

MINNIE: Dynamic Privacy Leak Detection in WeChat Miniapps via Unified Shadow Memory

Jianjia Yu*, Zhengyu Liu*, Zhihan Xia^{†¶}, Penghui Li[‡], Zifeng Kang^{§¶}, Junfeng Yang[‡], Yinzhi Cao*

*Johns Hopkins University, [†]Stanford University, [‡]Columbia University

[§]Beijing University of Posts and Telecommunications

{jyu122, zliu192, yinzhi.cao}@jhu.edu, zxia2172@stanford.edu

pl2689@columbia.edu, junfeng@cs.columbia.edu, zifengkang@bupt.edu.cn

Abstract—Mobile applications such as WeChat host millions of embedded applications, known as *miniapps*, which allow users to complete various tasks like hotel booking without leaving the host superapp. Since miniapps can access sensitive user data through application programming interfaces (APIs) provided by the superapp, a major concern is privacy leakage to third parties. Prior works on detecting such leaks predominantly rely on static analysis, which requires heavy API modeling and fails to capture dynamic JavaScript features, leading to false positives and false negatives. Dynamic analysis, however, faces unique challenges in the miniapp ecosystem: miniapp code operates in isolated *view/logic* layers that communicate through (de)serialization, where value-level taint tracking loses granularity. Additionally, privacy-relevant behaviors are often guarded by conditional branches that depend on external inputs from users or servers and are difficult to exercise under concrete execution.

We address these challenges with MINNIE, the first dynamic taint analysis framework for miniapp privacy leak detection. MINNIE introduces a unified byte-level shadow memory that combines a taint label, which identifies the sensitive source category, with a symbolic mark, which indicates whether the byte is symbolic. This representation enables holistic taint propagation across components and layers while supporting lightweight path exploration. Specifically, MINNIE propagates shadow memory through string operations, synchronizes it across serialization boundaries, and integrates a UI scheduler to systematically explore event sequences. In our evaluation of a large-scale dataset of 10,625 popular real-world WeChat miniapps, MINNIE flagged over five thousand candidate zero-day privacy leaks across over one thousand miniapps. We used an LLM-based pipeline to cross-check all flagged flows against their privacy policies, and manual validation yields a policy-checking precision of 81.6%. Confirmed privacy leaks constitute unauthorized sharing to third parties such as analytics, advertising, and tracking services, not user-intended functionality. Notable cases include popular miniapps like DiDi Chuxing and WeBank that leak user IDs to other miniapps, undermining WeChat’s intended privacy safeguards.

[¶]Both authors contributed to the work when they were students at Johns Hopkins University.

I. INTRODUCTION

Mobile applications such as WeChat [1], Telegram [2], LINE [3], Rakuten [4], and TikTok [5], known as *superapps*, host a large number of lightweight embedded applications, known as *miniapps*, that extend the superapp’s functionality without separate installation. The most popular superapp, WeChat, hosts over 3.5 million third-party miniapps on its market that engage with more than 600 million active daily users [6]. Despite their widespread adoption, miniapps face a critical issue in data governance and control: the potential leakage of private user data to third parties. Miniapps can access sensitive user data like phone numbers and geographic information via privileged superapp-provided Application Programming Interfaces (APIs), and covertly transmit this data to external parties without the user’s knowledge, through network or cross-miniapp channels. This emerging threat has been confirmed in news reports, where miniapps were found leaking personal data in banking services [7], and a breach of over 130,000 medical examination reports in medical services [8].

Automatic detection of privacy leaks in miniapps is essential, especially given the ecosystem scale with millions of miniapps. Previous solutions are predominantly static [9], [10], [11], [12], [13], [14], [15], [16], with MiniScope [14] additionally employing lightweight runtime API usage monitoring. These approaches analyze miniapp code to track sensitive data flows and are generally efficient. However, they face challenges with dynamic miniapp features such as asynchronous event callbacks and runtime-resolved UI bindings, and often fail to track taints across the *view* and *logic* layers (Section II-A), which run in separate execution environments. For example, recent static and hybrid approaches [12], [13], [14] produce false negatives from uncaptured dynamic data flows and false positives from imprecise modeling of UI event-callback bindings.

Dynamic taint analysis, on the other hand, naturally captures runtime behaviors, but miniapps’ *multi-component, multi-layer architecture* poses two unique challenges. First, taint flows traverse architectural boundaries where value-level tracking loses information. Horizontally, a miniapp runtime consists of many pages rendered through separate *view* layers with their data maintained in an accumulated *logic* layer. The *view* layers run in separate WebView instances, while the

combined *logic* layer runs in a JS engine. They exchange data via serialization and deserialization, across which value-level taint tracking loses byte-level correspondence. Vertically, the miniapp execution stack involves four components, from top to bottom: the third-party miniapp code (e.g., a hotel booking miniapp); the miniapp framework provided by the superapp, which defines and exposes APIs such as `wx.getUserInfo`; the JavaScript engine; and the native superapp. In the context of privacy leaks, taint sources and sinks are superapp-provided APIs, defined in the miniapp framework, invoked from third-party miniapp code, and intermixed with other JS operations executed by the engine. Bridging taint propagation across this vertical stack is challenging. Second, privacy-relevant code paths are often guarded by conditions on external inputs such as server responses and user inputs, which concrete execution alone cannot fully exercise.

Existing dynamic analysis tools do not address these challenges. WeJalangi [17] and MiniScope [14] target miniapps but with no taint tracking, performing runtime error diagnosis or API logging. TaintDroid [18], ViaLin [19], and browser-based engines [20], [21] provide dynamic taint tracking but target Android or the browser. None of them can exercise paths guarded by conditions on external inputs because none of them carries a symbolic mark. SymSan [22] carries symbolic expressions in shadow memory but addresses them from object pointers within a single address space. It does not apply to the miniapp’s cross-component, cross-layer architecture, where such shadow memory needs to be synchronized across the (de)serialization that carries data between isolated runtimes.

In this paper, we design and implement MINNIE, the first dynamic taint analysis framework for miniapp privacy leak detection. The core design of MINNIE is a *unified byte-level shadow memory* representation that combines taint tracking and lightweight concolic execution. For every byte in a string, the corresponding shadow memory maintains its taint source, if applicable, and a symbolic mark, indicating whether the byte is symbolic or concrete.

First, MINNIE propagates shadow memory within and between components and layers. When the analysis crosses components, e.g., from the miniapp framework to the JS engine, MINNIE carries the shadow memory by instrumenting both components. When the analysis goes between *view* and *logic* layers, MINNIE synchronizes the shadow memory using additional fields in the messages between WebView and the JS engine. Second, MINNIE leverages the symbolic mark for lightweight concolic execution. When MINNIE encounters a branching condition involving symbolic bytes, it constructs and solves constraints over the symbolic marks to explore both current and alternative execution paths. MINNIE also incorporates a UI exerciser that triggers different user inputs to drive the dynamic analysis.

We conducted a large-scale measurement using MINNIE on 10,625 real-world WeChat miniapps. MINNIE flagged over five thousand candidate zero-day privacy leaks spanning over one thousand miniapps. We cross-checked these flows against the miniapps’ privacy policies using the LLM-based pipeline

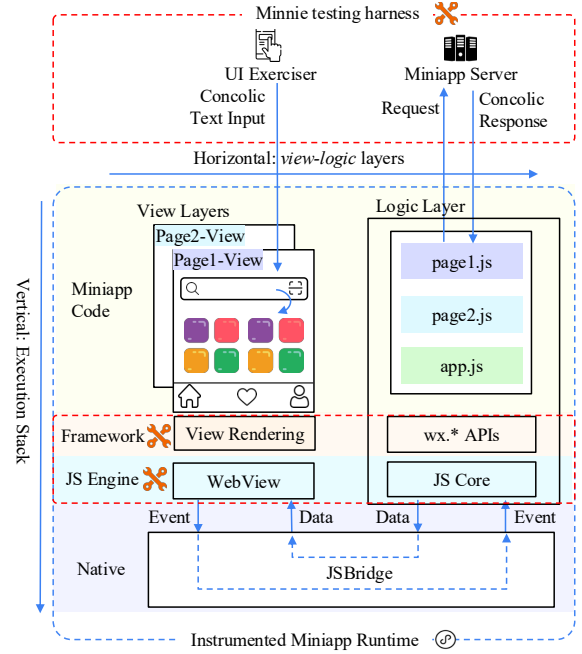


Fig. 1: Architecture of MINNIE’s instrumented miniapp runtime. The execution stack spans four components: miniapp code, framework, JS engine, and native superapp. View and logic layers run in different JS environments and communicate through the native JSBridge. MINNIE instruments the framework and JS engine (red). MINNIE’s testing harness (top) drives concolic execution by injecting symbolic text into UI elements and returning symbolic responses to network requests.

of prior work [15]. We measured the precision of the cross-checking by randomly sampling 100 miniapps and manually comparing their 609 flagged flows against their raw policy texts. We confirmed that 81.6% of the flows are genuine policy violations, and 83 of the 100 miniapps have at least one confirmed leak. Section V-B details the validation. Confirmed notable cases include DiDi Chuxing [23] and WeBank [24], which leak user IDs across miniapp boundaries, undermining WeChat’s per-miniapp identity isolation.

In summary, we make the following contributions:

- We present MINNIE, the first dynamic taint analysis framework for detecting user privacy leaks in WeChat miniapps.
- We introduce a unified byte-level shadow memory abstraction in which each byte carries a taint label and a symbolic mark, supporting holistic taint propagation across components and layers and lightweight concolic execution. We realize this abstraction across the JavaScript engine and the WeChat miniapp framework, together with a UI exerciser that systematically drives events and page transitions.
- We conduct a large-scale measurement of the WeChat miniapp ecosystem. From 10,625 miniapps, MINNIE flags over five thousand candidate zero-day privacy leaks across over one thousand miniapps. We manually validate the policy-violation labeling on a random 100 miniapp sample and report a measured precision of 81.6%. We release MINNIE, at <https://github.com/Suuuuuzy/Minnie>.

II. OVERVIEW

A. Miniapp Architecture

A superapp embeds lightweight *miniapps* inside it, combining the extensibility of web applications with the capabilities of native applications. We summarize the architecture along two axes, as shown in Figure 1.

Vertical stack. A miniapp spans four components from top to bottom, the third-party miniapp code, the miniapp framework, a JavaScript engine, and the native superapp. The miniapp code is written in JavaScript (for application logic) and a markup language (e.g., WXML for layout, WXSS for styling). It runs on top of the miniapp framework, which is provided by the superapp and exposes a set of APIs (e.g., `wx.getUserInfo`) for accessing user data, network resources, and UI services. The framework runs on a JavaScript engine, which in turn runs on the native superapp.

Horizontal layers. Each page in a miniapp has two layers: a *view* layer that renders the UI, and a *logic* layer that executes the application’s JavaScript. Different pages have separate view layers running in different WebView instances, while all pages share a single logic layer running in one JS engine. The two layers do not communicate directly; instead, all data and UI events are routed through the native layer’s JSBridge, which serializes data when crossing the boundary.

B. Threat Model

Our threat model considers privacy leaks in superapp-miniapp platforms, where third-party miniapps legitimately obtain sensitive user data through privileged APIs provided by the superapp but then expose such data to unauthorized parties through network or cross-miniapp channels, in violation of the miniapp’s privacy policy. Superapps such as WeChat collect rich Personally Identifiable Information (PII) including location, phone number, and contact list. This information is accessible to third-party miniapps via superapp-provided APIs, when requested by a miniapp. Miniapps can then transmit this data to untrusted third parties without user’s knowledge [8]. Specifically for WeChat, we consider miniapps that acquire users’ private information through miniapp library APIs and covertly share it through network channels (e.g., `wx.request`) or cross-miniapp channels (e.g., `wx.navigateToMiniProgram`). We provide the complete list of source and sink APIs in Table X (Appendix B).

Such privacy leaks break the superapp’s built-in cross-miniapp privacy protections. For example, WeChat assigns each user a unique `openid` per miniapp, to prevent identity linking across miniapps owned by different parties. Miniapps can bypass this restriction by sharing data through redirection to another destination miniapp. This is especially stealthy because the user prompt only displays the destination miniapp, not the transmitted data. Therefore, multiple miniapps can link the same user with different `openids`, thus enabling unauthorized cross-miniapp identity tracking and undermining WeChat’s intended privacy safeguards.

C. Motivating Example

We present a zero-day user privacy leak in the miniapp *Yes Accounting Exam Study* [25] discovered by MINNIE. The online classes in this miniapp have several million listeners. Despite its popularity, the miniapp leaks users’ phone numbers to other miniapps.

