The Skeleton Keys: A Large Scale Analysis of Credential Leakage in Mini-apps

Yizhe Shi*
Fudan University
yzshi23@m.fudan.edu.cn

Zhemin Yang*
Fudan University
yangzhemin@fudan.edu.cn

Kangwei Zhong
Fudan University
kwzhong23@m.fudan.edu.cn

Guangliang Yang
Fudan University
YangGL@fudan.edu.cn

Yifan Yang
Fudan University
yifanyang23@m.fudan.edu.cn

Xiaohan Zhang
Fudan University
xh_zhang@fudan.edu.cn

Min Yang
Fudan University
m_yang@fudan.edu.cn

***Abstract*—In recent years, the app-in-app paradigm, involving super-app and mini-app, has been becoming increasingly popular in the mobile ecosystem. Super-app platforms offer mini-app servers access to a suite of powerful and sensitive services, including payment processing and mini-app analytics. This access empowers mini-app servers to enhance their offerings with robust and practical functionalities and better serve their mini-apps. To safeguard these essential services, a credential-based authentication system has been implemented, facilitating secure access between super-app platforms and mini-app servers. However, the design and workflow of the crucial credential mechanism still remain unclear. More importantly, its security has not been comprehensively understood or explored to date.**

In this paper, we conduct the first systematic study of the credential system in the app-in-app paradigm and draw the security landscape of credential leakage risks. Consequently, our study shows that 21 popular super-app platforms delegate sensitive services to mini-app servers with seven types of credentials. Unfortunately, these credentials may suffer from leakage threats caused by malicious mini-app users, posing serious security threats to both super-app platforms and mini-app servers. Then, we design and implement a novel credential security verification tool, called KeyMagnet, that can effectively assess the security implications of credential leakage. To tackle unstructured and dynamically retrieved credentials in the app-in-app paradigm, KeyMagnet extracts and understands the semantics of credential-use in mini-apps and verifies their security. Last, by applying KeyMagnet on 413,775 real-world mini-apps of 6 super-app platforms, 84,491 credential leaks are detected, spanning over 54,728 mini-apps. We confirm credential leakage can cause serious security hazards, such as hijacking the accounts of all mini-app users and stealing users' sensitive data. In response, we have engaged in responsible vulnerability disclosure with the corresponding developers and are actively helping them resolve these issues. At the time of writing, 89 reported issues have been assigned with CVE IDs.

I. Introduction

In recent years, the app-in-app paradigm, involving super-app and mini-app, has been becoming increasingly popular in the mobile ecosystem. Mobile apps, often referred to as super-apps, offer a mini-app runtime environment, allowing them to delegate various functionalities to mini-apps. This integration enables mobile users to experience a wide range of functionalities within these super-apps. More than 20 leading mobile app platforms (e.g., WeChat and TikTok) have embraced this innovative paradigm, hosting an extensive collection of over 7 million mini-apps [1], which is more than four times the size of Google Play (1.68 million apps [2]). As an example, WeChat, one of the most globally-used instant-messaging mobile apps, functions as a super-app, and its mini-apps have engaged 600 million daily active users [3].

The app-in-app ecosystem's security, despite its paramount importance, still necessitates in-depth explorations and understanding. As shown in Figure 1, super-app platforms offer mini-app servers access to an extensive array of essential resources and services, e.g., payment processing, cloud use, AI service and mini-app analytics (step ②). This access empowers mini-app servers to enhance their offerings with robust and practical functionalities and better serve their mini-apps (step ③). To safeguard these essential services, a credential-based authentication system has been implemented. In particular, before accessing these services, the mini-app server first retrieves a credential from the super-app platform (step ①). Second, when a privileged service is called in mini-app server (step ②), it sends an authentication request with the credential. Last, the super-app platform verifies the credential for request approval. Similar to other credential-use scenarios [4], [5], [6], [7], [8], the credential mechanism may suffer from credential leakage risks. Recently, this has been demonstrated by concrete case studies [9], [10], which focus on the hard-coded credential leakage in mini-app clients. However, the

*Co-first authors.

Network and Distributed System Security (NDSS) Symposium 2025
23 - 28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.23273
www.ndss-symposium.org

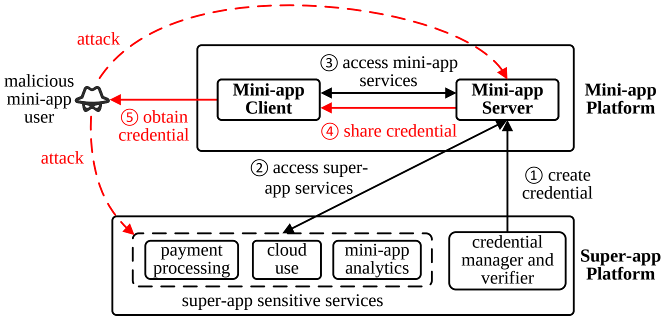


Fig. 1. The Credential System in the App-in-app Paradigm

design of the whole crucial credential mechanism still remains ambiguous. More importantly, its security has not been comprehensively understood or explored to date.

Understanding Credential Security. In this work, we conduct the first systematic study of the credential system in the app-in-app paradigm and draw the security landscape of credential leakage risks. We first understand what credentials are in the scenario of the app-in-app diagram and how they are used to secure super-app service access. We target 21 popular super-app platforms and conduct manual analysis on their documentation and sample mini-apps. We observe that most credentials are dynamically retrieved from the mini-app servers and do not have a fixed or unified format. This is totally different from other scenarios [4], [11], [12], [5], [13], [6], [14], [10], whose credential formats are hard-coded and structured. Moreover, we find that all super-app platforms enforce credential-based authentication. Our study unveils seven types of credentials in total, which perform different functionalities. For example, some of them are used to generate and manage other credentials, while some are applied to secure communication data. To better understand their use scenarios, we classify them into three novel categories.

Guided by the credential-use scenarios and categories, we manually analyse the security of credential usage in mini-apps. Our analysis reveals that the responsibility for safeguarding and managing credentials primarily falls on mini-app servers. When a mini-app (i.e., mini-app client) utilizes super-app services provided by its mini-app server (like image editing or optical character recognition), it is crucial for the server to authenticate the client and use its corresponding credential for accessing super-app services and returning the results (as shown in step ② and step ③ of Figure 1). However, unfortunately, mini-app developers may lack security awareness, resulting in a deficient understanding of the security and meanings of these credentials. They may improperly share their credentials with mini-app clients, i.e., credential migration, allowing their clients to interact directly with super-app platforms without server-side intervention (step ④). This common practice inevitably leads to security breaches when malicious mini-app users capture and abuse the shared credentials in their rooted devices. The credential leakage will threaten the security of both mini-app servers and super-app services, and the finding is further corroborated by our subsequent security impact assessment.

Detecting Credential Leakage. After thoroughly studying and understanding the security of credentials, we aim to learn and assess the security risks of credential leakage. Considering the large number of available mini-apps, an automated credential leakage detection tool is demanded. However, it is not an easy task to design and implement such a detection tool. In particular, as discussed in our aforementioned security study, the data value formats of the credentials used in the app-in-app authentication lack distinctive characteristics. Some of them are even transformed, e.g., encoded or encrypted. Existing credential detection solutions were designed only for structural and well-defined credentials, therefore, are not suitable for the scenarios focused on in this paper.

In this paper, we propose a novel credential security analysis approach, called KeyMagnet, that addresses the above challenge by understanding and verifying the credential-use semantics in both mini-app client- and server- sides. One key point behind our KeyMagnet approach is that during the credential migration from the mini-app server-side to the client-side, the credential-use behaviors still exhibit similar patterns on both sides when communicating with the super-app services. Therefore, our main idea is that we can compare the credential-use semantic patterns learned from both mini-app clients and servers. If a mini-app client has similar credential-use semantics, which should only be placed in the mini-app server-side, the risk of credential leakage may exist.

Guided by this, we design our KeyMagnet approach with three phases. First, KeyMagnet understands the credential-use semantics that should be held in mini-app servers. Although these semantics often occur in the background and are hardly retrieved, we find the mini-app development documentation provided by super-app platforms offers sufficient hints. Therefore, we conduct a documentation analysis to extract and formulate the credential semantics in mini-app servers. To achieve this, KeyMagnet designs and constructs a novel API-level semantic graph, called credential-use semantic graph (CSG), by mining credential-related interfaces and inferring their relationships from the large volume of documentation. Second, KeyMagnet learns the client-side credential semantics. Despite the credential formats being diverse and flexible, we observe that credentials are dynamically created and shared through the network. This means the client-side network behaviors must be a super-set of credential-use semantics. Thus, we download mini-app clients and conduct code semantic analysis by employing fine-grained data flow analysis in the mini-app client code. Through analyzing the data flows corresponding to network APIs, which reflect credential-use behaviors in the mini-app client, we can build a client-side behavior graph (CBG). Last, KeyMagnet compares the credential-use behaviors of the server- and client- sides, i.e., applying semantic-based analysis to detect similarity between the server- and client-side semantics (CSG and CBG). To this end, KeyMagnet bridges the semantic gap between CSG and CBG, and devises a novel graph matching technique to compute their similarity.

Our Findings. We implement the prototype of KeyMagnet, and apply it on 413,775 real-world mini-apps. Our results unveil that KeyMagnet accurately discovers leaked credentials in mini-apps (with a precision of 95.04% and a recall of 85.56%). Besides, we find that credential leakage is prevalent in the app-in-app ecosystem with 84,491 detected credential leaks, spanning over 54,728 mini-apps. Notably, our research points out that credential leakage appears in prominent corporations (e.g., Tencent and Baidu) and sensitive categories (e.g., Education and Government). In addition, severe security hazards may be caused by credential leakage, e.g., hijacking all users' accounts, stealing sensitive data of mini-app users and broadcasting malicious content through official channels. Upon further analysis, we discover that many credential leaks occur across mini-apps, e.g., a Baidu mini-app leaks the credential of its corresponding WeChat mini-app. We made efforts to collect the contact information of detected mini-apps and have reported the vulnerabilities to the corresponding developers. As of now, 89 reported issues have been assigned with CVE IDs and 38 of them are rated high severity.

