Better Safe than Sorry: Uncovering the Insecure Resource Management in App-in-App Cloud Services

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

Zhemin Yang
Fudan University
yangzhemin@fudan.edu.cn

Dingyi Liu
Fudan University
23210240084@m.fudan.edu.cn

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

Jiarun Dai
Fudan University
jrdai14@fudan.edu.cn

Min Yang
Fudan University
m_yang@fudan.edu.cn

***Abstract*—In the app-in-app ecosystem, super-apps provide mini-app developers access to various sensitive cloud services, such as cloud database and cloud storage. These services enable mini-app developers to efficiently s tore a nd m anage mini-app data in the super-app server. To protect these sensitive resources, super-apps implement an identity management mechanism, allowing mini-app developers to verify user identity and ensure that only authorized and trusted users can access specific resources. However, flaws exist in the implementation of resource management by mini-app developers, which can expose sensitive resources to attackers.**

In this paper, we conduct the first s ystematic s tudy o f t he insecure cloud resource management in the app-in-app ecosystem. We design and implement a tool, ICREMINER, that combines static analysis and dynamic probing to assess the security implications on 22,695 real-world mini-apps that access app-in-app cloud services in four super-app platforms. The results of our study reveal that 2,815 mini-apps (12.40%) are affected by the insecure resource management, involving 8,062 insecure cloud operations. We have identified that some mini-apps of prominent corporations are also vulnerable to these risks. Additionally, we conduct an in-depth analysis of the significant s ecurity hazards that can be caused by the vulnerability, such as allowing attackers to steal sensitive user information and pay for free. In response, we have engaged in responsible vulnerability disclosure to the super-app platforms and corresponding mini-app developers. We also provide several mitigation strategies to help them resolve the vulnerabilities.

I. INTRODUCTION

Nowadays, the OS-like app-in-app paradigm, which involves super-apps and mini-apps, has been becoming increasingly popular. Super-apps now host an extensive collection of over 7 million mini-apps [1]. The large number of mini-apps has led to the demand for cost-effective, scalable backend services, such as user authentication and data storage. To meet this demand, super-apps provide mini-app developers with cloud spaces, eliminating the need for mini-app developers to manage their own servers. Mini-app developers can conveniently store and manage sensitive data, such as user purchase history and confidential files, in the cloud spaces. As a result, the super-app becomes a vast information aggregator, housing sensitive data from massive mini-apps.

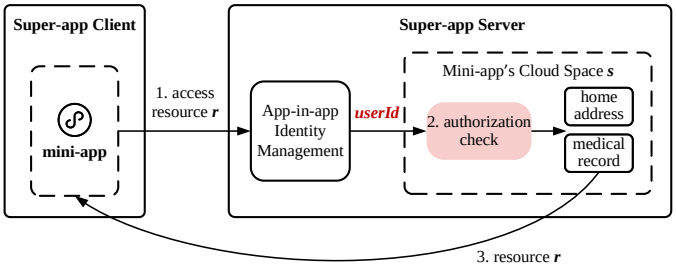

Fig. 1. Identity management mechanism in the app-in-app ecosystem.

To safeguard the sensitive resources stored in cloud spaces, access to specific resources, such as a mini-app user's medical record, is restricted to authorized mini-app users only. To secure and facilitate resource management for different mini-app users, super-apps provide a centralized user identity management mechanism. Specifically, super-apps assign each mini-app user a unique identifier, which enables access control to sensitive resources. Unlike traditional methods, where each mini-app manages its users' identities individually, mini-apps can directly retrieve user identities from super-apps.

As depicted in Figure 1, a mini-app user requests access to the sensitive resource r stored in the cloud space s. In step 1, this request is forwarded to the super-app server side. Before accessing the sensitive resources, the super-app server transfers the user's identity (i.e., $userId$) to the mini-app's cloud space s. Then the mini-app performs an authorization check to determine whether the user has access to r in step

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.230194
www.ndss-symposium.org

2, based on the user's identity. Upon successful verification, the sensitive resource r is returned to the mini-app client in step 3. In this mechanism, mini-app developers can focus on the authorization process to avoid unauthorized access, as user identity is protected by super-apps.

In contrast, to safeguard the sensitive resources, modern cloud infrastructures, such as AWS [2], Microsoft Azure [3], and Google Cloud Platform [4], primarily issue cloud credentials (such as AWS keys) to developers for accessing the cloud resources in the client side. Specifically, developers can configure permissions for cloud credentials, enabling mobile applications to access cloud resources with credentials. Previous studies have identified the misuse and leakage of high-privileged cloud credentials, which can be obtained and exploited by attackers, leading to significant privacy risks [5], [6]. Unlike traditional methods, centralized user identity management eliminates the need for mini-app developers to use cloud credentials in the client side, which can mitigate the risks of credential leakage.

Understanding the Security Risks. In this work, we conduct the first systematic study of the security of the identity management mechanism in the app-in-app ecosystem. To understand the security risks, we first perform an in-depth analysis of the mechanism based on the developer documentation to understand how it secures cloud resource access. Our analysis reveals that while super-apps provide mini-app developers with user identities, the primary responsibility for implementing user identity check and managing sensitive resources falls on mini-app developers in the cloud space. Specifically, when a mini-app user tries to access sensitive resources, the mini-app should retrieve and verify the user's identity in the cloud space to ensure the user has necessary access permissions (step 2). Unfortunately, mini-app developers may misinterpret the mechanism and fail to ensure a secure implementation of the authorization check, such as moving the check to the mini-app client side. In this paper, we identify four types of insecure practices, which undermine the security of sensitive resources stored in cloud spaces and can lead to severe hazards.

Security Hazards Assessment. After thoroughly studying and understanding the security of app-in-app cloud services, we aim to assess the security risks introduced by the **i**nsecure **c**loud **re**source **m**anagement (ICREM) of mini-app developers. Given the large number of available mini-apps, an automated tool is required. In light of this, we are motivated to develop a tool, called ICREMINER, to detect the security risks. This task presents several challenges. One major challenge is that the cloud-side code and resources are out of reach. As a result, it is difficult to gain a comprehensive understanding of sensitive resources managed in the cloud space, based solely on analysis of the mini-app client. Traditional methods primarily focus on analyzing the resources accessible through leaked credentials in the client side, which cannot be applied to our task. Furthermore, it is challenging to ensure that our assessment does not lead to the leakage of sensitive resources or pose risks to cloud services.

For the first challenge, since super-apps offer specific APIs for mini-apps to access cloud resources, ICREMINER conducts a fine-grained data flow analysis of cloud APIs to model and understand cloud resource access from the mini-app client side. To gain deeper insights into cloud resource management within cloud side, we propose a semantic-driven approach based on code semantics to infer cloud resources that are not directly accessed in the mini-app client side. This approach leverages large language models for inference, which have demonstrated strong capabilities in understanding code semantics [7], [8], [9]. For the second challenge, inspired by [5], [6], ICREMINER performs zero-leakage probing without risking access or modification of the cloud data. Additionally, we propose a series of assessment measures tailored to different types of app-in-app cloud services (Section IV-D).

Our Findings. We implement the prototype of ICREMINER and apply it to 1,248,815 real-world mini-apps. Our analysis reveals that 22,695 mini-apps manage sensitive resources in the app-in-app cloud spaces. Among them, 2,815 mini-apps (12.40%) are affected by the insecure resource management, including 8,062 insecure cloud operations. Notably, our research points out that the vulnerability affects mini-apps of prominent corporations (e.g., Tencent) as well as those in sensitive categories (e.g., Education and Government), which cause severe security hazards. For example, attackers can steal sensitive data belonging to mini-app users, such as personal resumes, medical records, ID card photos, and purchase histories. In addition, attackers can manipulate cloud resources, such as altering account balances without any payment. We have reported the vulnerabilities to the super-app platforms and corresponding mini-app developers. We have received 35 responses and 893 mini-app developers have fixed the issues or taken down their mini-apps.

In short, we make the following contributions.

- To the best of our knowledge, we conduct the first systematic study on security risks of insecure cloud resource management in the app-in-app ecosystem.
- We propose a novel approach, called ICREMINER[1], that combines static analysis and dynamic probing to automatically analyze the ICREM risks.
- We conduct a large-scale, empirical study on real-world mini-apps and have identified 2,815 mini-apps that are affected by this vulnerability. We also made responsible vulnerability disclosure and propose corresponding mitigation strategies.

II. BACKGROUND

A. App-in-App Cloud Services

To facilitate the development of mini-apps and ensure a secure environment to protect mini-app's data, super-apps provide mini-app developers with cloud services. These services enable mini-app developers to manage sensitive data through well-defined cloud APIs. Based on their functionalities, the

[1]The source code is avaliable at https://doi.org/10.5281/zenodo.16946146.

cloud services can be categorized into three types, i.e., cloud database, cloud storage, and cloud routine (or cloud function).

Cloud Database Service. The cloud database provides database services to store mini-app data, eliminating the need for mini-app developers to configure and maintain their own database servers. The cloud data is organized into collections, i.e., data tables, which are used to store different types of data, such as user profiles and purchase histories. Mini-apps can access these collections with collection names. Besides, when a mini-app user writes data into a collection from mini-app client, the collection creates a new row to store the data and attaches the user's identity to the data record.

Cloud Storage Service. The cloud storage offers a space for mini-apps to upload and download files, which supports various forms of unstructured data, such as videos and images. The file operations can be easily performed with cloud APIs. For example, mini-apps can invoke `wx.cloud.uploadFile` [10] to store files in the cloud storage. Additionally, each file is assigned a unique file ID for further access.

Cloud Routine Service. The cloud routine executes server-side code in the cloud space. When the cloud routine is called in the mini-app client, the defined code logic is executed in the cloud space. Mini-app developers can customize cloud routines for different functional requirements. Furthermore, unlike traditional authentication methods that rely on tokens or cookies which should be managed by mini-apps in the server side, mini-app developers can obtain user identities from super-apps in the cloud side.

B. Protection Measures in Cloud Services

Given the presence of sensitive data in cloud side, including user privacy and confidential files, robust protection measures are essential. Traditional cloud providers (e.g., AWS [2], Microsoft Azure [3], and Google Cloud Platform [4]), primarily issue cloud credentials to developers for accessing the cloud resources in the client side. Specifically, developers can configure permissions for cloud credentials, enabling mobile applications to access cloud resources with credentials. However, previous work has identified pervasive issues of cloud credentials [5], [6], such as the leakage of high-privileged credentials. In contrast, super-apps implement customized cloud architectures and manage the cloud resources based on user identities. Specifically, super-apps adopt several measures to safeguard the cloud resources.

