2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00151

WeMinT: Tainting Sensitive Data Leaks in WeChat Mini-Programs

Shi Meng[1,2], Liu Wang[*2], Shenao Wang[1], Kailong Wang[✉1], Xusheng Xiao[3], Guangdong Bai[4], Haoyu Wang[✉1]
[1†] Huazhong University of Science and Technology, China [2] Beijing University of Posts and Telecommunications, China
[3] Arizona State University, United States of America [4] The University of Queensland, Australia

***Abstract*—Mini-programs (MiniApps), lightweight versions of full-featured mobile apps that run inside a host app such as WeChat, have become increasingly popular due to their simplified and convenient user experiences. However, MiniApps raise new security and privacy concerns as they can access partially or all of host apps' system resources, including sensitive personal data. While taint detection has been proven effective in addressing this kind of concerns, existing taint detection techniques for mobile apps cannot be directly applied to MiniApps. The main reason is that the key logics of MiniApps are usually written in JavaScript, and its intrinsic characteristics (function-level scope, dynamic types, synchronous programming, and code obfuscation) prevent existing taint detection techniques from precisely propagating the taints. To address this problem, we propose a novel taint detection technique, WeMinT, that detects sensitive information leaks in MiniApps. Specifically, WeMinT facilitates taint propagation via building a context-based model based on the operational principle of MiniApps and JavaScript, and addresses asynchronous function calls by modeling their callbacks explicitly in taint rules. In addition, due to the adoption of Abstract Syntax Trees (ASTs) for code representation during taint detection, WeMinT exhibits better robustness against the commonly-applied code obfuscation. Our experimental results show that WeMinT can effectively detect sensitive information leaks in WeChat MiniApps, as well as trace the path of sensitive data flows. By applying WeMinT to over 20K suspicious MiniApps, we found that over 7.5K (36.5%) of them have sensitive data leaks, and WeMinT outperforms the state-of-the-art DoubleX based techniques in detecting these leaks.**

***Index Terms*—WeChat Mini-programs, Taint detection, Security, Privacy**

I. Introduction

Mini programs (or MiniApps), known as the lightweight version of the full-featured mobile applications running inside a host application (or SuperApp), have gained significant popularity in recent years by providing a seamless but simplified user experience (i.e., simplified navigation and limited but relevant features) [1]. For example, the users can accomplish a rich set of activities in an integrated manner, such as online shopping [2], watching and sharing social media contents [3], playing video games [4] or even seeing a doctor [5], inside the SuperApp without the nuisance for downloading multiple applications. These MiniApps thus gradually form unique ecosystems centering the SuperApps. Among them, WeChat has emerged as the most influential and dynamic, hosting over 7 million MiniApps that serve over 450 million DAUs (daily active users) [6].

The tremendous amount of data available from the WeChat ecosystem as well as the enormous size of active users have nurtured the thriving community of MiniApps[1] that is built upon. Accompanied by the conveniences and versatility boost for user experiences, the under-researched dynamics between WeChat and its MiniApps inadvertently open up new surfaces for ill-purposed parties to exploit potential privacy weaknesses. One of the primary concerns pertains to the collection and use of personal data by third-party developers. For instance, WeChat possesses a number of system-level permissions, and consequently the MiniApps could access partial or all of these system resources including personal data, without obtaining any form of notification or authorization. Another concern is the lack of transparency with respect to data collection and usage by third-party developers. The absence of clear information about how user data is collected and utilized renders it challenging for users to make informed decisions regarding their use of MiniApps.

As user private data protection has become increasingly critical in light of privacy laws such as GDPR [7], CCPA [8], APPI [9] and PDPA [10], WeChat has been actively incorporating guidelines and instructions [11] that impose more stringent scrutiny on data collection and processing from MiniApps. However, these are far from sufficient to safeguard the WeChat ecosystem from careless or even malicious developers who would accidentally or intentionally leak users' sensitive data. Although such issues from MiniApps have garnered mounting interest from the relevant research community, most of the recent works still focus on understanding the application ecosystem [4], [12]–[14] and measuring/characterizing its security/privacy issues [1], [15]–[17]. There still lacks a generic and systematic approach to track sensitive data flows and identify potential information leaks within MiniApps. The intuitive solution would be applying taint detection to this problem, as frameworks like FlowDroid [18] have been proven successful and powerful in the literature. Unfortunately, we have identified several critical obstacles that prevent the direct

[*]Liu Wang is the co-first author.

[✉]Haoyu Wang (haoyuwang@hust.edu.cn) and Kailong Wang (wangkl@hust.edu.cn) are the corresponding authors.

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

[1]MiniApps hereafter refers to WeChat MiniApps for simplicity, unless specified otherwise.

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00151

2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00151

application of existing taint detection techniques.

Given that the key logics of the MiniApps are written in JavaScript, most of those obstacles originate from the intrinsic characteristics of the language. The biggest obstacle is rooted in JavaScript's function-level scope, which specifies that variables declared inside a function are not visible outside of that function. This constraint makes it particularly difficult to propagate the taint, especially in larger code bases. In addition, the dynamically typed nature of JavaScript further exacerbates the situation, as the increased difficulty in resolving the types of variables or expressions hinders the taint propagation. The second obstacle is due to the asynchronous programming commonly seen in JavaScript. More specifically, the use of callbacks, promises, and async/await can make it complicated to determine the program execution flow. The third obstacle is related to the prevalent code obfuscation, which could invalidate techniques requiring clean source code.

Despite the abundance of tools and research for JavaScript analysis, they unfortunately fall short when directly applied to MiniApps. As a distinct subset of JavaScript applications, MiniApps incorporate a broad array of predefined and customized objects and APIs from the WeChat platform. These elements inevitably lead to disruptions in the data and control flows established by existing JavaScript static analysis tools, such as DoubleX [19] and TAJS [20], thereby complicating the analysis process. In particular, these disruptions could block or significantly reduce the effectiveness of taint propagation.

Our work. To advance the taint detection technique on MiniApps, we propose WeMinT that can effectively bridge the aforementioned gaps. WeMinT constructs a context-based analysis model based on the operational principle of MiniApps, which can enable effective taint propagation in different JavaScript function-level scopes (to be detailed in Section IV-B1). To provide better resolution for asynchronous functions, WeMinT leverages the taint detection model constructed according to the structural features of asynchronous APIs and their callback functions (to be detailed in Section IV-B2). Due to the adoption of Abstract Syntax Trees (ASTs) for code representation during taint detection, WeMinT exhibits better robustness against the commonly-applied code obfuscation, compared to approaches directly applied to clear-text source code such as those on JavaScript files. WeMinT starts with a taint detection to enable a coarse-grained search for potential data leaks. To capture in detail the suspicious private data leaks reported from the taint detection, we further develop a complementary data flow analysis that reports the explicit sensitive information transition paths from the target source to destination (to be detailed in Section IV-C).To facilitate the research in this area, we open source WeMinT at the online repository [21].

Contributions. In summary, this paper makes the following contributions:

- We propose WeMinT, a novel static taint detection framework designed to detect potential sensitive data leaks in WeChat MiniApps. WeMinT relies on a context-based analysis mode to address the challenges of JavaScript's function-level scopes and adopts abstract syntax trees for code representation, providing systematic taint detection and sensitive data flow path tracking in MiniApps.
- We conduct experiments to evaluate the effectiveness of WeMinT. Experimental results show that WeMinT is capable of tainting sensitive data leaks in MiniApps with promising accuracy and efficiency. Compared to the state-of-art, WeMinT exhibits better support for taint propagation and sensitive path construction in JavaScript.
- We perform a large-scale measurement study by applying WeMinT to over 20,000 MiniApps, seeking to measure the sensitive data leaks of MiniApps in the wild. We reveal the severity of sensitive data leaks in the WeChat MiniApp ecosystem.