In short, we make the following contributions.

- To the best of our knowledge, our work is the first to systematically study the app-in-app credential system and unveil its security implications.
- We propose a novel approach, called KeyMagnet[1], to detect the credential leakage in mini-apps.
- We have evaluated KeyMagnet with 413,775 mini-apps and have identified 84,491 credential leaks. We analyse the root causes of the prevalent leakage and propose corresponding mitigation strategies.

II. Study of Mini-app Credential System

A. App-in-App Background

Mobile apps recently introduced the app-in-app paradigm, which consists of super-app and mini-app platforms. Each platform has client-side and server-side. In a mobile device, a mobile app acts as super-app client and provides a web runtime to host mini-app client. A mini-app client is deployed as a web app. In the runtime, both super-app and mini-app client may communicate with their own servers. As introduced in Section I, super-app platforms provide mini-app servers a number of privileged resources and services, allowing mini-app servers to ease their mini-app clients' development and offer powerful services to their clients.

Super-app platforms apply a credential-based authentication to protect sensitive services. Specifically, as illustrated in Figure 1, the typical workflow of authentication is that super-app platforms issue credentials to mini-app servers. The credentials can only be used in the mini-app servers and cannot be disclosed to the mini-app clients. Before these services are accessed, the credentials should be provided to super-apps for verification. Thus, if credentials are leaked, anyone, in addition to the mini-app servers, can access these services. Recently, [9] and [10] have demonstrated the potential risks with concrete credential leakage cases. However, existing work only revealed a tip of the iceberg of the credential system. The design of the whole crucial credential mechanism still remains ambiguous. More importantly, its security has not been comprehensively understood or explored to date.

B. Understanding Credential System

We conduct a comprehensive study to understand the credential system in the app-in-app paradigm. To this end, we first collect and build our super-app dataset. We crawl popular Android apps from Google Play and their development documentation. In addition, we use the Google search engine to search for related keywords (e.g., "mini app" and "super app") and find more potential super-apps. By scanning these applications' documentation based on the keywords, we finally gathered 21 super-apps as our preliminary study dataset. These apps feature the app-in-app paradigm and are popular worldwide. In order to understand the credential mechanisms in these super-apps, two experts spend half a month on reading the developer documentation and analyzing the protocols designed by super-apps. The documentation explains the meanings of various parameters used in super-app provided services, and some documentation defines the credentials provided for mini-app developers to access these services. We can identify credentials based on the descriptions and understand their functionalities. Consequently, we successfully discover seven types of credentials across all super-apps (Table I). These credentials are mostly unstructured and dynamically retrieved from the mini-app servers. They are used to enforce the authentication before the service is accessed by mini-app servers.

Definition 1 (Unstructured Credentials): A structured credential is a type of credential that includes a fixed prefix, suffix, or string pattern. For instance, Amazon AWS credentials contain a fixed prefix string `AKIA` [5]. In contrast, unstructured credentials lack a fixed string pattern and are generated with random values. For instance, `AppSecret` in WeChat is a 32-character hexadecimal string, a format that many other strings also fit, making it difficult to distinguish the credentials from unrelated variables.

Definition 2 (Dynamically Retrieved Credentials): A dynamically retrieved credential is obtained from the server side and transmitted through network traffic. Compared to hard-coded credentials, they are not attached to the client, and thus cannot be discovered by static analysis of the client-side code.

We then study how these credentials are used, i.e., their functionalities and usage workflows. However, we find these credentials are designed in diverse ways, especially considering different super-app platforms are designed in inconsistent architectures. To unify and abstract their functionalities, we conducted an in-depth analysis of the credentials and found that most of them are used either to access privileged services

[1] The source code is avaliable at https://github.com/KeyMagnetProject2025/KeyMagnet.

TABLE I
Credential Utilization in High-profile Super-apps. Super-apps may have multiple credentials in certain types, such as Alipay and UnionPay have two types of data encryption keys. In total, there are 64 credentials across all super-apps.

	Credential Type	WeChat	Baidu	Alipay	Tiktok	Line	VK	WeCom	QQ	Feishu	JINRI toutiao	Watermelon	Pipixia	Kuaishou	Taobao	Cainiao	Koubei	Jingdong	Xiaohongshu	Paytm	UnionPay	DingTalk
Root Credential	Mini-app Root Credential	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓
	Corporation Root Credential							✓														✓
Access Credential	Mini-app Access Credential	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓
	Corporation Access Credential							✓		✓												✓
	Mini-app Group Access Key						✓															
Cryptographic Credential	Server-to-server Session Key	✓	✓		✓			✓	✓	✓	✓	✓	✓	✓	✓	✓			✓			
	Data Encryption Key	✓		✓													✓			✓	✓	

provided by super-apps or to safeguard data confidentiality during communication. Consequently, we grouped the credentials into three novel types: access credentials, cryptographic credentials, and root credentials. The first two types handle the functionalities mentioned above, while the last type is used to derive and manage the other two types. Next, we dive into the credential security. More details are discussed in the remaining subsections.

C. Threat Model

In the credential system, we assume that the super-app client and server, as well as the mini-app server, are trusted. In contrast, the mini-app client is untrusted. The attacker is a malicious mini-app user, which can impersonate a regular mini-app client to access sensitive resources and services with leaked credentials. Mini-app developers may migrate credentials to the mini-app client from mini-app server, which can be obtained by an attacker.

The attack objective is to compromise other mini-app clients (regular users), mini-app servers (developers), and super-app services. The attacker possesses the ability to manipulate the network traffic of the mini-app client, as the attacker's mobile device can be rooted. Besides, the attacker can directly launch attacks using leaked credentials without the need to develop malicious mini-apps, to hack into the mini-app server, or to install a malicious mini-app on user's mobile. We believe this attack scenario is practical and relatively easy to achieve.

D. Credential Leakage

Credential plays a crucial role in ensuring secure access to super-app services. In real-world practices, there is sophisticated access among different parties in the app-in-app diagram. As illustrated in Figure 1, the erroneous practice of mini-app servers is to share critical credentials with their mini-app clients. This creates a shortcut that allows their clients to directly communicate with super-app platforms without having to go through mini-app servers each time they access super-app services. As discussed in Section I, this inevitably leads to the credential leakage issue. We manually verify the credential security of the super-app platforms in our dataset and analyze the security hazards caused by the credential leakage. More details about the attack vectors are presented in Table I and Section II-E. Through stolen credentials, an attacker can easily bypass security enforcement deployed in both mini-app and super-app services, which can introduce serious security hazards, e.g., user account hijacking, user private information leakage and illegitimate paid services access.

E. Generalizing Attack Vectors

As summarized in Table I, we understand and generalize the attack vectors from the three proposed credential types (i.e., access, cryptographic, and root credentials), especially their functionalities and security mechanisms.

Access Credential. Access credential is used in super-app platforms to authenticate whether a requester has permission to access the resources and services (provided by super-app platforms). For instance, the high-profile short-form video app 'TikTok' utilizes an access credential (i.e., `accessToken`) to manage the access to its e-commerce service and fraud detection service. When a malicious mini-app user (e.g., *Mallory*) obtains an access credential, *Mallory* can freely access any services that should be paid, and steal the sensitive information, e.g., the logistics information and mini-app use data.

Cryptographic Credential. This class of credentials is widely adopted with popular cryptosystems, e.g., AES and RSA. They play a crucial role in ensuring the security of the communication channels between mini-apps and super-apps. Cryptographic credential is vital in protecting sensitive resources from being intercepted and tampered with malware data. For instance, Alipay implements a strict access mechanism combining symmetric and asymmetric keys to ensure the confidentiality and integrity of the communication process.

The secure practice for mini-app servers is to use the cryptographic credentials only in the communication between super-app platforms and mini-app servers. However, credential migration will leak cryptographic credentials. Specifically, many mini-app developers (e.g., a high-profile logistics mini-app 'Y***' and an aviation mini-app 'U***') directly send these credentials, i.e., sessionKey in WeChat, to the mini-app client-side.

When a cryptographic credential is compromised, serious attack effects can also be caused, e.g., hijacking all users' accounts belonging to the vulnerable mini-app servers. By using such a credential, a malicious mini-app user can contact either mini-app servers or super-app platforms and thus deceive them with fraudulent messages to launch security exploits. For instance, WeChat enables a 'One-Click Login' feature to let a mini-app client conveniently log into mini-app servers with a user's phone number. Technically, when a mini-app client requests One-Click Login, the super-app client and server support it by encrypting the user phone number n to n' with a sessionKey k. Please note that k is dynamically generated by the super-app platform and shared with the mini-app server. In normal cases, k and n's plain values are unknown to the mini-app client, which only holds n'. Then, the mini-app client forwards n' to the mini-app server for login approval. Last, the mini-app server verifies and decrypts n' with k to retrieve the user's phone number and let the mini-app client log in. However, oppositely, when k is leaked due to credential migration, a malicious mini-app user can capture k in the memory of his mini-app client and thus replace n with arbitrary phone numbers for content encryption. As a result, the attacker can successfully log into the mini-app server with other users' accounts and achieve mini-app account hijacking.

Root Credential. Different from the above two types of credentials, a root credential is not directly involved in the enforcement of service access control and communication confidentiality. Instead, it is applied to derive and manage other credentials. For example, AppSecret is the root credential in WeChat, and can be used to derive the access credential accessToken, which can control security access to all super-app services.

