MiniChecker: Detecting Data Privacy Risk of Abusive Permission Request Behavior in Mini-Programs

Yin Wang
Xi'an Jiaotong University
Xi'an, China
wy0724@stu.xjtu.edu.cn

Ming Fan*
Xi'an Jiaotong University
Xi'an, China
mingfan@mail.xjtu.edu.cn

Hao Zhou
The Hong Kong Polytechnic University
Hong Kong, China
hcnzhou@polyu.edu.hk

Haijun Wang
Xi'an Jiaotong University
Xi'an, China
haijunwang@xjtu.edu.cn

Wuxia Jin
Xi'an Jiaotong University
Xi'an, China
jinwuxia@mail.xjtu.edu.cn

Jiajia Li
Ant Group
Hangzhou, China
jiajia.lijj@antgroup.com

Wenbo Chen
Ant Group
Hangzhou, China
bill.cwb@antgroup.com

Shijie Li
Ant Group
Hangzhou, China
lishijie.lsj@antgroup.com

Yu Zhang
Ant Group
Hangzhou, China
zy249870@antgroup.com

Deqiang Han
Xi'an Jiaotong University
Xi'an, China
deqhan@mail.xjtu.edu.cn

Ting Liu
Xi'an Jiaotong University
Xi'an, China
tingliu@mail.xjtu.edu.cn

ABSTRACT

The rising popularity of mini-programs deployed on super-app platforms has drawn significant attention due to their convenience. However, developers' improper handling of data permission application in mini-programs has raised concerns about non-compliance and violations. Unfortunately, existing tools lack the capability to support the construction of a universal function call graph for the mini-program and the literature lacks a comprehensive and systematic study of the abusive issues. To bridge this gap, this paper introduces an automated tool, *MiniChecker*, to uncover the abusive permission request behavior in mini-programs. It defines five primary categories of abusive issues, namely homepage pop-up, overlaying pop-up, bothering pop-up, repeating pop-up, and looping pop-up, based on the request behavior features. *MiniChecker* achieves a detection precision rate of 82.4% and a recall rate of 95.3% on our benchmark, and identifies 3,866 risky mini-programs out of 20,000 real-world mini-programs. Our analysis reveals inherent design flaws in the mini-program permission mechanism, and we have shared our findings with several mini-program platforms.

*Corresponding author.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10.
https://doi.org/10.1145/3691620.3695534

CCS CONCEPTS

• **Software and its engineering**; • **Security and privacy** → **Software security engineering**;

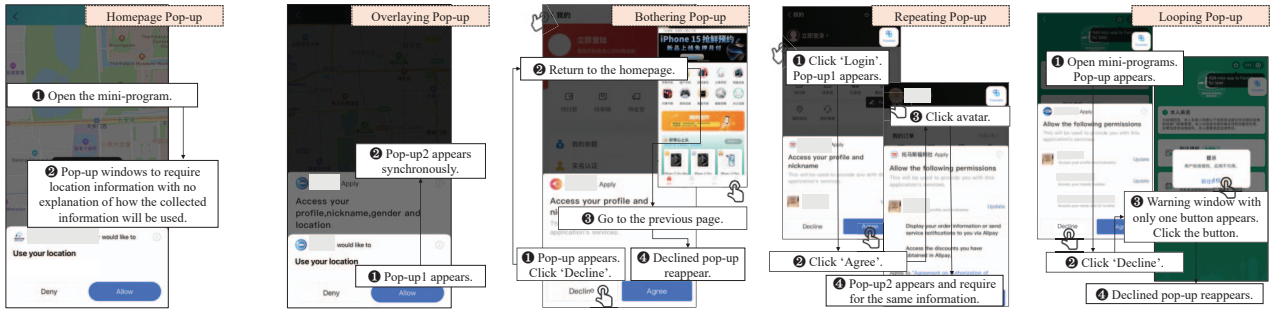
KEYWORDS

Mini-programs, data permission, privacy protection, mobile security.

ACM Reference Format:
Yin Wang, Ming Fan, Hao Zhou, Haijun Wang, Wuxia Jin, Jiajia Li, Wenbo Chen, Shijie Li, Yu Zhang, Deqiang Han, and Ting Liu. 2024. MiniChecker: Detecting Data Privacy Risk of Abusive Permission Request Behavior in Mini-Programs. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695534

1 INTRODUCTION

A new type of mobile application called "mini-program" or "mini-app"[38] is gradually gaining popularity. They are deployed on super-app platforms (e.g., WeChat, Alipay, etc.) and undergo platform review before publication. Mini-programs leverage web technologies and integrate native application capabilities[34]. In contrast with Android apps that occupy large memory space, mini-programs are small in size and do not require installations. The convenience of mini-programs has garnered significant attention from both users and developers. According to a report by Statistic[36] and QuestMobile[29], the number of mini-programs (3.8 million in WeChat and 3 million in Alipay) has surpassed the quantity of Android apps available on the Google Play Store (3.5 million).



(a) Homepage Pop-up (b) Overlaying Pop-up (c) Bothering Pop-up (d) Repeating Pop-up (e) Looping Pop-up

Figure 1: Examples of five abusive data permission request behaviors.

Permissions play a vital role as a data security management mechanism in Android apps[3, 10, 11, 42]. Similarly, mini-programs employ the "permission application" to "data acquisition" paradigm to obtain users' private personal data. For example, in Alipay, a mini-program can utilize the platform-provided API `my.getLocation` to apply for the data permission of user's location through a pop-up window. Once the user agrees, the mini-program can access various location-specific information, encompassing details such as cities, districts, countries, longitude, and latitude.

Unfortunately, developers often prioritize development convenience over user experience, frequently disregarding guidelines for requesting sensitive permissions, thereby causing misleading, forced and frequent permission requests. For example, developers may request multiple permissions simultaneously when the user first opens the mini-program, without providing clear explanations for their purposes. As a result, users are compelled to provide their personal data without fully understanding the associated risks. Additionally, some applications repeatedly request specific permissions through loop calls. If the user denies the permission request, the application becomes unable to perform subsequent functions, leaving the user with the choice to either grant authorization or exit the application.

The misuse of permission applications contravenes the principles of openness, transparency, and voluntary consent in data privacy laws. For instance, the General Data Protection Regulation (GDPR) Chapter 3, Article 12, mandates that "the controller shall take appropriate measures to provide any information and any communication relating to processing to the data subject in a concise, transparent, intelligible, and easily accessible form, using clear and plain language." Similarly, Article 5 of China's Personal Information Protection Law stipulates that "personal information shall not be processed through misleading, fraudulent, coercive, or other improper means." On the other hand, the frequently occurring data permission requests significantly impact user experience.

Research on permission security in Android apps has been extensively conducted, focusing on topics such as program behavior analysis[8, 28, 32, 33, 48], malware detection[1, 5, 7, 24, 26, 31, 43] and permission classification[4, 27, 45]. These studies utilize dynamic and static program analysis, as well as natural language processing techniques to understand program behaviors. However, there is a notable gap in research on the rationality of permission request behaviors in mini-programs.

Mini-programs present unique challenges: (*i*) They use a different language (JavaScript), file structure, and file association methods compared to Android apps, complicating the application of existing Android or traditional JavaScript program analysis tools. (*ii*) Analyzing permission usage in mini-programs requires examining the permission API calls and understanding the function call relationships within the mini-program code. While tools like TaintMini[39] and MiniTracker[21] exist for static analysis in mini-programs, they primarily focus on data flow analysis rather than function call flow graph analysis, making them unsuitable for permission request behavior analysis. (*iii*) Additionally, the design of permission mechanisms differs between Android and mini-programs. For instance, Android interprets a user's repeated denial of authorization as a permanent rejection, a feature absent in mini-programs. These new differences make the abusive permission requests in mini-programs an unexplored topic in current research.

Therefore, there is an urgent need to understand how mini-apps initiate permission requests, how they handle such behaviors with user interaction feedback, and to what extent these actions expose severe security and privacy risks. To this end, we developed a new static analysis framework *MiniChecker* for systematically and automatically detecting abusive permission requests in mini-programs. First, *MiniChecker* supplements the new call relationships in mini-programs. These include function definition of predefined objects `App` and `Page` and call relationship through JavaScript modules, XML templates, and component events, resulting in a universal function call flow graph. Then, *MiniChecker* uses a propagation method to obtain the execution sequence of the permission request behavior in the call flow graph, and divides it into active calls (i.e., asynchronous triggering with the event function) and passive calls (i.e., synchronous triggering with the lifecycle function). Finally, *MiniChecker* classifies permission request behaviors into different categories according to the relationships between different execution sequences (e.g., conditions before invocation, callbacks after invocation).

At a high level, we have classified five primary categories of abusive permission request behavior. [1] Figure 1 illustrates examples

[1]We discovered the five types of issues within three steps. Firstly, we examine the legal requirements for privacy management in mobile applications. Subsequently, we manually conducted practical tests to identify security issues in mobile applications based on these regulations. We communicated our preliminary findings to some mini-program platforms, who confirmed the issues and provided analytical support. Ultimately, we systematically summarized the characteristics of these violations, identifying five common types.

of each behavior, along with their characteristics, which are as follows:

(1) *Homepage pop-up*: This behavior refers to the automatic pop-ups when the user first opens the mini-program. In Figure 1(a), the user is asked to provide location information without any prior explanation (e.g., a privacy policy).

