: Clusters of mini-programs more than 5	3,109

TABLE II: Filtering Process for Web-View Domain Analysis in Mini-Programs.

of privileged data like user identifiers between the mini-program and web-view. This analysis focused on data flows from sensitive sources (e.g., user info APIs) propagating to the web-view through communication channels. By combining identified web-view domains and tracked privileged data flows, MICOSCAN can pinpoint instances where sensitive data may be shared across mini-programs via web-view cookies. These potential vulnerabilities are reported for further investigation and remediation.

VI. MEASUREMENT RESULTS

A. Experimental Setup

Dataset. To collect mini-programs, we leveraged the open-source MiniCrawler [26] to crawl and download package files from the WeChat mini-program ecosystem. Our initial dataset encompasses 415,550 mini-programs. However, after unpacking, only 351,483 mini-programs can be properly extracted and utilized for subsequent analysis.

Running Environment. Our experiments are conducted on a server running Ubuntu Linux of 22.04 version with two 64-core AMD EPYC 7713 and 256 GB RAM. The static analysis leverages the server's computational capacity by utilizing 32 threads, enabling high parallelism for efficient processing of the large mini-program dataset.

B. Vulnerability Prevalence

Web-View Domain Context Clustering. Due to the same-origin policy restriction, cookie sharing typically occurs within the same domain, hence we employ the domain name as the basis for mini-program clustering. Table II outlines the filtering process employed to identify mini-programs with web-views and cluster them based on their web-view domains. From the initial pool of 351,483 mini-programs, we first filtered for those containing the `<web-view>` component (Filter 1). Next, we performed web-view domain clustering (Filter 2), resulting in 45,448 groups of mini-programs with identifiable web-view domains. We then narrowed our focus to clusters containing more than 5 mini-programs (Filter 3), leaving 3,109 groups. These clusters represent potential candidates for further analysis, as they may exhibit vulnerabilities associated with cross-component data sharing and communication through web-views under the same domain.

Distribution of Web-View Domains. The web-view domain clustering results are illustrated in Figure 7. The green section represents the WeChat official domains, while the red section represents the third-party domains. The significantly larger size of the red section indicates that a considerable portion of the URLs in the dataset belong to third-party operators. Our analysis focused solely on grouping and investigating the domain names from the third-party operators. The rationale behind this approach is that the official domains associated with WeChat are trusted and unlikely to pose significant security risks. However, the third-party domains warrant closer examination, as they may potentially introduce vulnerabilities or be associated with unauthorized data sharing practices.

Privileged Data Transmitted. The results of our analysis, as presented in Table III, reveal widespread transmission of privileged data between mini-programs and their embedded web-views. Across all categories, we identified a total of 7,849 instances of sensitive data being sent from mini-programs to web-views, and 116 instances of data being received from web-views. The sensitive data transmitted includes phone numbers, nicknames, location data (latitude and longitude), avatar images, and user IDs. Among these, avatar images were the most frequently transmitted, with 2,815 instances, followed by user IDs (1,155 instances) and phone numbers (1,546 instances). Location data was also widely shared, with latitude and longitude transmitted in 858 and 859 instances, respectively. This extensive sharing of sensitive information not only exposes user privacy but also enables the possibility of tracking users across different mini-programs. Certain categories of

TABLE III: Privileged Data Transmitted between Mini-program and Web-View. **S** Denotes Privileged Data Sent from Mini-program to Web-View, **R** Denotes Data Received from Web-View.