We discover that some mini-apps (e.g., a high-profile car information mini-app 'D***' in WeChat and a popular retail business mini-app 'I***' in Line), particularly those developed by outsourcing companies, leak the root credentials. Since all other credentials can be derived, all super-app services and mini-app servers (e.g., all mini-app users' accounts) may be directly exposed.

III. KeyMagnet Design

In this work, we propose a novel vulnerability detection approach, named KeyMagnet, to vet the security of real-world mini-apps against the credential leakage issue. As discussed in Section I, it's challenging to detect the leaked credentials used in the app-in-app authentication. Different from existing related problems [4], [11], [12], [5], [13], [6], [14], [10], we focus on the credentials used in the app-in-app ecosystem, which are mostly not hard-coded and lack distinctive characteristics. KeyMagnet tackles the difficulty by proposing a novel semantic analysis and comparison technique with three analysis phases as illustrated in Figure 2. First, KeyMagnet employs documentation analysis to automatically handle the developer documentation provided by different super-apps to extract credential-related elements and learn server-side credential semantics. Then, KeyMagnet conducts a fine-grained data flow analysis to extract all potentially important behaviors of mini-app client related to crucial data, including credentials. Specifically, KeyMagnet constructs the inter-procedure control flow graph to track the mini-app behaviors. Besides, to compare the semantics of mini-app server-side and client-side, we propose two novel graph structures to represent the credential semantics to bridge the semantic gap. Finally, KeyMagnet conducts semantic comparison between the semantic graphs with a novel graph matching technique and verifies whether a credential semantic pattern of the mini-app server-side appears in the mini-app client-side.

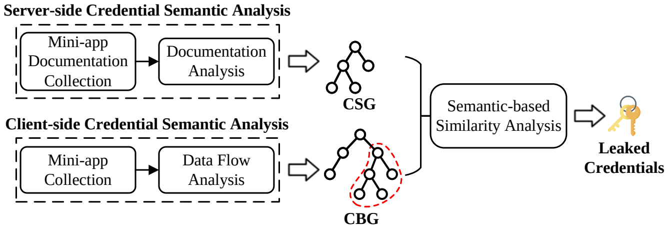
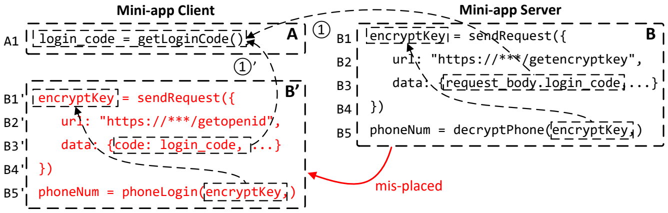

Server-side Credential Semantic Analysis
Mini-app Documentation Collection
Documentation Analysis
CSG
Client-side Credential Semantic Analysis
Mini-app Collection
Data Flow Analysis
CBG
Semantic-based Similarity Analysis
Leaked Credentials

Fig. 2. KeyMagnet Workflow

More technical details are presented in the remaining subsections. Specially, we first discuss the challenges and design insights, and then discuss each analysis phase.

A. Real-World Example and Design Insights



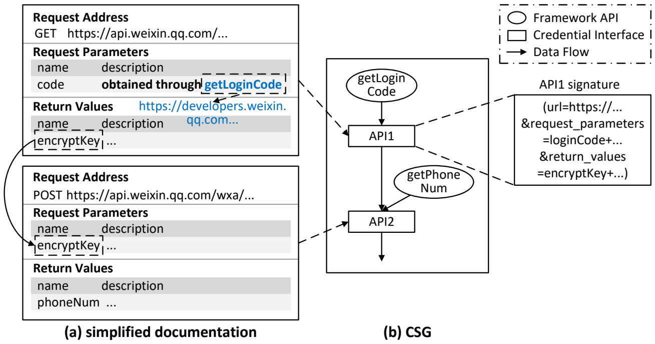
```
A1   login_code = getLoginCode()

B1'  encryptKey = sendRequest({
B2'      url: "https://***/getopenid",
B3'      data: {code: login_code, ...}
B4'  })
B5'  phoneNum = phoneLogin(encryptKey,)
```

```
B1   encryptKey = sendRequest({
B2       url: "https://***/getencryptkey",
B3       data: {request_body.login_code,...}
B4   })
B5   phoneNum = decryptPhone(encryptKey,)
```

Fig. 3. An Abstracted Example of Vulnerable Practice

We walk through a real-world example and explain our design motivation and insights as illustrated in Figure 3. In part A, the mini-app client obtains login_code from super-app platform using getLoginCode. The login_code serves as a unique identifier of current mini-app user. In part B, the mini-app server first extracts login_code from the request body in line B3. Then, the mini-app server sends a request to the super-app platform to obtain the cryptographic credential encryptKey in line B1. The secure practice should be to use the credential in the mini-app server-side, i.e., line B5 of part B. However, the mini-app developer may share the credential with the mini-app client, by misplacing operations B1-B5 to mini-app client (B1'-B5'). In line B1', the mini-app client sends a request to fetch the credential with the login_code obtained before, which will lead to the credential leakage.



(a) simplified documentation

(b) CSG

Fig. 4. Constructing Credential-use Semantic Graph

As illustrated by the motivating example, there are two main design insights:

Insight 1: Client-side Preserved Operations. The credential-based authentication provided by super-apps often requires the mini-apps to perform specific client-side operations to launch the authentication process. Unlike misplaced server-side preserved operations (part `B′` in Figure 3), such operations in mini-app clients are well-defined by mini-app framework interfaces, e.g., `getLoginCode`, which corresponds to `wx.login` in WeChat and `swan.login` in Baidu. Our analyzer first locates these operations and then infers the misplaced operations by analyzing similar contexts.

Insight 2: Data Dependency between Client-side and Server-side Preserved Operations. During credential migration, the credential-use behaviors still exhibit similar patterns in both sides when communicating with the super-app services. Specifically, mini-app developers will migrate corresponding interfaces to the mini-app client from mini-app server (with customized implementations) and the dependencies between client-side preserved operations and server-side preserved operations, whether they are misplaced or well-placed, remain unchanged (dependency ① and ①’). Thus, we can analyze the dependencies to identify if credential semantic patterns of the mini-app server-side appear in the mini-app client-side.

B. Server-side Credential Semantic Analysis

The credential-use semantics of the mini-app server-side are not straightforward, as many related operations run in the background and are out of reach. Nevertheless, the mini-app development documentation provided by super-apps offer some hints, e.g., credential interfaces (API) and their usage. Thus, our main idea for learning server-side semantics is conducting documentation analysis and building API-level credential semantics. To achieve this goal, we need to answer the following two crucial questions: 1) How to identify client-side preserved operations and server-side preserved operations? 2) Upon these found APIs, how to understand or reconstruct server-side credential semantics?

To answer the questions, the first step is to mine credential-related APIs for each super-app platform. Given that the development documentation presents each API in well-structured HTML elements, KeyMagnet can automatically parse the HTML content and extract the contained API information (such as API names, parameters, and descriptions). For client-side preserved operations, super-apps have provided their list, such as [15] in WeChat. For server-side preserved operations, KeyMagnet identifies these operations based on data dependencies between different APIs. We design a heuristic strategy to extract the API-level relationships according to the following observation: to facilitate mini-app developers to implement server-side operations, super-app platforms often indicate how to obtain parameters of one API in the descriptions. Specifically, API descriptions often directly attach hyperlinks pointing to the sources of the parameters, e.g., the description of `loginCode` contains a hyperlink, which points to its source API `getLoginCode` in Figure 4-a. Therefore, KeyMagnet starts from the client-side preserved operations and identifies the associated APIs based on the descriptions. This process continues iteratively until no new APIs are found. To comprehensively restore dependencies between APIs, we supplement the analysis by leveraging the naming similarity of corresponding variables, e.g., the WeChat cryptographic credential appearing in all APIs is named `session_key`. This approach can be easily extended to multiple super-app platforms due to well-organized documentation (i.e., needing less than 20 LoC updates on average for each super-app platform), and experiments demonstrate its scalability.

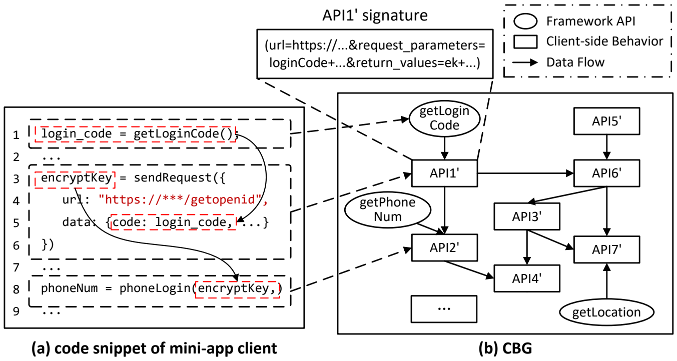
To represent the API-level semantics and accommodate the correlations between credential APIs, we design a novel graph, called credential-use semantic graph or CSG.

Definition 3 (Credential-use Semantic Graph): A credential-use semantic graph (CSG) represents the semantic of mini-app server-side and comprises credential-related operations. $CSG =< N, E >$, where N are extracted credential APIs and encompass two types of nodes: client-side preserved operations and server-side preserved operations. Besides, $E \subseteq N \times N$, which suggests the API dependencies.

We illustrate an example of CSG in Figure 4-b, in which each node represents the signature of different actions (e.g., API_1 and API_2), and the edge shows the data dependency between nodes. This graph shows the process of the acquisition and utilization of the credentials and contains semantic meanings. Then, we construct a collection of semantic graphs based on the usage patterns of different credentials, which will be utilized for further analysis.