Encrypted Channel. Traditional access to cloud services is typically achieved through direct network requests, where parameters are susceptible to attacker manipulation using mature tools, such as Burp Suite [11] and Charles [12]. For example, attackers can easily issue network requests to access cloud services using leaked cloud credentials. In contrast, super-apps employ proprietary protocols to secure the communications between the mini-app client side and the super-app cloud side, such as MMTLS in WeChat [13]. These proprietary protocols are customized, which increases the difficulty of reverse engineering and traffic inspection.

Identity Management. Super-apps implement an identity management mechanism to provide unique identities to mini-app users. Mini-app developers can determine whether a mini-app user has access to specific resources based on the user identity, thereby preventing unauthorized access. Specifically, in cloud spaces, mini-app developers can leverage proprietary APIs provided by super-apps to retrieve user identities for seamless authentication, such as `cloud.getWXContext` in WeChat [14]. This strategy eases the burden of mini-app developers and mitigates the issues of cloud credential leakage.

Typically, the protection of sensitive resources is controlled at the read/write level. Taking cloud database read permissions as an example, three primary strategies are employed based on the data's security level in different scenarios: (1) publicly readable to all users (for non-sensitive public information such as user comments), (2) readable only to the data owner (mini-app user) or mini-app developer (for private user information such as order details), and (3) readable only to mini-app developer (for system-level resources such as mini-app logs). Additionally, mini-app developers can adopt customized strategies to protect cloud resources based on user identity.

```
1  // mini-app appId: wxd3****c27
2  var t = wx.cloud.database({env: "chen***692"});
3  t.collection("course_users").where({
4    phone: e.phone
5  }).get().then(function(o) {
6      var i = o.data[0];
7      t.setData({name: i.realname, address: i.realaddr});
8      ...
9    }
10 });
```

Fig. 2. Code snippet demonstrating access to the cloud database 'course_users'.

III. Problem Statement

A. A Motivating Example

In this section, we present a motivating example to illustrate the attack vector targeting the insecure practices of mini-app developers in app-in-app cloud services. As illustrated in Figure 2, a mini-app (`wxd3***c27`) accesses the cloud database 'course_users' to retrieve user's detailed information, such as the real name and home address. The mini-app first gets a reference to the cloud database of environment `chen***692` in line 2. Then it targets the database 'course_users' and transfers the parameter 'phone' to fetch corresponding user data in line 3 and line 4. Finally, the data is returned to the mini-app client and the mini-app accesses the data in line 6. To safeguard mini-apps' cloud resources, super-apps implement various protection measures. For example, mini-apps other than `wxd3***c27` cannot access resources within the environment `chen***692`. Moreover, mini-app developers can configure access permissions based on user identities to ensure that users can only access their own resources.

Unfortunately, we find that mini-app developers often adopt flawed practices when implementing user identity checks. For

example, a secure approach for identity check is to obtain current user's identity, i.e., `userId`, in the cloud side and use it to determine the resources that the mini-app user is authorized to access. In this process, the user identity is securely managed by the super-app server, making it hard to tamper with. However, in current implementation, the mini-app transmits the customized parameter 'phone' from the mini-app client to the cloud in order to retrieve corresponding user data in line 4. This practice undermines the security mechanism, as a malicious user can manipulate the parameter 'phone' to impersonate the user identity and gain unauthorized access to other users' cloud resources.

B. Typical Attack Process

To execute such an attack, the attacker should be able to invoke cloud services with customized parameters, such as 'phone' in the motivating example, to access cloud resources of the target mini-apps. In traditional cloud infrastructures, this could be easily achieved by modifying network traffic. However, as discussed in Section II-B, super-apps have introduced proprietary security protocols, which hinder attackers from monitoring or tampering with traffic data.

However, we found that attackers can leverage the code caching mechanism of super-apps to achieve this on their own devices without the need to modify network traffic, as illustrated in Figure 3. Specifically, to optimize mini-app's performance, super-apps cache the mini-app code locally, enabling faster loading when the mini-app is launched again. The attacker can extract the cached code and inject a malicious payload (step 1 and 2). To ensure the integrity of the cached mini-app code, super-apps implement verification measures before loading the code into the mini-app runtime. However, these measures rely on local checks, such as verifying MD5 hashes or file signatures. Therefore, attackers can easily bypass the integrity check on their devices using techniques like hooking the verification functions in super-apps (step 4). Through reverse engineering of super-apps, we successfully identified and bypassed all file integrity checks implemented by super-apps, injecting manually crafted code into our test mini-app. The injected code was then loaded into memory and executed (step 5), enabling access to sensitive data. Beyond modifying cached code packages, attackers can also hook execution APIs to replace original code with manually crafted code before it executes. As the effects of different attack methods are similar, this paper focuses on describing attacks in the context of the caching mechanism.

C. Threat Model

Here, we discuss the threat model in our work, including the assumptions and the attacker's capabilities.

Assumptions. As illustrated in Figure 1, there are multiple parties involved in the cloud resource access and protection within the app-in-app ecosystem. We assume that the super-app server and the cloud space are trusted, as the server side is not visible to attackers. However, messages sent from the mini-app client are considered untrusted.

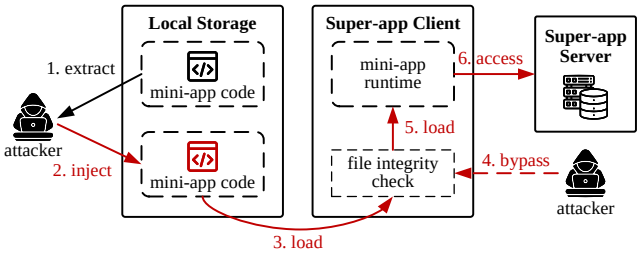

Fig. 3. The process by which the attacker injects malicious payload into the target mini-app.

Attacker's Capabilities. The attacker is a malicious mini-app user, whose objective is to access sensitive resources stored in the cloud side. The attacker has the capability to inject a crafted payload into the cached mini-app code. To inject the crafted payload, the attacker only needs a rooted mobile device and the ability to hook the integrity check APIs. Technically, since the local storage of an attacker's rooted device is under their control, any local checks can be bypassed. Besides, it is challenging to distinguish unexpected behaviors caused by the injected payload because the mini-app code should have the capability to invoke cloud APIs. Moreover, attackers do not need to develop a malicious mini-app and publish it in the mini-app market. They can execute the attack directly from their own devices. Therefore, we believe this attack scenario is practical and relatively easy to achieve.

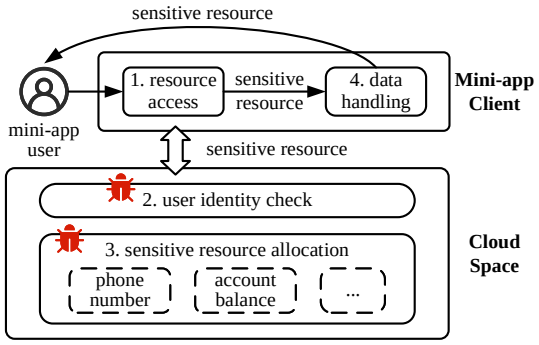

Fig. 4. Insecure practices in cloud resources management within the app-in-app ecosystem.

D. Insecure Practices in App-in-App Cloud Services

To systematically analyze insecure practices, we begin by analyzing the cloud resource management mechanisms of different super-apps. We analyze the developer documentation provided by super-app platforms to model their resource management workflows. As illustrated in Figure 4, the identity management mechanism comprises two key stages: verifying user identities before granting user resource access (*User Identity Check*) and determining the sensitive resources that can be allocated to mini-app users (*Sensitive Resource Allocation*). Next, we assess the security requirements that should be enforced at each stage. Specifically, we examine the security-related descriptions of each cloud service in the

documentation. The core principle guiding our analysis is the principle of least privilege, i.e., ensuring that mini-app users are granted only the necessary permissions. As a result, we identify four types of insecure practices:

- **User Identity Check.** Many mini-app developers incorrectly implement the user identity check in step 2. First, some mini-app developers move the identity check to the mini-app client side **(UIC-1)**, returning all resources to the mini-app client and filtering them based on client-side logic in step 4. An attacker can easily obtain all the exposed cloud resources. Second, some mini-app developers perform identity check based on the parameters transferred from the mini-app client **(UIC-2)**, such as user's phone number in step 3, rather than the user identity retrieved from super-apps. Previous work has found that customized user identifiers are often susceptible to privilege escalation vulnerabilities [15], [16], [17], making it easy for an attacker to forge their identities to retrieve other users' information.
- **Sensitive Resource Allocation.** Upon in-depth analysis of sensitive resources, we find that specific types of resources should only be accessed or manipulated by mini-app developers. First, privileged resources, such as the mini-app developer's GPT keys, should not be accessible to mini-app users. However, some mini-app developers improperly share privileged resources with mini-app users in step 3, thereby enabling unrestricted access for all mini-app users (**SRA-1**). Second, not all user information should be entirely under the user's control. For example, certain user information, such as account balance, should only be readable by mini-app users. Nevertheless, some mini-app developers group all user information under the same permission settings, granting users the ability to write to these resources (**SRA-2**). As a result, users can modify their account balances arbitrarily.

E. Generalizing Attack Vectors

Below we will discuss the attack vectors and the resulting security hazards of the four types of vulnerable practices from the perspective of different cloud services.

Cloud Database Service. The cloud database service serves as the foundational service in mini-apps, with nearly all business functions (e.g., login, registration, and payment) closely tied to it. Cloud databases are organized into collections, with each collection designed to store various types of resources, which can be accessed with the collection names.

A common issue is that mini-app developers expose cloud collections containing privileged resources (**SRA-1**). For example, some mini-apps share API keys with the mini-app client to conveniently access sensitive services, such as GPT service. Additionally, many mini-app developers write sensitive data directly to the cloud collections from the mini-app client (**SRA-2**). For example, after a successful payment, some mini-apps access the cloud collections and modify the user's account balance or shopping records in the mini-app client side. This practice implies that mini-app users have permission to write to corresponding collections, which should only be readable by them. Moreover, the insecure practices for **UIC-1** and **UIC-2** are also prevalent in cloud databases, where mini-app developers move the identity check to the mini-app client side or expose cloud resources of other users.