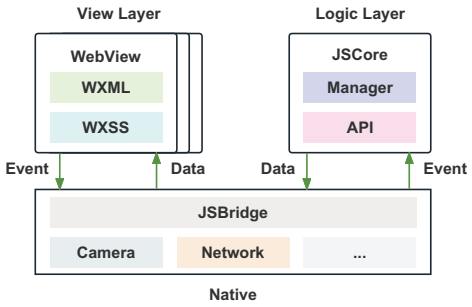

Fig. 1. The Runtime Framework of MiniApp

II. BACKGROUND

In this section, we provide an overview of WeChat MiniApps from both the perspective of code composition and the runtime framework.

A. WeChat MiniApp Code Composition

A WeChat MiniApp consists of two main parts: a set of code describing the overall MiniApp, and multiple sets of code describing each page of the MiniApp. The code describing the overall MiniApp consists of three files: (*i*) *app.json*, the global configuration file for the MiniApp; (*ii*) *app.js*, the file that registers the MiniApp instance with the *App()* method; (*iii*) *app.wxss*, the file that defines the global style. The code describing the MiniApp pages consists of four files: (*i*) *login.wxml*, which defines the page structure [22]; (*ii*) *login.js*, which defines the initial data, lifecycle callback functions and event handling functions for the page; (*iii*) *login.wxss*, which defines the page style; (*iv*) *login.json*, which configures the window representation of the page.

B. Runtime Framework for WeChat MiniApps

The framework of a WeChat MiniApp can be separated into two layers, the view layer and the logic layer, as shown in Figure 1. The view layer is composed of *.wxml* and *.wxss* files, and the logic layer comprises *.js* files. Multiple pages in an MiniApp correspond to multiple WebView threads in the view layer. The logic layer performs logical processing, data requests, interface calls, and so on through JSCore threads. The WebView thread and the JsCore thread communicate

through the JSBridge of the MiniApp host app (i.e., WeChat). Whenever the data in the logic layer changes, the *setData()* method is used to trigger updates in the view layer. The view layer generates an interaction event that invokes the corresponding event handler function in the logic layer, and if a change is made to the data in the event handler function, it triggers the update in the view layer again. In addition, some API calls in the logic layer are also handled through the host app, such as network requests are forwarded through the host app, cameras need to be started by the host app, etc.

III. Motivation and A Pilot Study

Due to the architectural design of WeChat MiniApps, MiniApps usually have frequent data exchange and sharing with the host apps and the backend server through the predefined interfaces, which opens up new surfaces for sensitive data leaks. One of the prominent examples is the leak of App Secret. The App Secret is a unique credential key of the MiniApp, and also an important parameter to get the Access Token that is the only backend API interface call credential for MiniApps. It is crucial for developers to keep this App Secret confidential, as WeChat officials have warned that the App Secret leaks can cause serious consequences such as identity fraud and sensitive data leaks. However, some developers still write the App Secret in the JavaScript code of the MiniApp logic layer, and use it as a parameter when calling the *wx.request()* to access the sensitive interface provided by WeChat and obtain some important return values. This approach is dangerous, as the code packages for MiniApps can be easily obtained and decompiled. If the App Secret is leaked, hackers can decompile the code package of the MiniApp and obtain the Access Token, potentially leading to an attack.

We conducted a pilot study and surprisingly found frequent App Secret leaks problem, even for those with a large user base. For instance, we found a WIFI information sharing MiniApp[2] (with 2.78 million users) vulnerable against the App Secret leaks in October 2022. Figure 2 shows that the developer defines a button in the view layer and binds the *getUserInfo()* callback function to initiate a user login operation. Within the function, the *requestLoginUserIdPost()* function in the same file is called, sending the App Secret as a parameter to the MiniApp backend to obtain the user information. It can be seen that the App Secret is explicitly written in the code of the MiniApp. Instead, the correct approach to save it is to encrypt and store it in the MiniApp's backend program.

IV. Approach

To enable automatic detection of sensitive data leaks from WeChat MiniApps, we develop a static analysis tool called WeMint. Figure 3 shows the overall architecture of WeMint.

[2]MiniApp name is anonymized for ethical considerations during paper review period. The developer has acknowledged and fixed the issue.

```
<view>
  <image class="index_welcom_bg" mode="aspectFill"
    src="../image/createWifiCodeBg.png">
  <button bindgetuserinfo="getUserInfo"
    openType="getUserInfo">Create My WiFi Code</button>
  </image>
</view>
```

```
getUserInfo: function(t) {
  /** some code **/
  wx.login({
    success: function(o) {
      e.requestLoginUserIdPost(o.code, t.detail.encryptedData,
        "wx01a1827d7f******", t.detail.iv);
    },
    fail: function(t) {}
  });
  /** some code **/
},
```

```
requestLoginUserIdPost: function(o, a, s, i) {
  /** some code **/
  // Declare the parameters when sending HTTPS requests
  var n = {code: o,encryptedData: a,appId: s,
    secret: "******36605cf7c736974dc33f******", // App Secret
    iv: i};
  // Call wx.request() to send a request
  wx.request({
    url: "https://wxapp.zt****.com/****/****/****/****",
    method: "POST", //Using the declared parameters
    data: n,
    header: {
      "content-type": "application/json;charset=utf-8"
    },
    /** some code **/
  });
},
```

Fig. 2. App Secret Leaks Example

A. Overview

WeMint is executed through the following three stages:

- **Source code processing:** The code package is decompiled into its original source code and file directory structure, and each page of the MiniApp is identified based on the page path defined in *app.json*.
- **Static analysis:** This stage consists of the core functionality of WeMint, including a taint detection engine for searching potential data leak locations on the coarse-grained level (to be detailed in Section IV-B), and a sensitive data flow path analyzer for tracking the specific data leak path on the fine-grained level (to be detailed in Section IV-C).
- **Results analysis:** Finally the results are aggregated and processed in this stage.

B. Taint-based Sensitive Data Leak Detection

Considering the security issues of MiniApps are primarily located in the JavaScript code of the logic layer, the taint-based detection technique faces the following challenges intrinsic to the JavaScript language. We first briefly enumerate them and our insights for mitigating them, followed by the details.

- **Difficulty of taint propagation due to the function-level scope.** We propose a context-based taint propagation model according to the rules of variable scopes in MiniApps and Javascript to bridge the gaps due to the scope restrictions (to be detailed in Section IV-B2).
- **Difficulty for handling asynchronous functions.** We propose an AST-based taint detection technique that utilize the AST structural features and callback functions for locating

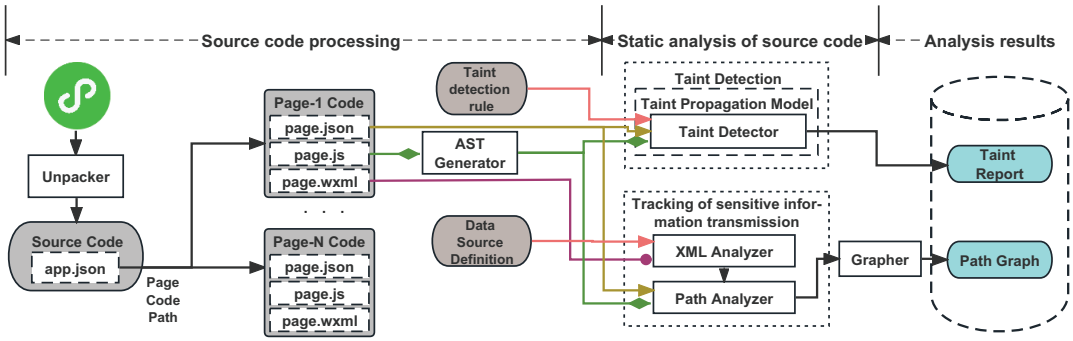

Fig. 3. Overall Architecture of WeMint

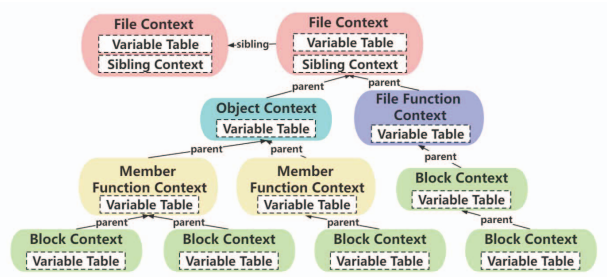

Fig. 4. Static Analysis Model of WeChat MiniApp

the taint. This technique also exhibits better robustness against obfuscation (to be detailed in Section IV-C).