Categories	Phone Number		Nickname		Latitude		Longitude		Avatar		User ID	
	S	R	S	R	S	R	S	R	S	R	S	R
Individual Seller	772	14	353	2	548	/	548	/	1,783	2	491	33
Life Service	112	/	72	/	175	/	175	3	459	/	133	/
Tools	106	/	33	/	6	/	6	/	87	/	73	23
Education	69	2	9	2	2	/	2	/	42	2	123	2
Local Service	9	4	/	/	/	/	/	/	3	/	12	/
Automotive Service	18	/	16	/	6	2	6	2	67	/	21	5
Real Estate	128	/	14	/	1	/	1	/	48	/	17	1
Traffic	13	/	/	/	2	/	4	/	3	/	7	/
Logistics	6	/	1	/	1	/	1	/	1	/	10	4
Business	109	/	25	/	13	/	13	/	113	/	82	4
Finance	2	/	3	/	2	/	2	/	1	/	5	/
Medical Service	25	/	14	/	2	/	2	/	30	/	10	4
IT	2	/	1	/	/	/	/	/	1	/	/	/
Social	1	/	/	/	1	/	1	/	3	/	7	/
Government	28	/	2	/	5	/	4	3	3	/	10	/
E-commerce	5	/	/	/	2	/	2	/	6	/	7	/
Sports	6	/	/	/	/	/	/	/	12	/	3	/
Dining	93	/	60	/	57	/	57	/	102	/	99	/
Entertainment	4	/	/	/	1	1	1	1	/	/	6	/
Charity	1	/	/	/	/	/	/	/	/	/	2	/
Tourism	26	/	8	/	34	/	34	/	36	/	13	/
News	11	/	5	/	/	/	/	/	15	/	24	/
Total	**1,546**	**20**	**616**	**4**	**858**	**3**	**859**	**9**	**2,815**	**4**	**1,155**	**76**

mini-programs exhibited particularly high levels of privileged data transmission. The *Individual Seller* category was the most significant contributor, accounting for 772 instances of phone number transmission, 548 instances of location data sharing, and 1,783 instances of avatar image transmission. Other notable contributors include the *Life Service* category, with 175 instances of location data sharing, and the *Real Estate* category, which accounted for 128 instances of phone number transmission. While the majority of privileged data transmission occurred in a one-way manner, from mini-programs to web-views, the 116 instances of data being received from web-views highlight additional privacy risks.

Identification of Overlapping Web-View Domains. Through our domain-based clustering and privileged data transmission analysis, we identified a total of 6,352 web-view domains that were involved in sending sensitive user data and 41 web-view domains that received such data. Among them, 7 domains exhibited both sending and receiving of sensitive data, indicating potential bi-directional data flows and higher risks of collusion attacks. These overlapping domains were further analyzed to assess their security and privacy implications.

Victim Analysis of Collusion Attacks. To understand the extent of the impact caused by cookie-sharing mechanisms, we analyzed the mini-programs affected by the 7 overlapping domains that exhibited both sending and receiving of sensitive data. Based on the data presented in Table IV, a total of 9,877 mini-programs were identified as potential victims of cookie-sharing practices. These mini-programs were exposed to significant privacy risks due to the improper handling of sensitive user data. Among the affected domains, `h5.youzan.com` had the greatest impact, involving

Web-View Domain	Cookie	Data	Count
h5.youzan.com	yz_log_uuid	user_id	5,952
	m	phone number	
	banner_id	product id	
	yz_log_uid	user_id	
watch-dog.huolala.cn	device_id	user_id	1,871
	user_id	user_id	
m.jd.com	jd_pin	username	1,213
w.fooww.com	nick_name	username	398
	headimgurl	avatar	
sf-express.com	device_id	user_id	279
	distinct_id	user_id	
dinghuo123.com	distinct_id	user_id	119
m.ctrip.com	DUID	user_id	45

TABLE IV: Collusion Attack via CMCS in the Wild.

5,952 mini-programs that transmitted sensitive data such as phone numbers, product identifications, and user IDs through cookies, including `m`, `banner_id`, and `yz_log_uid`. The domain `watch-dog.huolala.cn` affected 1,871 mini-programs, where device and user IDs were shared via cookies. Similarly, `m.jd.com` impacted 1,213 mini-programs through the sharing of usernames stored in the `jd_pin` cookie. These findings highlight the urgent need for stricter security controls and enhanced data protection measures to mitigate these risks and prevent further harm.

Findings. Our measurement study reveals critical risks in the mini-program ecosystem. The integration of web-view

components disrupts isolation mechanisms, enabling unauthorized data exchange and collusion attacks. We identified 7,965 instances of sensitive data transmission and uncovered 7 overlapping domains with bi-directional data flows, exposing 9,877 mini-programs to collusion attacks. These findings underscore the need for stronger isolation mechanisms, stricter scrutiny of web-view communication channels, and robust data protection policies to secure the ecosystem.