Cloud Storage Service. Cloud storage is used to store and retrieve various types of files. When uploading a file, mini-app developers must specify the file path for storage in the cloud space. Upon a successful upload, a unique file identifier, known as the file ID, is generated. The file ID consists of two components, i.e, a specific container ID and the cloud file path, with the container ID being constant. Subsequent operations, such as downloading the file, depend on this file ID.

Since cloud storage primarily consists of user files and are accessed using a specific file ID, the security issue mainly falls in type of **UIC-2**. Typically, mini-app developers do not retrieve `userId` from the super-apps to verify whether a mini-app user can access the files. Instead, they directly use the file ID to retrieve the corresponding files, assuming that only the current mini-app user possesses the file ID for their owned files. In this context, the file ID is considered as the mini-app user's identity. However, this assumption does not always hold, as many mini-app developers use predictable file paths, allowing attackers to guess the file ID and gain unauthorized access to the corresponding files.

Cloud Routine Service. Similar to RESTful APIs, cloud routines offer customized services that allow mini-apps to access code logic running in the cloud side. Unlike cloud services mentioned above, cloud routines are designed to be accessible to all mini-app users. Therefore, the cloud routines must not expose privileged resources to mini-app users. However, many insecure practices exist.

First, some mini-app developers move the user identity check to the mini-app client side (**UIC-1**). An attacker can easily access the exposed sensitive resources in the cloud routines. Furthermore, instead of using the `userId` retrieved from super-apps, many mini-app developers directly include a self-defined user ID in the parameters of cloud routines, such as phone number, and use it as the user identity for further authorization check (**UIC-2**). This practice undermines the security model of super-apps and introduces security risks of unauthorized access, as attackers can manipulate the user identities transmitted to the cloud side and access other users' sensitive information. Additionally, some cloud routines directly request privileged resources from the cloud side (**SRA-1**). An attacker can easily obtain leaked resources through the cloud routines. Besides, super-apps provide mini-app servers with various sensitive services, such as user notification and AI services, to simplify development and management of mini-apps. However, some mini-app developers expose these services directly to the mini-app client through cloud routines, allowing attackers to abuse these services. For example, we find a mini-app exposes the notification service to the mini-app client, which could be exploited by attackers to send malicious

content to mini-app users. Similar to cloud database service, many cloud routines expose write permissions for sensitive user data, such as modifying a user's balance, which should only be accessed in the server side after payment (**SRA-2**). Otherwise, an attacker could exploit the cloud routine to increase the balance without making a payment.

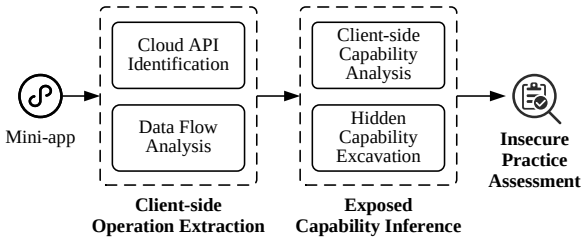

Fig. 5. The workflow to assess the insecure practices in app-in-app cloud services.

IV. Methodology

A. Design Overview

In this section, we aim to assess the impacts of mini-app developers' insecure practices in real-world mini-apps that expose sensitive cloud resources stored in the super-app server side. However, **it is difficult to understand the practices employed by mini-app developers and to determine whether these practices are vulnerable, as the cloud side is out of reach.** There are several studies that focus on vulnerabilities in cloud infrastructures [5], [6], [18], [19], [20], especially the permission misconfiguration of cloud credentials, which differs from the threat model in our work. Furthermore, previous work mainly focus on examining the accessibility of cloud services or cloud containers [6], such as "bucket" in AWS S3 [21]. However, in mini-apps, cloud containers are often accessible to all mini-app users because permission management is typically applied at the level of the stored elements rather than the container itself. In addition, super-apps provide mini-apps with various types of cloud services, such as cloud routines, which have not been studied. Therefore, traditional methods cannot be used to assess the new threats in the app-in-app ecosystem.

To address these challenges, ICREMINER conducts an in-depth analysis with three phases as illustrated in Figure 5. The main insight is that operations performed in the mini-app client side to access cloud services can reflect the potential practices conducted by mini-app developers. Therefore, ICREMINER first extracts all cloud operations performed in the mini-app client side. Specifically, ICREMINER locates cloud APIs used in mini-apps and performs data flow analysis to extract the context information, such as the cloud-related parameters and return values of cloud APIs. Second, ICREMINER combines static analysis and dynamic probing to infer the exposed capabilities, i.e., the cloud resources that can be accessed by the mini-apps. Specifically, ICREMINER conducts capability analysis from two perspectives. On one hand, some cloud resources are directly accessed in the mini-app client and ICREMINER analyzes corresponding cloud operations based on the context information. On the other hand, some cloud resources, while not directly accessed in the mini-app client side through cloud APIs, can still be reached via injected crafted payloads (referred to as *hidden capabilities* in this paper). To enhance our analysis, we propose several inference methods to uncover hidden capabilities based on code semantics. Finally, to assess whether the cloud practices are vulnerable and expose excessive capabilities in the mini-app client, ICREMINER applies security rules tailored to different types of cloud services, including cloud database, cloud storage, and cloud routine. Additionally, ICREMINER analyzes the types of insecure practices. More technical details are presented in the remaining subsections.

B. Client-side Operation Extraction

In this section, ICREMINER aims to extract all the cloud operations performed in the mini-app client side. Given that super-apps provide a variety of cloud APIs for mini-apps to access cloud services, we perform a documentation analysis to extract all related APIs. Then ICREMINER automatically locates these APIs to identify cloud operations in the mini-app client. To extract the context information of cloud operations, such as parameters, ICREMINER first models specific APIs provided by super-apps and constructs the inter-procedural control flow graph of the mini-apps. Then, ICREMINER conducts a fine-grained data flow analysis on the mini-app client code to extract data dependencies of cloud parameters. Moreover, ICREMINER further analyzes the values of the cloud parameters to recover their semantics. Specifically, some parameters are constant, such as the container name, ICREMINER extracts the instructions associated with them and retrieves the parameter values. Some cloud parameters are dynamically generated, such as cloud file names. ICREMINER collects the generation sequences of these parameters for further analysis.

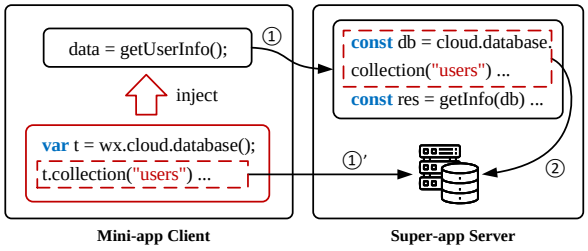

Fig. 6. An example of hidden capability.

C. Exposed Capability Inference

Before analyzing the insecure practices of mini-app developers, ICREMINER infers the exposed capabilities in cloud operations (i.e., cloud resources exposed to mini-app users), which are closely tied to the cloud-side practices. However, cloud operations in the mini-app client reveal only part of the exposed capabilities. For example, some cloud resources are not directly accessed in the mini-app client and thus remain

'hidden'. As illustrated in Figure 6, the mini-app invokes `getUserInfo` (step ①) and accesses the cloud database 'users' in the cloud side (step ②). Although the mini-app client does not directly interact with the cloud database, if the database name is exposed to attackers, an attacker could inject a malicious payload to access the cloud database (step ①'), risking the exposure of sensitive data. **The main challenge lies in that it is hard to identify hidden capabilities.**

Therefore, the capability analysis consists of two steps. First, ICREMINER infers the exposed cloud resources associated with cloud operations performed in the mini-app client side. For cloud database and cloud storage services, there are generally two types of access control policies regarding the readability of cloud elements: resources can either be readable by all users or only by the data owner (mini-app user). To assess whether an attacker can read specific cloud resources, we propose a dynamic probing method that infers exposed cloud resources without compromising user privacy. Due to ethical considerations, we adopt a zero-leakage probing to assess the security risks in section IV-D without risking access or modification of the cloud data. Moreover, due to the potential impacts of write operations on cloud resources, we do not directly verify whether mini-app developers have configured write permissions correctly. Instead, we assume that if a mini-app performs a write operation of specific cloud resources in the mini-app client side, it implies that the mini-app has granted write access to the corresponding resources for its users. Through static analysis, we can identify which cloud resources are writable by users based on the specific cloud APIs. For cloud routine services, access control policies are generally simple, and they are often designed to be accessible to all mini-app users. Therefore, we assume that attackers can access these cloud routines invoked in the mini-app client side. Besides, ICREMINER uses the context information of cloud routines to represent their semantics, including routine names, parameters and return values.

Second, ICREMINER infers the exposed cloud resources associated with hidden capabilities. Specifically, our goal is to identify the hidden cloud databases accessible from the mini-app client side, using valid database names. While other types of cloud services, such as hidden cloud storage and cloud routines may also exist, inferring the complex parameters of these services is highly impractical due to insufficient information available from the mini-app client side. Through an in-depth analysis of cloud database services accessed in mini-apps, we gain insights into the detection of hidden cloud databases and propose the following methods to infer hidden database names.

Common Name Augmentation. Unlike cloud routines and cloud storage, which are often tailored to specific mini-apps and have complex structures, the names of cloud databases tend to be more generic, commonly using terms like "users", "profiles", and "logs". Therefore, we conduct an analysis on our dataset to collect the top 10 most frequently used database names and their variations, such as changes in capitalization and singular/plural forms. We consider these database names are potentially used in other mini-apps.

LLM-based Inference. Database names are often closely related to data operations in mini-app client side. For example, the data operation `getUserInfo` retrieves user information from a cloud database, which is often named "users" or other similar formats. Based on this insight, ICREMINER conducts a static analysis to extract methods from the mini-app client that contains a specific token representing data operations, e.g., get, set, and add. However, it's hard to understand the semantics of these data operations. Recently, Large Language Models, such as GPT and Gemini, have demonstrated strong capabilities in processing text information and understand the semantics of code [7], [8], [9]. Unlike traditional semantic search methods based on NLP techniques [22], which rely on predefined name lists, LLMs exhibit stronger generalization capabilities to infer database names. Therefore, we design a name generation prompt that incorporates the contexts of data operation-related methods to generate potential database names based on Gemini. Furthermore, we evaluate the performance across different LLM models. Due to page limit, the results and the full prompt are presented in Appendix A.