1) Context-Based Taint Propagation Model (Solution for Difficulty #1): To facilitate the taint propagation, we propose a context-based model based on the rules of variable scopes in MiniApps and JavaScript, as shown in Figure 4. The general idea behind the model is to create a context for each code segment that defines new variable scopes, and construct a separate variable table in each context based on the propagation rules for variables in different scopes. The variable table encompasses all variable definitions in that context derived from static analysis. We specify the following 5 levels of context, as detailed below.

- *File Context*: The file-level context is an abstraction of the entire JavaScript file. From the rules of variable scopes in JavaScript, file-level variables and constants can be accessed and used in any scope of the file. The variable table of this context stores the definitions of all variables at the file level. In addition, due to the modularity introduced by the ES6 version of JavaScript, other files or modules may be introduced in the code via the *import* or *require* keywords. Thus, we also introduce the concept of sibling contexts in file-level contexts to construct relevant data dependencies. A file-level context may have one or more sibling contexts, and each sibling context is also a file-level context.
- *File Function Context*: This context is an abstraction of the global file-level functions. These functions can be called in member functions defined in the *Page()* of the MiniApps, where tainted code may be present. The variable table of this context stores the definitions of variables within the scope of the file-level functions. According to the rules for variable scopes in JavaScript, we consider the file-level context as the parent of file function context.
- *Object Context*: This context is an abstraction of the MiniApp objects. The variable table of this context stores the variable values that are parsed from the object's *data* property. Similar to the file function context, the parent of an object-level context is also a file-level context.
- *Member Function Context*: Member functions are functions defined within the object constructors, typically inside *App()* or *Page()* functions. The member function context is an abstraction of them. The variable table of this context stores the definitions of variables within the function scope. The parent of this context is the object-level context.
- *Block Context*: In JavaScript, a code block wrapped by '{}' creates a new block scope, e.g., *if* and *while* statements will create new variable scopes. The block-level context is an abstraction of them. The variable table of this context stores the definitions of variables in the current block scope. Additionally, the parent-child relationship at the block level arises when block nesting occurs.

2) AST-Based Taint Detection (Solution for Difficulty #2):

Detection flow. We generate an AST for each MiniApp's JavaScript code using Acorn [23]. We then traverse the AST in a depth-first manner and perform taint detection following the defined taint detection rules. The analysis process is illustrated in Algorithm 1. Specifically, we identify the variable, function, and *Page()* sections from the JavaScript code, and generate each level of context beginning from the File Context (i.e., root node) to lower-level contexts based on the variable scopes (lines 11 to 15).

The initial step takes place in the File Context where we identify the variables and constants (lines 8 to 10) defined at the file level. We perform a backward analysis to determine the values of these variables and constants, and save them in the variable table of the File Context. Next, we identify the functions declared at the file level. For each function, we generate a File Function Context that points to the File Context as its parent. We then perform taint location and variable analysis on these functions. Security issues in MiniApps can

Algorithm 1: Taint Detection Algorithm

Input: AST Root Node $root$,File Context fc,AST Node Visit Stack $S1$=[$root$],Context Visit Stack $S2$=[fc],Taint Checker $checker$
Output: Taint Report $report$

```
1  Function taintDetection(S1,S2,checker):
2      while S1! = ∅ do
3          node ← S1.pop();
4          context ← S2.pop();
5          if node.type ∈ checker.taintNodeTypeSet then
6              checker.checkByRule(node, context);
7          end
8          else if isVariableNode(node.type) then
9              variableValue ←
                 backTraceAnalysis(node, context)
               context.variableTable.update(node, variableValue);
10         end
11         else if isContextNode(node.type) then
12             newContext ← createContext();
13             newContext.parent ← context;
14             S2.push(newContext);
15         end
16         foreach child node childNode of node.children do
17             if childNode! = null then
18                 S1.push(childNode);
19             end
20         end
21     end
22     checker.analysisAndVerify();
23     report ← checker.generateReport();
24     return report;
25 End Function
```

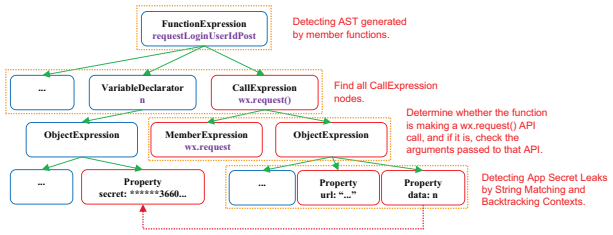

Fig. 5. App Secret Leaks Detection Rule

vary in type, and require different rules for taint location and analysis (lines 5 to 7). These rules need to be manually written based on the specific characteristics of the taint.

In the process of determining the specific values of key parameters in the tainted code, a backtracking operation is performed based on the static analysis model, starting from the current context and moving up the hierarchy to find the value of the key parameter. If the parent context at the top level cannot be found, the search continues in the sibling context.

Finally, we analyze the *Page()* function defined in the file and create an Object Context with the File Context as its parent. We first analyze all the member variables in the *data* property of the function and put the values obtained from the analysis into the variable table of this context. After that, we analyze all the member functions defined in the *Page()* function, and the analysis process is similar to the process of analyzing the file functions.

App Secret Detection Rules. Using a simple example shown in Figure 5, we explain how our App Secret detection rules can be derived from AST, and how they are utilized for the analysis. App Secrets typically function as parameters

```
1.let url1 = "https://demo.com/access?appid=" + getApp().appid +
"&secret=" + getApp().secret;  //member variables of App()
2.let url2 = "https://demo.com/access?appid=" + a.data.appid +
"&secret=" + a.data.secret;    //member variables of a
3.let url3 = "https://demo.com/access?appid=" + this.appid +
"&secret=" + this.secret;      //member variables referenced by this
4.let url4 = "https://demo.com/access?appid=" + appid + "&secret=" +
secret;                        //variables defined in code
```

(a)

```
1.function sendRequest(){
2.  let url = "https://demo.com/access?appid="; //define variable url
3.  wx.request({
4.    //string splice
5.    url:url + getApp().appid + "&secret" + getApp().secret,
6.    success:function(res){
7.      /** some code **/
8.    }
9.  })
10.}
```

(b)

Fig. 6. Other Obstacles of Taint Detection

when making HTTPS requests through *wx.request()*, which is represented as the *CallExpression* node in the AST. Therefore, we explicitly check for this type of node throughout the traversal process. Upon identifying such node, the function call within its child node *MemberExpression* is examined to see if it corresponds to *wx.request*. If a match is found, the parameters of this *CallExpression* node are inspected. Since these parameters are represented as an *ObjectExpression* type, all its properties are traversed to detect the presence of the App Secret within the *url* and *data* properties.

Handling asynchronous function calls. During the development process of MiniApps, various asynchronous scenarios need to be handled. The MiniApp SDK provides numerous asynchronous APIs that primarily use callback functions to achieve asynchrony. However, tainted code may exist in these callback functions. To deal with this scenario, we first try to summarize the characteristics of the AST structure for asynchronous functions. We find that on the AST, asynchronous functions also have a function type node in their parameters (child nodes), which is different from synchronous functions. This can help us identify whether an API is asynchronous or not. Specifically, we examined the code and AST structure of known asynchronous APIs, and we found that they are essentially function calls, with their AST structure consisting of a *CallExpression* type node. To use these APIs, an Object parameter defining the required properties and callback function is passed. This Object parameter is represented by an *ObjectExpression* node in the AST, which is a sub-node of the *Arguments* node of the *CallExpression* node. The callback function of the asynchronous API is a child element of this *ObjectExpression* node. Using this structure, we analyzed the callback functions of asynchronous APIs and the nested use of multi-layer asynchronous APIs to identify any potential tainted code within the callback functions.

3) Other Obstacles in Taint-Based Detection for JavaScript: Although we have tackled significant challenges through context-based model and AST, and yet some obstacles still remain in practice.