C. Client-side Credential Semantic Analysis

In this step, KeyMagnet attempts to learn client-side credential semantics. As discussed in Section I, it is challenging to do so as credentials do not have obvious characteristics in the mini-app client-side. To address this, we introduce data flow analysis for different types of mini-apps. Some super-apps provide customized runtime engines (e.g., WeChat, Baidu, and Alipay), and we can crawl mini-apps’ code packages. Therefore, we perform a fine-grained data flow analysis to track network data and extract the potential important behaviors in mini-apps, as illustrated in Figure 5. However, some super-apps run mini-apps in a WebView environment. This type



```
1 login_code = getLoginCode();
2   ...
3 encryptKey = sendRequest({
4     url: "https://***/getopenid",
5     data: {code: login_code, ...}
6 })
7   ...
8 phoneNum = phoneLogin(encryptKey,)
9   ...
```

Fig. 5. Constructing Client-side Semantics

of mini-app needs to load code from the mini-app's own server. To analyze these WebView-based mini-apps, we design a dynamic crawler to fetch mini-app client-side behaviors. Besides, we design a novel graph, called client-side behavior graph or CBG to represent the mini-app client-side semantics.

Definition 4 (Client-side Behavior Graph): A client-side behavior graph (CBG) represents the behaviors in mini-app client-side. It encompasses two types of nodes: mini-app framework interfaces and customized client-side interfaces. The edges within the graph depict the data dependencies between these nodes. Figure 5-b illustrates an example of CBG.

CBG Construction. The key to constructing the CBG lies in establishing data dependencies between different nodes, including both explicit and implicit dependencies.

To extract the explicit dependencies, we first perform a fine-grained data flow analysis on the mini-app client code. Based on a static analysis tool JAW[16], we model the specific APIs provided by different super-apps and construct the inter-procedure control flow graph (ICFG) of each file in mini-apps. Since many function calls are cross-file in mini-apps, we need to perform inter-file analysis. For inter-file function calls, we resolve the 'require' relations and connect inter-file call edges to enhance the call graph. Then we extract the credential-related context information of parameters along the data flow to enrich the behavior semantics. For example, as illustrated in Figure 5-a, the context information of the parameter `loginCode` in line 5 contains its key `code` and source API `getLoginCode`. Through inter-file analysis of mini-app code packages, we can easily extract the dependency relations along the data flow. For example, if the return value of API A is further used by API B, there is an edge between A and B.

We also implement a dynamic analysis to complement our result. We simulate user interactions to explore the WebView-based mini-apps based on Android UI Automator [17]. Due to that the dynamic analysis is greatly limited in code coverage [18], [19], [20], we adopt some heuristic strategies to optimize it. Specifically, we observe that mini-app developers often provide hints near input boxes to help users effectively fill in valid information, such as data type. Therefore, we first dump the GUI hierarchy and collect elements near the 'EditText' to extract the hints. Then, we fill in the input boxes with pre-defined data based on these hints. Given that super-apps are in different countries, we customize this method to cater to the specific languages of each country, e.g., Japan and Russia. Finally, we propose a value-based approach to establish correlations between different APIs. We compare the values of parameters and return values of different APIs to analyze dependencies between them. We do not take into account the scenario of encrypted traffic, which is out of our work.

To extract the implicit data dependencies, we thoroughly analyze the developer documentation and model the patterns of implicit dependencies to complement the CBG. For example, WeChat mini-apps usually use `wx.setStorageSync(key,value)` to store necessary data and use `wx.getStorageSync(key)` to fetch the stored data with the same key, which cannot be directly traced by aforementioned steps. We notice that these patterns are mostly paired, especially in getter-setter pattern, such as `wx.setStorageSync`/`wx.getStorageSync` and `wx.setStorage`/`wx.getStorage`. Therefore, this problem can be transformed into identifying the paired implicit dependencies among the nodes. Then, we manually analyze the developer documentation to find data-related operations, including data setting and data getting, and collect these patterns. In addition to getter-setter pairs, we also identify the global variables that can bridge implicit data dependencies. We then employ a pattern-based approach to identify potential implicit dependencies on the same variable and supplement edges to the CBG.

D. Semantic-based Similarity Analysis

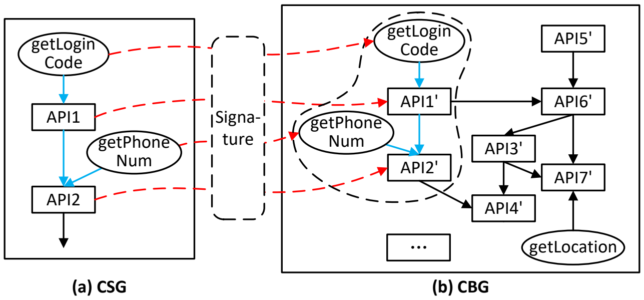


Fig. 6. Similarity Analysis Between CSG and CBG. The red dotted lines connect similar nodes and other lines represent the data flow.

After we understand the credential semantics in the mini-app client- and server-side, i.e., CSG and CBG, our objective is to determine whether the semantics of CSG exist within CBG, i.e., whether CSG corresponds to a subgraph of CBG. As these semantics are collected from different perspectives, there is a gap between these two-level semantics. To bridge the semantic gap and find if there are credential semantics in the mini-app client, we turn to prove the semantic isomorphism between the CSG and CBG, and design a novel semantic-based similarity analysis algorithm to detect credential leakage based on CSG and CBG. We achieve this by identifying the existence of a bijective function between the nodes of CSG and CBG, i.e., whether CSG corresponds to a subgraph of CBG.

For each CSG previously constructed, we can directly match client-side preserved operations, e.g., `getLoginCode` and `getPhoneNum`, which remain unchanged during credential migration. Concerning other operations, we find that operations with same functionalities have similar contexts or signatures (e.g., parameters and data dependencies) in mini-app client- and server-side. Consequently, we can match the nodes based on the signature information. As illustrated in Figure 6, API_1 in CSG and API_1' in CBG accept the same parameter from `getLoginCode`. Besides, API_2 and API_2' in the two graphs accept the same parameters from `getPhoneNum`, and similar parameters from API_1 and API_1'. Therefore, we can correlate these nodes based on similar signatures. In the simplest scenario, where mini-app developers migrate credential APIs (involving credentials in parameters or return values) directly to the mini-app client-side, we can readily identify vulnerable practices and do not need the following steps. As for other scenarios, we also need to check the dependencies between nodes to match the semantic graphs. In summary, we take a two-step way to perform subgraph matching: the first step is to match the nodes, and the second step is to compare the relations among these nodes. Based on the above observations, we design an algorithm that leverages the context information to bridge the semantic gap and prove the semantic isomorphism between CSG and CBG (Algorithm 1). The detailed steps are outlined below.

Node Inference. In the first step, we aim to find the equivalent nodes in CSG and CBG. If these nodes are client-side preserved operations, e.g., `wx.login`, we can directly match them based on the signature. It's challenging to detect misplaced server-side preserved operations as they are in various implementations in the mini-app client-side. The key insight is that the equivalent nodes have similar contexts or signatures. We first analyze the parameter information. As for each parameter of the nodes, we fetch the source of it based on data dependencies, that is, the client-side behavior that generates the parameter. If the sources are the same (e.g., the same framework APIs), we assume that the parameters are also the same. Although most custom parameters do not have distinct sources and even obfuscated, the information during data propagation and usage can help us understand the semantics of the parameters. For example, if a parameter `pvar` originates from a custom API `getPassword`, we can extract its semantic information related to 'password'.

Therefore, we first obtain the contexts of the parameters, which includes data dependencies (extracted based on the dataflow analysis) and parameter names. Then we perform similarity analysis on the context information using the Levenshtein Distance algorithm [21]. In order to increase the accuracy of node inference, we perform a preliminary study to choose the best similarity threshold for our evaluation. Inspired by previous work [22], [23], we change the similarity threshold from 0.8 to 0.95 with 0.05 as an interval, and manually check the corresponding detection result. Consequently, we choose a strict threshold of 0.9 to compare the context similarity while keeping low false positives. Finally, if a CBG node n_j has the same source or has similar contexts to a CSG node n_i, we can correlate these nodes.

Algorithm 1: Algorithm of Semantic-based Similarity Analysis

```
Input  : CSG $G_1$ of a credential and CBG $G_2$ of the target
         mini-app
Output: Matching result $isDetected$ between $G_1$ and $G_2$
Data   : Corresponding nodes map $M$
// step 1: infer the corresponding nodes
   between $G_1$ and $G_2$
$M \leftarrow \{\}$;
for action node $n_i$, id in $G_1$ do
    for action node $n_j$ in $G_2$ do
        if $n_i$.url == $n_j$.url then
            M[id].add ($n_j$);
            continue;
        if $n_i$ and $n_j$ are the same operations then
            M[id].add ($n_j$);
            continue;
        isMatched = true;
        for parameter $p$ in $n_i$ do
            if source of $p$ not in source of $n_j$ then
                if context of $p$ and context of $n_j$ are not similar
                  then
                    isMatched = false;
                    break;
        if isMatched == true then
            M[id].add ($n_j$);
// step 2: match the relations among nodes
$isDetected \leftarrow true$;
for edge $e$ in $G_1$ do
    $id_i, id_j$ = e.nodes; // extract nodes of the edge
    $hasEdge \leftarrow false$;
    for $node_{prev}$ in $M[id_i]$ do
        for $node_{next}$ in $M[id_j]$ do
            if an edge exists between $node_{prev}$ and $node_{next}$
              then
                hasEdge = true;
                break;
        if hasEdge then
            break;
    // no corresponding edge in CBG
    if not hasEdge then
        isDetected = false;
        break;
```

Similarity Analysis. In the second step, we compare the edges in these two graphs, that is, the data dependencies between the nodes. We apply a depth-first exploration strategy to explore the edges in CSG and check if there are corresponding edges between the nodes in CBG. Each time we select an edge from the CSG and traverse the corresponding nodes in CBG, i.e., $node_{prev}$ and $node_{next}$ in CBG which are acquired during the node inference phase. Then we check if there is an edge between $node_{prev}$ and $node_{next}$. Since there may be multiple corresponding nodes, as long as there are data dependencies between any pair of $node_{prev}$ and $node_{next}$, we consider there is an edge. We can also correlate $node_{prev}$ and $node_{next}$ based on the data dependencies when there are other nodes between them. If all the edges in CSG match the edges in CBG, the round of similarity analysis is finished, and it can be inferred that the mini-app leaks the credential corresponding to CSG. Since multiple credentials may be leaked in the mini-app, we will match all semantic graphs to check for any leaks.