Mini-app Correlation. Different mini-apps may share the same cloud databases, particularly those developed by the same mini-app developers. Therefore, we cluster mini-apps developed by the same developers, as they are likely to utilize similar database names for storing cloud data. Subsequently, we expand the database names of each mini-app by including names extracted from mini-apps within the same clusters. Besides, we also correlate mini-apps with the same cloud database names inspired by [6].

During the inference process, we ensure that our approach does not compromise the privacy of any mini-app user or disrupt the functionality of mini-apps. To prevent potential risks associated with write operations to cloud services, we avoid injecting any write operations into mini-apps. Additionally, we do not store any cloud data due to ethical considerations.

D. Insecure Practice Assessment

After identifying the exposed cloud resources in the mini-app client side, our objective is to assess whether the cloud-side practices of mini-app developers are insecure. Specifically, we aim to understand whether these capabilities are *more-than-expected*. **The fundamental principle is that mini-apps should not perform over-privileged operations.** For example, a mini-app user should not be able to perform read operations to access other users' sensitive information stored in the cloud space. To formalize this procedure, we propose several security rules for different cloud services to verify whether *more-than-expected* behaviors exist.

Assessment of Cloud Database Service. As discussed in Section III-E, there are potentially four types of insecure practices associated with cloud database service. For **SRA-1**, mini-app developers share privileged resources with the mini-app client. Specifically, to identify the resources stored

in cloud databases, we analyze the sensitive APIs provided by super-apps and variable names of the stored content in cloud databases of real-world mini-apps, inspired by [9], [23], [24]. Specifically, three experts analyze and categorize the content stored in the cloud database into different types of sensitive resources, and construct a corresponding set of keywords associated with each type. Then ICREMINER analyzes the collection names and return values of cloud databases collected from the mini-app client to determine the stored content based on keyword matching. However, since the content stored in hidden cloud databases is unknown, ICREMINER is unable to assess whether the hidden cloud database might also be vulnerable without actually retrieving the stored data. For **SRA-2**, the focus shifts to analyzing write operations to sensitive resources. ICREMINER extracts write operations based on the actions of cloud operations, such as setting or updating the cloud database. It then analyzes the parameter names and the data dependencies to determine whether any content that should not be modified by mini-app users (e.g., product price, account balance) is included in the write operations.

For other insecure practices, i.e., **UIC-1** and **UIC-2**, the focus is on detecting issues associated with user identity check. If mini-app developers move user identity check to the mini-app client side or rely on client-side parameters for user identity check, an attacker can inject a cloud operation to access the cloud resources belonging to other mini-app users. Therefore, ICREMINER analyzes whether the exposed capabilities detected in Section IV-C are more-than-expected and allow access to other users' resources. The primary challenge lies in identifying the resource owners, i.e., determining which data belongs to which users. We discovered that when a mini-app user adds cloud data to the cloud database from the mini-app client, super-apps attach the mini-app user's `userId` to the cloud data. Therefore, we can determine the resource owner based on `userId`.

Since there is no official API available to retrieve current user's `userId`, we propose a method to assess the security risk using new user accounts. Our key insight is that, for users who have not previously utilized the mini-app, their data should not exist in the cloud database. Therefore, if the user obtains a non-empty data record containing a `userId` in the cloud database, it indicates a potential exposure of another user's information. To avoid harming mini-app users' privacy and compromising the cloud services, we record only a binary response for the `userId`, indicating whether it exists or not. Besides, no researcher has access to the cloud resources or the `userId`. Furthermore, to analyze the specific type of insecure practice, ICREMINER conducts an in-depth analysis of cloud behaviors. Specifically, ICREMINER analyzes if mini-app developers moves the identity check to the mini-app client side (**UIC-1**) or if mini-apps transfer the customized user identity to the cloud side within the parameters (**UIC-2**).

Regarding hidden cloud databases, since the stored resources are unknown, it is impossible to determine whether privileged resources are exposed without analyzing the stored data. If a `userId` is identified, ICREMINER labels it as a special instance of insecure practice **UIC-1**, where client-side data handling does not exist due to the lack of resource access.

Assessment of Cloud Storage Service. Unlike cloud databases, the default permission for cloud storage allows all mini-app users to read stored files with file IDs. As discussed in Section III-E, the insecure practices of cloud storage mainly arise from the fact that many mini-app developers rely on file IDs for user identity check (**UIC-2**). If an attacker knows the cloud file path, they can easily access the stored files, such as users' passport pictures and confidential company files. Additionally, even if the mini-app developer does not set the file permissions to be publicly readable, the cloud path of the stored file itself can still pose significant security risks. Some popular cloud service providers (e.g., AWS and Azure) offer developers specific APIs, such as `s3:ListObjects` [25], to list container objects in cloud storage. These APIs require certain permissions and can only be accessed with privileged credentials. If the cloud path is guessable, an attacker could potentially list all the files stored in the cloud storage.

Therefore, we aim to determine whether a file path is guessable. However, it is not an easy task, as file paths are often generated through complex operations rather than being composed of constant values. In particular, many file paths consist of a sequence of strings, with each segment typically derived from a mini-app framework API or a JavaScript built-in string API, such as substring, concat, or slice.

TABLE I
Common patterns that generate guessable variables.

Pattern	Description
new Date().getTime()	Return the current timestamp.
Date().now	Return the current timestamp.
Math.round(100*Math.random())	Generate a random number with in range of 0-99.
Math.floor(100*Math.random())	Generate a random number within range of 0-99.
`str.match(/\.[^.]+?$/)`	Extract the file extensions from file path, e.g., '.txt' and '.jpg'.
`/\.\w +$/.exec(str)`	Extract the file extensions from file path, e.g., '.txt' and '.jpg'.

After analyzing the file paths extracted from mini-apps, we observe that various string operations can produce guessable file paths. For example, `new Date()` generates a timestamp for the current date, which could potentially be exploited by an attacker to obtain files created within a specific time frame. Besides, for strings that are initially unguessable, operations like regular expressions or substring manipulations can significantly reduce their complexity. For instance, the return value of `wx.chooseImage` is randomly generated, but the operation `file.match(/.\w+$/)` extracts only the file suffix, such as '.jpg', making the variable guessable. Therefore, ICREMINER models these operations and performs backward data flow analysis to trace the generation sequence of file paths. We summarize the common patterns that produce guessable variables in Table I.

Based on these patterns, we propose a heuristic strategy to iteratively analyze the generation sequence of a file path to determine whether it is guessable. Specifically, we consider a string guessable if its length is limited (e.g., less than three characters), or if its value range is restricted (e.g., less than 1,000). However, it is hard to process scenarios where the string contains regular expressions due to their complex structures and varied formats. Our key insight is that if a regular expression generates guessable variables, such as file suffixes, the execution results for different file paths should be identical or similar. To identify these cases, ICREMINER applies the extracted regular expressions to three randomly generated file paths and observe whether the outputs are guessable. Moreover, if the file path is dynamically generated from a remote server, it is considered unguessable because the server-side logic is unknown. Finally, if all the file components are guessable, the entire file path is considered guessable.

Assessment of Cloud Routine Service. Cloud routines provide tailored services for all mini-app users. Unlike the two aforementioned cloud services, mini-app developers can customize cloud routines in various formats and access the code logics in the cloud side, such as interacting with cloud databases and cloud storage. The types of insecure practices associated with cloud routines are similar to those found in cloud databases. For **UIC-1**, ICREMINER identifies the existence of data handling for return values in mini-app client side, such as retrieving specific user information from return values based on user identity. For **UIC-2**, ICREMINER analyzes the parameters of cloud routines to determine whether mini-app developers transfer customized user IDs to retrieve or manipulate cloud resources which violates secure practices for retrieving user identity in the server side.

For other types of insecure practices (**SRA-1** and **SRA-2**), which mainly focus on that whether cloud routines exposed privileged resources to mini-app users. ICREMINER analyzes the return values of cloud routines to identify the accessible cloud resources and examines the parameters to identify the controlled content in write operations. Then, ICREMINER analyzes whether privileged resources are present in the return values or parameters. Similarly, ICREMINER identifies the sensitive resources based on the previously categorized types and keyword set, to determine whether the keywords exist within the resources. Although these practices do not necessarily indicate actual exposure of sensitive cloud resources, they represent potential vulnerabilities, which are further verified in the evaluation section.

V. Security Assessment

In this section, we discuss our results in the detection of above insecure practices. We apply ICREMINER on a large-scale of real-world mini-apps to understand and assess the security hazards that can be caused by ICREM. We first introduce the experiment setup and then present the overview of our assessment results. Next, we discuss the interesting findings and the potential security hazards with case studies.

A. Experiment Setup

Analysis Statistics. To evaluate the effectiveness of ICREMINER and to uncover the landscape of ICREM risks in mini-apps, we conducted a large-scale analysis of 1,248,815 mini-apps, including 985,503 WeChat mini-apps, 83,312 TikTok mini-apps, 93,128 Alipay mini-apps, and 86,872 Baidu mini-apps. To assess the insecure cloud practices in mini-apps, we performed a preliminary analysis to filter mini-apps that access app-in-app cloud services based on cloud API identification. As a result, we identified 22,695 mini-apps. The experiments are conducted on an Ubuntu 18.04 LTS 64-bit server with 64 CPU cores (2.3GHz) and 206GB memory. On average, our analysis takes 13.6s in the client-side operation extraction and 63.9s in the capability inference phase for each mini-app.

Research Questions. In summary, we aim to answer the following two research questions:

- *RQ1: How many mini-apps are influenced by the ICREM risks in the wild?*
- *RQ2: What real-world impacts are posed by ICREM risks?*

B. Performance Validation

It is critical to understand the performance of ICREMINER for ensuring the reliability of our security assessment.

Accuracy of ICREMINER. To assess the accuracy, we manually checked all the detected vulnerabilities in the mini-apps which access the cloud services. Specifically, three domain experts manually analyzed the mini-app code to determine whether the detection results are true positives. A result was considered a true positive only if all experts reached a consensus. In cases of disagreement, the experts were asked to discuss their labeling criteria.

Furthermore, we conducted a systematic exploitability analysis in a controlled environment, and throughout the evaluation, ethical considerations are at the forefront. Specifically, we used two test accounts to simulate attacks against each other, demonstrating the exploitability and real-world impact, such as unauthorized file access and sensitive information leakage. For example, we modified the `fileId` of user A to that of user B to determine whether user A could access user B's resources. In total, we analyzed 1,943 mini-apps. The remaining mini-apps contain write operations to cloud resources. To avoid raising ethical concerns, we did not perform any write operations that could alter user balances or trigger unauthorized payments. Instead, we conducted code-level semantic analysis to analyze the functionalities and parameters involved, allowing us to assess their potential real-world impact. For example, a cloud database named "balance" suggests that it stores users' account balances.