One prominent obstacle we need to address is the variable value analysis. While analyzing the tainted code, some key

parameters may not be directly in the form of literal quantities, but instead may be member variables of the object (e.g., *a.data.secret,getApp().secret*), member variables referenced by the *this* keyword (e.g., *this.secret*) or variables defined directly in the code (e.g., *secret*), as shown in Figure 6(a). We respond to each of these situations with different tactics. When encountering the case of type such as *a.data.secret*, where *a* is an object variable, we traverse the AST generated by *a* in depth first, get each property in *a* and store it in a dictionary structure which is named 'a' to the variable table. We then use *a[data][secret]* to obtain the value of *secret*. When we encounter the *getApp().secret* case, we use app.js as a sibling context to the current File Context. The member variables in *App()* are analyzed in the same way and stored in a dictionary structure, which has a global scope and from which the value of secret is taken. As for member variables referenced by the *this* keyword, they are taken directly from the variable table of the current Object Context. The value of a directly defined variable, such as *secret*, is obtained from the variable table of the context in which the variable is located or by backtracking analysis. This helps overcome the challenges in the variable value analysis phase while analyzing tainted code.

Another obstacle we face is expression analysis. Some key parameters of the tainted code may perform arithmetic operations, as shown in Figure 6(b), where the *url* parameter passed to *wx.request()* performs a splicing operation. To get the specific value of the *url* after that operation, we have to perform the same operation. However, since JavaScript is a weakly typed language, it is difficult to determine the specific type of each operand in an expression during analysis, and thus the value of that expression. To solve this problem, we first determine the value of each unknown variable in the expression by backtracking the context. Then we construct a JavaScript code string based on the operator and the value of the unknown variable obtained from the analysis. Finally, we execute the expression by calling the *eval()* function in JavaScript to obtain the final result of the expression.

C. Sensitive Data Flow Path Analysis

To enable better understanding of the detected sensitive data leaks, WEMINT also performs detailed data flow analysis that reports the explicit transition paths of sensitive information from the target source to the destination. We achieve this through a three-step process: (*i*) identifying the data source; (*ii*) designing the data tracking algorithm; (*iii*) constructing and plotting the data propagation path for users to analyze. This allows developers to better understand how the sensitive data flows within their MiniApps, and help them pinpoint the exact location of the data leak and take appropriate measures to fix it.

Similar to traditional apps, the sensitive data accessed by MiniApps mainly comes from two sources: one is user input data, such as name, age, cell phone number; the other one is the sensitive API usage, as they would require user's privacy-related authorization. Thus, we configure the data source as sensitive data that potentially relates to taints propagated from the previous coarse-grained taint detection stage. Meanwhile, it is important to note that the flow analyzer is also capable of tracking other generic type of data, which enables further configurable and wider-range data flow analysis.

1) Data Tracking Algorithm: The objective of this step is to track the propagation paths of sensitive data derived from taint detection, which is different from the process of identifying tainted code. Instead, we adopt a forward-detecting approach to identify code involving sensitive data, and tag it during data tracking. Specifically, our approach analyzes taint code-located functions, event callback functions bound by UI components, and return values from sensitive APIs. In addition, other member functions of the *Page()* object may also contain the use of sensitive APIs, and for other functions, we only perform sensitive API return value tracing.

Algorithm 2: Data Flow Analysis Algorithm

```
   Input: AST Root Node root,Key Parameters Set ps,Page
          Object Properties pd,Page Object Member Functions
          pf
   Output: Data Flow Path path
1  Function dataFlowAnalysis(root,ps,pd,pf):
2      path ← CreateNewPath();
3      if isSuspiciousNode(root.type) then
4          path.isKeyPath ←
             suspiciousCheck(node, ps, pd, pf);
5      end
6      foreach child node childNode of node.children do
7          if childNode! = null then
8              childPath ←
                 dataFlowAnalysis(childNode,ps, pd,pf);
9              if childPath.isKeyPath then
10                 path.next.append(childPath);
11             end
12         end
13     end
14     return path;
15 End Function
```

The algorithm for sensitive data flow path analysis is shown in Algorithm 2. For the located callback function, we construct its AST and perform a depth-first traversal, setting that function node as the root node. To identify the key steps involved in the propagation of sensitive data, we introduce the concept of "critical paths", which refer to the AST nodes that pertain to the sensitive data identified (lines 3 to 5) during the depth-first traversal. For nodes of branch statement type, we then continue to execute the algorithm recursively (lines 6 to 13). As we execute the algorithm, we mainly solve the following problems.

(*i*) *Identification and parsing of sensitive data.*

Functions with tainted code. As sensitive information can be recklessly transmitted as parameters, we thus mark the parameters passed or variables used in the tainted code that involves data transmission as sensitive data for further verification. For example, App Secret could be passed as a parameter to *wx.request()*. Otherwise, we simply mark the paths containing the tainted code as critical for further analysis.

```
1. Page({
2.   getPhoneNumber(e){
3.     //set data method
4.     this.setData({
5.       /* Assign the value entered by the user
6.       to the member variable 'phoneNumber' */
7.       phoneNumber: e.detail.value
8.     });
9.   }
10.});
```

Fig. 7. Takes The Value Entered By User

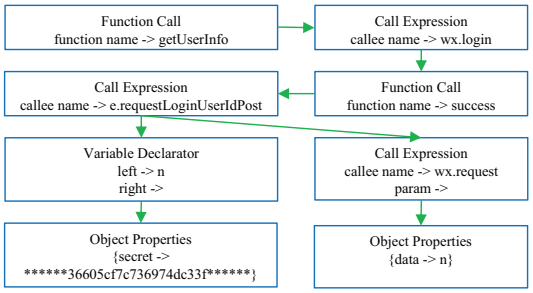

Fig. 8. Data Flow Graph

Callback function bound by UI components. After entering information in the input component, encapsulated in the event object as a parameter of the callback function for further use. The value is then extracted as a member variable, as shown in the Figure 7. The event object, represented by *e*, is passed to the callback function, and the user input can be retrieved using *e.detail.value*.

Return values from sensitive APIs. We have classified these APIs into two categories: synchronous and asynchronous, based on how they are used. Synchronous APIs directly include user privacy data in the return value, while asynchronous APIs accept an object type parameter in which a callback function can be defined. After a successful asynchronous API call, the privacy data will be included in the parameters of the *success* callback function.

Summary of analysis insights. When analyzing functions related to sensitive data, a variable set is created. It is initialized in various ways according to the generation of sensitive data. For example, sensitive information from UI component callback functions will be initialized and filled in the set with *e.detail*, while the return value of asynchronous sensitive APIs will be initialized and filled in the set with the parameters of the *success* callback function. When searching the AST nodes, we check whether the current node involves the key variables stored in the variable set. If there is a new variable generated by the key parameters in the variable set after operation or manipulation in the current node, then the node will be marked as a critical path node, and the new variable will be added to the variable set as a key variable. In case of complex expressions encountered during traversal, we transform the AST structure to code and use string matching to determine whether the expression is a critical path node.

(*ii*) *Analysis of page object member variables.* As aforementioned, member variables can be included in the page object by using *this.setData()*. The user input and the return values of the sensitive API can also be set as member variables of the page by *this.setData()*, making them accessible by the *this* keyword in other functions. We define a global *page-data set* to collect all key variables with values set by *this.setData()*. Then, the critical path node analysis also checks whether the current node uses sensitive data from the page-data set.

(*iii*) *Cross-functional analysis.* The main target of our analysis is the member functions defined in *Page()*. In MiniApps there may be calls to other member functions, causing a break in the data flow analysis. To overcome this, we maintain a mapping from the member function name to the AST node of that member function before analysis. Following the member function calling rules, other member functions will be called in the current function through *this.member function name*. Using the mapping based on the member function name, we can locate the corresponding AST node and continue to execute the analysis algorithm with this node serving as the root node, to address the problem of break in data flow analysis.