(2) *Overlaying pop-up*: This behavior refers to consecutive pop-ups of two different authorization windows within the same interface. In Figure 1(b), the pop-up windows overlay in the final step and the second window masks the prompt content of the first window. Users are likely to continuously click the "agree" button without carefully reading the prompt information of different authorization windows.

(3) *Bothering pop-up*: This behavior refers to the continued display of permission request windows for a permission that the user has already denied. In Figure 1(c), the mini-program does not remember the rejection and repeats the bothering pop-up when the user enters the previous page again.

(4) *Repeating pop-up*: This behavior refers to the continuous pop-ups for a specific category of permissions that the user has already granted. Different authorization APIs may apply for the same information content. In Figure 1(d), two different permission pop-ups request access to the same data (user's nickname and avatar).

(5) *Looping pop-up*: This behavior occurs when users are unable to exit an authorization page in a normal way and are forced to grant permission to exit. In Figure 1(e), after users' rejection, they become trapped on the homepage and can only engage in a continuous loop until they finally click "agree".

We have implemented a prototype of *MiniChecker*[1] and studied the current state of abusive permission request behavior by analyzing 20,000 mini-programs from Alipay platform. In particular, we have identified 168 mini-programs with homepage pop-ups, 379 with overlaying pop-ups, 65 with looping pop-ups, 322 with repeating pop-ups, and 3,866 with bothering pop-ups. Through analysis, we have identified certain design flaws within the permission mechanism of the mini-program. Subsequently, we have shared our findings with several mini-program platforms and received confirmation of our findings.

In short, we make the following contributions:

- To the best of our knowledge, we are the first to study abusive data permission request behaviors in mini-programs. In this study, we identify five primary abusive permission request behaviors.
- We develop *MiniChecker*, a systematic and automatic tool containing a suite novel static analysis techniques to build universal function call graph and detecti abusive permission request behaviors in mini-programs.
- We have evaluated *MiniChecker* on two datasets: a baseline dataset comprising 54 manually confirmed risky mini-programs, and a large-scale dataset incorporating 20,000 non-checked samples. In the first dataset, *MiniChecker* achieves a detection precision rate of 82.4% and a recall rate of 95.3%, beyond the JavaScript analysis tools TAJS[19], Approximate CG[9], and CodeQL[12]. In the second dataset, *MiniChecker* identifies 3,866 risky mini-programs with potential abusive permission request behaviors.

[1]https://github.com/xjtu-intsoft/MiniChecker

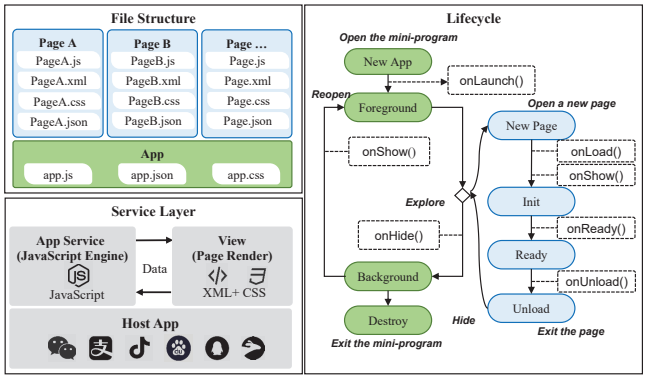


Figure 2: Running mechanism of mini-programs

2 BACKGROUND

2.1 Mechanism of Mini-programs

2.1.1 Running Mechanism. A mini-program consists of a two-level file structure: the app level and the page level[15], which is shown in Figure 2. The app level provides an overview of the entire application, while the page level describes individual pages within the program.

In the app level, there are three files: (*i*) `app.js` is used to define the overall application logic, allowing globally referenced object `App`; (*ii*) `app.json` is used to specify global settings, including a list of all page directories and plugin information; (*iii*) `app.css` is used to define global styles, providing globally referenced styles.

In the page level, there are four files: (*i*) `JavaScript` files are written to describe the logic of each page. Custom objects and methods are used to meet development requirements, such as utilizing the platform predefined object `Page` to represent mini-program pages or using custom APIs to obtain permissions and data; (*ii*) `XML` files contain the page's layout structure and event information, which can be combined with component systems and event systems to construct the page's user interface; (*iii*) `CSS` files define component styles that are combined with XML files for page rendering; (*iv*) `JSON` files declare configuration information for the page, such as the default style of the navigation bar.

The lifecycle of a mini-program is determined by its app instance. As depicted in Figure 2, the mini-program initiates through `onLaunch` in `app.js`, then transitions between the foreground and background platforms based on user interactions, and eventually terminates. Throughout the execution of the mini-program, multiple page lifecycles are traversed. The traversal procedure is implemented through the routing mechanism, which is integrated with the lifecycle mechanism to facilitate navigation between different pages. A new page is loaded using `onLoad` and `onShow` in `{pageName}.js`, and it exits through `onUnload`.

2.1.2 Permission Mechanism. Mini-programs offer various data interfaces that developers can utilize, such as location, address, and bluetooth[13]. Table 1 provides a categorization of different data permission interfaces within the Alipay platform. Depending on the source of permission data, these interface types can be categorized

Table 1: Permission request interface

Source	Information Type	Data Permission API	Implementation[1]
Device	Location	my.getLocation, my.openCarSevice, my.startContinuousLocation	①
	Camera	my.scan, cameraContext.takePhoto, cameraFrameListener.start	①
	Album	my.chooseImage, my.chooseVideo, my.saveImage, my.saveImageToPhotosAlbum, my.saveVideoToPhotosAlbum	①
	Recorder	my.startRecord, my.stopRecord, my.cancelRecord, RecorderManager.start, RecorderManager.stop	①
	Bluetooth	my.connectBLEDevice, my.openBluetoothAdapter, my.getBeacons	①
	Contact	my.choosePhoneContact, my.chooseAlipayContact, my.chooseContact	①
	Clipboard	my.getClipboard	①
	Carrier	my.getCarrierName	①
Platform	Basic Information[2]	my.getAuthCode, my.getOpenUserInfo, my.getPhoneNumber	①②

1. The implementation methods include: ① API invocation; ② Button click.

2.Basic information refers to the information stored on the platform server, including avatar, nickname, phone number, region, gender, date of birth, user ID, access token, etc, bound to the customer's account of super app.

as follows: (*i*) **Device**: This category involves obtaining data from the device's native layer through JSBridge. For instance, when a developer invokes `my.chooseImage` in a mini-program, the super-app utilizes JSBridge to execute JavaScript API `chooseImage` and access the device's photo album data. (*ii*) **Platform**: This category involves obtaining data from the platform's server, which is uploaded by users. For example, when a developer calls `my.getAuthCode` in a mini-program, the platform returns the user's unique authorization code on the server. Developers can subsequently use it to retrieve user ID, nickname, avatar, and other information.

Mini-programs offer two options for developers to trigger these interfaces: (*i*) **API invocation**: Developers can invoke permission request interfaces in JS files and utilize callback functions to handle subsequent actions after successful or failed permission requests. (*ii*) **Button clicks**: Developers can define the "open-type" attribute for a button in XML files. Depending on the attribute's value, different permission requests will be executed when the button is clicked[14].

Mini-programs have three behaviors for these interface triggering based on user actions: (*i*) **Not authorized**: When a user has not granted a specific permission, if the data permission API is invoked, the mini-program will pop up a window requesting authorization. The developers can access the interface's returned data after the user clicks "agree". (*ii*) **Authorized**: Once a user has already granted the permission, the mini-program will not display the permission window again when invoking the same API. It retrieves the data directly. (*iii*) **Denied**: If a user denies a specific permission and has selected the "Don't show again" option, the mini-program will directly return a failure message when the interface is called again. If the user has not chosen that option, the permission window will be displayed again.

2.2 Motivating Example

2.2.1 Examples. In Figure 3, we present a motivating example that abstracts from the real-world mini-program. This mini-program exhibits three abuse issues: (*i*) Upon the user's first opening of the mini-program, it requests the nickname and avatar without any user interaction, constituting a *homepage pop-up*. (*ii*) On the "home" interface, when the user clicks the avatar, the mini-program prompts for the user's nickname and avatar information again, leading to a *repeating pop-up*. (*iii*) If the user agrees to the authorization in the first instance, the mini-program navigates to the "location" page and requests the user's location information. If the user refuses, the mini-program restarts this page until the user consents, resulting in a *looping pop-up*. These behaviors indicate that the mini-program forcibly or semi-forcibly collects user information, negatively impacting user experience and posing a potential violation risk.