VII. Discussion

Generality of CMCS Vulnerability. We also tested mini-programs other than WeChat. We found that the mini-programs of Alipay, Douyin, and Baidu also provide the same web-view component for embedding external web pages as the WeChat mini-program. We used the officially provided mini-program development tool to build a local test demo that embeds external web pages in a web-view, and used a Node.js-based web backend to set cookies. We found that these super applications behave similarly to WeChat. In the web-view of these mini-programs, there is no complete isolation between the view component and the embedded web page. Although we demonstrated and measured the collusion attack enabled by CMCS vulnerabilities, its potential impacts could extend far beyond this scenario. For example, an attacker could exploit CMCS by loading the same web-view as the victim's mini-program to impersonate the victim and log into their account without authorization. By inheriting the cookies stored in the shared web-view, the attacker could bypass authentication mechanisms and gain access to sensitive user data or perform malicious operations on behalf of the victim. These potential impacts underscore the urgent need for stricter isolation mechanisms in mini-program ecosystems.

Risk Mitigation. To address the risks associated with cookie sharing attacks and enhance security in the mini-program ecosystem, we propose several mitigation strategies: First and important, isolate WebView instances per mini-program by default. Each WebView instance should be bound to the app ID of the mini-program that initiated it. This unique identifier would ensure that the WebView can only interact with the specific mini-program that launched it. By default, any attempt by a WebView to interact with other mini-programs should be blocked, thereby preventing unauthorized data sharing or cross-contamination. Additionally, for scenarios requiring cross-program interactions, we propose implementing an explicit user-consent mechanism. When a WebView initiated by a mini-program attempts to interact with another mini-program or access its data, the system should explicitly request the user's permission. This mechanism would clearly inform users of the intended data exchange and require their consent before proceeding, ensuring transparency and control. By implementing these measures, the mini-program ecosystem can achieve stronger isolation and greater transparency while maintaining flexibility for necessary interactions.

Limitations. While our study provides valuable insights into CMCS vulnerabilities, several limitations warrant consideration. First, the collusion attack requires interaction with the attacker-controlled two mini-programs, which may reduce the feasibility of the attack in certain scenarios. However, we emphasize that this interaction does not need to occur simultaneously. The sender and receiver mini-programs can be used at different times, leveraging the persistent shared cookie storage managed by the global WebView process within the host application. Second, MiCoScanis a large-scale static taint analysis framework that is path-insensitive but field-sensitive. It relies on predefined keywords to identify sensitive information, which means it does not provide strict guarantees of soundness and precision. Nevertheless, we believe that MiCoScanis sufficiently effective for identifying CMCS risks in mini-program ecosystems. Third, MiCoScancurrently resolves dynamically constructed URLs through static data flow tracking across WXML and JavaScript files. This includes common patterns such as assignments via 'getApp().globalData' and 'setData'. However, if the URL is derived from runtime user input or dynamically fetched remote content, static analysis is unable to resolve it. While dynamic analysis could improve recall in such cases, we have not integrated it into MiCoScanto maintain scalability for large-scale ecosystem analysis.

VIII. Related Work

Security and Privacy of Mini-programs. The security and privacy of mini-programs have garnered considerable attention in recent years [27], [28], [29], [30], [31], [32], with various studies addressing different aspects of this emerging application paradigm. Notably, a significant body of work has focused on identifying and addressing security vulnerabilities within mini-programs. For instance, Wang et al. [33] collected 83 real-world bugs and developed WeDetector, a tool that identifies WeBugs following three specific bug patterns. Another investigation [1] explored issues such as system resource exposure, subwindow deception, and subapp lifecycle hijacking in the mini-program ecosystem, conducting evaluations on 11 popular platforms to highlight the prevalence of these security problems. Additionally, a novel privacy leak issue in mini-programs was studied [9], which could potentially lead to private data theft by the mini-program platform, and an attack process exploiting this vulnerability was elucidated. Moreover, the Cross Miniapp Request Forgery (CMRF) vulnerability was identified [2] in mini-program communication, shedding light on its root cause and designing the CMRFScanner tool for its detection. Beyond security vulnerabilities, a series of studies have also emphasized the importance of privacy in the mini-program ecosystem. For example, TaintMini [25] introduced a framework for detecting flows of sensitive data within and across mini-programs using static taint analysis, while MiniTracker [34] constructed assignment flow graphs as a common representation across different host apps, revealing common privacy leakage patterns in a large-scale study of 150k mini-programs. Furthermore, several studies [35], [36], [37] have focused on taint analysis techniques to detect AppSecret leaks and some others [38], [39], [5] focused on the consistency of data collection and usage in mini-programs, emphasizing the urgent need for more precise privacy monitoring systems.