Finally, three experts took about one month each to evaluate the detected mini-apps. As illustrated in Table II, ICREMINER detected 3,057 vulnerable mini-apps and 8,289 vulnerable cloud operations in these mini-apps. Moreover, 97.26% of the detected cloud operations expose sensitive resources.

TABLE II
A summary of vulnerable mini-apps before and after manual filtering of false positives. op means cloud operations and %op represents the proportion of detected operations relative to the preliminary results.

Cloud Service	Preliminary Results			False Positives/Insensitive Operations			Sensitive Operations		
	# op	# app	% total	# op	# app	% op	# op	# app	% op
cloud database	6759	2133	9.45%	57	53	0.84%	6702	2112	99.16%
cloud storage	1147	1022	4.53%	124	103	10.81%	1023	709	89.19%
cloud routine	383	316	1.40%	46	42	12.01%	337	284	87.99%
Total	8289	3057	13.54%	227	197	2.74%	8062	2815	97.26%

TABLE III
Detection results of different methods to infer cloud database names. (1) Common Name Augmentation: Extend analysis using frequently used database names, (2) LLM-based Inference: Utilize LLM to infer cloud database names, (3) Mini-app Correlation: Correlate cloud database names across different mini-apps.

Method	Detection Results		Insecure Practices	
	# op	# app	# op	# app
Common Name Augmentation	1197	879	195	180
LLM-based Inference	3309	810	1100	390
Mini-app Correlation	141	57	40	20
ICREMINER	4202	1486	1262	539

Specifically, there are 53 false positives in the detection result associated with cloud databases, primarily due to that the exposed resources are not privacy-sensitive. For example, some mini-apps use 'credential' to name unrelated variables. Additionally, some cloud databases are used for testing and do not expose sensitive resources. Moreover, there are some hidden cloud operations that are not accessed in the mini-app client. Since the contents stored in them are unknown, we cannot determine whether the contents are sensitive and do not consider them in the the assessment. For cloud storage, we assess whether the cloud path is guessable and there are no false positives, indicating that ICREMINER can accurately identify guessable cloud paths. Furthermore, we analyze the file descriptions provided in the mini-app client and filter out stored files that are not privacy-sensitive, such as user avatars and public feed images. As for cloud routines, the false positives are primarily due to that the sensitivity of cloud resources varies depending on the context. For example, mini-app users pay for VIP service in a mini-app as***FM, and the mini-app invokes a cloud routine `setVIP` to change the user's status. An attacker could exploit this by directly invoking the cloud routine without making a payment. However, for unpaid services, this vulnerability does not exist since users can apply for VIP status for free, typically for promotional purposes.

To assess the false negatives, we randomly selected 100 mini-apps that were not flagged as vulnerable, and then manually analyzed the mini-app code to check whether they contain undetected vulnerabilities. Taking the cloud routine as an example, if we find that the cloud routine exposes sensitive resources that were not detected, we consider it a false negative. As a result, we do not find any false negatives.

TABLE IV
The statistics of large-scale assessment.

Super -app	Cloud Database		Cloud Storage		Cloud Routine	
	# app	% total	# app	% total	# app	% total
WeChat	2057	9.39%	673	3.07%	247	1.13%
TikTok	26	9.09%	10	3.50%	10	3.50%
Alipay	13	4.22%	15	4.87%	15	4.87%
Baidu	16	7.84%	11	5.39%	12	5.88%
Overall	2112	9.31%	709	3.12%	284	1.25%

Performance of Hidden Capability Excavation. As illustrate in Table III, we compare the effectiveness of our proposed methods for inferring hidden capabilities. By utilizing all these methods, ICREMINER was able to identify 4,202 hidden cloud database names in 1,486 mini-apps and 539 mini-apps (36.27%) are vulnerable, which expose excessive cloud resources to mini-app users. Although hidden capabilities do not necessarily indicate a problem, a secure implementation ensures that only necessary cloud resources can be accessed in the mini-app client, adhering to the "minimization principle".

Specifically, LLM can effectively assist in generating valid database names and "LLM-based Inference" method proved more effective, uncovering 390 vulnerable mini-apps. Additionally, the "Common Name Augmentation" method contributed to the detection of 1,197 valid database names in 879 mini-apps, which indicates that some database names are frequently used by different mini-apps. Furthermore, the "Mini-app Correlation" method also helped ICREMINER identify 20 vulnerable mini-apps.

C. RQ1: Landscape of ICREM Risks

To evaluate the prevalence of vulnerabilities in real-world mini-apps, we conduct a large-scale analysis of mini-apps from different super-apps and present our findings below. Additionally, we discuss the generality of ICREMINER in Section VI.

Prevalence of ICREM. As illustrated in Table II, 2,815 mini-apps that access cloud services suffer from insecure practices, leading to the exposure of sensitive resources. We provide detailed case studies of vulnerable mini-apps that access different types of cloud services in Appendix B.

We compare the detection results across different super-apps in Table IV. The majority of vulnerable mini-apps are WeChat mini-apps, primarily because a larger proportion of

TABLE V
The statistics of different types of insecure practices.

Super-app	User Identity Check						Sensitive Resource Allocation					
	UIC-1			UIC-2			SRA-1			SRA-2		
	# op	# app	% total	# op	# app	% total	# op	# app	% total	# op	# app	% total
WeChat	1765	886	32.65%	4306	1846	67.94%	93	81	2.98%	1762	984	36.22%
TikTok	12	12	30.77%	19	12	30.77%	15	10	25.64%	9	7	17.95%
Alipay	13	13	39.39%	24	22	66.67%	8	8	24.24%	0	0	0
Baidu	7	7	26.92%	22	16	61.54%	7	7	26.92%	3	2	7.69%
Overall	1797	919	32.65%	4371	1896	67.35%	123	106	3.77%	1774	993	35.28%

WeChat mini-apps integrate cloud services. In contrast, cloud services are less commonly used in the mini-apps of other super-app platforms. For instance, there are 204 Baidu mini-apps incorporate cloud services. As a result, WeChat mini-apps exhibit more security vulnerabilities. Furthermore, our detection results show that mini-apps in other platforms also pose severe risks and expose sensitive cloud resources. Our in-depth analysis of the mini-app client reveals that a significant portion of the vulnerable cloud databases (22.17%) are associated with user information, such as user's phone number and billing address, posing a serious risk to users' privacy. There are also many vulnerable mini-apps that expose sensitive files stored in the cloud storage. Since mini-app developers often store users' files, including educational materials and company documents, in easily guessable cloud paths, attackers can obtain the cloud paths to download the corresponding files.

Unlike cloud database or cloud storage, cloud routines used in mini-apps are often accessible to all mini-app users. To date, no systematic work has revealed their security risks. As illustrated in Table IV, many mini-app developers mistakenly expose sensitive operations through cloud routines, such as altering a user's account balance. Additionally, several privilege escalation issues exist within cloud routines. For example, many mini-app developers implement cloud routines that retrieve user information based on phone number or simple user ID, which attackers can tamper with to access other users' information.

Insecure Practices. As shown in Table V, the four types of insecure practices are prevalent in mini-apps. The most common insecure practices are related to user identity check (UIC-1 and UIC-2). Specifically, 919 mini-apps move the identity check to the mini-app client side, and 1,896 mini-apps use client-side parameters for the user identity check. However, an attacker can arbitrarily change the parameter to access cloud resources of other users or directly steal the exposed cloud resources before the client-side check. Moreover, many mini-app developers assign write permissions to sensitive resources, such as account balance (SRA-2). For example, an e-commercial mini-app hui*** accesses cloud database 'user' to change user's account balance in the mini-app client side after the user completes a payment. The attacker can directly access the cloud database to increase the account balance without any payment. Additionally, 3.77% of mini-apps expose privileged resources to the mini-app client side (SRA-1), most of which are API keys and `AppSecret`[26], the root credential of mini-apps.

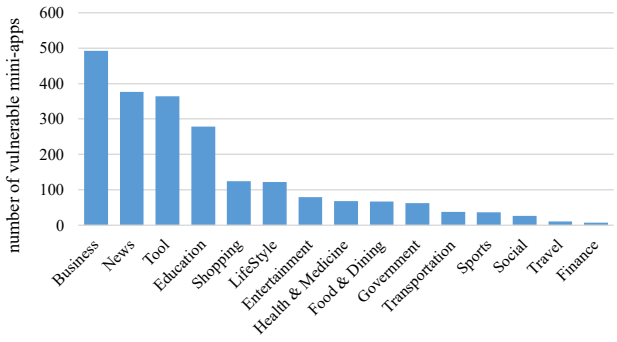

Fig. 7. Distribution of vulnerable mini-app categories.

Distribution Across Categories. We then analyze the number of vulnerable mini-apps across different categories. As illustrated in Figure 7, there are more vulnerable mini-apps in categories that aggregate a large amount of information, such as Business and Education. These mini-apps often contain sensitive information, like company files and student details, which pose a severe leakage risk. There are also many mini-apps in the categories that related to user daily life, such as News, Tool and Shopping, which also have a high proportion of ICREM risks. These categories mainly collect user's profile and online activities. Although there are relatively fewer mini-apps in some categories, such as Government, the exposed cloud resources can lead to severe consequences.

1) Key Observations: Here, we highlight two key observations obtained during the assessment of real-world mini-apps.

"Write Operation" Vulnerability. Although we cannot directly verify if mini-app developers have correctly configured write permissions, we can still uncover the "tip of the iceberg" regarding the security of write operations conducted in the mini-app client. The common write operations is to add or update user information, such as nickname or avatar. However, issues arise when certain write operations, which should be restricted, are accessible to users. For example, when users attempt to recharge their account balance, it should be updated only after the payment is successfully completed. However, some mini-app developers mistakenly handle the balance update operations in the mini-app client side, which

allows attackers to manipulate their balance without making any actual payment. Through static analysis, we can identify which cloud databases are writable by users based on specific cloud APIs. From the analysis of written content, we found that 993 mini-apps have vulnerable write operations.

Unsafe write operations can be categorized into two types. The first type allows users to write sensitive personal information. For example, in the mini-app AWS***, an attacker can add themselves to the admin list without undergoing authentication. Additionally, in a mini-app exp***, an attacker can arbitrarily change the account balance without any payment. The second type allows users to write to system resources. For example, a mini-app allows users to modify the notification content. Attackers might exploit this capability to alter notifications and deceive other users.