From the perspective of code execution, the sequence of these abuse issues is as follows: When the user launches the mini-program, it first loads the `app.js` file and executes the `onLaunch` lifecycle function within the `App` object. The permission request API call `my.getAuthCode` is defined within the scope of `handleAuth` (step ❶) and is executed when `handleAuth` is called (step ❷). Thus, `my.getAuthCode` is automatically executed when `onLaunch` starts (step ❸). Next, the mini-program loads the first page defined in the configuration file `app.json`, located at `pages/home`. All components on this page are defined in `pages/home/index.axml`. The avatar and username components are linked to `avatar.axml` and `userName.axml` respectively through template references. These components contain event functions (e.g., `onGetAuthorize`), which create event associations with the corresponding JavaScript file. When the avatar button is clicked, the `my.getOpenUserInfo` function defined in `onGetAuthorize` (step ❹) is executed. Similarly, the `my.getAuthCode` function defined in `goToMyInfoPage` (step ❺) is also executed, resulting in the user's avatar and nickname being collected a second time. After the user agrees to the authorization, the mini-program navigates to the `pages/location/index.axml` page via `my.navigateTo`. Here, it requests permission to obtain the user's location information by calling `my.getLocation` through a sequence of function return (step ❻), module export (step ❼), module import (step ❽), and the global function `getApp` (step ❾). If the user refuses to provide location information, the `handleLocation` function defined in `onGetLocation` (step ❿) is executed repeatedly due to recursion (step ⓫), causing the page to restart through the route API `my.reLaunch`.

2.2.2 Challenges. Based on the analysis of our motivating example, two key challenges must be addressed to develop *MiniChecker*.

How to universally identify the execution sequence associated with permission requests in a mini-program. While call graph analysis techniques have been well-studied in traditional JavaScript (e.g., TAJS[19], JSWALA[41], and Approximate CG[9]), they face challenges in generating complete function call flow graphs for current mini-programs. These challenges arise because some tools can only analyze individual JavaScript files, making it

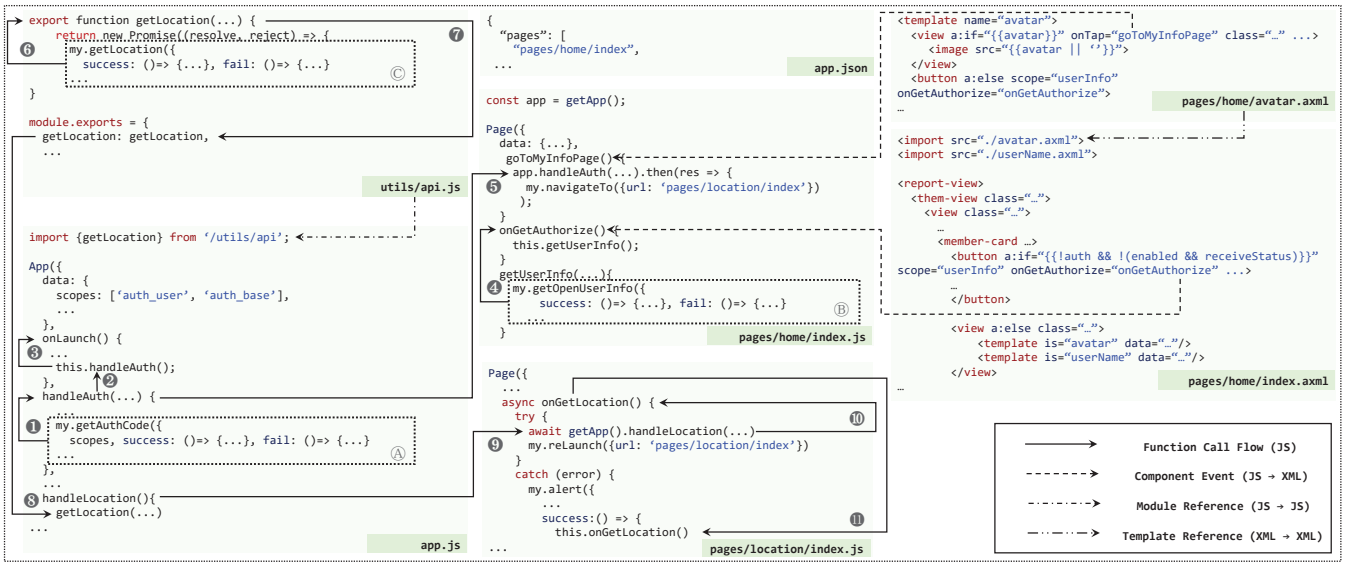


Figure 3: A simplified example of a risk mini-program containing homepage pop-up, repeating pop-up, and looping pop-up.

difficult to resolve inter-module call associations (e.g., step ❼ and ❽). Additionally, mini-programs use predefined object bodies, such as App and Page (e.g., step ❶ and ❸). There are also global functions, like getApp, and elements such as component associations and template references, which complicate the analysis.

How to recognize different abusive behaviors from the execution sequence. After obtaining the execution sequence, it is complicated to classify the execution sequence into different types of abuse issues. For example, for the homepage pop-up, we need to determine whether the execution location of the permission application is on the homepage, which is related to the lifecycle of the mini-program. For the looping pop-up, in addition to the loop or recursion of the function execution, we also need to pay attention to whether the mini-program forms a loop during the page jump process, which determines whether the user will repeatedly enter the same page to perform the same permission operation. Therefore, simply extracting the execution sequence is not enough to achieve the classification of abuse issues, and program lifecycle and user interaction need to be considered.

3 APPROACH

This section provides the detailed design of *MiniChecker*. Its high-level approach is to build a universal function call graph (UFCG) and scan the graph to detect abusive permission request behaviors. The analysis unfolds in three stages: universal function call graph extraction, behavior propagation and risk classification. Firstly, we extract the nodes related to function calls from the source code of a mini-program and analyze the dependency relationships between functions and files to constructing UFCG. Then, nodes involving permission requests are marked and propagated. The final call are classified into active calls or passive calls based on their trigger timings. Finally, *MiniChecker* further analyzes conditions, page routes and callbacks within the call sequence, completing an accurate risk classification by examining the preceding and succeeding features of the execution sequence.

3.1 Universal Function Call Graph Extraction

Initially, we build the function call graph in JavaScript files. The call graph is denoted as $G = (N, E)$, where N represents function call nodes in the code, and E represents the call relationships between functions. We construct G by identifying function definitions and call statements in the JavaScript code and considering them as nodes (i.e., function declaration, function expression, arrow function expression, function call, and method call). If two functions are found to have encapsulation and call relationships, we establish an edge from the child function node to the parent function node.

Then, we address the discontinuous call flows and fragmented information caused by the unique characteristics of the mini-program:

(1) **Functions in Predetermined Objects.** The mini-program provides the App and Page predetermined objects to execute the program logic. Global functions and event handlers are defined by "key-value" methods within these objects. Functions in these objects could reference each other by JavaScript keyword *this* (e.g., step ❷ in Figure 3). Moreover, these predetermined objects enable mini-programs to access globally defined App objects and functions available on other pages (e.g., step ❺ and ❾ in Figure 3).

MiniChecker first analyzes the parameter values of the node functions named App and Page. For "key-value" pairs within these objects, we add the values in the form of function expressions as nodes to the graph G. Subsequently, we analyze the intraprocedural data flow within each function definition node (N_{def}). For nodes that involve invocation with keyword *this* (N_{this_invk}) , we try to locate the outer App or Page object according to the scope hierarchy and add an edge in G ($N_{def} \rightarrow N_{this\_invk}$). Finally, we identify the call nodes of getApp (N_{getapp_invk}), whether they are direct calls or invoked through value, and perform a simple pointer attribute analysis. If the attribute accessed after the getApp call is a function definition node (N_{def}) in the getApp object, we establish edges between them ($N_{def} \rightarrow N_{getapp\_invk}$).

(2) **Module Functions across Files.**: The mini-programs adhere to the ECMA2015 standard[18] and supports module references

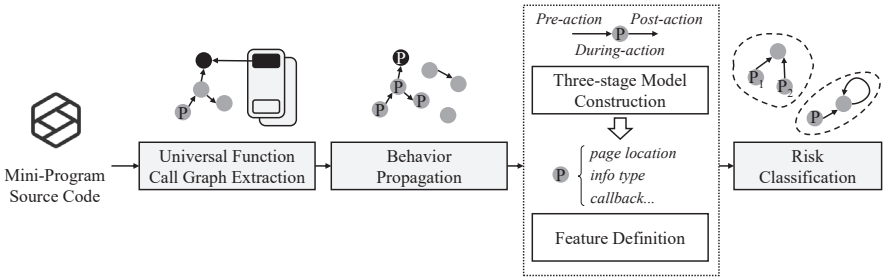


Figure 4: Architecture of *MiniChecker*

across files. Consequently, we analyze the module's export statements (i.e., `export`, step ❼ in Figure 3) and import statements (i.e., `import`, step ❽ in Figure 3), and add edges between definition nodes N_{exp_def} and invocation nodes N_{imp_invk} ($N_{exp\_def} \rightarrow N_{imp\_invk}$). Some mini-programs use default exports (e.g., `export default`) or aggregated export methods (e.g., `export {...}`), allowing import statements to omit specific function names. For unnamed exports, if there is a module reference relationship between two files, and the function name at call node $N_{unnamed\_invk}$ matches the definition node $N_{unnamed\_def}$, we create edges between the corresponding nodes ($N_{unnamed\_def} \rightarrow N_{unnamed\_invk}$). The analysis of module function calls enhances *MiniChecker*'s capability for multi-file code analysis in mini-programs.