Figure 2 and Listing 1 illustrate the end-to-end workflow and source code involved in the privacy leak, respectively. The leak spans two user-triggered steps in Miniapp A (S0 and S8 in Figure 2) and involves multiple components within and across the miniapp runtime, the miniapp server, and the superapp server. In the first step, the user navigates to `pages/login`, where the miniapp prompts them to tap the “authorizePhoneNumber” button for login (S0 in Figure 2). This triggers a request to the superapp server (S1), which returns a `phoneNumberCode` credential (S2). The miniapp runtime forwards this code to its own server (S3), which then requests the actual phone number from the superapp server (S4), receives it (S5), and returns it to the miniapp runtime (S6). The runtime then stores the phone number locally (S7), completing the login process. At the code level, this logic is implemented in the `authorizePhoneNumber` function (lines 2–13 in Listing 1). The function receives a response object `n`, which includes the credential `n.detail.phoneNumberCode`. It sends this to the miniapp server via a POST request (line 5). Upon receiving a response, it checks for the presence of `e.data.resInfo.phoneNumber` (line 9), verifies that it starts with expected country code (line 11), and stores the phone number using `setStorageSync` if the check succeeds (line 12).

In the second step, the user navigates to `pages/mine` and taps a button to join an online class, triggering the `goLearning` function (line 33), which in turn calls `y` (lines 15–32). This function is wrapped in a state-machine-like structure: it retrieves the stored phone number (line 22), constructs a URL embedding it (line 23), and eventually passes it to `wx.navigateToMiniProgram` (line 28) as part of the `path` argument. At the UI level, the user is prompted only with the name of the destination miniapp (S8 in Figure 2) and is not informed that their phone number is being retrieved from storage (S9) and then sent out to another party (S10). As a result, the destination miniapp (Miniapp B) obtains the user’s phone number without user consent or a paid API request [26]. This unauthorized access enables Miniapp B to track, re-identify, or potentially phish the user, posing serious privacy and security risks.

D. Challenges

The motivating example reveals two major challenges in automatically detecting privacy leaks in miniapps.

Data Communication across Components and Layers. In miniapps, sensitive data often propagates across multiple parties, including the miniapp runtime, the miniapp server, and the superapp server. For example, the user event “tap” propagates from the *view* layer to the *logic* layer, forming an

```

1 /* STEP 1: pages/login/login.js. The user taps the
   ↪ "authorizePhoneNumber" button.*/
2 authorizePhoneNumber: function (n) {
3   if ("getPhoneNumber:ok" == n.detail.errMsg) {
4     var code = n.detail.phoneNumberCode;
5     o.post_ajax(
6       "/users/nnv2/encryptedData",
7       {"code":code, "sessionKey":"XXX"},
8       function (e) {
9         if (e.data.resInfo.phoneNumber) {
10           var o = e.data.resInfo.phoneNumber;
11           if (o.startsWith('+86')){ // check country
12             ↪ code
13             wx.setStorageSync("mobile",o)} } } } } }
14 /* STEP 2: pages/mine/mine.js. The user taps "
   ↪ goLearning" button and confirms
   ↪ navigateToMiniProgram but is unaware of
   ↪ the transmission of their phone number.*/
15 function y() {
16   return t().wrap(
17     function (t) {
18       for (;;)
19       switch ((t.prev = t.next)) {
20         case 6:
21           return (
22             (r = wx.getStorageSync("mobile")),
23             (u = "pages/index?mobile=".concat(r)),
24             (t.next = 14)
25           );
26         case 14:
27           (x = t.sent.data),
28           wx.navigateToMiniProgram({
29             appId: (x || {}).appid || "wxea2***1
30             ↪ e25", path: u });
31         case "end":
32           return t.stop(); } } } }
33 goLearning: function (t, e) {
34   return y.apply(this, arguments);
35 }

```

Listing 1: Vulnerable code for the miniapp *Yes Accounting Exam Study*. The two user actions occur on different pages, and the data flow involves server response and storage access.

inter-layer data flow. After login, the miniapp stores the user’s phone number in app-wide storage (line 12 in Listing 1) and retrieves it on a different page (line 22) when a subsequent event is triggered, forming an intra-layer data flow because both operations reside in the *logic* layer. Furthermore, this privacy leak involves client-server communication, where a sensitive value such as `phoneNumberCode` is sent from the client to the server, which then responds with sensitive data (e.g., the user’s phone number).

Conditional Privacy Leaks based on External Inputs.

Miniapp behavior and the program control flow depend on values from external inputs like user input and server responses. To reach privacy-leaking code, these values must satisfy certain conditions. In the client-server communication above, the server response must start with “+86” to trigger the `setStorageSync` statement (line 11-12), which is a necessary step in the privacy leak. Such dependent control flows introduce challenges for dynamic analysis. If an analysis relies

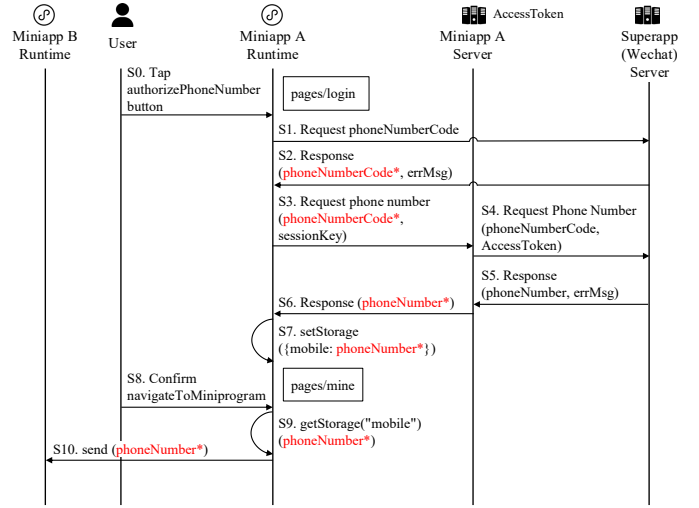


Fig. 2: Illustration of a privacy leak in the *Yes Accounting Exam Study* miniapp (labeled as Miniapp A). The leak requires two sequential user actions in Miniapp A: S0 on `pages/login` and S8 on `pages/mine`. Steps S0-S7 acquire the user’s phone number and save it in storage, and steps S8-S10 transmit the phone number externally to Miniapp B via `wx.navigateToMiniProgram`.

on predefined concrete values, many branches with specific conditions will not be exercised, leading to low coverage and false negatives.

E. Our Solution

We design MINNIE to solve these challenges. As illustrated in Figure 1, its core is an instrumented miniapp runtime using a unified shadow memory representation to support both holistic dynamic taint tracking and concolic execution. MINNIE addresses these challenges with the following mechanisms.

Holistic Dynamic Taint Propagation. MINNIE facilitates holistic dynamic taint propagation by synchronizing taint across components and layers. (1) Across components, MINNIE instruments both the JavaScript engine and miniapp framework to enable direct, variable-based taint propagation across components. With a unified taint representation, MINNIE is capable of tracking from source APIs in the miniapp framework, down to JS operations and built-ins, and finally back to sink APIs in the miniapp framework; (2) Across layers, MINNIE hooks serialization/deserialization processes to preserve taint information during message passing between *view* and *logic* layers, which run in separate JavaScript environments, enabling taint synchronization across JavaScript execution environments.

Concolic Execution. MINNIE incorporates concolic execution to systematically explore different code paths and branches that depend on external inputs. MINNIE models external inputs as symbolic values and collects immediate constraints (e.g., prefix or substring conditions). When encountering a conditional statement related to a concolic value, MINNIE solves the constraints and re-executes programs for different

branches. This approach improves coverage while validating generated inputs through concrete re-execution.

III. DESIGN

We present the design of MINNIE in this section. We begin with the system architecture in Section III-A, followed by our unified shadow memory design in Section III-B. We then describe dynamic taint tracking in Section III-C and concolic execution in Section III-D.

A. System Architecture

Figure 1 illustrates the architecture of MINNIE, which consists of two parts: an instrumented miniapp runtime (bottom) and a testing harness (top). The instrumented runtime forms the core of MINNIE: along the vertical stack (Section II-A), MINNIE instruments the miniapp framework to mark privacy-sensitive APIs (e.g., `wx.getUserInfo`) as taint sources and communication APIs (e.g., `wx.request`) as taint sinks. MINNIE also instruments the V8 JavaScript engine to (i) track byte-level taint through string operations such as `String.substring` and `JSON.stringify`, and (ii) synchronize taint across the horizontal *view-logic* boundary by hooking the (de)serialization that occurs when data is routed through the native JSBridge. On top of the instrumented runtime, the testing harness includes a UI exerciser that injects concolic text into interactive elements in the view layer, and a miniapp server that responds to network requests with concolic responses. Together, these components allow MINNIE to propagate taint holistically across components and layers and to systematically explore execution paths that depend on external inputs.

B. Unified Shadow Memory

The key idea of MINNIE’s design is its unified shadow memory representation, which enables both dynamic taint analysis and concolic execution within a single framework. Both techniques require maintaining and propagating metadata during program execution—specifically, taint labels for information flow tracking and symbolic marks for path exploration [27]. MINNIE adopts byte-level shadow memory for strings, which are the primary data type used in miniapp operations and API communications. As illustrated in Figure 3, each byte of a string is associated with one byte of shadow memory. The most significant bit serves as a symbolic mark indicating whether the byte represents a symbolic value subject to constraint solving. The four least significant bits encode the numeric identifier of the taint label, supporting up to 16 distinct values. We currently use 0 to denote untainted data, 1 to denote a generic taint without specific source attribution, and values 2–9 to denote sensitive source categories, as shown in Table X (Appendix B). The remaining identifiers are reserved for future extension.

This unified structure enables efficient simultaneous propagation of both taint and symbolic metadata during string operations. For instance, during string concatenation, MINNIE performs a single memory copy operation on the shadow bytes,

preserving both types of information without the overhead of maintaining separate tracking systems. This compact encoding provides precise per-byte source attribution with minimal memory overhead, while supporting the dual requirements of privacy leak detection and automated path exploration.

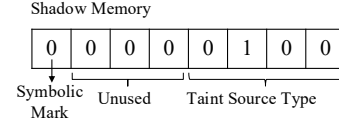


Fig. 3: A unified shadow memory representation in MINNIE.

C. Dynamic Taint Tracking

We first define the taint sources and sinks (Section III-C1), then present the taint propagation rules that MINNIE adopts to support taint propagation within and across components and layers in the miniapp architecture (Section III-C2).

1) *Taint Source and Sink Specifications*: The miniapp framework exposes a large number of APIs (479 for WeChat at the time of writing [28]) for developers, and several prior works defined their sets of source and sink APIs regarding privacy leaks. In this work, we selected sources and sinks in two steps. We first took the union of the source and sink APIs used by prior miniapp privacy studies [12], [13], [14], then manually reviewed all 479 APIs in WeChat’s official documentation [28] and added every API that either returns private user data (e.g., personal information, location) or transmits such data externally, such as over the network or via cross-miniapp communication. Two authors reviewed the API list independently and reconciled disagreements. In total, MINNIE tracks 38 source APIs, grouped into eight sensitive categories, and 13 sink APIs across the network and cross-miniapp channels. Table X gives the complete list. Each source category is assigned a numeric ID used as its taint label. Our sink set covers the network and cross-miniapp channels, but not other implicit channels like the clipboard or `<web-view>` components, and APIs that are undocumented or introduced after our review are not instrumented, which may lead to under-reporting.

TABLE I: Taint propagation mechanisms in miniapps and MINNIE’s design for handling each. MINNIE defines propagation rules (marked with $\star$) for mechanisms in the *Cross-Boundary* and *External Inputs* groups, formalized in Figure 4. Those in the *Direct* group are handled by MINNIE’s V8 string-level instrumentation.