IV. Large-Scale Assessment of Credential Leakage

We apply KeyMagnet to understand and assess the credential leakage issue on a large scale of real-world popular mini-apps. In this section, we first introduce the experiment setup and then present the overview of our assessment results. Next, we evaluate the accuracy of KeyMagnet. Last, we discuss the interesting findings and the potentially caused security hazards with case studies.

A. Experiment Setup

Prototype Implementation. We implement the first module (the documentation analysis) and the third phase (the graph matching algorithm) in Python with 3,205 lines of code (LoC) in total. We implement our fine-grained data flow analysis based on the static tool JAW [16] with 2,562 JavaScript LoC and 1,827 Python LoC, and our dynamic analysis based on UI Automator [17] with 910 Python LoC.

Dataset Collection. We find many super-apps listed in Table I belong to the same 'family' (e.g., both WeChat and QQ belong to Tencent) and have similar credential systems. To evaluate the effectiveness of our solution across different credential systems, we cluster these 21 super-apps and select popular ones for our further analysis. Then, we extend the open-source tool MiniCrawler [24] to support mini-app crawling of different super-apps. Finally, we totally collect 413,775 mini-apps as our database, including 214,602 WeChat mini-apps, 86,570 Baidu mini-apps, 93,130 Alipay mini-apps, 15,064 TikTok mini-apps, 3,984 Line mini-apps and 425 VK mini-apps.

B. Experiment Result Overview

We apply KeyMagnet on our dataset on a server with 64 CPU cores (2.3GHz) and 206GB memory. We successfully analyze 402,527 mini-apps and the remaining mini-apps fail to be analyzed due to timeout or AST parsing errors. On average, our analysis takes 23.6s for each mini-app. The results unveil that credential leakage is prevalent in the app-in-app ecosystem, with 84,491 detected credential leakage issues spanning over 54,728 mini-apps. Severe security hazards may be caused by credential leakage, e.g., hijacking all users' accounts belonging to mini-app servers, stealing the sensitive data of mini-app users and maliciously manipulating the functionalities of mini-app servers. Moreover, we find many credential leaks are cross-app and introduced by vulnerable mini-app development templates, which makes the issue more serious. For example, we find 35 vulnerable mini-apps in our dataset, developed with the same mini-app template, leak their root credentials due to the vulnerable template. As of now, 89 reported issues have been assigned with CVE IDs.

TABLE II
Performance of KeyMagnet

Super-app	TP	FP	TN	FN	Precision	Recall	F1-score
WeChat	466	34	389	111	93.20%	80.76%	86.54%
Baidu	483	17	488	12	96.60%	97.58%	97.09%
Alipay	478	22	430	70	95.60%	87.23%	91.22%
TikTok	476	24	363	137	95.20%	77.65%	85.53%
Line	110	8	490	10	93.22%	91.67%	92.44%
VK	0	0	425	0	-	-	-
Overall	2013	105	2585	340	95.04%	85.56%	90.05%

C. Accuracy

Given that there is no ground truth, we evaluate the false positives of KeyMagnet by randomly selecting 500 mini-apps identified as vulnerable from each super-app. To evaluate the false negatives, we randomly selected 500 mini-apps that KeyMagnet marked as without credential leakage. If the count of detected mini-apps is less than 500, we used the entire dataset for evaluation.

Then we manually check the sampled dataset. First, two experts separately verify these potentially leaked credentials with validation APIs (e.g., `getAccessToken` [25] as used in [10]), which can be successfully called if the credential is valid. We only consider the credential leakage a true positive if two experts both agree. Second, to check false negatives of KeyMagnet, the experts manually analyze the mini-app code and monitor network traffic to check for credential leakage. The results are summarized in Table II. KeyMagnet can achieve good performance in all super-app platforms and the F1-score even reaches 97.09% in Baidu. The average precision of KeyMagnet is 95.04% and the recall is 85.56%.

False Positives. Most of the false positives are introduced during the node inference phase (Section III-D). Unrelated APIs with similar semantic structures in mini-apps may be identified as credetial-related APIs, which can lead to the failure of node inference. Additionally, several mini-app developers use customized parameters in credential-related APIs, which maintain the data dependencies and can match the CSG. Consequently, KeyMagnet incorrectly considers these parameters as potential credentials, resulting in false positives.

False Negatives. The false negatives are primarily attributed to incomplete API dependencies. Mini-app developers share valid credentials with the mini-app client, but the credentials are not further used. KeyMagnet cannot determine whether it is a credential without credential-use semantics. Besides, some mini-apps leak credentials in the network traffic, but do not retrieve or use the the credentials in the mini-app client side.

TABLE III
The Statistics of Large-scale Assessment

Super -app	Root Credential		Access Credential		Crypto Credential	
	#app	%total	#app	%total	#app	%total
WeChat	22207	10.89%	20987	10.29%	**23421**	**11.48%**
Baidu	517	0.60%	336	0.39%	**1085**	**1.26%**
Alipay	3929	4.24%	**5916**	**6.38%**	3092	3.33%
TikTok	268	1.78%	**1889**	**12.56%**	726	4.83%
Line	**69**	**1.73%**	49	1.23%	0	0
VK	0	0	0	0	0	0
Overall	26990	6.71%	**29177**	**7.25%**	28324	7.04%

D. Landscape of Credential Leakage

Prevalence. We find that the credential leakage problem is prevalent in practice. As shown in Table III, we have detected 84,491 credential leakage issues in mini-apps. To make matters worse, our findings reveal that many vulnerable mini-apps are from prominent companies, including Tencent and Baidu, which pose a significant risk to the sensitive data of hundreds of millions of users.

The result shows that there are more vulnerable mini-apps in WeChat due to the large base. Additionally, the access credentials are leaked most because developers frequently employ them to access various services. The leakage of root credential and cryptographic credential is also widespread. The main reason is that mini-app developers often acquire access credentials with root credentials in mini-app client-side, which leads to the leakage of both credentials. Moreover, cryptographic credentials are frequently used in mini-apps to handle encrypted data, e.g., the phone number.

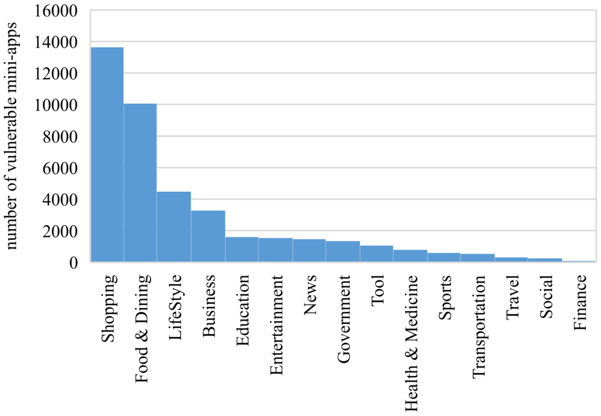


Fig. 7. Distribution of Vulnerable Mini-app Categories

Distribution. We then analyze the number of vulnerable mini-apps across different categories from WeChat and Baidu, which provide category information for the mini-apps. Finally, we obtain category information for 41,015 vulnerable mini-apps and Figure 7 illustrates the distribution. We find there are more vulnerable mini-apps in the categories that are frequently used by users, e.g., Shopping and Food. We randomly select 100 mini-apps from these categories and find most of them are developed by third parties with similar designs. It is worth noting that several crucial categories, e.g., Government and Education, also have a high proportion of credential leakage.

Credential Leakage Crossing Apps. By analyzing the results, we find that many mini-apps in different super-app ecosystems share the same security patterns, especially for the cases that are developed by the same mini-app developers or companies. Although different super-apps have their own interfaces and designs, the mini-apps developers still keep the same core code and only adjust a little bit of code for super-app specifications. In order to minimize the development costs on different super-apps, it is common practice for mini-app developers to use third-party mini-app frameworks, such as taro [26], uni-app [27], or code conversion tools [28], [29] with slight modifications to implement cross-superapp conversion for mini-apps. These approaches help to streamline the development process and reduce the overall cost of developing mini-apps for different super-apps. However, this also amplifies the problem of credential leakage. If there is credential leakage in a mini-app of one super-app, there may be credential leakage in other super-apps as well. For instance, a Baidu e-commerce mini-app ‘J***’ leaks the WeChat root credential of the corresponding mini-app in WeChat, due to that the vulnerable code is retained during mini-app development.

In addition to the cross-superapp leakage, we also find cross-miniapp credential leakage. We randomly sampled 100 WeChat mini-apps which leak the root credential and manually recover the credential value. Then, we use a regular expression `wx[a-f0-9]{16}` to search for possible appIds in the mini-app and check which appId the leaked root credential belongs to. We find 47 mini-apps that leak the root credentials of other mini-apps. Moreover, there may be cooperative relationships between different mini-apps, where one mini-app invokes services from other mini-apps (e.g., a hotel mini-app invokes the ‘getQRCode’ service with the root credential of a management mini-app ‘K***’), which can cause cross-miniapp credential leakage.