(3) **Event Functions in Templates.** We thoroughly examine the locations where permission requests may be triggered. Besides API calls, mini-programs also request permissions through event functions. To completely identify these event functions, *MiniChecker* not only builds event function nodes through event attribute correlations but also considers cross-file event information. Mini-program layout files support the `template` feature, enabling the instantiation of certain page template files during the loading process (e.g., `avatar.axml` in Figure 3). However, existing static analyses of mini-programs [21, 39] have overlooked this feature.

MiniChecker begins by parsing all `XML` files. For nodes associated with events—components with "on" or "catch" keyword properties (e.g., `onTap`)—it marks the associated function node N_{event_def} as an event trigger. Next, it parses the `XML` file associated with the `template` tag. If there is an association between the template layout XML_{temp} and the instance layout XML_{inst}, *MiniChecker* searches the event function definition N_{event_def} in the corresponding JavaScript file of XML_{inst} for the event attribute in XML_{temp} and marks N_{event_def} as an event trigger.

3.2 Behavior Propagation

Having described the approach of UFCG extraction, we now explain how *MiniChecker* tracks permission request behavior. We introduce Algorithm 1, a behavior propagation method designed to locate permission request behavior resulting from encapsulation.

Initially, we mark the data permission request API calls as the initial labeled nodes P within G (initialize and initial mark, Line 1 - 5). Subsequently, correlation analysis is performed on all nodes within the call flow graph (propagate request behavior, Line 6 - 20). If a node exhibits a connectivity relationship with any initial labeled node P, it will also be assigned a label P. Ultimately, should a node at the culmination of the behavior propagation correspond to an

Algorithm 1 Behavior Propagation

Require: Function call graph G=(N,E), data permission request API list L
1: **for** n in G **do** ▷ Initialize
2: **if** $n.API$ in L **then**
3: $n.mark \leftarrow True$ ▷ Initial mark
4: **end if**
5: **end for**
6: $waitingNode, visitedNode = [\,], [\,]$ ▷ Propagate request behavior
7: **for** n in G **do**
8: **if** $n.mark = True$ **then**
9: $waitingNode \leftarrow n.getConnectedNodes()$
10: $visitedNode \leftarrow n$
11: **while** $waitingNode \neq \emptyset$ **do**
12: $m \leftarrow waitingNode.pop()$
13: **if** m not in $visitedNode$ **then**
14: $m.mark \leftarrow True$ ▷ Propagated mark
15: $waitingNode \leftarrow m.getConnectedNodes()$
16: $visitedNode \leftarrow m$
17: **end if**
18: **end while**
19: **end if**
20: **end for**
21: **for** n in G **do** ▷ Trigger timing division
22: **if** $n.isEndPoint$ and $n.isEventFunction$ **then**
23: $n.triggerTiming \leftarrow$ "$ActiveEventTrigger(AET)$"
24: **end if**
25: **if** $n.isEndPoint$ and $n.isLifecycleFunction$ **then**
26: $n.triggerTiming \leftarrow$ "$PassiveLifeCycleTrigger(PLT)$"
27: **end if**
28: **end for**

event function or a lifecycle function (Section 2.1), we designate it as "active event trigger" (AET) or "passive lifecycle trigger" (PLT), which represents the timing of risk behavior activation (trigger timing division, Line 21 - 28). Determining the triggering timing is a fundamental feature of abuse issue classification, which will be further explained in Section 3.3.

For example, in Figure 3, `my.getAuthCode` and `handleAuth` in `app.js` have an encapsulation relationship. `my.getAuthCode` is marked at first. During the propagation process, `handleAuth` is also marked subsequently, and `onLaunch` is finally marked. Since `onLaunch` serves as a lifecycle function, it is automatically triggered when the mini-program initiates, thus this node is marked as PLT. Simultaneously, `onGetAuthorize` in `pages/home/index.js` is categorized as AET.

3.3 Risk Classification

To achieve accurate risk classification, *MiniChecker* incorporates behavioral features that influence the execution sequence. In this section, we first extract condition, interaction, and callback information using a three-stage model, then identify features associated with five abuse issues, and finally classify risks.

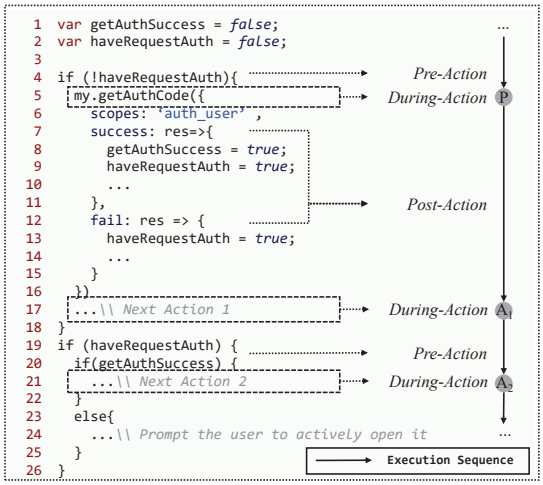


Figure 5: Three-stage model decomposition example. The execution node of mini-program API is denoted as A (or P for permission request API). The execution conditions, execution state, and subsequent callbacks can be categorized respectively as *pre-action*, *during-action*, and *post-action*.

3.3.1 Three-stage Model Construction. We first propose a three-stage model to consistently represent the various states of mini-programs' actions. In Figure 5, we illustrate our disassembly process using a code fragment with complete three-stage portions as an example. In cases where certain code portions are missing, it may cause abuse issues.

(1) **Pre-action**: This stage denotes the period preceding the behavior. The state during this stage is associated with conditions and branches. For instance, in Line 4 of the code fragment, the presence of branches (`!haveRequestAuth`) before `my.getAuthCode` indicates the indirect execution of this action, which is referred to as a "condition exists" (*ce*).

(2) **During-action**: This stage corresponds to the moment of the program behavior (e.g., the appearance of the pop-up window). The state during this stage depends on interactions within the mini-program. In this context, our focal interactions encompass platform-provided API invocations and user events. Platform-provided APIs related to user interaction encompass data permission APIs (Table 1), navigation APIs (i.e., `my.switchTo()`, `my.navigateTo()`, `my.navigateBack()`, `my.redirectTo()`, and `my.reLaunch()`), and the alarm API (i.e., `my.alarm()`).

We employ the concept of dominance from program compilation to ascertain the execution order of two behaviors. The considered dominance relationships are categorized based on the following rules: (*i*) For platform-provided API (corresponding to PLT), if there are no branches and conditions between the preceding and subsequent actions, as seen between `my.getAuthCode` and `NextAction1`, the preceding action is considered to "dominate" (*domi*) the subsequent action. (*ii*) For user events (corresponding to AET), considering that events always execute after the initiation lifecycle function (i.e., `onLoad()`, `onShow()`, `onReady()`, and `onLaunch()`), and the order of event triggering relies on user actions and is arbitrary, we assert that the initiation lifecycle function "dominates" any event function, and there is a mutual domination relationship among independent events. (*iii*) For interactions between different pages, we extract the parameters of the routing API and construct a routing flow graph to represent the navigational links between pages. For each behavior node, if there is a navigational link on the page it resides on, the behavior node preceding the navigation is considered to "dominate" the behavior node following the navigation.

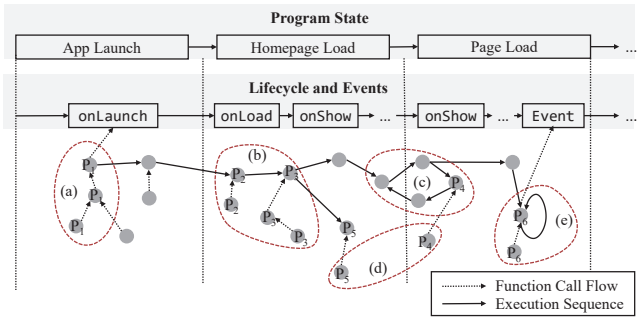


Figure 6: Classification examples.

(3) **Post-action**: This stage refers to the period subsequent to the execution of the behavior and the receipt of user feedback. The state during this stage is dependent on whether the user agrees or rejects the request, and it is associated with the callback function of the request call. For example, in Line 7 to Line 15, both the success callback and fail callback influence the variable `haveRequestAuth`, which in turn affects the next trigger of this API. Therefore, if the callback function impacts the pre-action branches, it is referred to as a "condition update" (*cu*).

3.3.2 Feature Definition. Based on the distinct characteristics observed in the three stages, the features and patterns of various abusive permission request behaviors can be defined. We use the following notations: (*i*) P represents the invocation of a specific permission request behavior (during-action); (*ii*) $P.loc$ indicates the location of the request in the page file; (*iii*) $P.info$ denotes the type of user information collected by the request (categorized in Table 1); (*iv*) $P.ce$ signifies the presence of a conditional branch before the action (pre-action); (*v*) $P.fcbe$ refers to the presence of a failure callback function after the action (post-action); (*vi*) $P.cu$ indicates an update in the condition within the callback function after the action (post-action). According to the definitions proposed in the Introduction (Section 1), we define each pattern as follows:

(1) *Homepage pop-up*: This pattern occurs when there exists a P_i satisfies that $P_i.loc$ is `app.js` or the first page defined in `app.json`.