Taint Propagation Mechanisms		MINNIE’s Design
Direct	Page/App-wide variable	V8 string-level taint propagation
	Encoding/decoding ops	$\star$ Byte-level taint redistribution
Cross-Boundary	App-wide storage	
	Page routing <i>view</i> $\rightarrow$ <i>logic</i> (UI event) <i>logic</i> $\rightarrow$ <i>view</i> (<code>setData</code>)	$\star$ Ser./Deser. taint synchronization
External Inputs	Server response	
	User input (input box, form)	$\star$ Concolic execution

2) *Taint Propagation*: We first summarize the various taint propagation mechanisms in miniapps in Table I. Apart from regular variable assignments where taint propagates

directly with values, two categories of mechanisms require MINNIE’s propagation rules: (i) *cross-boundary transformations*, where data crosses character, structural, or layer boundaries through encoding/decoding (e.g., `btoa`), serialization (e.g., `JSON.stringify`, app-wide storage, page routing), or message passing between view and logic layers, all of which cause value-level taint to lose byte correspondence; and (ii) *external inputs*, including server responses and user inputs, which influence control flow that fixed concrete inputs cannot exercise. MINNIE’s design addresses these mechanisms through the propagation rules shown in Table I, namely, byte-level taint redistribution, serialization/deserialization taint synchronization, and concolic execution. We formalize these rules through operational semantics over the JavaScript fragment relevant to the taint propagation, in the style of dynamic taint analysis formalizations such as Schwartz et al. [27].

Before presenting the rules, we introduce the notation, summarized in Table II. Configuration transitions are written $\langle e, \sigma \rangle \rightarrow \langle e', \sigma' \rangle$, indicating that under MINNIE’s instrumented semantics, expression e in state σ steps to e' in state σ' . The state $\sigma = (C, \Delta)$ pairs a JavaScript context C with a taint mapping Δ . The context C identifies the JavaScript execution environment: C_l is the shared *logic* context across all pages, while C_{v_i} is the *view* context of page i (each running in its own WebView/JS engine instance). The taint mapping Δ associates each variable with its current taint, and $\Delta[x \mapsto t]$ denotes updating Δ to bind variable x to taint t . The notation *call* $f(arg)$ denotes an API call site as it appears in the program source, subject to MINNIE’s instrumentation, while premises of the form $x = f(arg)$ describe the meta-level evaluation of f ’s native semantics. Auxiliary maps include $\Lambda[f]$, which assigns a taint source identifier to each source API f , and the API sets such as *SyncSrc*, *AsyncSrc*, *Sink* classify miniapp APIs by role.

Using this notation, we formalize the propagation rules in Figure 4, organized into three groups: SOURCE & SINK (framework-level instrumentation that marks the entry and exit of taint flows), CROSS-BOUNDARY PROPAGATION and EXTERNAL INPUTS (corresponding to the two groups in Table I). We explain each group below.

Source and Sink. Source initialization and sink checking are instrumented at the framework level. SOURCE INIT (SYNC) and SOURCE INIT (ASYNC) (Figure 4) handle the two API styles in WeChat: synchronous APIs return tainted values directly, while asynchronous APIs invoke success, fail, and complete callbacks with tainted arguments. SINK CHECKING verifies arguments via `checkTaint` before the sink executes; tainted arguments at a sink indicate a candidate sensitive-data flow.

Cross-Boundary Transformations. Within the runtime, taint must remain consistent when data goes through byte-level transformation across character, structural, or layer boundaries. ENCODING and DECODING propagate taint across reversible character-to-byte transformations such as `base64`. SERIALIZATION and DESERIALIZATION synchronize taint across the view-logic boundary (where data is implicitly

serialized when crossing) and across explicit calls to `JSON.stringify/JSON.parse`. In all four rules, taint is preserved at the byte level via MINNIE’s shadow memory: each byte segment in the output inherits the taint of its source segment in the input. The notation $x[i, j]$ and $x_{res}[i, j]$ refers to corresponding byte segments before and after the transformation; the index pair (i, j) is reused as a label for paired segments rather than as a literal byte position in both.

External Inputs. SERVER-RESPONSE and USER-INPUT (Figure 4) introduce symbolic values for concolic execution. The server-response rule mocks the network call via `mockRequest` and marks both the response and error as symbolic. MINNIE then explores the success and fail callbacks in separate executions and invokes `complete` accordingly, instead of committing to a single real network outcome. The user-input rule marks the user-supplied value as symbolic before dispatching the input event handler. Concolic execution itself is described in Section III-D.

D. Concolic Execution

Miniapps often receive external inputs from server responses (e.g., credential tokens, user information) and user input (e.g., input boxes and forms) that may introduce taint or affect the control flow. MINNIE applies concolic execution for these external inputs to increase code coverage and support holistic taint tracking. Specifically, each symbolic value is initialized as an empty string with a symbolic mark.

Our key insight is that the one-bit-per-byte symbolic mark modeling can effectively preserve symbolic information through common string operations by tracking symbolic values at the byte level. For example, consider `str=x+"\n";str=="admin"` where x is a symbolic variable. MINNIE reads each byte of `str` to determine whether it originates from a symbolic or concrete source. It then reconstructs the constraint as $(= (\text{str}++ x \text{"n"} \text{"admin"}))$ ¹ and sends it to the solver. As a result, the solver produces a solution such as `"admi"`. This byte-level symbolic marking enables MINNIE to precisely recover the constraint logic while avoiding the overhead of maintaining full symbolic expressions. Furthermore, the per-byte symbolic mark can even be more efficient in some operations. For instance, with statements `str="a"+x; str=="bn"`, the condition resolves to `false` instantly after comparing the first character, without solving `str`’s constraints.

1) *Concolic Operations:* With the one-bit-per-byte symbolic mark, MINNIE supports *byte-preserving* operations such as string concatenation, where symbolic marks retain symbolic influence across composed strings. For other operations like `str.startsWith(prefix)` or `str.includes(x)`, MINNIE also derives corresponding Z3 expressions by reading the symbolic mark of each string byte.

MINNIE also supports property access on symbolic values (e.g., `x[prop]` where x is symbolic), which is especially

¹This is in SMT-LIB syntax offered by Z3, where `str++` concatenates two string operands.

SOURCE & SINK $\frac{\sigma=(C_l, \Delta), t=\Delta[f], f \in \text{SyncSrc}, x_{\text{ret}}=f(\text{arg})}{\langle \text{call } f(\text{arg}), \sigma \rangle \rightarrow \langle \text{setTaint}(x_{\text{ret}}, t), \sigma \rangle} \text{SOURCE INIT (SYNC).}$ $\frac{\sigma=(C_l, \Delta), t=\Delta[f], f \in \text{AsyncSrc}, \text{arg}=\{\text{success}:f_{\text{succ}}, \text{fail}:f_{\text{fail}}, \text{complete}:f_{\text{comp}}\}, (x_{\text{res}}, x_{\text{err}})=f(\text{arg})}{\langle \text{call } f(\text{arg}), \sigma \rangle \rightarrow ((\text{if } x_{\text{res}} \text{ then } \text{setTaint}(x_{\text{res}}, t); f_{\text{succ}}(x_{\text{res}}) \text{ else } f_{\text{fail}}(x_{\text{err}})); f_{\text{comp}}(x_{\text{res}}, x_{\text{err}}), \sigma)} \text{SOURCE INIT (ASYNC).}$ $\frac{\sigma=(C_l, \Delta), f \in \text{Sink}, x_{\text{ret}}=f(\text{arg})}{\langle \text{call } f(\text{arg}), \sigma \rangle \rightarrow \langle \text{checkTaint}(\text{arg}); x_{\text{ret}}, \sigma \rangle} \text{SINK CHECKING}$	
CROSS-BOUNDARY TRANSFORMATIONS $\frac{\sigma=(C, \Delta), f \in \text{Encode}, t=\text{getTaint}(x), x_{\text{res}}=f(x)}{\langle \text{call } f(x), \sigma \rangle \rightarrow \langle x_{\text{res}}, (C, \Delta[x_{\text{res}} \mapsto t]) \rangle} \text{ENCODING} \quad \frac{\sigma=(C, \Delta), f \in \text{Decode}, x_{\text{res}}=f(x), t_i=\Delta[x_i], x_i=\text{enc}(x_{\text{res}}[i])}{\langle \text{call } f(x), \sigma \rangle \rightarrow \langle x_{\text{res}}, (C, \Delta[x_{\text{res}}[i] \mapsto t_i] \text{ for } x_i \in x) \rangle} \text{DECODING}$ $\frac{\sigma=(C, \Delta), f \in \text{Serialize}, x_{\text{res}}=f(x), t_i=\Delta[x[i, j]]}{\langle \text{call } f(x), \sigma \rangle \rightarrow \langle x_{\text{res}}, (C, \Delta[x_{\text{res}}[i, j] \mapsto t_i] \text{ for } x_{\text{res}}[i, j] \in x_{\text{res}}) \rangle} \text{SER.} \quad \frac{\sigma=(C, \Delta), f \in \text{Deserialize}, x_{\text{res}}=f(x), t_i=\Delta[x[i, j]]}{\langle \text{call } f(x), \sigma \rangle \rightarrow \langle x_{\text{res}}, (C, \Delta[x_{\text{res}}[i, j] \mapsto t_i] \text{ for } x[i, j] \in x) \rangle} \text{DESER.}$	
EXTERNAL INPUTS $\frac{\sigma=(C_l, \Delta), \text{arg}=\{\text{url}:u, \text{data}:d, \text{success}:f_{\text{succ}}, \text{fail}:f_{\text{fail}}, \text{complete}:f_{\text{comp}}\}}{(x_{\text{res}}, x_{\text{err}})=\text{mockRequest}(u, d), \text{setSymbol}(x_{\text{res}}), \text{setSymbol}(x_{\text{err}}), t=\text{getTaint}(d)} \text{SERVER-RESPONSE}$ $\frac{\sigma=(C_v, \Delta), e=\{\text{detail}:\{\text{value}:x\}\}, f=\text{getHandler}(el, \text{"input"})}{\langle \text{trigger}(el, \text{"input"}), \sigma \rangle \rightarrow \langle \text{setSymbol}(x); f(e), \sigma \rangle} \text{USER-INPUT}$	

Fig. 4: Operational semantics of MINNIE’s taint propagation rules, organized by category: source/sink instrumentation, cross-boundary propagation (encoding, decoding, serialization, deserialization), and external inputs (server response, user input). Notation is defined in Table II and Section III-C2.

important for handling server responses that are often structured as objects. When such a property access occurs, MINNIE transitions the symbolic value x into an object and creates the accessed property with a default symbolic value. This process is recursive: when nested property accesses occur, newly created symbolic properties are themselves treated as symbolic objects, enabling symbolic tracking to propagate across deep object structures. So in the case of the motivating example, o is resolved to a symbolic string (line 10). On the other hand, for *non-byte-preserving* transformations (e.g., `replace`, `split`), taint propagation is preserved, but the loss of symbolic precision may prevent MINNIE from reaching branches guarded by them. Table XI (Appendix C) lists operations supporting precise symbolic constraints and those retaining taint only.

2) *Constraint Solving and Forking*: MINNIE instruments the JavaScript engine’s bytecode interpreter to identify conditional expressions. Conditional expressions are recognized by inspecting the relations between bytecodes. For example, when the current bytecode is a test operator such as `TestEqual` and the next bytecode is a control-flow related operator, such as `JumpIfTrue`. When MINNIE encounters a branch condition involving a string, it reads the symbolic mark of each byte, constructs the expression, and sends it to Z3 for solving. After obtaining a satisfying value for each branch, MINNIE reconstructs the original symbolic input (e.g., the full server response object) by embedding the solved value at the appropriate property, with other symbolic properties filled by default symbolic values (empty strings with symbolic marks) that can be solved lazily upon their own access. MINNIE then re-triggers the corresponding UI event or network request with the reconstructed input, allowing the program to execute along the chosen branch. Both branches are explored by re-triggering with each satisfying value.

E. A Walkthrough Example

We illustrate MINNIE’s end-to-end taint tracking using the phone number leak from the motivating example (Listing 1, Section II-C). The leak spans two pages, an asynchronous source, app-wide storage, and a cross-miniapp sink, exercising five of the nine rules in Figure 4: SOURCE INIT (ASYNC), SERVER-RESPONSE, SER., DESER., and SINK CHECKING.

Step 1: Source initialization. When the user taps the `authorizePhoneNumber` button (line 2 of Listing 1), the framework invokes the `getPhoneNumber` API. MINNIE recognizes `getPhoneNumber` as an asynchronous source ($f \in \text{AsyncSrc}$) and applies the SOURCE INIT (ASYNC) rule, attaching the Phone taint label (ID 6) to the returned `phoneNumberCode` via `setTaint` before invoking the success callback.