Despite these significant contributions to understanding the privacy of mini-programs, there is still a need for further exploration and research. Different from the existing works, our study focused on both the user tracking and privacy risks of cross mini-program cookie sharing caused by the absence of web-view isolation in cross mini-program channels.

Collusion Attacks. Collusion attacks, where multiple malicious applications collude to bypass security boundaries and gain unauthorized access to sensitive data, have been explored in the Android ecosystem. Several researches [40], [41], [42] have investigated collusion attacks facilitated by different attack vectors. AppHolmes [43] conducted the first in-depth study on app collusion in the mobile ecosystem, where one app covertly launches others in the background, leading to performance, efficiency, and security implications. Bosu et al. [40] conducted a large-scale analysis of 110,150 Android apps to detect collusive and vulnerable apps based on inter-app ICC data flows, providing real-world evidence and insights into various types of ICC abuse for collusion attacks. Bhandari et al. [44] proposed a model-checking based approach for detecting collusion among Android apps by analyzing simultaneous inter-app communication and information leakage paths, improving upon prior techniques' detection capabilities. It is important to note that while the existing research provides valuable insights into collusion attacks in various contexts, the unique characteristics and security models of the mini-program ecosystem necessitate tailored research and mitigation strategies. The introduction of shared web-view components and the interplay between mini-programs and web-view content pose distinct challenges that require specialized analysis and countermeasures.

IX. Conclusion

In this paper, we systematically investigated the security and privacy risks posed by CMCS vulnerabilities in the app-in-app ecosystem. Our analysis revealed that CMCS vulnerabilities exist in four major platforms, including WeChat, AliPay, TikTok, and Baidu, and can be exploited to perform collusion attacks. To address this issue, we proposed MiCoScan, a static analysis tool that integrates web-view context modeling and cross-webview data flow analysis to detect CMCS vulnerabilities. Using MiCoScan, we conducted the first large-scale measurement study of 351,483 real-world mini-programs, uncovering 45,448 clusters of mini-programs sharing web-view domains and 9,877 vulnerable mini-programs exposed to collusion attacks. These findings highlight the widespread impact of CMCS vulnerabilities and underscore the urgent need for improved isolation mechanisms and stricter data protection practices in mini-program ecosystems.

Acknowledgement

This work was supported in part by the National Natural Science Foundation of China (grants No.62572209, 62502168) and the Hubei Provincial Key Research and Development Program (grant No. 2025BAB057).

References

[1] 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.

[2] 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.

[3] W3C, "Miniapp standardization white paper," 2022, https://www.w3.org/TR/mini-app-white-paper.

[4] S. Wang, Y. Zhao, K. Wang, and H. Wang, "On the usage-scenario-based data minimization in mini programs," in *Proceedings of the 2023 ACM Workshop on Secure and Trustworthy Superapps*, ser. SaTS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 29–32. [Online]. Available: https://doi.org/10.1145/3605762.3624435

[5] S. Wang, Y. Li, K. Wang, Y. Liu, H. Li, Y. Liu, and H. Wang, "Miniscope: Automated ui exploration and privacy inconsistency detection of miniapps via two-phase iterative hybrid analysis," *ACM Trans. Softw. Eng. Methodol.*, Dec. 2024, just Accepted. [Online]. Available: https://doi.org/10.1145/3709351

[6] Y. Wang, M. Fan, H. Zhou, H. Wang, W. Jin, J. Li, W. Chen, S. Li, Y. Zhang, D. Han, and T. Liu, "Minichecker: Detecting data privacy risk of abusive permission request behavior in mini-programs," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1667–1679. [Online]. Available: https://doi.org/10.1145/3691620.3695534

[7] Y. Han, Z. Xiao, Z. Wang, and J. Zhang, "Privacy policy compliance in miniapps: An analytical study," in *Proceedings of the ACM Workshop on Secure and Trustworthy Superapps*, ser. SaTS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 11–17. [Online]. Available: https://doi.org/10.1145/3689941.3695777

[8] X. Deng, M. Zhang, X. Dong, and X. Hu, "Detect counterfeit mini-apps: A case study on wechat," in *Proceedings of the ACM Workshop on Secure and Trustworthy Superapps*, ser. SaTS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1–10. [Online]. Available: https://doi.org/10.1145/3689941.3695773

[9] 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, pp. 1597–1613.