2) Construction of Sensitive Data Flow Paths: To construct the sensitive data flow paths, we create different path nodes based on the type of the AST nodes and create branch node paths directly when encountering branch type nodes. When encountering a critical path node, we create the corresponding path based on the type of the critical path node. Once all paths have been created, we draw the sensitive data flow graph using Graphviz [24], an open source graph visualization tool that can draw many types of graphs, especially suitable for generating image results such as data flow diagrams.

3) Example Illustration: Figure 8 shows an example of a path diagram of tainted flow in a MiniApp. The diagram describes the tracking result of the App Secret leaks problem in Figure 2. When the user clicks the Button to trigger the *getUserInfo()* event function, *wx.login()* API is called in the function, and the *requestLoginUserIdPost()* function is called in the *success* callback function of the API. In the *requestLoginUserIdPost()* function, the App Secret is hard-coded into the Object variable *n*, which is sent as a parameter to *wx.request()* API. Figure 8 fully describes all the key paths related to the tainted flow propagation.

V. Evaluation

In this section, we conduct experiments to evaluate WeMint. Our experiments were conducted on a Windows 10 system, powered by an 8-core Intel(R) Core(TM) i7-4790 processor clocked at 3.60GHz, and equipped with 16GiB of RAM. In particular, we seek to answer the following research questions (RQs):

- **RQ1.** How effective is WeMint in detecting and preventing sensitive data leaks in MiniApps?
- **RQ2.** What is the prevalence of MiniApps with sensitive data leaks in the wild?
- **RQ3.** How well does WeMint perform compared with state-of-the-art approach?

A. Evaluation Subject

We used *Mini-Crawler* [14], an open source crawler that is able to crawl MiniApps automatically, to collect a large number of MiniApp packages from the app market of WeChat to form our evaluation dataset. In total, we collected 115,392 MiniApps, occupying 261 GB of storage. To demonstrate the effectiveness of WeMinT, we evaluate how WeMinT detects App Secret leaks, a representative type of sensitive data leaks in MiniApps. App Secret leaks are mainly caused by carelessly programming, i.e., the developer explicitly writes the App Secret in the MiniApp's code. To find the MiniApps where App Secret leaks may occur and filter out useless samples, we first applied regular matching to pinpoint MiniApps from our dataset that may potentially suffer from this security problem. Specifically, we conducted a code scan on each MiniApp using regular matching pattern to identify a 32-bit long string composed of lowercase letters and numbers (i.e., the string pattern of App Secret). If it matches, we assume the scanned MiniApp as a suspicious MiniApp that may have App Secret leaks. As a result, we identified 20,766 suspicious MiniApps for further investigation.

B. RQ1: Effectiveness of WeMinT

1) Effectiveness of WeMinT's Taint Detection: In this evaluation, we first manually curate a benchmark of MiniApps. We randomly selected 100 MiniApp packages for manual inspection, and confirmed that 47 of them are positive samples with App Secret leaks. We then apply WeMinT to these 100 MiniApps. The results showed that for the 47 positive samples, WeMinT correctly identified 42 of them without reporting false positives. Thus, WeMinT achieves a precision of 100% (42/(42+0)), a recall of 89% (42/(42+5)) and an accuracy of 95% ((42+53)/100). In fact, as our detection rule is tailored to the usual scenarios of App Secret leaks, WeMinT is expected to exhibit zero false positives but might encounter some potential misses (ignoring the corner cases where decompilation fails). In addition, the average time cost for taint detection of each MiniApp was 3.11 seconds. These results show that WeMinT can effectively and efficiently detect data leaks in MiniApps.

2) Effectiveness of WeMinT's Sensitive Data Flow Path Analysis: WeMinT's sensitive data flow path analyzer is designed initially to track specific data leak paths, but in practice it can serve as a generic data flow analysis tool that enables tracing of any data of interest. For example, in MiniApps, the uses of sensitive APIs pose a potential security risk because they provide access to data that is highly valuable and confidential. Thus, tracking the data flow of sensitive API return values is an important way to ensure that sensitive data is stored and accessed securely. WeMinT, fortunately, is equipped with this capacity. Thus, we conduct another evaluation of WeMinT's generic data tracking capability. Specifically, we apply WeMinT to track the data flows of sensitive data commonly used in MiniApps and manually inspect the data flow paths to evaluate the effectiveness of the data flow path analyzer.

TABLE I
ELEMENTS OF THE UI INTERFACE THAT CAN INTERACT WITH THE USER

Classification	Example	Features
Components with *open-type* attributes that provide special functionality	<button>	Bind event callback functions using properties with the *bind* keyword
Forms, input boxes, and other components that collect user input	<form>,<input>,etc	Bind event callback functions using properties with the *bind* or *catch* keywords
Developer-defined components	None	The location of the code defining the components can be found in the configuration file
Collect user data using canvas, media and map	map, media selection and other components	The corresponding API can be found in the logic layer callback function

Evaluation Setup. In this evaluation, we concern about two types of sensitive data in MiniApps, i.e., user input data and sensitive API return values, as they are representative data sources for taints (as discussed in Section IV-C). Specifically, we use the data entered through UI and the data returned by calling sensitive APIs of MiniApps as the primary sources for sensitive data flow path analysis.

In the UI layout file, we have classified UI elements that can obtain user data and trigger event operations into four categories in Table I. (*i*) WeChat-offered components with the *open-type* attribute, such as the button component, which can specify button behavior and obtain user information through specified callback functions; (*ii*) Components that collect user input data, such as form and input box components, which use properties containing the "bind" or "catch" keywords to bind callback functions for receiving or processing user input data; (*iii*) Custom components developed by the developer for component reuse, which may include basic components for accepting user input, and whose paths can be found in the page configuration file; and (*iv*) Components that use canvas, media, map, and other way to collect user input information, which require the use of APIs provided by WeChat for collecting user information and can be found directly in the logical layer code without analyzing the UI interface.

In terms of sensitive APIs, since there is no predefined list of sensitive APIs available, we manually reviewed the developer documentation for all APIs and filtered them according to their functions, identifying 27 APIs that involve users' private data, as shown in Table II.

We randomly selected 20 MiniApps, and used WeMinT to extract event callback functions from the view layer as well as the return values of sensitive APIs to track them. Next, we manually analyzed the tracking results to assess the effectiveness of WeMinT's sensitive data flow path analysis.
Results. Table III shows the detailed results of our sensitive data flow path analysis for 20 MiniApps. Specifically, $\#\ of\ pages$ indicates how many pages are included in the MiniApp, $User\ Input\ Data$ and $Sensitive\ API$ indicates that the sources of sensitive data are from user interaction events in the view layer or return values of the sensitive APIs, respectively. Through manual inspection, we categorize the

TABLE II
SYNCHRONOUS AND ASYNCHRONOUS SENSITIVE APIS.

Classification	APIs	Sensitive Info
Sync APIs	wx.getSystemSetting	System Info
	wx.getSystemInfoSync	System Info
	wx.getDeviceInfo	Device Info
	wx.createCameraContext	Camera
Async APIs	wx.getSystemInfoAsync	System Info
	wx.getSystemInfo	System Info
	wx.requestPluginPayment	Payment Info
	wx.requestPayment	Payment Info
	wx.chooseVideo	Video
	wx.chooseMedia	Image&Video
	wx.startRecord	Audio
	wx.chooseImage	Image
	wx.startLocationUpdateBackground	Location
	wx.startLocationUpdate	Location
	wx.getLocation	Location
	wx.onLocationChange	Location
	wx.getFuzzyLocation	Location
	wx.chooseLocation	Location
	wx.choosePoi	Location
	wx.chooseAddress	Address
	wx.login	User Info
	wx.getUserProfile	User Info
	wx.getUserInfo	User Info
	wx.getWeRunData	WeRun
	wx.chooseLicensePlate	CarPlate
	wx.chooseContact	Contact
	wx.getClipboardData	Clipboard