Step 2: Server response and concolic execution. The miniapp sends `phoneNumberCode` (tainted) to its own server via `post_ajax` (line 5). MINNIE mocks the server response via `mockRequest` and applies the SERVER-RESPONSE rule: it marks the response e as symbolic via `setSymbol`. When the miniapp accesses nested properties `e.data.resInfo.phoneNumber` (line 9-10), MINNIE lazily propagates the symbolic mark: each property access on a symbolic value creates a new symbolic property, so `e.data`, `e.data.resInfo`, and finally `e.data.resInfo.phoneNumber` (referred to as o) all become symbolic. When execution reaches the conditional `o.startsWith('+86')` (line 11), MINNIE’s concolic engine (Section III-D) reads the symbolic marks of o , constructs the Z3 constraint (`str.prefixof "+86" o`), and solves it for both outcomes (e.g., `" +86..."` for true and `" +1..."` for false). MINNIE then reconstructs the response object e for each outcome, embedding the concretized o at `e.data.resInfo.phoneNumber`. Other unconstrained

TABLE II: Notation used in MINNIE’s operational semantics.

Name	Description
<i>State</i>	
σ	The current state, decomposed as $\sigma = (C, \Delta)$.
C	A JavaScript context (one per WebView/JS engine). C_l : the <i>logic</i> context. C_{v_i} : the <i>view</i> context for page i . C : either <i>view</i> or <i>logic</i> .
Δ	Map from a variable to its taint.
$\Delta[x \mapsto t]$	Update Δ to bind variable x to taint t .
<i>Notation</i>	
$\langle e, \sigma \rangle \rightarrow \langle e', \sigma' \rangle$	Configuration $\langle e, \sigma \rangle$ transitions to $\langle e', \sigma' \rangle$ under MINNIE’s instrumented semantics.
$\text{call } f(\arg)$	A call site in the program source, subject to MINNIE’s instrumentation.
$x = f(\arg)$	In a premise: meta-level evaluation of f ’s native semantics, independent of instrumentation.
<i>API Sets</i>	
<i>SyncSrc</i>	The set of synchronous source APIs.
<i>AsyncSrc</i>	The set of asynchronous source APIs.
<i>Sink</i>	The set of sink APIs.
<i>Encode</i>	The set of encoding APIs (e.g., <code>btoa</code>).
<i>Decode</i>	The set of decoding APIs (e.g., <code>atob</code>).
<i>Serialize</i>	The set of serialization APIs (e.g., <code>JSON.stringify</code>); also includes implicit serialization across the view-logic boundary.
<i>Deserialize</i>	The set of deserialization APIs.
$\Delta[f]$	The taint source identifier for source API f .
<i>Operations</i>	
$\text{setTaint}(x, t)$	Tag x with taint t .
$\text{getTaint}(x)$	Retrieve x ’s current taint.
$\text{checkTaint}(x)$	Check x ’s taint at a sink and report violations.
$\text{setSymbol}(x)$	Mark x as a symbolic value for concolic execution.
$\text{getHandler}(el, evt)$	Retrieve the registered handler for event evt on element el .
$\text{mockRequest}(u, d)$	Mock a network request to u with payload d , returning a symbolic response/error pair.
$\text{enc}(\cdot)$	The encoding operation that decoding APIs invert (used in the Decoding rule).
<i>Variables</i>	
$x, \arg, e$	Values (byte, string, number, array, object) and event objects.
t, t_i	Taint values.
$f_{\text{succ}}, f_{\text{fail}}, f_{\text{comp}}$	Success, fail, and complete callbacks of an asynchronous API.
$x[i, j], x_{\text{res}}[i, j]$	Corresponding byte segments related by a string transformation; the index pair (i, j) is reused as a label, not a literal position in both.

properties retain their default symbolic values (empty string with symbolic mark) and are solved upon later access or comparison. MINNIE re-triggers the `authorizePhoneNumber` event with the mock server returning the reconstructed objects e , exploring both branches. The true branch reaches `wx.setStorageSync` (line 12).

Step 3: Cross-boundary serialization. At line 12, `wx.setStorageSync` serializes the tainted phone number for storage. The SER. rule synchronizes byte-level taint: each byte of the serialized representation in storage inherits the taint of its source bytes in `phoneNumber`, preserving correspondence across the storage boundary.

Step 4: Storage retrieval and propagation. On a different page, at line 22, `wx.getStorageSync("mobile")` retrieves the tainted phone number; MINNIE applies

the DESER. rule to map storage taint back to the runtime value r . Subsequent string concatenation `"pages/index?mobile="+r` (line 23) propagates taint via V8’s instrumented `String.concat`, producing a tainted URL.

Step 5: Sink checking. Finally, in this same page, `wx.navigateToMiniProgram` is called with the tainted URL as path (line 28). MINNIE applies the SINK CHECKING rule, invoking `checkTaint` on the arguments. The Phone taint is detected, and MINNIE reports the leak: phone number flows from `getPhoneNumber` (source) to `wx.navigateToMiniProgram` (sink) across two pages, storage, and reaches a cross-miniapp sink API.

F. UI Exerciser

The UI exerciser explores pages and UI events in a miniapp to expose privacy leaks. During testing, the schedulers dynamically adjust the execution order of events and pages to improve coverage.

1) Data Dependencies between UI Events and Pages:

MINNIE tracks data dependencies between UI events and pages using the Event Dependency Graph (EDG) and the Page Routing Graph (PRG). The EDG captures data flows between UI events through shared variables or storage, while the PRG records routing transitions and the data being passed. For the EDG, MINNIE identifies a data dependency when one event sets/writes a value and another event later reads the same value. For example, a text input event is followed by a tap event that depends on the input value. Note that we only consider events that are triggered separately; otherwise, their dependencies are already resolved by MINNIE’s taint engine and do not require additional scheduling. Similarly, the PRG connects pages based on observed routings that are invoked with separate UI events. This helps MINNIE to optimize event/page coverages.

2) Scheduling Algorithm: We present the scheduling algorithm in Algorithm 1, which takes the initial page queue PgQ and the event dictionary EvD as input. PgQ is the list of pages extracted from the miniapp’s configuration file, and initially prioritized based on sensitive API appearance. EvD is a map from a page to the events on that page. Each page’s events and handlers are extracted by matching bindings in the *view* layer (e.g., `<button bindtap="handleTap">`) to their handlers in the *logic* layer (e.g., `function handleTap()`). The events are initially prioritized heuristically with an order of input/form, tap, other events, and finally page routings. As the exercising starts, MINNIE fetches the top page from PgQ and adopts a depth-first search (DFS) exploration strategy on the events (lines 3-4). In the exploration, MINNIE iterates through the events on the current page *page* (line 8-9) and checks for event dependencies (line 10). For dependent events, MINNIE recursively triggers them again and updates the EDG (line 11-15). If a page routing happens, MINNIE updates the PRG, reorders the page queue PgQ , and explores the new page with the same strategy (line 16-21). When all the events on the

Algorithm 1: Algorithm for page and event scheduling.

```
Input : Page queue  $PgQ$ , Event dic  $EvD$ 
1   $EDG \leftarrow \{ \}, PRG \leftarrow \{ \}$ 
2  while  $PgQ \neq \emptyset$  do
3     $page \leftarrow \text{pop}(PgQ)$ 
4     $DFSE\text{ExplorePage}(page)$ 
5  end
6  function  $DFSE\text{ExplorePage}(page)$  :
7    while  $EvD[page] \neq \emptyset$  do
8       $event \leftarrow \text{pop}(EvD[page])$ 
9       $\text{Trigger}(event)$  // Exercise a UI event
10      $depEvList \leftarrow \text{getWriteDependence}(event)$  // find
        dependent events of writes
11     for  $e \in depEvList$  do
12        $\text{Trigger}(e)$ 
13        $\text{IteEDG}(e)$  // iteratively trace back EDG to
        exercise dependent events
14        $EDG \leftarrow EDG \cup \{(event, e)\}$ 
15     end
16     if  $\text{routingTo}(page2)$  then
17        $PRG \leftarrow PRG \cup \{(page, page2)\}$ 
18        $\text{enqueue}(PgQ, page)$ 
19        $\text{PageSchedule}(PgQ)$  // Reorder pages
20        $DFSE\text{ExplorePage}(page2)$ 
21     end
22   end
23    $PgQ.\text{remove}(page)$ 
24 end
```

current page have been triggered at least once, the exploration of the page finishes (line 7-23).

IV. IMPLEMENTATION

MINNIE instruments both the V8 JavaScript engine and the WeChat miniapp framework to enable comprehensive taint tracking. We adopt a two-level instrumentation strategy: first, the V8 JavaScript engine provides fundamental byte-level taint tracking and concolic execution primitives that are generic across JavaScript programs; second, the WeChat miniapp framework handles source initialization and sink checking, since all sensitive APIs are defined at the framework level rather than in the core JavaScript language. This separation may allow the V8-level instrumentation to transfer to other V8-based miniapp platforms, while the framework-level instrumentation remains platform-specific.

V8 Engine Instrumentation. WeChat miniapps use V8 [29] as their JavaScript engine. We build on prior open-source dynamic taint tracking in Chromium [20], [30], porting and adapting the byte-level taint propagation primitives to V8 in Chromium 92.0.4515.107 to match the version used by WeChat. MINNIE extends these primitives with a unified shadow memory representation combining taint labels with symbolic marks (Section III-B), V8 bytecode interpreter hooks for concolic constraint collection, and Z3 integration for path forking. The patch covers byte-level shadow memory management, instrumentation for string operations (e.g., `substring`, `concat`), bytecode interpreter hooks, and Z3 integration.

MINNIE exposes three JavaScript APIs callable from within JavaScript code: `setTaint()` initializes taint on a string, `getTaint()` retrieves a string's taint, and `checkTaint()` verifies whether a string carries specific taint. Each byte of

taint metadata uses its four least significant bits to represent 8 sensitive categories and a generic taint label (see Appendix B). To handle non-string types such as numbers, we convert them to strings, attach taint to the converted representation, and convert them back at sinks.

Framework-Level Source and Sink Instrumentation.

Source initialization and sink checking occur at the framework level. For source APIs, MINNIE rewrites the callback to tag the returned data with `setTaint()`. For sink APIs, MINNIE hooks the execution and calls `checkTaint()` on relevant parameters before the API runs. For example, `wx.navigateToMiniProgram` is checked on `path` and `extraData` (which may leak data to another miniapp) but not `appId` (which only specifies the target).

UI Exerciser. MINNIE's UI exerciser is built on Minium [31], WeChat's official automation testing framework, which provides programmatic access to UI elements and triggers events such as taps, scrolls, and text inputs. On top of Minium, MINNIE implements the event scheduler, page scheduler, and concolic input injection. The schedulers maintain the EDG and PRG (Section III-F) and reorder events and pages dynamically based on observed dependencies.

V. EVALUATION

We structure our evaluation of MINNIE around the following Research Questions (RQs).

RQ1: Zero-Days. How many zero-day privacy leaks does MINNIE detect? How prevalent are privacy leaks in the WeChat ecosystem? What are the category and popularity distributions of the analyzed miniapps?

RQ2: FPs and FNs. What are the false positives and false negatives of MINNIE?

RQ3: Coverage. What code coverage does MINNIE achieve, in terms of page exploration, element interaction, event-binding function calling, and code coverage?

RQ4: Component Ablation. What are the results of MINNIE without specific components, such as unified shadow memory, cross-layer propagation, concolic execution, and scheduler-based testing?

RQ5: Performance Overhead. What is the performance overhead of MINNIE?

A. Experimental Setup

1) *Datasets:* We use the following datasets.

Large-scale Real-world Dataset. We performed our own large-scale data collection because MiniCrawler [10] has become obsolete. The collection included extracting keywords from the Miniapp Document [32] using jieba [33] tokenizer, and using Appium [34] to automatically search for these keywords in WeChat on Android. We developed an LSpoused [35] module to capture miniapp metadata such as name, ID, and ratings, as well as downloading the miniapp packages, which

TABLE III: [RQ1] Selected zero-day privacy leaks detected by MINNIE. We have responsibly disclosed all listed leaks to the WeChat platform. See the Ethical Considerations section.