(2) *Overlaying pop-up*: This pattern occurs when there exists different P_i and P_j such that $P_i.loc = P_j.loc$, and $P_i.domi(P_j)$.

(3) *Bothering pop-up*: This pattern occurs when there exists a P_i that does not satisfy $P_i.ce$ & $P_i.fcbe$ & $P_i.cu$.

(4) *Repeating pop-up*: This pattern occurs when there exists different P_i and P_j such that $P_i.info \cap P_j.info \neq \emptyset$, and $P_i.domi(P_j)$.

(5) *Looping pop-up*: This pattern occurs when there exists a P_i that satisfies $P_i.domi(P_i)$.

3.3.3 Examples. In this section, we give five risk classification examples, as shown in Figure 6. *MiniChecker* uses the features obtained from the three-stage decomposition to match risks on the function call flow graph after propagation. The matching process for each example is as follows:

(1) *Homepage pop-up*: *MiniChecker* checks the lifecycle function related to the homepage to detect homepage pop-up issues. As shown in Figure 6(a), the permission request P_1, after propagation, connects to the lifecycle function of the `app.js`, indicating that P_1 may be executed when the mini-program starts.

(2) *Overlaying pop-up*: *MiniChecker* checks the node connection relationships of different permission requests to detect overlaying pop-up issues. As shown in Figure 6(b), for P_2 and P_3, after propagation, they are directly connected in the execution sequence, indicating that P_2 and P_3 have potential continuous execution risks.

(3) *Bothering pop-up*: *MiniChecker* checks the conditions and callbacks before and after the execution sequence of the permission request to check for bothering pop-up issues. As shown in Figure 6(c), for P_4, there is a possibility of cyclic execution. If the execution sequence does not record the execution of P_4 (i.e., no write operations are performed on pre-action variables during the post-action phase), it will cause bothering pop-up issues of P_4.

(4) *Repeating pop-up*: *MiniChecker* checks the information scope of different permission requests to detect repeating pop-up issues. As shown in Figure 6(d), when calling P_4 and P_5, if the types of information requested by P_4 and P_5 overlap, and their executions do not affect each other, it will result in repeating pop-up issues.

(5) *Looping pop-up*: *MiniChecker* checks the execution sequence of the permission request for loops without exits to detect looping pop-up issues. As shown in Figure 6(e), P_6 permissions exhibit recursion during execution, which is considered as a looping pop-up issue.

4 EVALUATION

In this section, we first introduce the study setup of our experiment, including the constructed dataset and the running environment. Then, we evaluate our method by answering the following three research questions.

RQ1: What is the effectiveness of our approach in the detection of abusive issues?

RQ2: How much improvement can our method achieve in risk detection compared to traditional JavaScript analysis tools?

RQ3: To what extent are abusive issues prevalent in mini-programs in large-scale real-world environments?

4.1 Study Setup

4.1.1 Benchmark Dataset. Given the absence of an existing dataset pertaining to abusive permission request behavior research in mini-programs, we undertook the task of constructing our benchmark dataset. With the help of the Alipay platform, we obtained a batch of unchecked mini-program samples. We first ran these mini-programs dynamically on real devices to check for the presence of five abusive issues. Then we examined the code to filter out those with severe obfuscation, and finally identified the code location of the abusive permission request. We analyzed 1,000 randomly selected mini-program samples based on their `appIDs`. Three students with programming and code analysis experience conducted the test process. During the dynamic testing phase, each mini-program sample was used by a tester for 5 minutes to achieve as high coverage as much as possible. Risky samples were confirmed again by the remaining two testers to ensure accurate analysis and determine the exact location of the risk-generating code.

When constructing the benchmark dataset, we observed that approximately 60% of the mini-programs contained obfuscated code, often due to the use of libraries like `webpack` for code packaging. Obfuscation is typically employed to protect the mini-program or facilitate multi-platform reuse. However, it complicates function execution and disrupts the AST structure of the code, making it difficult to analyze according to standard mini-program or JavaScript formats. We have removed unsuitable samples and plan to address the challenge caused by obfuscation in future work.

In conclusion, we compiled a benchmark dataset consisting of 54 mini-program samples with 128 distinct risk behaviors. Specifically, there are 30, 19, 54, 9 and 16 mini-programs exhibiting homepage pop-up, overlaying pop-up, bothering pop-up, repeating pop-up and looping pop-up respectively. These mini-programs represent six different types: shop, beauty, express delivery, government, hospital, and service. Among these, some are newly launched with version numbers starting at 0, while others have undergone multiple updates and maintenance. The dataset includes popular mini-programs published by governments or renowned brands and developed by professional developers. For instance, the Shanghai Dongfang Hospital mini-program, developed by a medical institution, has over 7,000 monthly users. Similarly, the Arabica mini-program, published by a popular coffee brand, has a monthly user base of 1,000. We also included mini-programs developed by individual developers that have relatively few users.

4.1.2 Environment. In terms of software, *MiniChecker* is implemented atop *CodeFuse-Query*[16]. *CodeFuse-Query* is a code querying tool designed to construct a database of program information (e.g., AST, ASG, Class Hierarchies, Documentation), facilitating queries for diverse programming language information (e.g., Java, JavaScript, XML). It adopts a data-driven and standardized model for code queries, resulting in a faster computational speed than SQL. In terms of hardware, we conducted the experimental analysis using a MacBook Pro, which is equipped with a six-core Intel Core i7 processor and 16GB of memory.

4.2 Effectiveness of Detecting Risk Behaviors

4.2.1 Benchmark Experiment. We applied *MiniChecker* to detect these risk behaviors, and the detection results are presented in Table 3. The columns **Lab.** represent the presence of a risk in code for five abusive issues (• means presence while blank indicates absence) and the columns **Mc.** represent the detection results of the risks in code (✓represents presence while blank indicates absence). We discuss the evaluation results as follows:

(1) In our analysis of 128 confirmed risk behaviors, we identified a total of 148 risky behaviors. Of these, 122 were accurately detected (True Positive, TP), 26 were mistakenly labeled as risky (False Positive, FP), and 6 went undetected (False Negative, FN). Our method achieves a detection precision rate of 82.4% (122/148) and a recall rate of 95.3% (122/128). *MiniChecker* proves effective in identifying risk behaviors within the samples.

(2) The false alarm rate of *MiniChecker* is 17.6% (26/148). False positives predominantly involve homepage, overlaying and looping pop-ups. The primary reason for false positives is the presence of

Table 2: Evaluation of *MiniChecker*

Mini-Program Name	Homepage		Overlaying		Bothering		Repeating		Looping	
	Lab.	Mc.	Lab.	Mc.	Lab.	Mc.	Lab.	Mc.	Lab.	Mc.
Youbei Rent		✓		✓	•	✓				
Haiqitong	•				•	✓				
Luquan Jinpai Image Design	•	✓	•	✓	•	✓				
Yumantang					•	✓				
Kuaile Rocket					•	✓			•	✓
Douyou Share	•	✓	•	✓	•	✓			•	✓
Black Dragonfly	•	✓	•	✓	•	✓				
Air Legal Consult					•	✓				
Jinyang Lake Bicycle	•	✓	•	✓	•	✓				
Baofeng Charge	•	✓	•	✓	•	✓				
Yunchongba Station		✓	•	✓	•	✓				✓
Love Sports Welfare		✓		✓	•	✓	•	✓		
Oze Hardware	•	✓	•	✓	•	✓				
Thomas Welfare		✓		✓	•	✓	•	✓		
Xuzhou Health Report	•	✓	•	✓	•	✓				
Hebei Tang Hosipital					•	✓		✓		
Jilin Baixin Shoes					•	✓			•	✓
Tianquantailai Market	•	✓			•	✓	•	✓		
Xiangxie Priority Living	•	✓			•	✓	•	✓		
Anyue Charge	•	✓	•	✓	•	✓			•	✓
Shentong Express		✓	•	✓	•	✓				
Tongwei TCM Hospital					•	✓				
Naizhi					•	✓				
Jinqian Express Query	•	✓		✓	•	✓				✓
Mengyin Payment					•	✓			•	✓
Hefeng Power	•	✓	•	✓	•	✓				
Yellow Duck Choice					•	✓				
Zhangqing Third Hospital	•	✓			•	✓			•	✓
Jilin Huili Plan Company					•	✓			•	✓
Shannxi Filling	•	✓	•	✓	•	✓			•	✓
Sichuan Miliang Market	•	✓			•	✓	•			✓
Yanwu		✓			•	✓				
Shanghai Dongfang Hosiptal					•	✓			•	✓
Shangcheng Go		✓			•	✓				
Nanjing Walk	•	✓	•	✓	•	✓				
Dianxiaoya Charge	•	✓	•	✓	•	✓			•	✓
Xiangguan					•	✓				
Xingqishu					•	✓				
Arabica	•		•		•	✓				
Tianzhu Peoples Hospital					•	✓				
Liuan Hospital		✓			•	✓			•	
Shengzhou Hospital					•	✓			•	✓
Lihao Welfare	•	✓		✓	•	✓	•	✓		✓
Tehui Buy	•	✓		✓	•	✓	•	✓		✓
Lidani	•	✓		✓	•	✓	•	✓		✓
Leshan Shop	•	✓			•	✓	•			✓
Shunfayi	•	✓	•	✓	•	✓			•	✓
Xiaobo Care	•	✓	•	✓	•	✓			•	✓
Yilian Hair	•	✓	•	✓	•	✓			•	✓
Yujin Beauty	•	✓	•	✓	•	✓			•	✓
Zhai Phone Rent	•	✓			•	✓				
Zhiyi Beauty		✓		✓	•	✓				✓
Tuchuang Network	•	✓			•	✓				
Najing Deda Copier	•	✓			•	✓				
Total	30	28/9/2	19	18/8/1	54	54/0/0	9	7/1/2	16	15/8/1