Template-based Leakage. We find tens of thousands of mini-apps are developed with mini-app templates and thus share similar vulnerabilities. Due to the lack of development capabilities, many merchants delegate the development and management of their mini-apps to third-party platforms (e.g., youzan[30], weimob[31], CloudBase[32]). These platforms use mini-app templates to provide development services and more than 300 platforms that provide template services are found in our dataset. However, the templates have introduced new attack vectors and led to the propagation of vulnerabilities. For example, if a mini-app has credential leakage, it is highly probable that other mini-apps developed with the same template or even by the same third-party platform also suffer credential leakage.

Leakage Scenarios. We conduct an in-depth analysis of scenarios where credentials are frequently leaked, that is, the functional scenarios that mini-app developers tend to share credentials with mini-app clients. Leveraging the credential

TABLE IV
Functionality of Credentials in Mini-apps

Service	WeChat	Baidu	Alipay	Tiktok	Line	VK
Mini-app Login	✓	✓	✓	✓	✓	✓
UserInfo Retrival	✓			✓		✓
Data Analysis Service	✓	✓		✓		✓
Customer Service	✓	✓	✓	✓		
Message Service	✓	✓	✓	✓	✓	✓
Cloud Development	✓					
Payment Service	✓		✓	✓	✓	
AI Service	✓					
Logistics Service	✓	✓				
Shopping Service	✓		✓	✓		
Promotion Service		✓		✓		✓
Live Streaming Service	✓			✓		
Bio-authentication	✓		✓			
Content Security Check	✓	✓		✓		✓
Short Link Generation	✓	✓	✓	✓		✓

semantic graphs previously constructed, we can pinpoint the credential semantics associated with the credential leakage and the credential-use behaviors in the mini-app client-side.

We find the most common vulnerable scenarios are 'phoneLogin' and 'createQRCode'. The first scenario corresponds to the 'One-Click Login' feature provided by super-apps to easily log into mini-apps with the cryptographic credential. The second one is to generate the QR code of mini-apps with the access credential, which is also a common functionality to prompt the mini-apps. However, the access credential can also be used to access many privileged services, e.g., mini-app use analysis. Mini-app developers may not be aware of security hazards and expose the credentials.

E. Security Hazards and Case Studies

In order to understand the security hazards that can be caused by credential leakage, we analyze the super-app resources and services that can be accessed with credentials. By analyzing development documentation provided by super-apps, we have identified 15 sensitive services provided by super-apps, as illustrated in Table IV. Most of the vulnerable services can introduce security impacts on mini-app users, e.g., Mini-app Login, UserInfo Retrival and Message Service. We have verified that when a credential is leaked, these sensitive services can be freely abused, which can cause severe hazards, including account hijacking, phishing attack, and sensitive information theft.

Furthermore, many services are used for mini-app development, e.g., payment service and cloud development. This implies that leakage attacks can also pose security threats to mini-app developers. Attackers can exploit these services under the guise of the developers' identities, potentially disrupting the normal operations of developers in mini-apps. Besides, attackers may forge requests as the victim developers to deceive the mini-app server with leaked credentials, thereby bypassing the necessary security checks.

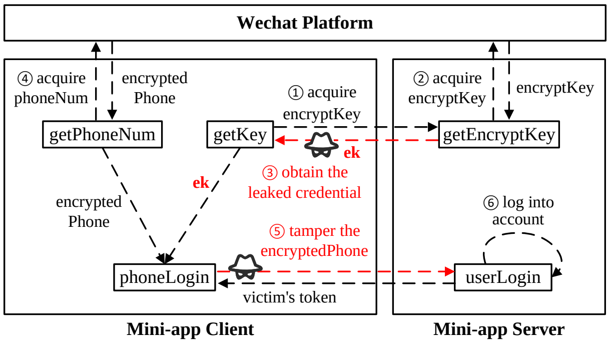


Fig. 8. Process of Account Hijacking. The boxes represent the mini-app client- and server-side behaviors and the red lines represent vulnerable parts that can be monitored or manipulated by attackers.

Below, we discuss more details of the caused security consequences with case studies.

Account Hijacking. 'One-Click Login' is a common service used to quickly log into mini-apps, which may suffer account hijacking attack due to credential leakage, as illustrated in Figure 8. In the error practice, the cryptographic credential `encryptKey` is leaked in the response of `getKey` (step ①). Specifically, an attacker can monitor the network traffic and obtain the leaked credential in step ③. Then, the attacker can intercept the network traffic and get the `encryptedPhone`, which is retrieved by `getPhoneNum` (step ④). With the leaked credential, the attacker can tamper the `encryptedPhone` with any user's phone number in step ⑤ and send the tampered data to the mini-app server. When the mini-app server receives the tampered data, it extracts the tampered phone number and returns the token bound with the tampered phone number for request approval. Finally, the attacker can successfully hijack any user's account and fetch all the user's information stored in the mini-app, e.g., health status, billing address, and education information.

We have found that a high-profile logistics mini-app 'Y***' in WeChat, which more than 100,000 users have visited recently, leaks the cryptographic credential. We successfully hijack the testing account with the testing phone number. Then, the victim's sensitive information, such as purchase history and billing address, will be exposed. We have reported this issue to the corresponding developers. They promptly acknowledged the issue and fixed it before the article was submitted.

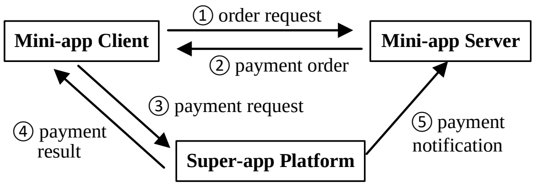


Fig. 9. Workflow of Payment Service

Payment Deception. Some credentials are used to safeguard the payment process, which are called payment credentials. We illustrate the payment workflow in Figure 9. The mini-app client first sends an order request to the mini-app server (step ①). Then the mini-app server generates a signed order with a payment credential and sends it to the mini-app client (step ②). To launch the payment process, the mini-app client sends a payment request to the super-app platform with the signed order (step ③). When the payment is finished, the super-app platform will return the result to the mini-app client (step ④) and send an asynchronous notification to a notification url which is set by the merchant in the meantime (step ⑤).

Yang et al[4] defined several security rules for in-app payment, and one rule emphasises that never to place any credential (e.g., the private key for signing) in the merchant app (i.e., the mini-app client). However, several mini-app developers violate these rules. For example, we find a case of cross-superapp leakage in a mini-app designed for selling building materials. Specifically, the WeChat mini-app leaks Alipay's cryptographic credentials. With the leaked credentials, an attacker can manipulate the order information, such as tampering with the price, and generate a valid signature, and even shop for free.

Phishing Attack. Line provides a messaging channel to mini-app developers and lets them send messages/notifications to the users. In the process, an access credential is used as a mean of authentication for channels. The root credential in Line can be used to retrieve the access credential. When the access credential or the root credential gets leaked, an attacker can access and use the messaging channel to broadcast malicious messages. Because users cannot distinguish whether the message is sent by the developer or the attacker from the client-side, they are susceptible to deception. For example, we find that a retail mini-app 'M***' leaks the access credential in the response of ***`/members_card`(mini-app url).

Sensitive Information Theft. Access credentials can be used to safeguard sensitive resources, such as the logistics information, user portrait and mini-apps' realtime logs, which are stored in the super-app server. When their leakage occurs, severe security issues may be caused. For example, an attacker can use leaked access credentials to obtain express orders and mini-app logs from super-app servers, which include records of user activities within the mini-apps. This process requires no additional requirements, such as installing a malicious mini-app on user's mobile, as these information is managed by the super-app platforms. To understand the impacts, we perform an analysis on the logs in 11,955 mini-apps which uses the log API [33]. Then we leverage the variable name in the mini-app code to understand the semantic meaning. In the end, we find 1,930 mini-apps record sensitive information in logs and present the top-6 ones in Table V, e.g., user's location and account information.

Mini-app Function Manipulation. With the leaked access credentials, attackers can manipulate certain functionalities integrated in mini-apps provided by super-apps, e.g., live streaming service in WeChat [34]. Attackers can manipulate the product information sold in live rooms, such as modifying prices or directly deleting products. As for the logistics service, apart from sensitive information leakage, attackers can cancel users' delivery orders, which can greatly disrupt the normal use of the mini-app.

TABLE V
Top Private Information Recorded in Mini-app Logs

Item	# mini-app	Item	# mini-app
Location	31	Account info	33
Phone number	12	Device info	95
Credential	60	Setting info	221

V. Discussion

Ethical Considerations and Vulnerability Disclosure. Throughout the course of our research, ethical considerations are at the forefront. We have complied with all relevant laws/regulations and followed community practices, such as [35]. This whole research has been approved by our institution's IRB and this study is considered as "minimal risk" when consulting with IRB staff.

To check the validity of credentials, we take multiple precautions to call the validation APIs and do not store any data returned from the calls. To understand the security hazards caused by credential leakage, we develop our own testing mini-apps for the in-depth analysis and restrict the vulnerability exploitation verification to internal tests using our own accounts, with the explicit consent of all participants involved. Furthermore, to prevent unintended access, we configure the testing mini-apps to be unsearchable. We make sure not to store any leaked credentials or launch any attacks against third-party entities to collect or manipulate their data.

Finally, we adhere to responsible disclosure practices of our findings to the super-app platforms and mini-app developers. We actively worked together with them to fix these problems. Given the widespread and serious nature of the credential leakage issue, we reported the detected vulnerabilities to the super-app platforms and provided several mitigation measures. Then we tried to notify the affected developers. Unlike the marketplaces of mobile applications (such as Google Play), which provide developer contact information [36], super-app platforms do not provide such information. Therefore, we made efforts to extract contact information from various sources, including privacy policies, and the official websites of the companies behind the mini-app. Consequently, we collected emails for 18,398 mini-apps. Most of the remaining mini-apps possess no privacy policy and were developed by individual developers, making it difficult to collect the corresponding contact information. We have reported the vulnerabilities to the corresponding developers.