TABLE III
RESULTS OF SENSITIVE DATA FLOW PATH ANALYSIS

MiniApp ID	# of pages	Data Flow		Total	Correct	Error	Miss	Time spent
		User Input Data	Sensitive APIs					
wx000b96cc50******	5	9	5	14	12	0	2	45.12s
wxbf990623a5******	13	18	4	22	20	1	1	36.86s
wx0b72e410af******	11	18	0	18	18	0	0	63.88s
wx0b21a6cf7a******	23	55	11	66	62	1	3	54.16s
wx00dbd76979******	16	34	3	37	36	0	1	25.16s
wx0c87dc4e32******	17	14	1	15	11	3	1	28.82s
wx0cb2d24f07******	9	18	0	18	16	2	0	34.47s
wx0ac876a29a******	7	8	10	18	15	0	3	56.95s
wx0f1f69246e******	9	10	2	12	12	0	0	29.21s
wx2aefd30c27******	18	28	6	34	28	2	4	28.90s
wx0d78a195bf******	5	5	1	6	4	1	1	24.26s
wx1f67e18769******	9	14	0	14	14	0	0	32.86s
wx3ba311e0a4******	7	2	6	8	6	0	2	32.18s
wx2a58adda69******	13	18	2	20	18	1	1	43.03s
wx0e08ba7d02******	9	14	0	14	14	0	0	34.14s
wx0e8be68d7f******	8	26	2	28	24	1	3	75.64s
wx0e3a760cb2******	15	30	8	38	36	0	2	36.51s
wx0f160dddb0******	6	31	3	34	34	0	0	31.05s
wx1b4851d9c7******	6	6	1	7	6	0	1	12.11s
wx0f4b126998******	11	14	2	16	16	0	0	32.40s
Total	/	372	67	439	402	12	25	/

quality of the generated data flow paths using three labels: $Correct$ denotes that a recognized data flow path is correct and complete; $Error$ denotes that a recognized data flow is wrong or broken; $Miss$ denotes that a data flow in the code failed to be recognized. Overall, WEMINT identified a total of 439 data flow paths, of which 402 were correctly identified, 12 were incorrectly identified, and 25 were missed. WEMINT achieved an accuracy of 91.57% in the module of sensitive data flow path analysis. Note that in our evaluation we define only sources but not sinks. This is because some APIs that can be used as sink can be encapsulated by developers, and defining an explicit sink point may cause WEMINT to miss some data flows.

We also measure the time cost of WEMINT for sensitive data flow path analysis (Column $Timespent$). The average time spent for a MiniApp was 37.89s, with the longest time being 75.64s, and the shortest time being 12.11s. We observe that the time spent is not proportional to the number of pages, because time-consuming operations are mainly spent on AST traversal and analysis. In general, the data flow path analysis takes a longer time for more complex JavaScript code structure (e.g., having a large number of code blocks and nested callback functions).

TABLE IV
DISTRIBUTION OF THE NUMBER OF USERS OF MINIAPPS WITH APP SECRET LEAKS.

Visit Total	Count	Percentage
[0,10000)	7011	92.43%
[10000,20000)	162	2.14%
[20000,50000)	204	2.69%
[50000,100000)	88	1.16%
[100000,1000000)	101	1.33%
[1000000,10000000)	18	0.24%
[10000000,60000000)	1	0.01%

C. RQ2: Prevalence of Sensitive Data Leaks

RQ2 aims to understand how many MiniApps with App Secret leaks WEMINT can find in the wild. For the selected 20,766 suspicious MiniApps, we applied WEMINT to perform taint detection, and successfully confirmed 7,585 MiniApps with App Secret leaks, accounting for 36.5% of the MiniApps in our dataset. Note that the AST-based detection for App Secret utilized by WEMINT is highly reliable. As indicated by the results in RQ1, all the MiniApps flagged by WEMINT are true positives with App Secret leak issues. Therefore, the identified 7,585 leaks are confirmative. In other words, there are at least 7,585 MiniApps in our dataset that have security leak issues.

To show the impacts of these data leaks, we next measure the number of users for these 7,585 MiniApps with App Secret leaks, as shown in Table IV. $Visit\ Total$ represents the cumulative number of users of a MiniApp, which is obtained by making requests to the interface provided by WeChat after acquiring the Access Token through App Secret and MiniApp ID. Note that the statistics are up to October 1, 2022. We can see that these MiniApps with such sensitive data leaks have a user base ranging from 2 to 53 millions. Over half of the MiniApps have a user base of less than 238 and 92% have a user base less than 10,000. This indicates that most of the MiniApps with sensitive data leaks have a limited number of users, while there are indeed a few of them with large user bases. Table V shows the top 10 MiniApps ranked by the cumulative number of users. $Share\ PV$ represents the number of retweets of the MiniApp and $Share\ UV$ represents the number of users who retweeted the MiniApp. We can see that the highest ranked MiniApp has 53,000,256 users, and the other MiniApps have a minimum of 1.5+ million users. Any attack launched by hackers on these MiniApps would have severe consequences given their user bases.

TABLE V
TOP 10 MINIAPPS WITH APP SECRET LEAKS BY NUMBER OF USERS.

MiniApp ID	Visit Total	MiniApp ID	Visit Total
wx845a2f34af******	53,000,**6	wx01a1827d7f******	2,789,**5
wxd3448d9870******	7,414,**5	wx781693d468******	2,741,**0
wx3db9d150cb******	5,147,**2	wx3da0e7ce42******	1,932,**5
wx5103f6e064******	4,127,**7	wxad40758d7c******	1,620,**3
wx507477e9ca******	3,197,**3	wx8c01564d8a******	1,565,**0

D. RQ3: Comparison with State-of-the-art

In this RQ, we compare WEMINT with the state-of-the-art open-source tool TAINTMINI [25] on ground truth dataset. TAINTMINI is a static taint analysis framework developed based on DoubleX[3] [19] for detecting flows of sensitive data in MiniApps. To this end, we utilize both tools to detect App Secret leaks and analyze sensitive data flow paths, and compare their output results. Specifically, we apply the tools to 20 representative MiniApps with App Secret leaks and conduct quantitative and qualitative analysis to demonstrate the superiority of WEMINT.

App Secret Leak Detection. Table VI illustrates the comparative results of WEMINT and TAINTMINI in App Secret leak detection. None of these App Secret Identifiers are recognized by TAINTMINI, which indicates the necessity of the ingenious design of WEMINT on specific tasks. We attribute the poor performance of TAINTMINI to its focus on identifying only `Object` identifiers while overlooking `key-value` pairs within it. For instance, in the most common App Secret leakage scenario, developers hardcode the Secret value in the `globalData` object of `App.js`, represented by the key-value pair {`secret: hardcode-hex-string`}. However, TAINTMINI only considers the object identifier `globalData` in the data flow analysis and overlooks the fine-grained analysis of key-value pairs.

Sensitive Data Flow Path Analysis. For a more detailed and comprehensive comparison, we assign the same source and sink APIs to both WEMINT and TAINTMINI, and evaluate the performance of detecting sensitive data flow paths on the ground truth benchmark. Table VII records the detailed results of sensitive data flow paths detected by both frameworks. For the 224 sensitive data flow paths in the ground truth, WEMINT successfully detected 202 out of them, while TAINTMINI detected 152. Additionally, TAINTMINI produced 2 false positives due to mistakenly detecting JavaScript files from unregistered pages.