Lab. represents the label of a risk in code (• means presence while blank indicates absence), and **Mc.** represents the *MiniChecker*'s detection result of the risk in code (✓represents presence while blank indicates absence). **Total** represents the total number of statistical results. In child columns of **Mc.**, three numbers represent the number of **TP**, **FP**, and **FN** samples, respectively.

dynamic conditions and branches in the triggering path of risky behaviors, posing challenges for accurate determination through static methods. For example, the API `my.request` is commonly used for network communication in mini-programs, which determines whether to execute subsequent actions based on its return results. However, static analysis methods cannot capture the dynamic return values (e.g., the status code), making it impossible to ascertain if these subsequent actions will be performed. *MiniChecker* considers all possible executions, which sometimes results in non-triggered behaviors being misidentified as risky behaviors.

Table 3: Comparative Experiment with TAJS, JSWALA, Approximate CG and CodeQL.

Tool	Homepage		Overlaying		Bothering		Repeating		Looping		Average	
	Pre.	Rec.	Pre.	Rec.	Pre.	Rec.	Pre.	Rec.	Pre.	Rec.	Pre.	Rec.
TAJS	-	0	-	0	-	0	-	0	-	0	-	0
Approximate CG	-	0	-	0	**1.0**	0.30	-	0	-	0	**1.0**	0.13
CodeQL	**0.86**	0.26	-	0	**1.0**	0.98	-	0	-	0	0.98	0.49
MiniChecker	0.76	**0.93**	**0.69**	**0.95**	**1.0**	**1.0**	**0.875**	**0.78**	**0.65**	**0.94**	0.82	**0.95**

In child columns for each risk, **Pre.** represents precision and **Rec.** represents recall. The symbol "-" indicates that no risk samples have been detected, therefore the precision cannot be calculated.

(3) *MiniChecker* exhibits an underreporting rate of 4.7% (6/128). The missed report is attributed to the utilization of dynamically loaded third-party libraries. For instance, the third-party library `antmove` provides an API packaging interface designed to facilitate the portability of mini-programs across various platforms. This involves substituting the API prefix specific to each platform with a standardized API prefix, thereby facilitating platform migration with minimal code modifications (e.g., obtain `my.getAuthCode` by calling `require("antmove/api")(...).getAuthCode`). However, this approach replaces the original API name, resulting in the inability to locate the API for subsequent risk detection.

(4) We measured the time required for *MiniChecker* to analyze the samples in the benchmark dataset. Each mini-program took an average of 104.2 seconds. The primary factors influencing *MiniChecker*'s processing time are graph construction and graph operations. Specifically, data preprocessing (i.e., building the UFCG and completing behavior propagation) took the longest time, averaging 72.0 seconds. Calculating the looping pop-up required 32.1 seconds on average. Other risk queries each took less than 0.1 seconds on average.

Answer to RQ1: *MiniChecker* proves effective in identifying risk behaviors within the benchmark, achieving a detection precision rate of 82.4% (122/148) and a recall rate of 95.3% (122/128).

4.2.2 Comparative Experiment. Although there exist several static analysis tools for mini-programs, such as TaintMini and MiniTracker, these tools primarily focus on data flow analysis rather than call flow graph analysis. TaintMini employs the concept of event groups to handle the correlation between different functions. However, it does not differentiate between event types, such as lifecycle events and user events. On the other hand, MiniTracker utilizes AliasMap to handle function calls and directly conducts data flow analysis based on the AST, while also disregarding the execution order between different functions. Neither tool offers an interface for constructing call flow graph.

To evaluate the effectiveness of *MiniChecker* in constructing UFCG, we compared it with several popular JavaScript call flow graph analysis tools: TAJS[19], Approximate CG[9], and CodeQL[12]. TAJS is a data flow analysis tool for type inference and call flow graph generation and has frequently served as a benchmark in JavaScript static analysis research [2, 20]. Approximate CG implements a field-based call graph construction algorithm, supporting more JavaScript features (e.g., ES6 and Vue) compared to TAJS. CodeQL, released by GitHub, is a code review tool which converts semantic code information into a database and allows users to customize queries for finding vulnerabilities and other issues. CodeQL's JavaScript module supports flow graph query. TaintMini[39] and MiniTracker[21] were excluded from our comparison because

they focus on data flow analysis rather than function call flow graph analysis. We utilized the tools to analyze the mini-program code in the benchmark, generating a call flow graph. Then call flow graphs were then employed to detect risks according to our risk detection method. Based on our comparison, we have drawn the following conclusions:

(1) TAJS cannot effectively analyze mini-program code due to the extensive use of ES6 features, such as arrow functions and spread operators, which cause parsing errors. Although we attempted to use the `Babel` library to transpile the code, the converted code remained unanalyzable by TAJS.

(2) Approximate CG achieved 100% accuracy in detecting bothering pop-up issues, but only for this specific risk. It encountered difficulties with key-value function definitions and references within object bodies. While Approximate CG can associate APIs with predefined functions such as `App` or `Page`, it failed to accurately capture function call information within object-defined functions. Consequently, it cannot determine the call relationships within the lifecycle, leading to an inability to accurately detect other abusive issues.

(3) CodeQL exhibited a higher detection accuracy for homepage pop-ups due to its ability to identify more function points, including anonymous functions, which *MiniChecker* does not account for. Given that anonymous functions significantly increase the complexity of graph calculations, *MiniChecker* limits its analysis to named functions only. However, CodeQL was unable to associate layout XML files, which prevented it from identifying event functions. Consequently, it cannot detect issues such as loop pop-ups that require user interaction.

Answer to RQ2: Compared to traditional JavaScript tools TAJS, Approximate CG and CodeQL, *MiniChecker* achieves the highest recall in abusive permission request behavior detection.

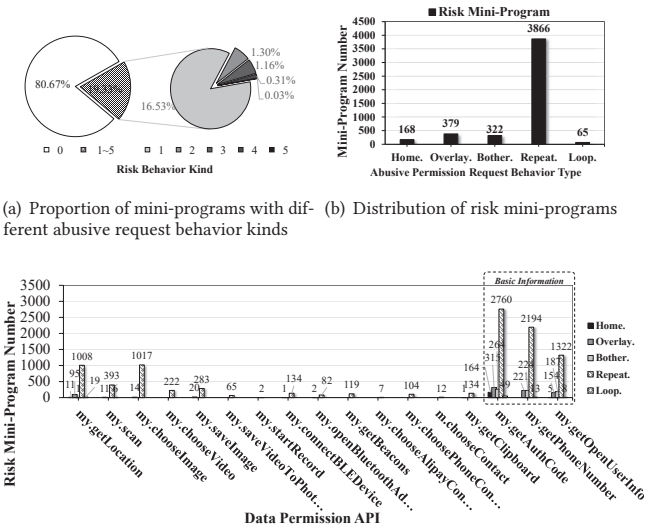
4.3 Risks in Real-world Environment

We analyzed 20,000 mini-programs from the real-world environment and successfully obtained analysis results for 3,866 mini-programs exhibiting risky behavior. The analysis results are shown in Figure 7. Through the results, we have the following conclusions:

(1) 19.33% (3,866) of them have at least one risk issue (Figure 7(a)), and all risk mini-programs exhibit bothering pop-ups (Figure 7(b)). Developers frequently fail to appropriately manage API callbacks, leading to repeated requests of rejected permission.

(2) As shown in Figure 7(b), looping pop-up behavior occurs in 0.33% (65) of mini-programs. We analyzed samples that required forced authorization and found that the information requested primarily pertained to the core functions of the applications. Developers often overlooked the possibility of users refusing authorization, leading to issues with application functionality. For instance, an application designed to locate electric vehicle charging stations cannot function properly without access to location data. The developer assumed users would grant authorization and thus did not adequately address the scenario of a user denying access.

(3) Among the data permission APIs, the basic information API is called with the highest frequency and mostly appears in the risky mini-programs (dashed box in Figure 7(c)). The reason is that mini-programs are more commonly used as service interfaces, and



(a) Proportion of mini-programs with different abusive request behavior kinds (b) Distribution of risk mini-programs

(c) Distribution of data permission APIs in each abusive behavior. The dashed box represents the information type of data permission APIs. The folded vertical coordinates from left to right are `my.saveVideoToPhonesAlbum`, `my.openBluetoothAdapter`, `my.chooseAlipayContact` and `my.choosePhoneContact`.