Miniapp ID	Miniapp Name	Rating Score	# Ratings	Actual Privacy Leak	Source	Sink
wxaf35****0b2a	DiDi Chuxing	4.3	1,093,010	User ID	wx.getUserInfo	wx.openEmbeddedMiniProgram
wx66a9****b9ae	WeBank	2.4	158,558	User ID	wx.getStorage	wx.navigateToMiniProgram
wx114b****0eeb	Guangzhou Metro	4.2	61,865	User ID	wx.getUserInfo	wx.navigateBackMiniProgram
wxa6d4****62bc	Safety Education Platform	3.8	40,806	Name, ID Number, Phone	wx.getUserInfo	wx.navigateToMiniProgram
wx26da****1226	Tencent QQ	2.6	34,534	Name, User ID	wx.getUserInfo	wx.navigateToMiniProgram
wx20d4****d636	PICC Property and Casualty	3.8	32,775	ID Number	wx.getUserInfo	wx.navigateBackMiniProgram
wxb582****f01a	ShanSong Delivery	4.7	25,800	User ID, City, Address	wx.getStorage	wx.request
wxb937****117e	Hellobike	2.2	23,580	User Profile	wx.getUserInfo	wx.navigateToMiniProgram
wx6f17****ca30	ICBC e Life plus	3.8	15,035	User ID	wx.getStorage	wx.navigateToMiniProgram
wx9fd6****c5f9	QQ Music	3.5	12,693	Location	wx.startLocationUpdate	wx.sendSocketMessage

were later extracted from the Android device. The dataset contains 12,009 unique miniapps. We further collected privacy policies from all 12,009 miniapps in our large-scale dataset. We found that 10,625 out of 12,009 miniapps (88.5%) provide privacy policies, with 193 miniapps (1.6%) providing additional custom privacy policies (e.g., feature-specific policies for location or payment). This high policy-availability rate makes the dataset suitable for policy-based privacy leak validation. Our large-scale experiments are conducted on the dataset of 10,625 miniapps that provide privacy policies. We show the distribution of miniapps ratings in Table IV. Notably, 216 miniapps have over 10,000 ratings, representing the top 2.03% of the dataset. These miniapps with over 10,000 ratings include widely used and recognized ones such as Meituan (food delivery) and DiDi Chuxing (rideshare). We also list the top categories of analyzed miniapps in the same table. As shown, the dataset spans WeChat’s full category taxonomy, led by E-commerce & Shopping (11.29%), Business & Enterprise (7.99%), Education (7.15%), Transportation & Travel (7.08%), and Government & Public Services (7.07%).

MiniTracker Dataset. We utilized MiniTracker [13] curated benchmark dataset, with 65 cases (54 positive, 11 negative). The dataset consists of crafted code snippets and simplified real-world implementations with complex JavaScript dynamic behaviors. After manual validation, we excluded 12 cases because they are either unexecutable or do not contain valid privacy leaks. In the end, the refined dataset has 53 cases with 45 positives and eight negatives.

2) *Running environment:* We collected the data on WeChat 8.0.49, running on a rooted Android Pixel 6a. The evaluation was conducted on WeChat Devtools 1.06.2310080, running on a server with 128 GB of memory, 20 Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz cores, and Ubuntu 22.04.5 LTS.

B. RQ1: Zero-Day Privacy Leaks & Impacts

We now discuss the zero-day privacy leaks detected by MINNIE. We define a zero-day privacy leak as one that satisfies both of the following conditions: (1) it is not reported and cannot be detected by prior works including TaintMini [12], MiniTracker [13], and MiniScope [14], and (2) it violates the miniapp’s privacy policy.

Overall results. We ran MINNIE and all prior works on the large-scale dataset with 10,625 miniapps that come with a privacy policy, with a time budget of five minutes per miniapp.

TABLE IV: [RQ1] Characterization of the analyzed (10,625) and flagged (1,047) miniapps by category and popularity.

	All (10,625)	Flagged (1,047)
<i>Top categories</i>		
E-commerce & Shopping	11.29%	13.17%
Business & Enterprise	7.99%	16.62%
Education	7.15%	6.68%
Transportation & Travel	7.08%	8.47%
Government & Public Services	7.07%	6.77%
<i>Popularity (number of ratings)</i>		
No or hidden ratings	44.98%	35.34%
1–999	46.32%	53.01%
1,000–9,999	6.67%	8.31%
≥10,000	2.03%	3.34%

Then we filtered out flows that are detected by any of the three prior works and excluded cross-miniapp leak cases where source and destination miniapps are registered by the same operator. To validate whether detected flows constitute privacy leaks, we followed the methodology of prior work [15], which uses large language models (LLMs) to extract data collection and sharing promises from privacy policy texts, then cross-checks these against the sources and sinks of detected data flows. Through this analysis, MINNIE flagged 5,501 candidate zero-day privacy leaks across 1,047 miniapps.

Manual validation of the policy cross-check. We used the pipeline of Wang et al. [15], which uses an LLM to extract the collection and sharing statements from a privacy policy and match them against reported data flows, to check whether each flow is disclosed by its miniapp’s policy. To measure the precision of this pipeline, we drew a random sample of 100 miniapps from the 1,047 flagged miniapps, with 609 flagged flows in total. Two authors independently compared each flow against the raw policy text of its miniapp and labeled it *confirmed* (the policy discloses neither the collection nor the sharing that the flow performs) or *disclosed* (the policy does cover it, so the pipeline reports a false positive). Disagreements were resolved by discussion. Of the 609 flows, 497 (81.6%) were confirmed as genuine privacy-policy violations, and 83 of the 100 miniapps contained at least one confirmed leak. All 112 unconfirmed flows correspond to specific disclosures present in the raw policy but were missed by the LLM.

Impact on real-world miniapps. The 1,047 MINNIE-flagged miniapps span the same major categories as the overall dataset,

with Business & Enterprise (16.62%) as the largest category, as seen in Table IV. Besides, 35 of flagged miniapps (3.34%) have more than 10,000 ratings, slightly above the 2.03% share of the corpus at that level. Many well-known companies, despite their reputation and large user bases, operate miniapps that leak private user data in violation of their own policies. Table III shows selected examples of zero-day leaks detected by MINNIE. Specifically, DiDi Chuxing [23], a ride-sharing service miniapp with over one million user ratings, implying a substantially larger user base, was found to leak User ID, which can often be exploited to link individual user profiles or activities. WeBank [24], a financial service miniapp, also leaks user IDs. Interestingly, we found that even miniapps developed by Tencent [36] (the company that operates WeChat)—such as Tencent QQ [37]—leaked multiple types of sensitive user data, despite operating within Tencent’s own ecosystem. This suggests that privacy risks are systemic and not limited to third-party developers, raising concerns about insufficient platform-level analysis, even by the platform provider itself. The results also include several government miniapps, such as those for tax payments or high courts, for which Tencent does not disclose the number of user ratings. Privacy leaks in these miniapps are particularly concerning, as users are more likely to trust official services and willingly authorize access to their private data.

Source and sink characterization. We characterize the taint sources and sinks of the flagged candidate leaks in Table V. The Usage column shows the number of miniapps that use the corresponding APIs. Among the taint sources, storage APIs are the most common source category among detected leaks, as shown in the first row of the table. Some miniapps face location and device information leaks, which may be more severe than storage leaks since such information can be directly used for entity identification. Regarding the sink channels, flows through cross-miniapp channels are much less common than those through network channels. After excluding three same-operator cases, the retained cross-miniapp leaks involve 15 miniapps spanning four categories, with ratings ranging from 172 to over 1,000,000. All retained source and destination miniapps are registered by different operators.

Multi-source leaks. Figure 5 shows the distribution of the number of distinct sensitive source categories present in a single leaked payload. We observe that a majority of candidate zero-day leaks involve multiple source categories, with nearly 40% of leaks combining four or more distinct categories, indicating that privacy leaks in practice are often richly structured sensitive data rather than isolated values. This multiple-source complexity of real-world privacy leaks also demonstrates the necessity of designing an automated detection approach like MINNIE. Note that when multiple sources influence the same byte, MINNIE keeps only one label, as the encoding is not one-hot. The leak is still detected but attributed to one source.

C. RQ2: False Positives and False Negatives

We evaluate the false positives (FPs) and false negatives (FNs) of MINNIE compared to three prior works: Taint-

TABLE V: [RQ1] Distribution of MINNIE-flagged miniapps by taint source category and sink type. For each source category, we show an example API, the number of miniapps using APIs in that category, the number of miniapps in which data from that category reaches a sink, and a breakdown of the leaking miniapps by network and cross-miniapp channels. Percentages are computed over the number of miniapps using the corresponding source category.

Source	APIs (Example)	# Usage	# Leak	Network		Cross-miniapp	
				#	%	#	%
Storage	<code>wx.getStorageSync</code>	9,773	539	529	5.41	10	0.10
Profile	<code>wx.getUserInfo</code>	7,108	433	428	6.02	5	0.07
Location	<code>wx.getLocation</code>	5,344	402	402	7.52	0	0.00
Media	<code>wx.chooseVideo</code>	4,767	268	268	5.62	0	0.00
Device	<code>wx.getNetworkType</code>	3,999	58	58	1.45	0	0.00

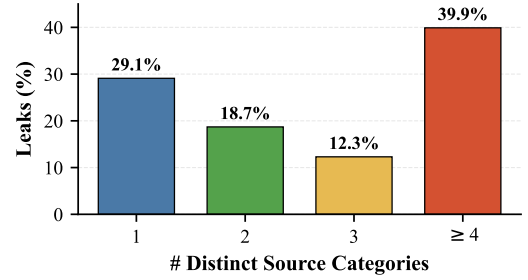


Fig. 5: [RQ1] Distribution of the number of distinct sensitive source categories present in a single leaked payload. Around 40% of candidate zero-day leaks combine four or more categories of sensitive information within the same transmission.

Mini [12], MiniScope [14] and MiniTracker [13]. We use the MiniTracker dataset and 100 randomly selected miniapps from the large-scale dataset.² We ran the four tools on same datasets with the same time limit of five minutes per miniapp, and manually inspected the miniapp code to verify their reports, with detailed results in Table VI.

100 Random Miniapp Dataset. We build the groundtruth set of 100 random miniapps as follows. We ran the four tools on 100 randomly sampled miniapps with a five-minute timeout per miniapp and collected every reported privacy-leak flow. We then took the union of all flows reported by any tool, yielding 264 flows in total, spanning 33 of the 100 miniapps. Two authors independently inspected each flow in the miniapp code and labeled it a true leak or not, and disagreements were resolved by discussion. 214 of the 264 were confirmed as true leaks. This union-based ground truth may overestimate absolute recall because leaks missed by all four tools cannot enter the reference set. We therefore interpret recall relative to this common reference set rather than as absolute recall.

As shown in Table VI, MINNIE identifies 146 true positives with 14 FPs and 68 FNs, achieving the highest precision and recall among all evaluated tools. The FNs of MINNIE primarily result from incomplete dynamic exploration due to execution paths guarded by complex constraints, such as mandatory account registration and login. In these cases,

²We did not evaluate the entire large-scale dataset because obtaining the ground truth requires substantial manual effort.

MINNIE cannot reach certain functionalities, although it still detects taint flows on pages accessible without authentication. The 68 FNs span eight miniapps. We collected the per-miniapp testing logs and excluded one that completed exploration in under 184s, which is well within the five-minute budget, then re-ran the remaining seven with a one-hour budget. MINNIE did not report any additional flows under the one-hour limit. This result suggests that extending the time budget alone does not resolve these FNs. The FPs of MINNIE mainly come from conservative taint propagation across server responses, where the returned object is tainted as a whole even when only specific properties are tainted.

In contrast, MiniTracker and TaintMini exhibit both high false positive and false negative rates. By inspecting their reports, we observe that both tools are effective at recognizing simple and local data flows, such as cases where sources and sinks appear within the same function. However, they suffer from imprecise taint propagation and frequently over-taint intermediate values, leading to a large number of FPs. Moreover, both tools struggle to capture data flows involving dynamically imported functions, runtime-resolved callbacks, or heavily obfuscated code, resulting in substantial FNs. Besides, MiniScope also exhibits a large number of FNs. A major source of this limitation is its reliance on static analysis to associate UI events with their corresponding callbacks, which are often dynamically resolved at runtime in miniapps. For example, given an event binding such as `bindtap="toggle"`, MiniScope attempts to statically resolve the callback name `toggle`, but this resolution often depends on runtime object state or dynamic binding, which cannot be reliably inferred statically. In contrast, MINNIE avoids this by directly exercising UI actions via a UI automator and observing the actual callback execution through dynamic taint tracking. This design enables MINNIE to capture privacy leaks that depend on dynamic event resolution.

MiniTracker Dataset. The MiniTracker dataset contains 53 labeled cases, including 45 positives and eight negatives. As shown in Table VI, TaintMini reports 12 taint flows, correctly identifying ten true positives and producing two FPs, while missing 35 positive cases. MiniScope reports 21 flows, of which 19 are true positives and two are FPs, resulting in 26 FNs. MiniTracker reports 41 flows, identifying 37 true positives and four FPs, but still misses eight positive cases.

In comparison, MINNIE identifies 42 true positives with only three FNs and correctly distinguishes all eight true negatives, yielding zero FPs on this dataset. The remaining FNs of MINNIE are caused by execution paths that are difficult to exercise automatically within the given time limit.