To ease the burden on mini-app developers in understanding the vulnerability report, we first explained the nature of credentials in mini-apps and the credential leakage issue. Additionally, we put the detailed information in an attached report and proposed several fix suggestions to remove the credential logic from the mini-app client. To help developers verify and fix the vulnerability themselves, we explained the potential vulnerable APIs used in the mini-app client. This allows mini-app developers to easily analyze the server-side logic and determine if these APIs are susceptible to credential leakage issue. Note that the responsible disclosure is private and only available to the relevantly affected mini-app developers. All vulnerable mini-apps discussed in this paper are anonymized or masked. Most of the mini-app developers were previously unaware of the security risks associated with sharing credentials in the mini-app client-side. A mini-app developer responded that they were aware of the issue and did it this way to meet the development deadline.

As of the time of writing, 89 vulnerabilities are assigned with CVE IDs, with 38 of them rated as high severity, while the remaining are rated as medium severity. For example, CVE-2023-3***2, which is related to root credential leakage, has a CVSS score of 8.2. Although all the CVEs are credential leakage, it appears to involve a degree of subjective judgment regarding CVSS score. For mini-apps with similar functionalities, the "Confidentiality Impact" scores for some reported mini-apps were all high, whereas the scores for others were all low, resulting in some being rated as medium. Upon sampling 100 reported vulnerabilities and rechecking them, we find that 26 mini-app developers have fixed the issues or taken down their mini-apps.

Generality of KeyMagnet. It is important to note that our approach is extensive and scalable. Our approach relied on the analysis of the development documentation (provided by super-app platforms) to extract necessary information (e.g., credential-related APIs). Since the documentation of different super-app platforms is often well-organized, we can easily extend our approach to multiple super-app platforms with minimal human effort. The primary effort involves updating the documentation sources and HTML patterns (i.e., needing less than 20 LoC updates on average for each super-app platform). Besides, we have conducted analyses on six different super-app platforms. The experimental results demonstrate our method's effectiveness in detecting credential leakage on different super-app platforms.

Limitations and Future Work. The assessment shows the effectiveness of KeyMagnet in detecting credential leakage in the app-in-app ecosystem. Below we will discuss the limitations of KeyMagnet.

First, as stated in Section IV-D, our method will introduce some false negatives, mainly because the leaked credentials are not further used. Besides, developers may use credentials in a way that does not follow the documentation. We now take a conservative strategy to improve the soundness and plan to adopt a more relaxed strategy to cover these situations. Due to limitations in code coverage during dynamic analysis, we might overlook network traffic that exposes credentials. Additionally, our strategies can not fully bypass scenarios that require login or registration to proceed in certain mini-apps. Such scenarios necessitate human intervention, and in the future, reinforcement learning strategies may be employed to improve the progress of dynamic analysis. Furthermore, our current work focuses on the analysis of mini-apps in 6 prominent super-apps. The evaluation can demonstrate the scalability of KeyMagnet, and we plan to extend KeyMagnet to analyze the mini-apps in other super-apps in the future.

Mitigations. In order to mitigate the vulnerabilities, mini-app developers must be vigilant and ensure that credentials are strictly utilized in the mini-app server-side, avoiding any embedding or transmission to the mini-app client-side. Besides, mini-app developers should check if any credentials are contained in the traffic data that can be obtained by attackers. Upon discovering a mini-app that leaks its credentials, mini-app developers should check whether other developed mini-apps in different super-apps have similar vulnerabilities. Additionally, third-party development platforms need to be especially vigilant regarding this issue. Given that many mini-apps are developed with mini-app development templates, they should be responsible for ensuring the prevention of credential leakage before offering development services.

As for the super-apps, it is crucial to provide mini-app developers with clear and accurate documentation, particularly ready-to-use example code, to help them understand how to correctly use credentials. Existing developer documentation tends to be convoluted, and the official examples are rather abstract, posing challenges for mini-app developers in understanding. What's worse, the examples provided by official sources may not necessarily be secure, and those incorrect usage examples can lead to mini-app developers' erroneous practices. For example, we find an official example provided by a famous online shopping super-app for implementing RSA encryption suggests mini-app developers write the private key in the client-side. We have reported these issues to the corresponding platforms. Additionally, before a mini-app is released, super-app platforms need to detect potential credential leakage. We believe that KeyMagnet provides a valuable tool for detecting and mitigating credential leakage in the app-in-app ecosystem, and it is feasible to integrate our checks into the workflow of super-app platforms.

From the perspective of protection after credential leakage, super-app platforms can provide developers with measures such as IP whitelist to prevent malicious access by attackers. For cryptographic credentials, super-app platforms can introduce verification mechanisms to check the integrity of encrypted data and prevent tampering by attackers. Furthermore, these credentials can be rotated regularly and have a revocation mechanism in place. Besides, super-app platforms can implement fine-grained access control, restricting the resources and services that the credential holder can access.

More importantly, super-app platforms can establish a comprehensive vulnerability disclosure mechanism for mini-apps. It is often difficult for security researchers to contact mini-app developers directly. Therefore, super-app platforms can offer a reporting channel to help quickly fix vulnerabilities and enhance the security of the entire app-in-app ecosystem.

VI. Related Work

Detection of Credential Leakage. Credential leakage is a serious security problem. In past years, several techniques have been proposed to detect the problem. However, as discussed in this work, existing techniques mainly worked on the detection of hard-coded and well-structured credentials. Existing techniques mainly focused on open-source repositories [5], [7], [37], [8] and revealed numerous credentials, including API keys, passwords, and cryptographic keys, might leaked. It is particularly concerning that many developers stored the crucial credentials in plain text in client-side [5], [8], [37], which posed a significant security risk.

Pattern-based search and heuristics-driven filtering techniques are the primary methods for detecting credential leakage [5]. However, these approaches introduce many false positives and false negatives. In recent years, machine learning has emerged as a powerful tool for learning the characters [38]. A number of studies [7], [37], [39] have explored the use of machine learning techniques to decrease the false positives of traditional approaches in credential leakage detection. For example, PassFinder [38] utilized contextual information to detect passwords in source code.

The most similar studies to us are [10], [9]. They used a pattern-based method to detect credential leakage. [10] used a regular expression with a 32-byte hex-string format of `[a-f0-9]{32}` to search for possible credentials in the mini-apps, and [9] extended the regular expression to find more potential credentials. However, like the previous work, this introduced many false negatives and could only detect the hard-coded credentials. Table VI highlights how our paper compares with other existing related work. Based on the forms of credential leaks, we categorize them into three types. For the hard-coded credentials, some of them have fixed structures, such as AWS keys, but most are unstructured. In addition, some developers may employ transformation techniques to safeguard their credentials. Most of current work cannot detect the unstructured credentials and the dynamically leaked credentials. Therefore, existing techniques cannot be applied or extended to address the research problem that this paper aims to address. Our paper proposes a novel method based on credential semantics that can effectively detect credential leakage among sophisticated scenarios.

Studies on Mini-app and Its Security. As the popularity of mini-apps, there are several studies focusing on their security. [43] first systematically studied on the resource management in app-in-app systems and revealed some privilege escalation vulnerabilities. [24] developed MiniCrawler and conducted a large-scale analysis on mini-apps. [44] performed a systematic study of privacy over-collection in mini-apps. [45] discovered identity confusion vulnerabilities against mini-apps, which allow an attacker to invoke privileged capabilities and can lead to severe consequences, such as manipulating users' financial accounts. Another vulnerability has been discovered by [46], and they found the security issues of cross-miniapp communication, which allow attackers to inject a forged request into a vulnerable mini-app leading to various security consequences, such as sensitive information leakage and even shopping for free. [47] and [48] conducted research on discrepancies for super-apps in different platforms and found several discrepancies in WeChat. [49] revealed hidden undocumented APIs in super-apps, which can be potentially expolited to access protected resources.

TABLE VI
Comparison with existing work on credential leakage detection. "Structured" means credentials in fixed format; "Transformed" means credentials with transformation measures such as concatenation and encryption; "Dynamic" means dynamically retrieved credentials without fixed and well-defined formats.

Detector	Structured	Transformed	Dynamic
PlayDrone[40]	✓	✗	✗
Sinha et al.[12]	✓	✗	✗
Yang et al.[4]	✓	✗	✗
Wen et al.[41]	✓	✗	✗
Meli et al.[5]	✓	✗	✗
Saha et al.[7]	✓	✗	✗
Lounici et al.[37]	✓	✗	✗
GitLeaks[8]	✓	✗	✗
PassFinder[38]	✓	✗	✗
PrivRuler[42]	✓	✗	✗
Zhang et al.[10]	✓	✗	✗
Baskaran et al.[9]	✓	✗	✗
CredMiner[13]	✓	✓	✗
LeakScope[6]	✓	✓	✗
KeyMagnet	✓	✓	✓

Most recently, [10], [9] proposed a concrete credential leakage problem in mini-apps and found that many mini-app developers hard-code the `AppSecret` in mini-apps. However, most credentials are dynamically generated by super-app platforms and do not have a fixed or unified format. In this work, we conduct the first systematic study on the app-in-app credential system and propose a novel approach to unveil the vulnerabilities inside it.

Security Analysis of JavaScript Programs. There has been a significant amount of researches conducted on the analysis of web and JavaScript programs [50], [51], [52], [53], [54], [55]. Numerous studies have focused on identifying vulnerabilities and potential attacks in JavaScript code and developing techniques for detecting and mitigating these security risks. For instance, JAW [16] proposed a hybrid structure to detect client-side CSRF vulnerabilities. Kang et al. proposed ProbeTheProto [56] and measured the client-side prototype pollution among websites. However, there are still challenges to be addressed due to the unique features of mini-apps. Recently, Chao et al. proposed TaintMini [57] and built a universal data flow graph to detect the flow of sensitive data in mini-apps. Li et al. proposed MiniTracker [58] to detect privacy leakage automatically in mini-apps. KeyMagnet extend the capabilities to analyze mini-apps in different super-apps.