Figure 7: Risks in Large-scale Real-world Dataset

developers need to obtain user identity information (e.g., user IDs and access tokens) for further service. Therefore, basic information application behavior is very common. When developers do not have enough code writing awareness to correct their improper calls, abusive permission request behavior occurs.

Answer to RQ3: In the real environment, bothering pop-ups are the most prevalent and looping pop-ups occur when developers inadequately handle user rejections. Compared to other data permission APIs, risk behaviors appear more frequently when applying for basic information (e.g., avatar and phone number).

5 DISCUSSION

5.1 Findings and Suggestions

Based on our analysis, we find two primary factors contributing to the prevalence of abusive permission requests:

F1: Implementation Mechanism. Certain API designs do not adequately account for variations in the data collected. For instance, with `my.getAuthCode`, developers have the capability to gather fundamental user data by configuring various parameters. These parameters encompass fields such as the avatar nickname, retrievable through `my.getOpenUserInfo`, and mobile phone number details, accessible via `my.getPhoneNumber`. Furthermore, unlike Android, where permissions are executed asynchronously, mini-program permissions operate synchronously, potentially resulting in multiple permission requests occurring simultaneously.

F2: Developer Intentions. Some developers may lack privacy awareness, leading them to neglect appropriate methods for handling sensitive information, such as utilizing overlay pop-ups. Additionally, there are cases where developers intentionally create user inconvenience to coerce them into sharing sensitive data, exemplified by the use of repetitive pop-ups.

Based on the findings, we propose the following suggestions:

S1: Introduce Options. It is advised to incorporate a "Do not show again" functionality into all data permission pop-up windows. We have observed that while this feature is available for certain permission requests (e.g., `my.getLocation`), it is absent for others (e.g., `my.getAuthCode`). The inclusion of this option can alleviate the frequent appearance of pop-ups.

S2: Classify Data Types. The primary cause of repetitive pop-ups stems from the overlapping ranges of data types across different APIs. If distinct data permission APIs access segregated data types, and uniform JavaScript API interfaces are employed for the same data collection, developers can perform operations on identical data types without triggering additional pop-up windows, provided that the user has already granted permission.

We have conveyed our discovery of abuse and design issues in the permission mechanism to several mini-program platforms, submitted our recommendations, and received their confirmation.

5.2 Threats to Validity

Impact of Categories. *MiniChecker* has false positives in detecting homepage, overlaying, and looping pop-ups. Its reliance on static analysis hinders its ability to capture dynamic program execution values. For example, some mini-programs depend on these status codes to decide whether to proceed with future permission requests. Unlike bothering and repeating issues, which often necessitate multiple permission requests, these three types of problems require only a single request and have more stringent triggering conditions. To ensure comprehensive detection, we have considered all possible conditions. This may lead to an increased number of false positives for unexecuted permission requests. In the future, we plan to address this issue by ascertaining the execution status of complex branches through part dynamic execution. This involves collecting relevant variables within the conditions, analyzing their dependencies, and executing condition statements locally.

Code Obfuscation. During dataset construction, we encountered a considerable proportion of mini-program code with obfuscation issues, which are prevalent in the real world as well. For example, some obfuscation techniques cause code structure fragmentation by shuffling the sequence of function executions and assigning meaningless names, such as arbitrary letters or numbers. This process renders the original AST highly intricate, making it challenging to discern call relationships using standard rules. Moreover, certain programs consolidate all code into a single file, resulting in an exceedingly large AST structure. This significantly increases both time and space consumption for analyzing. Currently, there is limited research on JavaScript, and no anti-obfuscation methods specifically tailored to mini-programs exist.

Here we propose a feasible method, which we plan to develop in our future work, to counter known obfuscation techniques. First, we will collect and analyze both successful and failed samples to obtain their AST structures. Next, we will manually identify obfuscation patterns in the failed samples, compile a list of known obfuscation methods, and label the samples accordingly. Using the AST features and obfuscation labels, we will build a training data set to design a classifier capable of detecting obfuscation categories. Finally, we will apply appropriate anti-obfuscation techniques or tools to the classified samples to restore them to their pre-obfuscated states and employ *MiniChecker* to detect permission request issues.

Platform Migration. During our analysis, we observed similar permission application abuse issues in mini-programs on other platforms. For example, the mini-program `Shanghai Dongfang Hospital` with the same name as one in the benchmark dataset also has the issue of pop-up loops on the WeChat platform. After investigation, we found that mini-apps on various super-app platforms exhibit architectures similar to those of Alipay. The primary difference among these platforms lies in their distinct APIs. For example, unlike Alipay, WeChat's mini-programs can access a user's motion data through the `wx.getWeRunData` API. Omitting such APIs could result in incomplete data. In our future work, we plan to expand the API list of *MiniChecker* to support multiple platforms.

6 RELATED WORK

Mini-program Analysis. Mini-programs have gained popularity recently, and related research is still emerging[6, 22, 23, 39, 40, 44, 46]. Zhang et al. [47] constructed a dataset of WeChat mini-programs by injecting the WeChat client to obtain download links for mini-program code and analyzed their ecological characteristics. Several works have explored mini-program code analysis, including TaintMini[39], MiniTracker[21], and Wemint[25]. However, these studies primarily concentrate on analyzing data flows rather than call graphs. Our research represents the first attempt to investigate abusive permission request behaviors in mini-programs, which is not addressed in the aforementioned works.

Android Permission Security. The related work on permission analysis is primarily focused on Android applications[17, 28, 30, 35]. Felt et al.[11] evaluated permission risks in Android apps introduced by third-party plugins, considering user consent and defense mechanisms. Zhang et al. [48] proposed Vetdroid, analyzing app resource access for sensitive behavior detection. Taylor et al. [37] studied permission differences in apps with similar functions. However, the permission mechanisms in Android and mini-programs exhibit distinctions, resulting in varied manifestations of permission issues in mini-programs. Consequently, we introduce *MiniChecker*, aimed at identifying and addressing instances of permission application abuse in mini-programs.

7 CONCLUSION

This paper tackles the emerging security issue of detecting abusive data permission request behaviors within mini-programs. We propose *MiniChecker* for detecting five abusive issues in mini-programs and establish a baseline dataset to validate its effectiveness. By analyzing real-world mini-programs, we find some risky mini-programs and offer suggestions for preventing these risks.

ACKNOWLEDGEMENT

This work was supported by National Natural Science Foundation of China (62232014, 62272377, 62372368, 62372367), CCF-AFSG Research Fund, and Young Talent Fund of Association for Science and Technology in Shaanxi, China.

REFERENCES

[1] Moutaz Alazab, Mamoun Alazab, Andrii Shalaginov, Abdelwadood Mesleh, and Albara Awajan. 2020. Intelligent mobile malware detection using permission requests and API calls. *Future Generation Computer Systems* 107 (2020), 509–521. https://doi.org/10.1016/j.future.2020.02.002

[2] Gábor Antal, Péter Hegedus, Zoltán Tóth, Rudolf Ferenc, and Tibor Gyimóthy. 2018. Static JavaScript Call Graphs: A Comparative Study. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, Madrid, Spain, 177–186. https://doi.org/10.1109/SCAM.2018.00028

[3] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (Raleigh, North Carolina, USA) *(CCS '12)*. Association for Computing Machinery, New York, NY, USA, 217–228. https://doi.org/10.1145/2382196.2382222

[4] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. 2016. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 1101–1118. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/backes_android

[5] Patrick P. K. Chan and Wen-Kai Song. 2014. Static detection of Android malware by using permissions and API calls. In *2014 International Conference on Machine Learning and Cybernetics*, Vol. 1. IEEE, Lanzhou, China, 82–87. https://doi.org/10.1109/ICMLC.2014.7009096

[6] Kathleen Cheng, Maximilian Schreieck, Manuel Wiesche, Krcmar, and Helmut. 2020. Emergence of a Post-App Era – An Exploratory Case Study of the WeChat Mini-Program Ecosystem. In *15th International Conference on Wirtschaftsinformatik*. Potsdam, Germany.

[7] Ming Fan, Jun Liu, Xiapu Luo, Kai Chen, Zhenzhou Tian, Qinghua Zheng, and Ting Liu. 2018. Android Malware Familial Classification and Representative Sample Selection via Frequent Subgraph Analysis. *IEEE Transactions on Information Forensics and Security* 13, 8 (2018), 1890–1905. https://doi.org/10.1109/TIFS.2018.2806891

[8] Zheran Fang, Weili Han, and Yingjiu Li. 2014. Permission based Android security: Issues and countermeasures. *Computers and Security* 43 (2014), 205–218. https://doi.org/10.1016/j.cose.2014.02.007

[9] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient construction of approximate call graphs for JavaScript IDE services. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, San Francisco, CA, USA, 752–761. https://doi.org/10.1109/ICSE.2013.6606621

[10] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '11)*. Association for Computing Machinery, New York, NY, USA, 627–638. https://doi.org/10.1145/2046707.2046779

[11] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. 2012. Android Permissions: User Attention, Comprehension, and Behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security* (Washington, D.C.) *(SOUPS '12)*. Association for Computing Machinery, New York, NY, USA, Article 3, 14 pages. https://doi.org/10.1145/2335356.2335360

[12] GitHub. 2023. CodeQL: Industry-leading Semantic Code analysis Engine. https://codeql.github.com/

[13] Ant Group. 2023. Alipay Mini-app Document: API Overview. https://opendocs.alipay.com/mini/api

[14] Ant Group. 2023. Alipay Mini-app Document: Button. https://opendocs.alipay.com/mini/component/button

[15] Ant Group. 2023. Alipay Mini-app Document: Framework Overview. https://opendocs.alipay.com/mini/framework/overview

[16] Ant Group. 2024. CodeFuse Query. https://github.com/codefuse-ai/CodeFuse-Query

[17] Muhammad Ikram, Narseo Vallina-Rodriguez, Suranga Seneviratne, Mohamed Ali Kaafar, and Vern Paxson. 2016. An Analysis of the Privacy and Security Risks of Android VPN Permission-Enabled Apps. In *Proceedings of the 2016 Internet Measurement Conference* (Santa Monica, California, USA) *(IMC '16)*. Association for Computing Machinery, New York, NY, USA, 349–364. https://doi.org/10.1145/2987443.2987471

[18] ECMA International. 2023. ECMAScript Language Specification. https://262.ecma-international.org/14.0/

[19] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium (SAS) (LNCS, Vol. 5673)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 238–255.