TABLE VI
COMPARASION WITH TAINTMINI ON THE EFFECTIVENESS OF TAINT DETECTION

MiniApp ID	Identifier	WEMINT	TAINTMINI
wx000b96cc50******	`app_key_ald`	✓	✗
wx000d416d0d******	`e[appSecret]`	✓	✗
wx00a796d7b4******	`page[secret]`	✓	✗
wx00b1b2de34******	`wx.request.data[secret]`	✓	✗
wx00b74f75f6******	`globalData[secret]`	✓	✗
wx00b77e32b1******	`globalData[secret]`	✓	✗
wx00b94fba35******	`globalData[secret]`	✓	✗
wx00bb88373f******	`c[APPSECRET]`	✓	✗
wx00c462d1bc******	`globalData[secret]`	✓	✗
wx00cc9fcdbf******	`d[appKey]`	✓	✗
wx00cd02ab1e******	`c[APPSECTRE]`	✓	✗
wx00dbd76979******	`globalData[secret]`	✓	✗
wx00e1bf5a0c******	`e[APPSECRET]`	✓	✗
wx00e400c360******	`globalData[secret]`	✓	✗
wx00e4a41b44******	`globalData[secret]`	✓	✗
wx00e589a1ff******	`globalData[secret]`	✓	✗
wx00eb4375fb******	`e[APPSECRET]`	✓	✗
wx00eb6c28c7******	`globalData[secret]`	✓	✗
wx00eee55cb5******	`globalData[secret]`	✓	✗
wx00fdc948cb******	`_[APP_SECRET]`	✓	✗

TABLE VII
COMPARASION WITH TAINTMINI ON THE EFFECTIVENESS OF SENSITIVE DATA FLOW PATH ANALYSIS

MiniApp ID	GT[1]	WEMINT			TAINTMINI		
		TP	FP	FN	TP	FP	FN
wx000b96cc50******	3	3	0	0	3	2	0
wx000d416d0d******	6	6	0	0	5	0	1
wx00a796d7b4******	8	8	0	0	8	0	0
wx00b1b2de34******	2	2	0	0	2	0	0
wx00b74f75f6******	6	5	0	1	5	0	1
wx00b77e32b1******	2	2	0	0	1	0	1
wx00b94fba35******	3	3	0	0	3	0	0
wx00bb88373f******	20	20	0	0	5	0	15
wx00c462d1bc******	20	18	0	2	13	0	7
wx00cc9fcdbf******	6	6	0	0	6	0	0
wx00cd02ab1e******	19	19	0	0	5	0	14
wx00dbd76979******	3	3	0	0	3	0	0
wx00e1bf5a0c******	36	28	0	8	17	0	19
wx00e400c360******	45	44	0	1	40	0	5
wx00e4a41b44******	14	9	0	5	9	0	5
wx00e589a1ff******	3	3	0	0	3	0	0
wx00eb4375fb******	11	8	0	3	9	0	2
wx00eb6c28c7******	4	3	0	1	3	0	1
wx00eee55cb5******	1	0	0	1	0	0	1
wx00fdc948cb******	12	12	0	0	12	0	0
Total	224	202	0	22	152	2	72

[1] GT stands for the ground truth dataset.

VI. DISCUSSION

1) Performance Influencing Factors: First, JavaScript is a programming language that supports both procedural and functional programming, with all values in JavaScript being objects. The language's design has been enhanced by its event-driven and non-blocking capabilities since ES6. These features make it very flexible when writing JavaScript code, but their complexity also limits the accuracy of the analysis algorithm due to the large number of statement combinations that it cannot cover. In addition, since our static analysis process is performed on the MiniApps' source code, we use an unpacker tool to decompile the MiniApps to access to the source code. There are a small number of MiniApps experienced issues during the decompilation process, such as the code package cannot be properly decompiled, or the normal structure of the decompiled code is destroyed, or the decompiled code is obfuscated to a high degree, etc., which can have an impact on the static analysis results. Nevertheless, only 0.2% of MiniApps in our dataset were affected by the decompilation

[3]DoubleX is a static analysis tool designed to help developers identify potential data leaks in browser extensions. As both browser extensions and MiniApps are primarily developed in JavaScript, DoubleX can be used to analyze MiniApps.

issues, indicating a very limited impact.

2) Limitations: While WEMINT has proved to be an effective tool for identifying sensitive data leaks in WeChat MiniApps, it has some limitations. First, the accuracy of WEMINT's analysis results is affected by the complexity of the JavaScript code. If the code is highly obfuscated and has a complex structure, the detection coverage of the analysis results may be reduced. Second, during the sensitive data flow path analysis, we could identify data returned by sensitive APIs as sensitive data, but for user input data we were unable to determine if it was indeed sensitive data. In response, WEMINT tracks all the user input data and may produce false positives. This can be mitigated by adapting existing works [26] to analyze the sensitive data types of user input data. Third, the existing detection strategy in WEMINT may not be able to cover all code combinations because it is generated by manual analysis, which inevitably leads to some omissions. Continuous improvement is required to enhance the capabilities of the tool.

3) Extension and Generalization of WEMINT*:* In this study, WEMINT is designed to discover sensitive data leaks for WeChat MiniApps, and tracking sensitive data flow paths. In fact, WEMINT is extensible, allowing developers to develop a customized detection strategy for MiniApps based on the characteristics of the bugs they want to identify. This flexibility enables WEMINT to adapt to the ever-changing threat landscape and stay effective in detecting new types of sensitive data leaks. In terms of generalization, as WEMINT works on the basis of ASTs generated for the JavaScript language, it is applicable to all traditional MiniApps frameworks that use JavaScript as the development language for the logic layer (including WeChat, Alipay, etc.). However, for some MiniApps developed by multi-terminal unified development frameworks such as uni-app, Taro etc., WEMINT still needs future work to adapt.

VII. RELATED WORK

A. Analysis on MiniApps

As an emerging application paradigm in parallel to the existing web [27], [28] and mobile [29] systems, MiniApps have received attention from the research community only in the recent years. We characterize the prior works according to the following three aspects.

1) Understanding MiniApp Application and Architecture: A considerable portion of the research focuses on understanding the application and architecture of MiniApps [4], [12], [13]. More recently, Zhang et al. [14] designed MiniClawer, and then analyzed the security practices of the crawled MiniApps. They focus on whether the MiniApp code was obfuscated and which security-related APIs were involved.

2) Security Analysis of MiniApps: Several studies have focused on the security of MiniApps [1], [16]. In particular, Wang et al. [15] collected 83 MiniApp bugs from the real world and proposed WeDetector to detect WeBugs with three bug patterns. Zhang et al. [17] identified a novel privacy disclosure problem in MiniApps that can lead to the theft of private data held by the MiniApp platform. They illustrated an attack process that exploits this vulnerability.

3) MiniApp Analysis and Optimization Tools: A few studies have focused on MiniApp analysis and optimization tools. Liu et al. [30] designed WeJalangi, an efficient dynamic analysis framework for WeChat MiniApps based on the existing JavaScript dynamic analysis framework Jalangi [31]. However, they did not open-source it. Additionally, Li et al. [32] proposed a cross-learning search model for user fuzzy search. This model can assist users in finding the desired search results more easily.

B. Analysis on Javascript

There are numerous studies on JavaScript in the context of web security [33]–[37]. Melicher et al. [38] modified browsers and used dynamic taint analysis to detect possible vulnerabilities in web applications. DoubleX [19] investigated security issues in browser extensions and examined the JavaScript code of browser extensions using static analysis. In terms of JavaScript code analysis tools, there have been continued contributions in the community [39], [40]. JSAI [41] can convert JavaScript code into an intermediate language (IR) for static analysis. JSFlow [42] implements information flow tracking by performing dynamic analysis of JavaScript. SAFE [43] is an extensible parser that supports JavaScript AST rewriting and control flow graph analysis. Additionally, several tools are available to detect concurrency in Node.js, such as NodeAV [44], NRace [45], NodeRacer [46], and Node.fz [47]. Some work has also been done to detect analysis of client-side JavaScript, such as RClassify [48], AutoFLox [49], SymJS [50], JAW [51], and Adgraph [52].

VIII. CONCLUSION

Based on the analysis of potential security and privacy risks in WeChat MiniApps, we propose a novel framework, WEMINT, that uses abstract syntax trees to identify tainted code and trace the path of sensitive data flows. Our experimental results demonstrate that WEMINT can effectively and efficiently detect vulnerabilities to sensitive information leaks in MiniApps. By applying WEMINT to over 20K suspicious MiniApps, we discovered that over 7.5K (36.5%) of them have sensitive data leaks. Furthermore, our results indicate that WEMINT outperforms the state-of-the-art DoubleX based system in some aspects. These findings highlight the importance of using innovative approaches to ensure the security and privacy of WeChat MiniApps. In the future, we plan to extend WEMINT to address other types of security and privacy risks in MiniApps and explore the feasibility of integrating WEMINT into the development process of WeChat MiniApps.