Templated Cloud Services. Some mini-apps are developed by the same developers using the same templates, which results in similar vulnerability patterns across these mini-apps and exacerbates the vulnerabilities. Among our detection results, we found that 125 mini-apps (4.4%) were developed using templates. On average, each template is used to develop approximately 12 mini-apps. For instance, we identified a third-party mini-app template for entertainment services that exposes sensitive cloud routines to the client side, which can be exploited by attackers to send notifications to other mini-app users. This vulnerability affects 15 mini-apps developed using this template. Additionally, the correlation results in the hidden capability inference demonstrate that mini-apps developed by the same developers often have similar or templated implementations of cloud services. These mini-apps often have the same insecure practices and expose sensitive cloud resources.

D. RQ2: Real-world Impact and Case Studies

To demonstrate the real-world impact of ICREM risks, we first analyze the scale of affected users. According to data provided by super-app platforms, 115 vulnerable mini-apps each have over 100,000 users, and the cumulative number of affected users exceeds 70 million. Furthermore, we demonstrate the severity of ICREM risks by presenting the security hazards implied in real-world mini-apps, combined with detailed case studies.

Data Leakage in Cloud Services. Many mini-apps suffer from data leakage, particularly those that expose excessive privileges for users to access sensitive cloud data. For example, a hospital mini-app g*** exposes the patient's information stored in the cloud database `user`, including patient's name, age, phone number and medical information.

As for the cloud storage, which is often made publicly readable, if an attacker knows the file path, they can access the cloud files. To keep the file path secret, super-apps provide mini-apps APIs such as `wx.chooseImage` to choose local images and the file name is randomly generated. However, mini-app developers may use guessable file names, such as timestamps. For example, an education mini-app b*** stores students' educational materials with names based on timestamps, which can be easily guessed and traversed by an attacker. Using two test accounts, we successfully retrieved the files of another account stored in the cloud server. Specifically, we first logged into account A, uploaded files and recorded the filenames. Then, we logged into account B and injected access to account A's files into the target mini-app. Finally, we successfully obtained account A's files.

Pay for Free. In the app-in-app ecosystem, the typical recharge process involves two steps: the user completes the payment, and then the mini-app updates the balance in the cloud side. However, some mini-app developers execute the second step in the mini-app client side, which can be exploited by a malicious user. For instance, in a third-party ChatGPT mini-app, AI***, users can purchase tokens to access GPT services. After a payment is made, the mini-app directly invokes the cloud database service `user***Tokens` to update the user's account balance. This enables an attacker to manipulate the cloud database and arbitrarily increase token amounts without completing a payment.

Privilege Escalation. A mini-app typically includes various user roles, such as ordinary users, VIP users, and administrators. These roles can change in certain scenarios. For example, ordinary users can upgrade to VIP status by making a payment or entering an activation code. Some mini-apps wrongly place role-change functionalities in the mini-app client side. For example, in an education mini-app, the cloud database `vip***` is exposed to mini-app users. When a user enters the correct activation code, the mini-app adds them to the cloud database as a VIP user. Attackers could exploit this by directly writing to the cloud database, adding themselves to the VIP user list.

VI. Discussion

Generality of ICREMiner and Limitations. To understand the impacts of ICREM risks on different super-apps, we conducted an in-depth analysis of several prominent super-apps, i.e., WeChat, TikTok, Baidu, and Alipay. The cloud services in these super-apps have similar implementations, and the cloud APIs have similar names. For example, WeChat uses `wx.cloud.uploadFile` [10], while Alipay employs `CloudContext.uploadFile` [27] for uploading files to cloud storage. Besides, to perform our attacks, the attacker needs to inject malicious payload into the target mini-app, using methods such as exploiting the caching mechanism as discussed in Section III-B. Through further analysis of super-apps, we found that while they implement different integrity checks, these local checks can be easily bypassed. Therefore, ICREMiner can be easily extended to different super-apps. Moreover, attackers do not need to publish a malicious mini-app in the mini-app market and they can execute the attack from their own devices, making it practical.

Furthermore, there are several limitations in our proposed method. First, as discussed in Section V-B, our method may introduce some false positives, primarily because some data are not privacy-sensitive. ICREMiner determines whether

data is privacy-sensitive based on variable names and data dependencies. Considering additional semantics from the mini-app client side may bring improvement, which we leave for future work. Second, hidden cloud operations cannot be fully discovered. Although we have proposed several inference methods, it is impossible to achieve complete inference based on client-side analysis. Third, our current work focuses on the analysis in four prominent super-apps. The assessment can demonstrate the scalability of ICREMINER, and we also analyze the security issues in other super-apps, such as Line and VK, as presented in Appendix C.

Ecosystem-specific Factors. Unlike traditional cloud architectures, where cloud credentials issued to developers are used for both authentication and authorization, the app-in-app ecosystem decouples these two processes. Specifically, the super-app platform provides a centralized authentication mechanism, managing user identities across all mini-apps. This design often leads mini-app developers to assume that adequate security protections have been enforced by the super-app platform. However, super-app platforms only manage user identities and do not enforce access control over the cloud resources accessible to users. This fundamental change causes inconsistent authorization enforcement in different mini-apps and introduces new security risks. We believe that super-app platforms should play a more proactive role to provide stronger support for secure authorization. Furthermore, super-app platforms do not ensure the integrity of the mini-app code they host, which further exacerbates the risks. We have reported and discussed our findings with the super-app platform.

Mitigations. For super-apps, it is essential to provide mini-app developers with clear documentation to help them understand the security mechanism of app-in-app cloud services and how to correctly safeguard sensitive resources. Super-apps should remind mini-app developers to set appropriate permissions for sensitive resources. Furthermore, super-apps can implement stricter access control mechanisms. For example, sensitive resources can be tagged with user IDs, allowing the super-app to verify resource ownership when a malicious user attempts to access them. Besides, super-apps can take measures to detect runtime code injection (e.g., Xposed, Frida) to mitigate the security issues. In addition, super-apps can assess the security of the current runtime environment, such as checking whether the device has been rooted or whether other security compromises are present.

For mini-app developers, it is crucial to avoid placing sensitive operations, such as retrieving root credentials, in the mini-app client side. Besides, mini-app developers must ensure that access to sensitive resources is properly secured, especially by performing user identity check in the cloud side. Moreover, mini-app developers need to focus on the hidden cloud operations, which can also expose sensitive resources. We believe that ICREMINER offers a valuable tool for detecting and mitigating insecure resource management by mini-app developers. Furthermore, It is feasible for mini-app developers to utilize our tool to find potential security risks.

VII. RELATED WORK

Studies on App-in-App Ecosystem. In recent years, many studies focus on the security of the app-in-app ecosystem, including the security of mini-apps [24], [26], [28] and super-apps [29], [30], [31], [32], [33]. For example, Lu et al. [29] systematically analyzed the resource management flaws in super-apps, which fail to provide effective control to protect privileged APIs. [33] studied hidden APIs in super-apps that are undocumented and can expose sensitive resources. [26], [28], [34] identified credential leakage issues in mini-apps. [31] discovered discrepancies in the sensitive APIs of WeChat across platforms, which can be exploited by attackers to steal user's privacy. Zhang et al. [30] conducted a systematic study on identity confusion vulnerabilities, where super-apps fail to properly verify the mini-app identity before granting access to privileged APIs and found three types of confusion vulnerabilities. In contrast, our work focuses on a new issue: insecure practices in cloud resource management by mini-apps. Besides, the vulnerabilities we identified have not been uncovered in previous work.

Identity Checks. Many studies have examined the security flaws in identity check. Previous research [15], [35] mainly focused on the check of user's identity in the server side, to prevent unauthorized access to other users' resources. For web applications, many research focused on the security issues during multi-origin communications [36], [37], [38], [39], especially the issues within postMessage Handler. Furthermore, Zhang et al. [30] conducted the first systematic study of identity confusions in this ecosystem and identified three types of confusion vulnerabilities. Yang et al. [24] identified the issue of missing appId checks in the receiver mini-app during cross-mini-app communication. We perform the first systematic study of the identity management mechanism for mini-app users in the app-in-app ecosystem.

Broken Access Control. The identified insecure practices in mini-apps fall under the category of "Broken Access Control" in the OWASP Top 10 [40]. This issue is prevalent in mobile and web applications and has been studied in prior work [41], [35], [15], [42], [17], [43]. These work primarily focus on horizontal privilege escalation. For example, Zuo et al.[42] investigated post-authentication security and identified the vulnerable access control in mobile services. Li et al.[17] analyzed user tag spoofing attacks in mobile apps, and Liu et al.[43] studied vulnerabilities caused by missing owner checks in web applications. In contrast to these works, we focus on systematically analyzing security issues in cloud resource management within the app-in-app ecosystem, which differ significantly in both architecture and threat models.

Cloud Security. Serverless services have experienced rapid growth in recent years, and many studies focus on the cloud security. Several work used formal methods to verify the security of permission configurations in cloud side [44], [45], [46], [19], [47]. Shevrin et al. [19] formally examined multi-step attacks that exploit cloud policy misconfigurations. Besides, some studies investigated data breaches in cloud

infrastructures [20], [48], [49], where sensitive information is exposed through publicly accessible cloud storage buckets. Additionally, may work focused on the misuse or permission misconfigurations of cloud credentials [5], [6], where high-privileged credentials are exposed to the client side, allowing attackers to access sensitive resources.

The work most related to us is PrivRuler [6]. PrivRuler conducted a fine-grained assessment of the cloud capabilities owned by mobile apps, and verified if these capabilities exceed the legitimate requirements of the apps. In contrast, our study focuses on the security implications of ICREM, with sensitive cloud resource exposure as part of the implication. It's not trivial to apply traditional method to this task. First, unlike traditional cloud architectures, which issue cloud credentials to developers to manage access to cloud resources, the app-in-app cloud architecture is based on the identities, which are managed by the super-apps. Second, while PrivRuler only focuses on cloud storage and notification service, we conduct a systematic analysis of three types of app-in-app cloud services, i.e., cloud database, cloud storage and cloud routine. Third, PrivRuler aims to infer accessible cloud containers as it assumes that cloud policies are often defined at the granularity of container objects. However, mini-apps operate with permissions granted at the element level. Consequently, PrivRuler's vulnerability patterns are unsuitable for detecting vulnerabilities in this context. We propose a novel method that employs semantic inference to identify potential vulnerabilities in mini-apps, including the hidden capabilities.