[10] Y. Yang, C. Wang, Y. Zhang, and Z. Lin, "SoK: Decoding the Super App Enigma: The Security Mechanisms, Threats, and Trade-offs in OS-alike Apps," *arXiv preprint*, 2023.

[11] Z. Zhang, J. Du, W. Diao, and J. Wu, "Minible: Exploring insecure ble api usages in mini-programs," in *Proceedings of the ACM Workshop on Secure and Trustworthy Superapps*, ser. SaTS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 18–22. [Online]. Available: https://doi.org/10.1145/3689941.3695774

[12] Z. Yan, M. Fan, Y. Wang, J. Shi, H. Wang, and T. Liu, "Muid: Detecting sensitive user inputs in miniapp ecosystems," in *Proceedings of the 2023 ACM Workshop on Secure and Trustworthy Superapps*, ser. SaTS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 17–21. [Online]. Available: https://doi.org/10.1145/3605762.3624429

[13] C. Wang, Y. Zhang, and Z. Lin, "Rootfree attacks: Exploiting mobile platform's super apps from desktop," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 830–842. [Online]. Available: https://doi.org/10.1145/3634737.3645001

[14] Y. Wang, Y. Yao, S. Shi, W. Chen, and L. Huang, "Towards a better super-app architecture from a browser security perspective," in *Proceedings of the 2023 ACM Workshop on Secure and Trustworthy Superapps*, ser. SaTS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 23–28. [Online]. Available: https://doi.org/10.1145/3605762.3624427

[15] Z. Zhang, Z. Zhang, K. Lian, G. Yang, L. Zhang, Y. Zhang, and M. Yang, "Trusteddomain compromise attack in app-in-app ecosystems," in *Proceedings of the 2023 ACM Workshop on Secure and Trustworthy Superapps*, ser. SaTS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 51–57. [Online]. Available: https://doi.org/10.1145/3605762.3624430

[16] Y. Han, X. Ji, Z. Wang, and J. Zhang, "Systematic analysis of security and vulnerabilities in miniapps," in *Proceedings of the 2023 ACM Workshop on Secure and Trustworthy Superapps*, ser. SaTS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1–9. [Online]. Available: https://doi.org/10.1145/3605762.3624432

[17] Android, "Application sandbox," https://source.android.com/docs/security/app-sandbox, 2025, accessed: 2025-05-31.

[18] WeChat, "File system of wechat mini-programs," https://developers.weixin.qq.com/minigame/en/dev/guide/base-ability/file-system.html, 2025, accessed: 2025-05-31.

[19] DARKNAVY, "Achieving persistent client-side attacks with a single wechat message," https://www.darknavy.org/blog/achieving_persistent_client_side_attacks_with_a_single_wechat_message/, 2025, accessed: 2025-05-31.

[20] WeChat, "Webview component in wechat mini-programs," https://developers.weixin.qq.com/miniprogram/en/dev/component/web-view.html, 2025, accessed: 2025-05-31.

[21] ——, "Mini program development platform," https://developers.weixin.qq.com/miniprogram/dev/framework/, 2025, accessed: 2025-05-31.

[22] AliPay, "Mini program development platform," https://miniprogram.alipay.com/, 2025, accessed: 2025-05-31.

[23] TikTok, "Mini program development platform," https://developer.open-douyin.com/docs/resource/zh-CN/mini-app/introduction/usage-guide, 2025, accessed: 2025-05-31.

[24] Baidu, "Mini program development platform," https://smartprogram.baidu.com/docs/develop/tutorial/intro/, 2025, accessed: 2025-05-31.

[25] C. Wang, R. Ko, Y. Zhang, Y. Yang, and Z. Lin, "Taintmini: Detecting flow of sensitive data in mini-programs with static taint analysis," in *Proceedings of the 45th International Conference on Software Engineering*, 2023.

[26] Y. Zhang, B. Turkistani, A. Y. Yang, C. Zuo, and Z. Lin, "A measurement study of wechat mini-apps," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 5, no. 2, pp. 1–25, 2021.