VII. Conclusion

In this work, we make the first step to systematically understand the credential system and its security in the app-in-app paradigm and point out the root cause of credential leakage. We propose a novel semantic-oriented security-verification approach, called KeyMagnet, that can detect the vulnerability based on the similarity of credential semantics between the client- and server-side. By applying KeyMagnet on 413,775 real-world mini-apps, 84,491 credential leaks are detected and 54,728 mini-apps are subject to credential leakage attacks. Our experiment results show that KeyMagnet is scalable, effective, and accurate. In addition, the results remind us that we need to pay attention to the security issues caused by third-party development platforms. Our research can provide a successful experience for super-apps to take actions to ensure the security of the app-in-app ecosystem.

Acknowledgement

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Key Research and Development Program (2021YFB3101200), National Natural Science Foundation of China (62172104, 62172105, 62472096, 62102093, 62102091, 62302101, 62402114, 62402116, 62202106). Zhemin Yang was supported in part by the Funding of Ministry of Industry and Information Technology of the People's Republic of China under Grant TC220H079. Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

References

[1] QPSoftware. (Accessed September 11, 2024) The development trends of wechat mini-program. [Online]. Available: https://qpsoftware.net/blog/development-trends-wechat-mini-program

[2] Statista. (Accessed September 11, 2024) Google Play: number of available apps 2017-2024. [Online]. Available: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[3] QPSoftware. (Accessed September 11, 2024) Wechat mini program - all you need to know. [Online]. Available: https://qpsoftware.net/blog/wechat-mini-program-all-you-need-know

[4] W. Yang, Y. Zhang, J. Li, H. Liu, Q. Wang, Y. Zhang, and D. Gu, "Show me the money! finding flawed implementations of third-party in-app payment in android apps." in *Network and Distributed System Security Symposium (NDSS)*, 2017.

[5] M. Meli, M. R. McNiece, and B. Reaves, "How bad can it git? characterizing secret leakage in public github repositories." in *Network and Distributed System Security Symposium (NDSS)*, 2019.

[6] C. Zuo, Z. Lin, and Y. Zhang, "Why does your data leak? uncovering the data leakage in cloud from mobile apps," in *2019 IEEE Symposium on Security and Privacy (S&P)*, 2019.

[7] A. Saha, T. Denning, V. Srikumar, and S. K. Kasera, "Secrets in source code: Reducing false positives using machine learning," in *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*, 2020.

[8] C. Farinella, A. Ahmed, and C. Watterson, "Git leaks: Boosting detection effectiveness through endpoint visibility," in *20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2021.

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

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

[11] H. Hosseini, J. Rengstorf, and T. Hupperich, "Automated search for leaked private keys on the internet: Has your private key been pwned?" in *Proceedings of the 17th International Conference on Software Technologies (ICSOFT)*, 2022.

[12] V. S. Sinha, D. Saha, P. Dhoolia, R. Padhye, and S. Mani, "Detecting and mitigating secret-key leaks in source code repositories," in *12th IEEE/ACM Working Conference on Mining Software Repositories (MSR)*, 2015.

[13] Y. Zhou, L. Wu, Z. Wang, and X. Jiang, "Harvesting developer credentials in android apps," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*, 2015.

[14] N. Samarasinghe, A. Adhikari, M. Mannan, and A. Youssef, "Et tu, brute? privacy analysis of government websites and mobile apps," in *Proceedings of the ACM Web Conference (WWW)*, 2022.

[15] WeChat. (Accessed September 11, 2024) WeChat API list. [Online]. Available: https://developers.weixin.qq.com/miniprogram/en/dev/api/

[16] S. Khodayari and G. Pellegrino, "Jaw: Studying client-side csrf with hybrid property graphs and declarative traversals," in *30th USENIX Security Symposium (Usenix Security)*, 2021.

[17] Android Developers community. (Accessed September 11, 2024) Write automated tests with UI Automator. [Online]. Available: https://developer.android.com/training/testing/other-components/ui-automator

[18] Y. He, L. Zhang, Z. Yang, Y. Cao, K. Lian, S. Li, W. Yang, Z. Zhang, M. Yang, Y. Zhang, and H. Duan, "Textexerciser: Feedback-driven text input exercising for android applications," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[19] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2020.

[20] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie, "An empirical study of android test generation tools in industrial cases," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[21] K. Beijering, C. Gooskens, and W. Heeringa, "Predicting intelligibility and perceived linguistic distance by means of the levenshtein algorithm," *Linguistics in the Netherlands*, vol. 25, no. 1, pp. 13–24, 2008.

[22] Y. Lin, R. Liu, D. M. Divakaran, J. Y. Ng, Q. Z. Chan, Y. Lu, Y. Si, F. Zhang, and J. S. Dong, "Phishpedia: A hybrid deep learning based approach to visually identify phishing webpages," in *30th USENIX Security Symposium (USENIX Security)*, 2021.

[23] W. Kang, B. Son, and K. Heo, "Tracer: Signature-based static analysis for detecting recurring vulnerabilities," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[24] 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 (POMACS)*, vol. 5, no. 2, 2021.

[25] WeChat. (Accessed September 11, 2024) auth.getAccessToken. [Online]. Available: https://developers.weixin.qq.com/miniprogram/en/dev/api-backend/open-api/access-token/auth.getAccessToken.html

[26] JD. (Accessed September 11, 2024) Taro. [Online]. Available: https://docs.taro.zone/en/docs

[27] DCloud. (Accessed September 11, 2024) Uni-app. [Online]. Available: https://en.uniapp.dcloud.io/

[28] Antmove. (Accessed September 11, 2024) Antmove. [Online]. Available: https://github.com/ant-move/Antmove

[29] Npm. (Accessed September 11, 2024) Wx2my. [Online]. Available: https://www.npmjs.com/package/wx2my

[30] Youzan. (Accessed September 11, 2024) Youzan. [Online]. Available: https://www.youzan.com/

[31] Weimob. (Accessed September 11, 2024) Weimob. [Online]. Available: https://weimob.com/

[32] Tencent. (Accessed September 11, 2024) CloudBase. [Online]. Available: https://www.cloudbase.net/

[33] WeChat. (Accessed September 11, 2024) Realtime Log. [Online]. Available: https://developers.weixin.qq.com/miniprogram/en/dev/framework/realtimelog/

[34] ——. (Accessed September 11, 2024) Live Streaming Service. [Online]. Available: https://developers.weixin.qq.com/miniprogram/dev/OpenApiDoc/livebroadcast/studio-management/createRoom.html

[35] (Accessed September 11, 2024) Vulnerability disclosure cheat sheet. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Vulnerability_Disclosure_Cheat_Sheet.html

[36] Kobi Gluck. (Accessed September 11, 2024) New policy update to boost trust and transparency on google play. [Online]. Available: https://android-developers.googleblog.com/2023/07/boosting-trust-and-transparency-in-google-play.html

[37] S. Lounici, M. Rosa, C. M. Negri, S. Trabelsi, and M. Önen, "Optimizing leak detection in open-source platforms with machine learning techniques." in *2021 International Conference on Information Systems Security and Privacy (ICISSP)*, 2021.

[38] R. Feng, Z. Yan, S. Peng, and Y. Zhang, "Automated detection of password leakage from public github repositories," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, 2022.

[39] S. Shi, X. Wang, K. Zeng, R. Yang, and W. C. Lau, "An empirical study on mobile payment credential leaks and their exploits," in *Security and Privacy in Communication Networks (SecureComm)*, 2021.

[40] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of google play," in *The 2014 ACM international conference on Measurement and modeling of computer systems (SIGMETRICS)*, 2014.

[41] H. Wen, J. Li, Y. Zhang, and D. Gu, "An empirical study of sdk credential misuse in ios apps," in *25th Asia-Pacific Software Engineering Conference (APSEC)*, 2018.

[42] X. Wang, Y. Sun, S. Nanda, and X. Wang, "Credit karma: Understanding security implications of exposed cloud services through automated capability inference," in *32nd USENIX Security Symposium (USENIX Security)*, 2023.

[43] 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 (CCS)*, 2020.

[44] 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.

[45] 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)*, 2022.

[46] 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 (CCS)*, 2022.

[47] 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)*, 2023.

[48] ——, "Rootfree attacks: Exploiting mobile platform's super apps from desktop," 2024.

[49] ——, "Uncovering and exploiting hidden apis in mobile super apps," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023.

[50] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for javascript," in *Proceedings of the 16th International Static Analysis Symposium (SAS)*, 2009.

[51] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu, "Safe: Formal specification and implementation of a scalable analysis framework for ecmascript," in *19th International Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2012.

[52] IBM T.J. Watson Research Center. (Accessed September 11, 2024) WALA, The T. J. Watson Libraries for Analysis. [Online]. Available: https://github.com/wala/WALA

[53] S. Li, M. Kang, J. Hou, and Y. Cao, "Mining node. js vulnerabilities via object dependence graph and query," in *31st USENIX Security Symposium (USENIX Security)*, 2022.

[54] A. Fass, D. F. Somé, M. Backes, and B. Stock, "Doublex: Statically detecting vulnerable data flows in browser extensions at scale," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[55] M. Kang, Y. Xu, S. Li, R. Gjomemo, J. Hou, V. Venkatakrishnan, and Y. Cao, "Scaling javascript abstract interpretation to detect and exploit node. js taint-style vulnerability," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[56] Z. Kang, S. Li, and Y. Cao, "Probe the proto: Measuring client-side prototype pollution vulnerabilities of one million real-world websites," in *Network and Distributed System Security Symposium (NDSS)*, 2022.

[57] 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 (ICSE)*, 2023.

[58] 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 (TDSC)*, pp. 1–17, 2023.