VIII. Conclusion

In this paper, we perform the first systematic study of the insecure cloud resource management in the app-in-app ecosystem. We design and implement a novel approach, called ICREMiner, combined with static analysis and dynamic probing to assess the security implications. We also propose methods to detect hidden vulnerability in the cloud side with the help of LLM. By applying ICREMiner on 1,248,815 real-world mini-apps, we find 22,695 mini-apps access app-in-app cloud services and 2,815 of them (12.40%) are affected by the insecure resource management. Finally, we have made responsive disclosure to the super-app platforms and corresponding mini-app developers. We also provide several mitigations to enhance the security of the app-in-app ecosystem.

IX. Ethics Considerations

Throughout the course of our research, we ensure compliance with all relevant laws and regulations, and adhere to community practices and guidelines, such as those outlined by [50] and [51]. Our work was reviewed by our institution's IRB and this study is considered as "minimal risk". Furthermore, we followed best practices established by prior related work to assess the cloud resources unauthorized access [5], [6], [20]. To perform necessary evaluation, we only used our own accounts with explicit consent of all participants involved. Besides, we did not store any data or launch any attacks against the cloud server to collect or manipulate sensitive resources.

To avoid impacting mini-apps' cloud services, we proposed a variety of innovative methods to analyze the security of cloud services based on the semantics in the mini-app client side. During the dynamic probing, we carefully followed the practices of prior researches [5], [6], [52], [53], [20], [54], which focus on identifying vulnerable or malicious services, including the cloud services. Besides, we adhered to principles that prevent the leakage of real user's data and avoid introducing changes to the cloud services. In addition to these guidelines, we made extra precautions to minimize any potential harm to cloud services or mini-app users. First, we did not perform any write operations and conducted security inferences without accessing the cloud data. Second, we did not conduct fuzz testing on cloud services. Instead, we limited the detection rate to simulate manual access, thereby preventing any potential damage to the cloud services. Moreover, for the mini-app dataset collection, we adhered to the methods established in previous studies [26], [34]. To avoid overloading the super-app's servers, we also limited the download speed to a few seconds per mini-app.

Regarding vulnerability disclosure, to help address the vulnerabilities and prevent the exposure of mini-apps' sensitive data, we adhered to responsible disclosure practices to super-app platforms and mini-app developers. The super-app platforms have recognized this vulnerability. Besides, we actively worked together with them to fix these problems. To notify the affected mini-app developers, we made efforts to extract contact information from various sources, including privacy policies, and official websites of the companies behind the mini-app. Consequently, we collected emails for 1,869 mini-apps. Most of the remaining mini-apps were developed by individual developers, making it difficult to collect the contact information. We have reported detected vulnerabilities to the corresponding developers. To date, we have received 35 responses and 893 mini-app developers have fixed the issues or taken down their mini-apps. The mini-app developers have expressed their gratitude for our efforts in detecting vulnerabilities. They have actively sought our assistance in resolving the issues, and we have helped them fix the vulnerabilities.

Acknowledgment

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 Natural Science Foundation of China (62172104, 6240211). Zhemin Yang and Min Yang are the corresponding authors. Min Yang is 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 April 23, 2025) The development trends of wechat mini-program. [Online]. Available: https://qpsoftware.net/blog/development-trends-wechat-mini-program

[2] Amazon. (Accessed April 23, 2025) Amazon web services. [Online]. Available: https://aws.amazon.com

[3] Microsoft. (Accessed April 23, 2025) Microsoft azure. [Online]. Available: https://azure.microsoft.com

[4] Google. (Accessed April 23, 2025) Google cloud: Cloud computing services. [Online]. Available: https://cloud.google.com

[5] 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 (SP)*, 2019.

[6] 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.

[7] C. Fang, N. Miao, S. Srivastav, J. Liu, R. Zhang, R. Fang, A. Asmita, R. Tsang, N. Nazari, H. Wang *et al.*, "Large language models for code analysis: Do llms really do their job?" 2024.

[8] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu, "Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis," 2024.

[9] C. Yue, K. Chen, Z. Guo, J. Dai, X. Sun, and Y. Yang, "What's done is not what's claimed: Detecting and interpreting inconsistencies in app behaviors," in *Network and Distributed System Security Symposium (NDSS)*, 2025.

[10] WeChat. (Accessed April 23, 2025) uploadFile. [Online]. Available: https://developers.weixin.qq.com/minigame/en/dev/wxcloud/reference-client-api/storage/uploadFile.html

[11] PortSwigger. (Accessed July 9, 2025) Getting started with burp suite. [Online]. Available: https://portswigger.net/burp/documentation/desktop/getting-started

[12] Charles. (Accessed July 9, 2025). [Online]. Available: https://www.charlesproxy.com/

[13] Tencent. (Accessed April 23, 2025) Mmtls: Introduction of tlsv1.3 based tecent security communication prtocol. [Online]. Available: https://github.com/WeMobileDev/article/blob/master/SUMMARY.md

[14] WeChat. (Accessed April 23, 2025) getWXContext. [Online]. Available: https://developers.weixin.qq.com/miniprogram/en/dev/wxcloud/reference-server-api/utils/getWXContext.html

[15] M. Monshizadeh, P. Naldurg, and V. Venkatakrishnan, "Mace: Detecting privilege escalation vulnerabilities in web applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[16] F. He, Y. Jia, J. Zhao, Y. Fang, J. Wang, M. Feng, P. Liu, and Y. Zhang, "Maginot line: Assessing a new cross-app threat to pii-as-factor authentication in chinese mobile apps," in *Network and Distributed System Security Symposium (NDSS)*, 2024.

[17] S. Li, Z. Yang, G. Yang, H. Zhang, N. Hua, Y. Huang, and M. Yang, "Notice the imposter! a study on user tag spoofing attack in mobile apps," in *32nd USENIX Security Symposium (USENIX Security)*, 2023.

[18] 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.

[19] I. Shevrin and O. Margalit, "Detecting {Multi-Step} {IAM} attacks in {AWS} environments via model checking," in *32nd USENIX Security Symposium (USENIX Security)*, 2023.

[20] S. El Yadmani, O. Gadyatskaya, and Y. Zhauniarovich, "The file that contained the keys has been removed: An empirical analysis of secret leaks in cloud buckets and responsible disclosure outcomes," in *2025 IEEE Symposium on Security and Privacy (SP)*, 2024.

[21] Amazon. (Accessed April 23, 2025) Aws bucket. [Online]. Available: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-s3-bucket.html

[22] J. Lee, J.-K. Min, and C.-W. Chung, "An effective semantic search technique using ontology," in *Proceedings of the 18th international conference on World Wide Web (WWW)*, 2009.

[23] Y. Nan, Z. Yang, X. Wang, Y. Zhang, D. Zhu, and M. Yang, "Finding clues for your secrets: Semantics-driven, learning-based privacy discovery in mobile apps." in *Network and Distributed System Security Symposium (NDSS)*, 2018.

[24] 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.

[25] Amazon AWS. (Accessed April 23, 2025) ListObjects. [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/API/API_ListObjects.html

[26] 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.

[27] Alipay. (Accessed April 23, 2025) uploadFile. [Online]. Available: https://opendocs.alipay.com/cloud/09tn5v?pathHash=5b248a39

[28] 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.

[29] 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.

[30] 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.

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

[32] ——, "Rootfree attacks: Exploiting mobile platform's super apps from desktop," in *The 19th ACM ASIA Conference on Computer and Communications Security (ACM AsiaCCS)*, 2024.

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

[34] Y. Shi, Z. Yang, K. Zhong, G. Yang, Y. Yang, X. Zhang, and M. Yang, "The skeleton keys: A large scale analysis of credential leakage in mini-apps," in *Network and Distributed System Security Symposium (NDSS)*, 2025.

[35] S. Son, K. S. McKinley, and V. Shmatikov, "Fix me up: Repairing access-control bugs in web applications." in *Network and Distributed System Security Symposium (NDSS)*, 2013.

[36] Y. Cao, V. Rastogi, Z. Li, Y. Chen, and A. Moshchuk, "Redefining web browser principals with a configurable origin policy," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013.

[37] G. Yang, J. Huang, and G. Gu, "{Iframes/Popups} are dangerous in mobile {WebView}: Studying and mitigating differential context vulnerabilities," in *28th USENIX Security Symposium (USENIX Security)*, 2019.

[38] G. Yang, J. Huang, G. Gu, and A. Mendoza, "Study and mitigation of origin stripping vulnerabilities in hybrid-postmessage enabled mobile applications," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[39] M. Steffens and B. Stock, "Pmforce: Systematically analyzing postmessage handlers at scale," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[40] (Accessed September 6, 2024) Owasp top 10. [Online]. Available: https://owasp.org/Top10/

[41] F. Sun, L. Xu, and Z. Su, "Static detection of access control vulnerabilities in web applications," in *20th USENIX Security Symposium (USENIX Security)*, 2011.

[42] C. Zuo, Q. Zhao, and Z. Lin, "Authscope: Towards automatic discovery of vulnerable authorizations in online services," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[43] F. Liu, Y. Shi, Y. Zhang, G. Yang, E. Li, and M. Yang, "Mocguard: Automatically detecting missing-owner-check vulnerabilities in java web applications," in *2025 IEEE Symposium on Security and Privacy (SP)*, 2025.

[44] M. Bouchet, B. Cook, B. Cutler, A. Druzkina, A. Gacek, L. Hadarean, R. Jhala, B. Marshall, D. Peebles, N. Rungta *et al.*, "Block public access: trust safety verification of access control policies," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE/ESEC)*, 2020.

[45] W. Eiers, G. Sankaran, A. Li, E. O'Mahony, B. Prince, and T. Bultan, "Quantifying permissiveness of access control policies," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, 2022.

[46] Z. Jin, L. Xing, Y. Fang, Y. Jia, B. Yuan, and Q. Liu, "P-verifier: Understanding and mitigating security risks in cloud-based iot access policies," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[47] J. Backes, U. Berrueco, T. Bray, D. Brim, B. Cook, A. Gacek, R. Jhala, K. Luckow, S. McLaughlin, M. Menon, D. Peebles, U. Pugalia, N. Rungta, C. Schlesinger, A. Schodde, A. Tanuku, C. Varming, and D. Viswanathan, "Stratified abstraction of access control policies," in *Computer Aided Verification: 32nd International Conference*, 2020.