[27] C. Wang, Y. Zhang, and Z. Lin, "One size does not fit all: Uncovering and exploiting cross platform discrepant {APIs} in {WeChat}," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 6629–6646.

[28] ——, "Uncovering and exploiting hidden apis in mobile super apps," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2471–2485.

[29] Z. Zhang, L. Zhang, G. Yang, Y. Chen, J. Xu, and M. 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*, 2024.

[30] Z. Zhang, Q. Hou, L. Ying, W. Diao, Y. Gu, R. Li, S. Guo, and H. Duan, "Minicat: Understanding and detecting cross-page request forgery vulnerabilities in mini-programs," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, B. Luo, X. Liao, J. Xu, E. Kirda, and D. Lie, Eds. ACM, 2024, pp. 525–539. [Online]. Available: https://doi.org/10.1145/3658644.3670294

[31] Y. Cai, Z. Zhang, M. Yao, J. Liu, X. Zhao, X. Fu, R. Li, Z. Li, X. Chen, Y. Guo, and D. Li, "I can tell your secrets: Inferring privacy attributes from mini-app interaction history in super-apps," *CoRR*, vol. abs/2503.10239, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2503.10239

[32] Y. Yang, Y. Zhang, and Z. Lin, "Understanding miniapp malware: Identification, dissection, and characterization," in *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/understanding-miniapp-malware-identification-dissection-and-characterization/

[33] T. Wang, Q. Xu, X. Chang, W. Dou, J. Zhu, J. Xie, Y. Deng, J. Yang, J. Yang, J. Wei *et al.*, "Characterizing and detecting bugs in wechat mini-programs," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 363–375.

[34] W. Li, B. Yang, H. Ye, L. Xiang, Q. Tao, X. Wang, and C. Zhou, "Minitracker: Large-scale sensitive information tracking in mini apps," *IEEE Transactions on Dependable and Secure Computing*, 2023.

[35] Y. Zhang, Y. Yang, and Z. Lin, "Don't leak your keys: Understanding, measuring, and exploiting the appsecret leaks in mini-programs," *arXiv preprint arXiv:2306.08151*, 2023.

[36] S. Baskaran, L. Zhao, M. Mannan, and A. Youssef, "Measuring the leakage and exploitability of authentication secrets in super-apps: The wechat case," *arXiv preprint arXiv:2307.09317*, 2023.

[37] S. Meng, L. Wang, S. Wang, K. Wang, X. Xiao, G. Bai, and H. Wang, "Wemint: Tainting sensitive data leaks in wechat mini-programs," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1403–1415.

[38] Y. Wang, M. Fan, J. Liu, J. Tao, W. Jin, H. Wang, Q. Xiong, and T. Liu, "Do as you say: Consistency detection of data practice in program code and privacy policy in mini-app," *IEEE Transactions on Software Engineering*, vol. 50, no. 12, pp. 3225–3248, 2024.

[39] X. Zhang, Y. Wang, X. Zhang, Z. Huang, L. Zhang, and M. Yang, "Understanding privacy over-collection in wechat sub-app ecosystem," *arXiv preprint arXiv:2306.08391*, 2023.

[40] A. Bosu, F. Liu, D. Yao, and G. Wang, "Collusive data leak and more: Large-scale threat analysis of inter-app communications," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017, pp. 71–85.

[41] M. Alhanahnah, Q. Yan, H. Bagheri, H. Zhou, Y. Tsutano, W. Srisa-An, and X. Luo, "Detecting vulnerable android inter-app communication in dynamically loaded code," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 550–558.

[42] B. Wang, C. Yang, and J. Ma, "Iafdroid: Demystifying collusion attacks in android ecosystem via precise inter-app analysis," *IEEE Transactions on Information Forensics and Security*, 2023.

[43] M. Xu, Y. Ma, X. Liu, F. X. Lin, and Y. Liu, "Appholmes: Detecting and characterizing app collusion among third-party android markets," in *Proceedings of the 26th international conference on World Wide Web*, 2017, pp. 143–152.

[44] S. Bhandari, F. Herbreteau, V. Laxmi, A. Zemmari, P. S. Roop, and M. S. Gaur, "Detecting inter-app information leakage paths," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017, pp. 908–910.