The FNs of TaintMini, MiniScope, and MiniTracker are primarily due to two limitations: (i) inadequate modeling of miniapp runtime data flows across multiple execution layers, and (ii) insufficient handling of JavaScript’s dynamic features, which is further exacerbated by widespread code minification in real-world miniapps. Overall, MINNIE consistently outperforms prior approaches on this dataset by significantly

TABLE VI: [RQ2] Detection results comparing MINNIE with three prior works on two datasets.

Approach	100 Random Miniapp Dataset			MiniTracker Dataset			
	# TP	# FP	# FN	# TP	# FN	# TN	# FP
TaintMini [12]	7	7	207	10	35	6	2
MiniScope [14]	16	12	198	19	26	6	2
MiniTracker [13]	60	24	154	37	8	4	4
MINNIE	146	14	68	42	3	8	0

TABLE VII: [RQ3] Comparison of different page testing strategies in MINNIE with a time limit of ten minutes on 100 random miniapps. # represents the number of triggered items, and % represents the coverage percentage relative to the total.

Strategy	Pages*		Elements†		Bindings†		# Flows
	#	%	#	%	#	%	
Monkey Testing	819	59.87	378	40.00	1,285	31.45	44
Scheduler-based	987	72.15	547	57.88	2,411	59.01	66

* Total pages (1,368) are collected from each miniapp’s app.json file.

† Total interacted input/form elements (945) and triggered binding functions (4,086) are collected dynamically upon page visiting.

reducing both FPs and FNs. This improvement is mainly attributed to MINNIE’s unified taint engine, which tracks taint flows across multiple miniapp layers, as well as its dynamic execution model that naturally captures runtime behaviors.

D. RQ3: Coverage

We perform experiments to measure MINNIE’s coverage and illustrate the effectiveness of the page/event scheduler-based testing strategy. As a baseline, we implemented a monkey-testing approach (i.e., random page interactions) following the strategy of gremlins.js [38] using Minium [31].

To demonstrate the effectiveness of the event/page scheduler-based strategy, we randomly selected 100 miniapps from the dataset and recorded the total number of visited pages, triggered input/form elements, triggered binding functions, and detected taint flows after running for ten minutes with both strategies. As shown in Table VII, the scheduler-based strategy visited 20.5% more pages than monkey testing. On each visited page, the monkey testing triggered 40.00% of input/form elements and 31.45% of binding functions, while the scheduler-based strategy triggered 57.88% of input/form elements and 59.01% of binding functions—17.88% and 27.56% higher, respectively. This also enabled MINNIE to detect 22 additional taint flows.

In Figure 6, we further show the metrics at five-second intervals over a ten-minute run on three randomly selected miniapps. Our observations are as follows. First, in most of the testing period, the scheduler-based testing strategy consistently triggers more input/form elements and binding functions than monkey testing. This is because the event/page scheduler is designed to maximize code coverage on each page. Upon page load, the event scheduler generates a prioritized queue of events, focusing on input/form-related events, and dispatches them sequentially. When a triggered event causes page navigation, the page scheduler saves the current page’s remaining elements and comes back once the

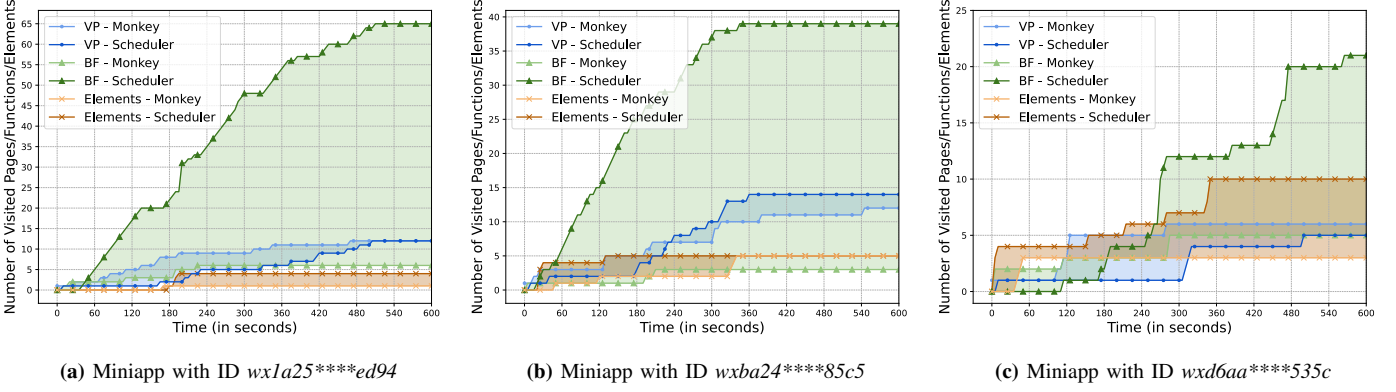


Fig. 6: [RQ3] A comparison of different UI testing strategies (i.e., monkey testing and scheduler-based strategy). The terms VP, BF, and Elements in the legend represent Visited Pages, Triggered Binding Functions, and Triggered Input/Form Elements, respectively. The shaded area between lines shows the gap between the two strategies over the same metric and time period.

event completes. In contrast, the stateless nature of monkey testing causes repeated interactions with the same elements instead of exploring new ones.

Second, while the monkey testing strategy may visit more pages than the scheduler-based strategy within the same time frame, both strategies tend to cover a similar number of pages by the end of testing. This difference arises because the monkey testing strategy would randomly navigate to different pages, even if the current page has not been fully tested. In contrast, the scheduler-based strategy prioritizes testing events on the current page before navigating to the next. Regarding code coverage, MINNIE explored >95% of pages and binding functions in 79 and 77 of 100 miniapps, respectively, after ten minutes. The average statement coverage is 73.34%, and 56.52% of pages reach coverage $\geq 75\%$, after ten minutes.

E. RQ4: Component Ablation

MINNIE combines four mechanisms: unified byte-level shadow memory, cross-layer taint synchronization, concolic execution, and scheduler-based UI exploration. To isolate the contribution of each mechanism, we construct four ablated configurations and evaluate them on the same 100-miniapp dataset as Section V-C, using a five-minute budget per miniapp.

TABLE VIII: [RQ4] Component ablation. *Unified*: unified byte-level metadata. *Sync*: cross-layer taint synchronization at the (de)serialization boundary. *Concolic*: concolic execution. *Sched.*: scheduler-based UI exploration (home page loading otherwise).

Config	Components				100 Random Miniapps		
	<i>Unified</i>	<i>Sync</i>	<i>Conc.</i>	<i>Sched.</i>	#TP	#FP	#FN
(i) MINNIE w/ separate metadata	✓	✓	✓	✓	146	14	68
(ii) MINNIE w/o cross-layer	✓	✗	✓	✓	24	5	190
(iii) MINNIE w/o concolic	✓	✓	✗	✓	69	0	145
(iv) MINNIE w/o scheduler	✓	✓	✓	✗	23	6	191
(v) MINNIE	✓	✓	✓	✓	146	14	68

We first compare MINNIE’s unified metadata representation with a separate-metadata design. In MINNIE, each shadow byte jointly encodes a taint label and a symbolic mark, whereas the separate-metadata design maintains taint labels and symbolic

marks in two independent byte-aligned metadata structures. As shown in Table VIII, the two designs yield the same detection results, since they preserve the same taint and symbolic information. While the unified representation allows both types of metadata to share the same propagation and synchronization mechanisms across string operations and (de)serialization boundaries, avoiding separate metadata management. Under the byte-aligned separate-metadata design, the same information requires two metadata bytes per data byte instead of one. To approximate the additional bookkeeping required by separate metadata structures, we mirror each shadow-memory access into a second metadata structure. Across string lengths from 25 to 400 bytes and across `prefix`, `substring/concat`, and equality operations, the separate shadow memory incurs 11–17% runtime overhead on strings of 100 bytes or more. Removing cross-layer synchronization reduces the number of TPs from 146 to 24 and increases the number of FNs from 68 to 190, because taint can no longer be preserved across serialization boundaries between the view and logic layers. Disabling concolic execution yields 69 TPs and 145 FNs, showing that concolic exploration helps reach paths guarded by constraints over external inputs. Finally, disabling scheduler-based exploration yields 23 TPs and 191 FNs as only the home page is loaded and fewer events are exercised.

We further characterize the practical scope and behavior of MINNIE’s concolic execution. Across approximately 6.63 million string-operation call sites in our dataset, 76.55% are byte-preserving operations for which MINNIE supports precise constraint solving, while the remaining 23.45% are non-byte-preserving transformations for which MINNIE preserves taint but does not retain symbolic constraints. On the same 100-miniapp dataset, MINNIE invoked the solver 1,205 times, of which 85.3% were satisfiable, 11.6% were unsatisfiable, and 3.1% timed out. The median solving time was 9.0 ms, and satisfiable queries yielded 1,028 concrete assignments used to drive alternative branches. Each solver invocation was capped at 200 ms, and timed out constraints were not further explored.

F. RQ5: Performance

We evaluate MINNIE’s operation-level and end-to-end runtime overhead. First, we measure MINNIE’s overhead on representative string operations, encoding/decoding operations, and (de)serialization operations, as shown in Table IX. Each operation was executed over 1,000,000 iterations, and the total execution time was recorded. The five character-level operations incur at most 12.4% overhead. Encoding/decoding and (de)serialization operations incur $2.8\text{--}5.0\times$ overhead, except for `JSON.parse` (-1.2%, within measurement noise). The higher overhead mainly results from repeated per-string and per-character taint hooks during output construction. Second, we measure MINNIE’s end-to-end performance overhead by comparing the instrumented runtime with the original miniapp engine. We ran 10 randomly selected miniapps for ten minutes each on both runtimes. Within the same ten-minute budget, the median peak memory on the subset for MINNIE and the original engine was 2,903 MB vs 2,871 MB, indicating that MINNIE’s per-string taint-byte shadow memory brings reasonable memory overhead over the original WeChat DevTools.

TABLE IX: [RQ5] Execution-time overhead of representative operations in the original and MINNIE-instrumented runtimes.

Operations	Original (ms)	Instr. (ms)	Overhead
<code>charAt()</code>	84.01	94.41	+12.4%
<code>String equality (===)</code>	58.88	61.90	+5.1%
<code>concat</code>	108.80	110.00	+1.1%
<code>toUpperCase()</code>	118.80	125.00	+5.2%
<code>toLowerCase()</code>	144.90	152.80	+5.5%
<code>encodeURIComponent</code>	359.80	1795.00	$5.0\times$
<code>decodeURIComponent</code>	483.80	1845.80	$3.8\times$
<code>JSON.stringify</code>	1462.40	4117.00	$2.8\times$
<code>JSON.parse</code>	1709.60	1689.00	-1.2%

VI. DISCUSSION

Coverage. Despite MINNIE’s efforts to maximize code coverage by systematically exploring events and pages, two primary challenges remain: implicit data-flow and time constraints. First, MINNIE cannot detect implicit data-flow dependencies where one event modifies the shared states that indirectly affect another. Second, fully exploring the extensive state space of miniapps requires significant time. To address this, MINNIE sets a timeout limit for each page and prioritizes interactions with heuristic rules, which helps MINNIE balance coverage and efficiency.

Trade-offs in Cross-Boundary Taint Propagation. MINNIE adopts a conservative taint-propagation strategy when information crosses boundaries where byte-level correspondence is not preserved, including at the client-server boundary. In particular, MINNIE taints server responses only when the corresponding outbound request contains tainted data. This design reflects a common miniapp interaction pattern, in which applications transmit user-specific tokens or identifiers to backend services and receive personalized or privacy-sensitive results in return. In practice, this trade-off enables MINNIE

to capture complex privacy leak behaviors that span multiple parties including the miniapp, its server, and the WeChat server, which would otherwise be missed by approaches that terminate taint propagation at network interfaces.

Extensibility. MINNIE is implemented and evaluated only on WeChat. Some engine-level components, including V8 taint tracking and concolic execution, may transfer to other V8-based miniapp platforms, while framework-level support, including source/sink specifications, UI automation and event/page modeling, and miniapp collection require platform-specific adaptation. We leave porting and evaluation on other miniapp platforms to future work.