ACKNOWLEDGMENTS

This work was supported in part by National Key R&D Program of China (2021YFB2701000), the National Natural Science Foundation of China (grant No.62072046, 62172049), Knowledge Innovation Program of Wuhan-Basic Research and HUST-FiberHome Joint Research Center for Network Security.

REFERENCES

[1] Y. Yang, Y. Zhang, and Z. Lin, "Cross miniapp request forgery: Root causes, attacks, and vulnerability detection," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 3079–3092.

[2] "ecommerce saas solution by wechat: a complete guide," https://wechatwiki.com/wechat-resources/wechat-mini-shop-ecommerce-solution/, 2022.

[3] Q. Rao and E. Ko, "Impulsive purchasing and luxury brand loyalty in wechat mini program," *Asia Pacific Journal of Marketing and Logistics*, 2021.

[4] L. Hao, F. Wan, N. Ma, and Y. Wang, "Analysis of the development of wechat mini program," in *Journal of Physics: Conference Series*, vol. 1087, no. 6. IOP Publishing, 2018, p. 062040.

[5] Y. Qian and A. Hanser, "How did wuhan residents cope with a 76-day lockdown?" *Chinese Sociological Review*, vol. 53, no. 1, pp. 55–86, 2021.

[6] "White paper on internet development of miniapps in 2021," https://aldzs.com/viewpointarticle?id=16175, 2022.

[7] "General data protection regulation," https://commission.europa.eu/law/law-topic/data-protection_en, 2022.

[8] "California consumer privacy act," https://oag.ca.gov/privacy/ccpa, 2022.

[9] "Act on the protection of personal information," https://www.ppc.go.jp/, 2022.

[10] "Personal data protection act," https://www.pdpc.gov.sg/, 2022.

[11] "Configuring user privacy protection guidelines for miniapps," https://www.aldzs.com/viewpointarticle?id=16573, 2023.

[12] A. Cheng, G. Ren, T. Hong, K. Nam, and C. Koo, "An exploratory analysis of travel-related wechat mini program usage: affordance theory perspective," in *Information and Communication Technologies in Tourism 2019: Proceedings of the International Conference in Nicosia, Cyprus, January 30–February 1, 2019*. Springer, 2019, pp. 333–343.

[13] L. Ma, L. Wang, and E. Jiang, "Empirical study on the wechat mini program acceptance based on uta ut model take the pearl river delta as an example," in *2018 15th International Conference on Service Systems and Service Management (ICSSSM)*. IEEE, 2018, pp. 1–6.

[14] Y. Zhang, B. Turkistani, A. Y. Yang, C. Zuo, and Z. Lin, "A measurement study of wechat mini-apps," *ACM SIGMETRICS Performance Evaluation Review*, vol. 49, no. 1, pp. 19–20, 2021.

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

[16] H. Lu, L. Xing, Y. Xiao, Y. Zhang, X. Liao, X. Wang, and X. Wang, "Demystifying resource management risks in emerging mobile app-in-app ecosystems," in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications Security*, 2020, pp. 569–585.

[17] L. Zhang, Z. Zhang, A. Liu, Y. Cao, X. Zhang, Y. Chen, Y. Zhang, G. Yang, and M. Yang, "Identity confusion in {WebView-based} mobile app-in-app ecosystems," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1597–1613.

[18] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[19] A. Fass, D. F. Somé, M. Backes, and B. Stock, "Doublex: Statically detecting vulnerable data flows in browser extensions at scale," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1789–1804.

[20] S. H. Jensen, M. Madsen, and A. Møller, "Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications," in *FSE*, 2011, p. 59–69.

[21] "Wemint," https://anonymous.4open.science/r/WEMINT, 2023.

[22] "Weixin markup language," https://developers.weixin.qq.com/miniprogram/dev/reference/wxml, 2022.

[23] "Acorn-a small, fast, javascript-based javascript parser," https://github.com/acornjs/acorn, 2022.

[24] "Graphviz is open source graph visualization software," https://graphviz.org/, 2022.

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

[26] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang, "{SUPOR}: Precise and scalable sensitive user input detection for android apps," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 977–992.

[27] K. Wang, J. Zhang, G. Bai, R. Ko, and J. S. Dong, "It's Not Just the Site, It's the Contents: Intra-domain Fingerprinting Social Media Websites Through CDN Bursts," in *WWW*, 2021.

[28] K. Wang, Y. Ling, H. Wang, G. Bai, and J. S. Dong, "Are they Toeing the Line? Auditing Privacy Compliance among Browser Extensions," SIGMETRICS.

[29] K. Wang, Y. Zheng, Q. Zhang, G. Bai, Q. Mingchuang, D. Zhang, and J. S. Dong, "Assessing Certificate Validation User Interfaces of WPA Supplicants," in *MobiCom*, 2022.

[30] Y. Liu, J. Xie, J. Yang, S. Guo, Y. Deng, S. Li, Y. Wu, and Y. Liu, "Industry practice of javascript dynamic analysis on wechat mini-programs," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1189–1193.

[31] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for javascript," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 488–498.

[32] H. Li, Z. Liu, S. Xu, Z. Lin, and X. Chen, "How to find it better? cross-learning for wechat mini programs," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, 2019, pp. 2753–2761.

[33] S. Lekies, B. Stock, and M. Johns, "25 million flows later: large-scale detection of dom-based xss," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 1193–1204.

[34] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett, "Vex: Vetting browser extensions for security vulnerabilities." in *USENIX Security Symposium*, vol. 10, 2010, pp. 339–354.

[35] B. B. Nielsen, B. Hassanshahi, and F. Gauthier, "Nodest: Feedback-driven static analysis of node.js applications," ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 455–465. [Online]. Available: https://doi.org/10.1145/3338906.3338933

[36] S. Wei and B. G. Ryder, "Practical blended taint analysis for javascript," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 336–346. [Online]. Available: https://doi.org/10.1145/2483760.2483788

[37] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg, "Saving the world wide web from vulnerable javascript," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 177–187. [Online]. Available: https://doi.org/10.1145/2001420.2001442

[38] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia, "Riding out domsday: Towards detecting and preventing dom cross-site scripting," in *2018 Network and Distributed System Security Symposium (NDSS)*, 2018.

[39] S. Guarnieri and V. B. Livshits, "Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code." in *USENIX Security Symposium*, vol. 10, 2009, pp. 78–85.

[40] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for javascript." in *SAS*, vol. 9. Springer, 2009, pp. 238–255.

[41] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf, "Jsai: A static analysis platform for javascript," in *Proceedings of the 22nd ACM SIGSOFT international symposium on Foundations of Software Engineering*, 2014, pp. 121–132.

[42] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, "Jsflow: Tracking information flow in javascript and its apis," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, 2014, pp. 1663–1671.

[43] J. Park, Y. Ryou, J. Park, and S. Ryu, "Analysis of javascript web applications using safe 2.0," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 59–62.

[44] X. Chang, W. Dou, Y. Gao, J. Wang, J. Wei, and T. Huang, "Detecting atomicity violations for event-driven node. js applications," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 631–642.

[45] X. Chang, W. Dou, J. Wei, T. Huang, J. Xie, Y. Deng, J. Yang, and J. Yang, "Race detection for event-driven node. js applications," in

2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2021, pp. 480–491.

[46] A. T. Endo and A. Møller, "Noderacer: Event race detection for node. js applications," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 120–130.

[47] J. Davis, A. Thekumparampil, and D. Lee, "Node. fz: Fuzzing the server-side event-driven architecture," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 145–160.

[48] L. Zhang and C. Wang, "Rclassify: classifying race conditions in web applications via deterministic replay," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 278–288.

[49] F. S. Ocariza Jr, K. Pattabiraman, and A. Mesbah, "Autoflox: An automatic fault localizer for client-side javascript," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 31–40.

[50] G. Li, E. Andreasen, and I. Ghosh, "Symjs: automatic symbolic testing of javascript web applications," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 449–459.

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

[52] U. Iqbal, P. Snyder, S. Zhu, B. Livshits, Z. Qian, and Z. Shafiq, "Adgraph: A graph-based approach to ad and tracker blocking," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 763–776.