[20] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: a static analysis platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 121–132. https://doi.org/10.1145/2635868.2635904

[21] Wei Li, Borui Yang, Hangyu Ye, Liyao Xiang, Qingxiao Tao, Xinbing Wang, and Chenghu Zhou. 2024. MiniTracker: Large-Scale Sensitive Information Tracking in Mini Apps. *IEEE Transactions on Dependable and Secure Computing* 21, 4 (2024), 2099–2114. https://doi.org/10.1109/TDSC.2023.3299945

[22] Yi Liu, Jinhui Xie, Jianbo Yang, Shiyu Guo, Yuetang Deng, Shuqing Li, Yechang Wu, and Yepang Liu. 2021. Industry Practice of Javascript Dynamic Analysis on WeChat Mini-Programs. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1189–1193. https://doi.org/10.1145/3324884.3421842

[23] Haoran Lu, Luyi Xing, Yue Xiao, Yifan Zhang, Xiaojing Liao, XiaoFeng Wang, and Xueqiang Wang. 2020. Demystifying Resource Management Risks in Emerging Mobile App-in-App Ecosystems. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 569–585. https://doi.org/10.1145/3372297.3417255

[24] Akshay Mathur, Laxmi Mounika Podila, Keyur Kulkarni, Quamar Niyaz, and Ahmad Y. Javaid. 2021. NATICUSdroid: A malware detection framework for Android using native and custom permissions. *Journal of Information Security and Applications* 58 (2021), 102696. https://doi.org/10.1016/j.jisa.2020.102696

[25] Shi Meng, Liu Wang, Shenao Wang, Kailong Wang, Xusheng Xiao, Guangdong Bai, and Haoyu Wang. 2023. Wemint:Tainting Sensitive Data Leaks in WeChat Mini-Programs. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1403–1415. https://doi.org/10.1109/ASE56229.2023.00151

[26] Uğur Pehlivan, Nuray Baltaci, Cengiz Acartürk, and Nazife Baykal. 2014. The analysis of feature selection methods and classification algorithms in permission based Android malware detection. In *2014 IEEE Symposium on Computational Intelligence in Cyber Security (CICS)*. IEEE, Orlando, FL, USA, 1–8. https://doi.org/10.1109/CICYBS.2014.7013371

[27] Naser Peiravian and Xingquan Zhu. 2013. Machine Learning for Android Malware Detection Using Permission and API Calls. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*. IEEE, Washington, DC, United States, 300–305. https://doi.org/10.1109/ICTAI.2013.53

[28] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. 2014. AutoCog: Measuring the Description-to-Permission Fidelity in Android Applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. Association for Computing Machinery, New York, NY, USA, 1354–1365. https://doi.org/10.1145/2660267.2660287

[29] QuestMobile2023. 2023. 2023 Panoramic Ecological Flow Report. https://research.tencent.com/article?id=A7Z

[30] Ittipon Rassameeroj and Yuzuru Tanahashi. 2011. Various approaches in analyzing Android applications with its permission-based security models. In *2011 IEEE INTERNATIONAL CONFERENCE ON ELECTRO/INFORMATION TECHNOLOGY*. IEEE, Mankato, MN, USA, 1–6. https://doi.org/10.1109/EIT.2011.5978583

[31] Hemant Rathore, Sanjay K. Sahay, Ritvik Rajvanshi, and Mohit Sewak. 2021. Identification of Significant Permissions for Efficient Android Malware Detection. In *Broadband Communications, Networks, and Systems*, Honghao Gao, Ramón J. Durán Barroso, Pang Shanchen, and Rui Li (Eds.). Springer International Publishing, Cham, 33–52.

[32] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 2019. 50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions System. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 603–620. https://www.usenix.org/conference/usenixsecurity19/presentation/reardon

[33] Bhaskar Pratim Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. 2012. Android Permissions: A Perspective Combining Risks and Benefits. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies* (Newark, New Jersey, USA) *(SACMAT '12)*. Association for Computing Machinery, New York, NY, USA, 13–22. https://doi.org/10.1145/2295136.2295141

[34] Maximilian Schreieck, Ange Ou, and Helmut Krcmar. 2023. Mini-App Ecosystems. *Bus. Inf. Syst. Eng.* 65, 1 (2023), 85–93. https://doi.org/10.1007/s12599-022-00773-9

[35] Wook Shin, Shinsaku Kiyomoto, Kazuhide Fukushima, and Toshiaki Tanaka. 2009. Towards Formal Analysis of the Permission-Based Security Model for Android. In *2009 Fifth International Conference on Wireless and Mobile Communications*. IEEE, Cannes/La Bocca, France, 87–92. https://doi.org/10.1109/ICWMC.2009.21

[36] Statistics. 2022. Number of available applications in the Google Play Store from December 2009 to March 2022. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[37] Vincent F. Taylor and Ivan Martinovic. 2016. SecuRank: Starving Permission-Hungry Apps Using Contextual Permission Analysis. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices* (Vienna, Austria) *(SPSM '16)*. Association for Computing Machinery, New York, NY, USA, 43–52. https://doi.org/10.1145/2994459.2994474

[38] W3C. 2022. MiniApp Standardization White Paper. https://www.w3.org/TR/mini-app-white-paper/

[39] Chao Wang, Ronny Ko, Yue Zhang, Yuqing Yang, and Zhiqiang Lin. 2023. TaintMini: Detecting Flow of Sensitive Data in Mini-Programs with Static Taint Analysis. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, Melbourne, Victoria, Australia, 932–944. https://doi.org/10.1109/ICSE48619.2023.00086

[40] Tao Wang, Qingxin Xu, Xiaoning Chang, Wensheng Dou, Jiaxin Zhu, Jinhui Xie, Yuetang Deng, Jianbo Yang, Jiaheng Yang, Jun Wei, and Tao Huang. 2022. Characterizing and Detecting Bugs in WeChat Mini-Programs. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 363–375. https://doi.org/10.1145/3510003.3510114

[41] T.J. Watson. 2016. JS WALA. https://github.com/wala/JS_WALA

[42] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. 2012. Permission Evolution in the Android Ecosystem. In *Proceedings of the 28th Annual Computer Security Applications Conference* (Orlando, Florida, USA) *(ACSAC '12)*. Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/2420950.2420956

[43] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. 2012. DroidMat: Android Malware Detection through Manifest and API Calls Tracing. In *2012 Seventh Asia Joint Conference on Information Security*. IEEE, Tokyo, Japan, 62–69. https://doi.org/10.1109/AsiaJCIS.2012.18

[44] Yuqing Yang, Yue Zhang, and Zhiqiang Lin. 2022. Cross Miniapp Request Forgery: Root Causes, Attacks, and Vulnerability Detection. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 3079–3092. https://doi.org/10.1145/3548606.3560597

[45] Muhammad Yusof, Madihah Mohd Saudi, and Farida Ridzuan. 2017. A new mobile botnet classification based on permission and API calls. In *2017 Seventh International Conference on Emerging Security Technologies (EST)*. IEEE, Canterbury, United Kingdom, 122–127. https://doi.org/10.1109/EST.2017.8090410

[46] Jianyi Zhang, Leixin Yang, Yuyang Han, Zixiao Xiang, and Xiali Hei. 2023. A Small Leak Will Sink Many Ships: Vulnerabilities Related to mini-programs Permissions. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, Torino, Italy, 595–606. https://doi.org/10.1109/COMPSAC57700.2023.00085

[47] Yue Zhang, Bayan Turkistani, Allen Yuqing Yang, Chaoshun Zuo, and Zhiqiang Lin. 2021. A Measurement Study of Wechat Mini-Apps. *Proc. ACM Meas. Anal. Comput. Syst.* 5, 2, Article 14 (jun 2021), 25 pages. https://doi.org/10.1145/3460081

[48] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang. 2013. Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (Berlin, Germany) *(CCS '13)*. Association for Computing Machinery, New York, NY, USA, 611–622. https://doi.org/10.1145/2508859.2516689