Mitigations. We discuss four practical defenses. First, miniapp developers should minimize cross-miniapp data sharing, avoid placing sensitive information in navigation parameters, and validate each transmitted field against the miniapp’s documented requirements. Second, the superapp platform could enforce taint-aware sink policies that block or require user consent for data flows not disclosed in a miniapp’s privacy policy. Third, the platform could revise the cross-miniapp navigation API so that user prompts display both the transmitted data and its recipient, rather than only the destination miniapp’s name. Finally, the platform could run MINNIE during miniapp pre-release auditing and compare detected flows with the developer-declared privacy policy. Together, these defenses preserve legitimate miniapp functionality while distributing responsibility between developers, who control what data is transmitted, and the platform, which is better positioned to detect and regulate flows that users cannot observe.

Limitations. Several factors may affect MINNIE’s results. First, MINNIE’s UI exerciser may not fully exercise execution paths guarded by complex constraints (e.g., mandatory account registration and login), leading to uncovered branches. Second, our policy-violation detection relies on LLM-based extraction from privacy policy texts. On a manually validated sample of 609 flagged flows across 100 miniapps, the precision of this labeling is 81.6% (Section V-B). Most false positives arise when the LLM pipeline fails to extract disclosures that are present in the raw privacy policy. Third, MINNIE emits byte-level predicates over equality, prefix, suffix, substring, and inclusion, which Z3 decides efficiently. We do not emit regular-expression or string-replacement constraints: their theories are incomplete for the JavaScript found in real applications, and a byte-level mark cannot in any case reconstruct the derivation such a constraint would need. This is a scope decision with a measured cost (Section V-E), not an unbounded source of error, because taint propagation is preserved for executed paths, but loss of symbolic precision can reduce path coverage and cause false negatives.

VII. RELATED WORK

Privacy Leak Detection in Miniapps. Prior works on privacy leak detection in miniapps mainly leverage static analysis. Wang *et al.* [12] present Taintmini that tracks the flow of sensitive data to detect privacy-sensitive data leaks. Li *et*

al. [13] propose MiniTracker, capable of revealing some dynamic feature leak patterns. However, these static frameworks suffer from incorrect or incomplete modeling of JavaScript dynamic features and cross-boundary data flows in miniapps, resulting in significant false positives and false negatives. In contrast, MINNIE—the first holistic dynamic analysis for detecting privacy leaks—directly executes and checks runtime behavior, eliminating the need for static approximations.

Vulnerabilities in Superapps and Miniapps. Beyond privacy leaks, prior works have also discovered other types of security vulnerabilities in miniapps, such as cross-miniapp request forgery [39], cross-site scripting [11], and credential leakage [40], [41], [16]. Furthermore, the associated superapps hosting miniapps are also found susceptible to vulnerabilities including unauthorized data access [42], [43], privilege escalation [44] and identity confusion [45]. While detecting these vulnerabilities is beyond the scope of this work, MINNIE is built on unified shadow memory and implemented as an instrumented miniapp runtime with UI-driven execution, making it conceptually portable to these settings, which we leave for future work.

Dynamic Taint Tracking and Concolic Execution. There have been some prior works in dynamic taint tracking and concolic execution, but they do not address MINNIE’s challenges of cross-component and cross-layer propagation. Foxhound [21] attaches character ranges with operations to strings, following flows from SpiderMonkey into Gecko within one browser process. DBTaint [46] adds taint tags via SQL rewriting, bridging a web app to its database, not multiple JS runtimes. Fontus [47] tracks per-character taint in strings within a single JVM. TaintDroid [18] carries tags across binder IPC at message granularity, and ViaLin [19] tracks taint within the Android runtime alone. SymSan [22] implements shadow memory-based concolic execution for native code via compile-time LLVM instrumentation in a single process and address space, which does not apply to the miniapp architecture. In contrast, MINNIE’s novelty lies in (i) the single byte-level encoding carrying both a taint label and a symbolic mark, and (ii) taint synchronization across multiple isolated JS runtimes that communicate via (de)serialization. None of these systems combines both properties in the miniapp runtime setting.

Web and Mobile Privacy and Security. Miniapps are built on standard web technologies. MINNIE’s formalization of dynamic taint analysis follows the operational-semantics style established by Schwartz *et al.* [27]. Prior research on traditional web privacy and security is closely related, including the detection of Cross-Site Scripting [20], [48], [49], [30], [50], [51], [52], Cross-Site Request Forgery [53], [54], [55], [56], [57], Cross-Site Leaks [58], [59], [60], Arbitrary Code Execution [61], [62], [63], [64], [65], prototype pollution [66], [67], [62], [68], [69], [70], and code reuse attacks [71], [72], [73], [74], [75]. In comparison, MINNIE focuses on analyzing cross-component, cross-layer data flows with concolic execution for the systematic detection of privacy leaks

in miniapps. MINNIE is also closely related to the broader domain of mobile app security and privacy, a well-established field. Prior work has applied static analysis to detect data leaks [76], [77], [78], [79], [80], [81] and sensitive user inputs [82], [83], [84]. Other approaches employ dynamic analysis techniques for purposes such as privacy monitoring [18], [85], [19], identifying permission misuses [86], and inspecting behaviors of commercial Android packers [87]. Hybrid analysis—combining static and dynamic techniques—has also been adopted by Oltrogge *et al.* [88] to assess the security impact of online app generators, and by Wang *et al.* [89], [90] to assess OAuth implementations in android applications. Despite these advances, existing solutions are often ineffective when applied to miniapp ecosystems, due to the key challenges outlined above—challenges that MINNIE is explicitly designed to address and overcome.

VIII. CONCLUSION

This paper presents MINNIE, the first dynamic taint analysis framework to detect privacy leaks in miniapps. MINNIE comprises an instrumented miniapp runtime with unified shadow memory and a UI testing harness. MINNIE’s unified shadow memory design efficiently supports both taint tracking and concolic execution within a single framework. Our large-scale evaluation flagged over five thousand candidate zero-day privacy leaks across over one thousand miniapps, with a manually measured policy-labeling precision of 81.6%. By enabling systematic detection at scale, MINNIE lays the groundwork for future efforts in shaping a privacy-preserving miniapp ecosystem.

ACKNOWLEDGEMENT

We want to thank the reviewers for their helpful comments and feedback. This work was supported in part by the National Science Foundation (NSF) under grants CNS-21-54404 and CNS-20-46361. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF.

ETHICAL CONSIDERATIONS

We responsibly reported all detected privacy leaks to the WeChat platform with a 45-day disclosure window; the vendor has acknowledged receipt of our reports, but has not yet provided substantive responses as of the time of writing. Our experiments did not transmit any real user data to third parties. Although we used our own WeChat accounts to drive miniapp execution, MINNIE hooks all sink APIs (e.g., `wx.request`, `wx.navigateToMiniProgram`) and intercepts the tainted data before it leaves the runtime, ensuring that no sensitive information is actually sent to external miniapps or servers. We conducted all evaluations in controlled settings to avoid excessive load or disruptions to the miniapp ecosystem, and we limited our analysis to privacy-leak detection without exploiting or redistributing sensitive data.