[48] J. Cable, D. Gregory, L. Izhikevich, and Z. Durumeric, "Stratosphere: Finding vulnerable cloud storage buckets," in *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2021.

[49] A. Continella, M. Polino, M. Pogliani, and S. Zanero, "There's a hole in that bucket! a large-scale analysis of misconfigured s3 buckets," in *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, 2018.

[50] (Accessed April 23, 2025) Vulnerability disclosure cheat sheet. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Vulnerability_Disclosure_Cheat_Sheet.html

[51] (Accessed September 6, 2024) Owasp web security testing guide. [Online]. Available: https://owasp.org/www-project-web-security-testing-guide/

[52] A. Cui and S. J. Stolfo, "A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan," in *Proceedings of the 26th annual computer security applications conference (ACSAC)*, 2010.

[53] Z. Durumeric, E. Wustrow, and J. A. Halderman, "ZMap: Fast internet-wide scanning and its security applications," in *22nd USENIX Security Symposium (USENIX Security)*, 2013.

[54] Z. Xu, A. Nappa, R. Baykov, G. Yang, J. Caballero, and G. Gu, "Autoprobe: Towards automatic active malicious server probing using dynamic binary analysis," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

TABLE VI
Detection results of different methods to infer cloud database names.

Method	Detection Results		Insecure Practices	
	# op	# app	# op	# app
Gemini-based Inference	3309	810	1100	390
GPT-based Inference	3103	740	1050	363
Deepseek-based Inference	2691	629	931	308
Llama-based Inference	3131	789	1053	372
Bert-based Inference	1180	565	191	176

APPENDIX A
COMPARISON OF DATABASE NAME INFERENCE METHODS

To better evaluate and explain the differences among large language models in LLM-based inference, we assess the performance of several representative models, including Gemini (gemini-pro), GPT (gpt-3.5-turbo), DeepSeek (deepseek-r1:70b), and Llama (llama3.3:70b). All models are evaluated using their default configurations. The full prompt used is shown in Table VII. Furthermore, to demonstrate the effectiveness of LLM-based methods, we also compare our approach with the BERT-based semantic search technique. The evaluation results are shown in Table VI. We can observe that all LLM-based methods outperform the BERT-based method. The semantic search approach can only infer simple database names and fails to generalize when mini-apps use customized or unusual names. Among the LLM-based methods, Gemini-based inference achieves the best performance in our task. As a result, ICREMINER adopts Gemini for database name inference.

APPENDIX B
VULNERABLE MINI-APP EXAMPLES

In this section, we randomly select 10 vulnerable mini-apps from each app-in-app cloud service and provide a detailed description in Table VIII. Due to ethical considerations, we do not provide detailed information about these mini-apps. The table illustrates the data stored in cloud databases and cloud storage, as well as the operations performed by cloud routines. Specifically, we manually analyze the mini-app code to understand the stored data in cloud databases and the detailed information of files stored in cloud storage. For cloud routines, we analyze the parameters and return values to understand their functionalities.

APPENDIX C
MEASURING THE INSECURE CLOUD RESOURCE MANAGEMENT IN OTHER SUPER-APPS

We conducted an analysis of other emerging super-apps, i.e., Line and VK, and crawled 4,324 Line mini-apps and 862 VK mini-apps for evaluation. We found that 15 Line mini-apps and 7 VK mini-apps integrate cloud services. Among them, 3 Line mini-apps and 2 VK mini-apps are vulnerable, and we have reported these issues to the corresponding developers. Since these super-app platforms host a relatively small number of mini-apps, there are not so many vulnerable mini-apps in these super-app platforms. Nevertheless, the results demonstrate that such risks also influence other super-app platforms.

TABLE VII
The prompt used for LLM-based inference.

Section	Details
Role	You are an expert in mini-app development, with in-depth knowledge of the app-in-app cloud mechanism, especially the cloud databases. Your objective is to infer relevant database names for a mini-app based on the provided function information.
Task Description	**Task: Database Name Inference** 1. Analyze the functionalities and the semantic purpose of the mini-app's functions. 2. Identify the key data entities implied in the function body. 3. Generate potential database names accordingly.
Input Format	**Mini-app Description:** {description} **Function Name:** {function_name} **Function Body:** {function_body} **Existing Database Names:** A list of existing database names of the mini-app: {existing_db_names}
Output Format	The output should be a valid Python list, such as ["xxx", "xxx", "xxx"]. Each database name should be a string with a length not exceeding 30 characters. The names should be as unique as possible and not repeated. Please only return all the database names without anything else.
Constraints	1. Only focus on generating database names for mini-apps and reject unrelated topics. 2. Do not reuse the input function name directly. 3. Output must be in the specified Python list format.

TABLE VIII
Detailed information of stored data or performed cloud operations in vulnerable mini-apps.

Cloud Database		**Cloud Storage**	**Cloud Routine**	
collection name	cloud data	file content	function name	description
userinfo	user name, gender, country, etc	user photo	payment	update user's balance after a payment is made
orders	buyer name, buyer phone number, goods list, etc	employee information	groupbill	get user information using the user ID
shop_orders	user information	student card picture	getuserinfo	get user information using the user ID
user_info	user name, phone number, password, etc	vehicle license	queryuser	get user information using the user ID
loginUser	account, company name, password, etc	medical license	getqysessionkey	get the credential session_key
admins	admin name, account, password, etc	user photo	login	login into the account (return the credential app_secret)
Employee	user name, email address, position, etc	user photo	login	login into the account (return the credential session_key)
users	country, gender, phone number, etc	user certificate	gethistory	get user's browser history using the user ID
payment	user name, card number, order information, etc	curriculum vitae	getrole	get user information using the phone number
todos	action, user name, etc	ID card picture	sendtongyipaymsg	send messages to other mini-app users

APPENDIX D
ARTIFACT APPENDIX

A. Description & Requirements

In this section, we list the information necessary to recreate the experimental setup we used to run our artifact, including the source code, hardware and software requirements.

1) How to access: The source code of the artifact, along with the required scripts, can be downloaded from: https://doi.org/10.5281/zenodo.16946146. Detailed installation and execution steps are provided in the `README` file of the repository.

2) Hardware dependencies:

- A standard Windows desktop or laptop (preferably running Windows 11).

3) Software dependencies:

- **On the Windows computer**:
  - Python (version $\geq$ 3.8) for executing automation scripts and performing static analysis.
  - Node.js (version $\geq$ 18) for running auxiliary JavaScript utilities.
  - Python dependencies can be installed using `pip install -r requirements.txt`.

4) Benchmarks: We provide the test dataset in the directory: "*./StaticAnalysis/componentAnalyze/test_dataset*".

B. Artifact Installation & Configuration

The source code of the artifact can be obtained from the repository. To prepare the environment, run the following command: `pip install -r requirements.txt`

C. Experiment Workflow

Our experiment focuses on identifying insecure practices in cloud resource management across the real-world mini-apps to assess the associated security risks. The workflow consists of the following stages: first, we apply our analysis system (ICREMINER) to detect insecure practices in mini-apps and analyze the detection results to demonstrate the prevalence of such security issues; second, we evaluate the effectiveness of different inference techniques in uncovering hidden cloud resources. Furthermore, to assess detection accuracy, we manually analyzed the mini-app code to determine whether the detection results are true positives. A result was considered a true positive only if all experts reached a consensus. In cases of disagreement, the experts were asked to discuss their labeling. Since there is no prior work addressing the same problem, we do not include comparative experiments against existing approaches.

D. Major Claims

- (C1): ICREMINER identifies a large number of vulnerable mini-apps by analyzing their cloud resource management practices, as demonstrated in Experiment (E1). The results are presented in Table IV and Table V.
- (C2): The LLM-based inference method significantly improves the effectiveness of uncovering insecure cloud operations by inferring hidden database names. This is supported by the results of Experiment (E2), as shown in Table III.

E. Evaluation

This section outlines the operational steps required to execute and evaluate the performance of our system.

1) Experiment (E1): This experiment is the core experiment of our paper, which aims to identify vulnerable mini-apps and categorize the types of insecure cloud resource management practices they exhibit. The process involves three main components executed in sequence:

[How to]

1) **Static analysis component (client-side operation extraction):** This phase analyzes the mini-app client code to identify mini-apps that integrate cloud services and extracts all cloud operations performed in the mini-app client side. It also extracts the context information related to cloud service usage, such as database names.
2) **Dynamic analysis component (exposed capability inference):** This phase performs dynamic analysis to infer the exposed capabilities in cloud operations, i.e., the cloud resources exposed to mini-app users, in a controlled environment.
3) **Security assessment component (insecure practice assessment):** Based on the results of the static and dynamic analysis, this phase identifies insecure practices in mini-apps by applying security rules tailored to different types of cloud services.

[Preparation] To prepare the environment, run the following command: `pip install -r requirements.txt`

[Execution]

1) **Static analysis component:** Execute the following commands to extract mini-apps that integrate cloud services and identify specific cloud components, along with the context information:

```
python3 cloudComponentExtractor.py
python3 mpRunner.py
```

2) **Dynamic analysis component:** Run the following scripts to infer hidden cloud database names and automatically test mini-app access to the cloud resources:

```
python3 nameInference.py
python3 autoTest.py
```

3) **Security assessment component:** Run the final script to evaluate insecure practices in different types of cloud services:

```
python3 main.py
```

[Results] The output of this experiment is stored in the directory `./SecurityAssessment/res`, which contains the detection results for insecure practices across various types of cloud services in the test dataset.

2) Experiment (E2): This experiment aims to evaluate the effectiveness of the proposed inference methods for identifying hidden cloud database names, which contribute to detecting insecure cloud resource management. The three methods under comparison are:

- Common Name Augmentation: Extend analysis using frequently used database names.
- LLM-based Inference: Utilize LLM to infer cloud database names.
- Mini-app Correlation: Correlate cloud database names across different mini-apps.

[How to] This experiment builds upon the detection results produced in Experiment E1. It analyzes which insecure practices were identified as a result of database names inferred by each of the above methods.

[Preparation] No additional preparation is required beyond the output of Experiment E1.

[Execution] Navigate to the `./SecurityAssessment` directory and run the following command:

```
python3 resultAnalyze.py
```

This script compares the contributions of each inference method by analyzing the detection results produced using their inferred database names.

[Results] The results of this experiment are output to the command line and include the detection results of different inference methods.