REFERENCES

- [1] About mini programs. <https://developers.weixin.qq.com/miniprogram/en/dev/framework/quickstart/>.
- [2] Telegram, “Telegram mini apps,” <https://core.telegram.org/bots/webapps>, accessed: 2025-04-10.
- [3] LINE Corporation, “Line mini app,” <https://developers.line.biz/en/services/line-mini-app/>, n.d., accessed: 2025-03-28.
- [4] Rakuten Tech. Readme reference. [Online]. Available: <https://rakutentech.github.io/ios-miniapp/readme.html#how-can-we-develop-a-miniapp>
- [5] TikTok, “Mini programs,” <https://www.tiktok.com/discover/mini-programs>, accessed: 2025-03-28.
- [6] QPSoftware, “Wechat mini program: All you need to know,” 2023, accessed: 2024-10-31. [Online]. Available: <https://qpsoftware.net/blog/wechat-mini-program-all-you-need-know>
- [7] Wechat mini programs for banking pose ‘significant’ risks of personal data leakage, says report. <https://www.scmp.com/tech/tech-trends/article/3142239/wechat-mini-programs-banking-pose-significant-risks-personal-data>.
- [8] CCTV News. (2025) Cctv 3.15 gala: Mini programs hide “privacy assassins”, over 130,000 medical examination reports were stolen. Accessed: 2025-07-16. [Online]. Available: <https://news.cctv.com/2025/07/16/ARTIvAoJkONdHxLLnFnrmfUY250716.shtml>
- [9] X. Zhang, Y. Zhang, X. Zhang, Z. Huang, L. Zhang, and M. Yang, “Understanding privacy over-collection in wechat sub-app ecosystem,” *CoRR*, vol. abs/2306.08391, 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2306.08391>
- [10] Y. Zhang, B. Turkistani, A. Y. Yang, C. Zuo, and Z. Lin, “A measurement study of wechat mini-apps,” *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 5, no. 2, jun 2021. [Online]. Available: <https://doi.org/10.1145/3460081>
- [11] Y. Han, X. Ji, Z. Wang, and J. Zhang, “Systematic analysis of security and vulnerabilities in miniapps,” in *Proceedings of the 2023 ACM Workshop on Secure and Trustworthy Superapps*, ser. SaTS ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1–9. [Online]. Available: <https://doi.org/10.1145/3605762.3624432>
- [12] 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 *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 932–944.
- [13] W. Li, B. Yang, H. Ye, L. Xiang, Q. Tao, X. Wang, and C. Zhou, “Minitracker: Large-scale sensitive information tracking in mini apps,” *IEEE Transactions on Dependable and Secure Computing*, 2023.
- [14] S. Wang, Y. Li, K. Wang, Y. Liu, H. Li, Y. Liu, and H. Wang, “Miniscope: Automated ui exploration and privacy inconsistency detection of miniapps via two-phase iterative hybrid analysis,” *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 6, Jul. 2025. [Online]. Available: <https://doi.org/10.1145/3709351>
- [15] Y. Wang, M. Fan, J. Liu, J. Tao, W. Jin, H. Wang, Q. Xiong, and T. Liu, “Do as you say: Consistency detection of data practice in program code and privacy policy in mini-app,” *IEEE Transactions on Software Engineering*, 2024.
- [16] S. Meng, L. Wang, S. Wang, K. Wang, X. Xiao, G. Bai, and H. Wang, “Wemint:tainting sensitive data leaks in wechat mini-programs,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 1403–1415.
- [17] 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.
- [18] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, “Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones,” *ACM Trans. Comput. Syst.*, vol. 32, no. 2, Jun. 2014. [Online]. Available: <https://doi.org/10.1145/2619091>
- [19] K. Ahmed, Y. Wang, M. Lis, and J. Rubin, “Vialin: Path-aware dynamic taint analysis for android,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1598–1610.
- [20] 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.
- [21] D. Klein, T. Barber, S. Bensalim, B. Stock, and M. Johns, “Hand sanitizers in the wild: A large-scale study of custom javascript sanitizer functions,” in *Proc. of the IEEE European Symposium on Security and Privacy*, Jun. 2022.
- [22] J. Chen, W. Han, M. Yin, H. Zeng, C. Song, B. Lee, H. Yin, and I. Shin, “SYMSAN: Time and space efficient concolic execution via dynamic data-flow analysis,” in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 2531–2548. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity22/presentation/chen-ju>
- [23] DiDi Chuxing, “Didi chuxing official website,” 2025, accessed: 2025-04-17. [Online]. Available: <https://www.didiglobal.com>
- [24] WeBank Co., Ltd., “Webank official website,” <https://www.webank.com>, 2025, accessed: 2025-04-17.
- [25] WeChat, “Yes accounting exam study,” 2025, accessed: 2025-03-17. Note: This link can only be accessed within the WeChat app. [Online]. Available: <https://mp.weixin.qq.com/wxawap/waverifyinfo?action=get&appid=wx0746c49127c6ac62>
- [26] WeChat Mini Program Documentation, “getPhoneNumber,” <https://developers.weixin.qq.com/miniprogram/en/dev/framework/open-ability/getPhoneNumber.html>, n.d., accessed: 2025-03-26.
- [27] E. J. Schwartz, T. Avgerinos, and D. Brumley, “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask),” in *2010 IEEE symposium on Security and privacy*. IEEE, 2010, pp. 317–331.
- [28] Weixin Official Documentation, “Mini program api documentation,” <https://developers.weixin.qq.com/miniprogram/dev/api/>, 2025, accessed: 2025-07-18.
- [29] Google Inc., “V8 JavaScript Engine,” <https://v8.dev>, 2008, accessed: 2025-04-16.
- [30] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia, “Riding out domsdays: Towards detecting and preventing dom cross-site scripting,” in *Network and Distributed System Security Symposium*, 2018. [Online]. Available: <https://api.semanticscholar.org/CorpusID:3389782>
- [31] Minium, “Minium python readme,” 2024, <https://minitest.weixin.qq.com/#/minium/Python/readme>.
- [32] W. Developers, “Wechat mini program service categories,” 2024, accessed: 2024-11-13. [Online]. Available: <https://developers.weixin.qq.com/miniprogram/product/material/>
- [33] SunJunyi, “jieba: Chinese text segmentation,” 2023, pyPI package. [Online]. Available: <https://pypi.org/project/jieba/>
- [34] Appium Contributors, *Appium Documentation*, 2024, accessed: 2024-11-13. [Online]. Available: <https://appium.io/docs/en/latest/>
- [35] JingMatrix, “LSPosed v1.10.0,” 2023, gitHub release. [Online]. Available: <https://github.com/JingMatrix/LSPosed/releases/tag/v1.10.0>
- [36] Tencent Holdings Ltd., “Tencent official website,” <https://www.tencent.com/en-us/>, 2025, accessed: 2025-04-17.
- [37] —, “Tencent qq,” <https://im.qq.com/>, 2025, accessed: 2025-04-17.
- [38] “A monkey testing library written in JavaScript, for Node.js and the browser.” <https://github.com/marmelab/gremlins.js/>.
- [39] 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. [Online]. Available: <https://dl.acm.org/doi/pdf/10.1145/3548606.3560597>
- [40] Y. Shi, Z. Yang, K. Zhong, G. Yang, Y. Yang, X. Zhang, and M. Yang, “The skeleton keys: A large scale analysis of credential leakage in mini-apps,” in *Proceedings of the 32nd Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA: Internet Society, Feb. 2025.
- [41] Y. Zhang, Y. Yang, and Z. Lin, “Don’t leak your keys: Understanding, measuring, and exploiting the appsecret leaks in mini-programs,” in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2411–2425.
- [42] S. Baskaran, L. Zhao, M. Mannan, and A. Youssef, “Measuring the leakage and exploitability of authentication secrets in super-apps: The wechat case,” in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, 2023, pp. 727–743.
- [43] C. Wang, Y. Zhang, and Z. Lin, “Uncovering and exploiting hidden apis in mobile super apps,” in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2471–2485.
- [44] 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.
- [45] L. Zhang, Z. Zhang, A. Liu, Y. Cao, X. Zhang, Y. Chen, Y. Zhang, G. Yang, and M. Yang, “Identity confusion in WebView-based mobile app-in-app ecosystems,” in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 1597–1613.
- [46] B. Davis and H. Chen, “Dbtaint: cross-application information flow tracking via databases,” in *Proceedings of the 2010 USENIX Conference on Web Application Development*, ser. WebApps’10. USA: USENIX Association, 2010, p. 12.
- [47] D. Klein, B. Rolle, T. Barber, M. Karl, and M. Johns, “General Data Protection Runtime: Enforcing Transparent GDPR Compliance for Existing Applications,” in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2023.
- [48] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, and P. Saxena, “Auto-patching dom-based xss at scale,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 272–283. [Online]. Available: <https://doi.org/10.1145/2786805.2786821>
- [49] A. S. Buyukkayhan, C. Gemiciglu, T. Lauinger, A. Oprea, W. Robertson, and E. Kirda, “What’s in an exploit? an empirical analysis of reflected server XSS exploitation techniques,” in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. San Sebastian: USENIX Association, Oct. 2020, pp. 107–120. [Online]. Available: <https://www.usenix.org/conference/raid2020/presentation/buyukkayhan>
- [50] M. Steffens, C. Rossow, M. Johns, and B. Stock, “Don’t trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild.” 2019.
- [51] DOMinator. <https://github.com/wisec/DOMinator>.
- [52] J. Yu, S. Li, J. Zhu, and Y. Cao, “Coco: Efficient browser extension vulnerability detection via coverage-guided, concurrent abstract interpretation,” in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 2441–2455. [Online]. Available: <https://doi.org/10.1145/3576915.3616584>
- [53] S. Khodayari and G. Pellegrino, “[JAW]: Studying client-side {CSRF} with hybrid property graphs and declarative traversals,” in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2525–2542.
- [54] S. Calzavara, M. Conti, R. Focardi, A. Rabitti, and G. Tolomei, “Mitch: A machine learning approach to the black-box detection of csrf vulnerabilities,” in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 528–543.
- [55] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. Rossow, “Deemon: Detecting csrf with dynamic analysis and property graphs,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1757–1771.
- [56] H. Shahriar and M. Zulkernine, “Client-side detection of cross-site request forgery attacks,” in *2010 IEEE 21st International Symposium on Software Reliability Engineering*. IEEE, 2010, pp. 358–367.
- [57] A. Sudhodanan, R. Carbone, L. Compagna, N. Dolgin, A. Armando, and U. Morelli, “Large-scale analysis & detection of authentication cross-site request forgeries,” in *2017 IEEE European symposium on security and privacy (EuroS&P)*. IEEE, 2017, pp. 350–365.
- [58] T. Van Goethem, G. Franken, I. Sanchez-Rola, D. Dworken, and W. Joosen, “Sok: Exploring current and future research directions on xs-leaks through an extended formal model,” in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 2022, pp. 784–798.
- [59] L. Knittel, C. Mainka, M. Niemietz, D. T. Noß, and J. Schwenk, “Xsinator. com: From a formal model to the automatic evaluation of cross-site leaks in web browsers,” in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1771–1788.
- [60] A. Sudhodanan, S. Khodayari, and J. Caballero, “Cross-origin state inference (cosi) attacks: Leaking web site states through xs-leaks,” *arXiv preprint arXiv:1908.02204*, 2019.
- [61] M. Shcherbakov, M. Balliu, and C.-A. Staicu, “Silent spring: Prototype pollution leads to remote code execution in node. js,” in *USENIX Security Symposium 2023*, 2023.
- [62] S. Li, M. Kang, J. Hou, and Y. Cao, “Mining node. js vulnerabilities via object dependence graph and query,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 143–160.
- [63] M. Kang, Y. Xu, S. Li, R. Gjomemo, J. Hou, V. Venkatakrishnan, and Y. Cao, “Scaling javascript abstract interpretation to detect and exploit node. js taint-style vulnerability,” in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2023, pp. 1059–1076.
- [64] S. H. Jensen, P. A. Jonsson, and A. Möller, “Remedying the eval that men do,” in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 34–44.
- [65] B. B. Nielsen, B. Hassanshahi, and F. Gauthier, “Nodest: feedback-driven static analysis of node. js applications,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 455–465.
- [66] Z. Kang, S. Li, and Y. Cao, “Probe the proto: Measuring client-side prototype pollution vulnerabilities of one million real-world websites,” in *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*, 2022.
- [67] S. Li, M. Kang, J. Hou, and Y. Cao, “Detecting node. js prototype pollution vulnerabilities via object lookup analysis,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 268–279.
- [68] H. Y. Kim, J. H. Kim, H. K. Oh, B. J. Lee, S. W. Mun, J. H. Shin, and K. Kim, “Dapp: automatic detection and analysis of prototype pollution vulnerability in node. js modules,” *International Journal of Information Security*, vol. 21, no. 1, pp. 1–23, 2022.
- [69] A. Alhamdan and C.-A. Staicu, “Welcome to jurassic park: A comprehensive study of security risks in deno and its ecosystem,” 2025.
- [70] O. Arteau, “Prototype pollution attack in nodejs application,” https://github.com/HoLyVieR/prototype-pollution-nsec18/blob/master/paper/JavaScript_prototype_pollution_attack_in_NodeJS.pdf, 2018, online; Accessed on 18 Feb 2021.
- [71] Z. Liu, K. An, and Y. Cao, “Undefined-oriented programming: Detecting and chaining prototype pollution gadgets in node.js template engines for malicious consequences,” in *2024 IEEE Symposium on Security and Privacy*, 2024.
- [72] Z. Kang, M. Lyu, Z. Liu, J. Yu, R. Fan, S. Li, and Y. Cao, “Follow my flow: Unveiling client-side prototype pollution gadgets from one million real-world websites,” in *2025 IEEE Symposium on Security and Privacy (SP)*, 2025, pp. 991–1008.
- [73] S. Lekies, K. Kotowicz, S. Groß, E. A. Vela Nava, and M. Johns, “Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1709–1723.
- [74] E. Cornelissen, M. Shcherbakov, and M. Balliu, “Ghunter: Universal prototype pollution gadgets in javascript runtimes,” in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 3693–3710.
- [75] M. Steffens, “Understanding emerging client-side web vulnerabilities using dynamic program analysis,” Ph.D. thesis, Saarland University, 2021.
- [76] 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,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 259–269. [Online]. Available: <https://doi.org/10.1145/2594291.2594299>
- [77] C. Gibler, J. Crussell, J. Erickson, and H. Chen, “Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale,” in *International conference on trust and trustworthy computing*. Springer, 2012, pp. 291–307.
- [78] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, “Pios: Detecting privacy leaks in ios applications,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2011.
- [79] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, “Information flow analysis of android applications in droidsafe,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.
- [80] M. Benz, E. K. Kristensen, L. Luo, N. P. Borges Jr, E. Bodden, and A. Zeller, “Heaps’n leaks: How heap snapshots improve android taint analysis,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1061–1072.

- [81] S. Rahaman, I. Neamtiu, and X. Yin, “Algebraic-datatype taint tracking, with applications to understanding android identifier leaks,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 70–82.
- [82] B. Andow, A. Acharya, D. Li, W. Enck, K. Singh, and T. Xie, “Uiref: analysis of sensitive user inputs in android applications,” in *Proceedings of the 10th acm conference on security and privacy in wireless and mobile networks*, 2017, pp. 23–34.
- [83] 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.
- [84] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang, “UIPicker: User-Input privacy identification in mobile applications,” in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 993–1008.
- [85] M. Sun, T. Wei, and J. C. Lui, “Taintart: A practical multi-level information-flow tracking system for android runtime,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 331–342.
- [86] R. Li, W. Diao, Z. Li, J. Du, and S. Guo, “Android custom permissions demystified: From privilege escalation to design shortcomings,” in *2021 IEEE Symposium on security and privacy (SP)*. IEEE, 2021, pp. 70–86.
- [87] Z. Dong, H. Liu, L. Wang, X. Luo, Y. Guo, G. Xu, X. Xiao, and H. Wang, “What did you pack in my app? a systematic analysis of commercial android packers,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1430–1440.
- [88] M. Oltrogge, E. Derr, C. Stransky, Y. Acar, S. Fahl, C. Rossow, G. Pellegrino, S. Bugiel, and M. Backes, “The rise of the citizen developer: Assessing the security impact of online app generators,” in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 634–647.
- [89] H. Wang, Y. Zhang, J. Li, H. Liu, W. Yang, B. Li, and D. Gu, “Vulnerability assessment of oauth implementations in android applications,” in *Proceedings of the 31st annual computer security applications conference*, 2015, pp. 61–70.
- [90] H. Wang, Y. Zhang, J. Li, and D. Gu, “The achilles heel of oauth: a multi-platform study of oauth-based authentication,” in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 167–176.

APPENDIX

A. An Example of MINNIE-Flagged Taint



Fig. 7: An example of byte-level taint recovery at a network sink. MINNIE recovers the concrete payload fields carrying sensitive data (e.g., longitude and latitude) and associates them with their corresponding taint source identifiers, providing direct runtime evidence of privacy leakage.

B. WeChat Miniapp Taint Sources and Sinks

TABLE X: WeChat miniapp source and sink APIs. Different sensitive source categories are assigned distinct numeric IDs. ID 0 denotes untainted data, while ID 1 denotes a generic taint without specific source attribution.

ID	Category	APIs
2	Storage	wx.getStorageSync, wx.getStorage
3	Profile	<button open-type="chooseAvatar"> <input type="nickname">, wx.login wx.getUserInfo, wx.getUserProfile <button open-type="userInfo">
4	Location	wx.getLocation, wx.chooseLocation wx.startLocationUpdate wx.startLocationUpdateBackground wx.getFuzzyLocation, wx.choosePoi MapContext.moveToLocation wx.onLocationChange
5	Address	wx.chooseAddress
6	Phone	getPhoneNumber getRealtimePhoneNumber
7	Device	wx.getConnectedWifi, wx.getNetworkType wx.createVKSession, wx.startRecord wx.createCameraContext wx.getBLEDeviceCharacteristics wx.openBluetoothAdapter wx.createBLEPeripheralServer
8	Media	wx.chooseImage, wx.chooseMedia wx.chooseVideo, wx.getFileInfo wx.chooseMessageFile
9	Others	wx.getWeRunData, wx.chooseInvoiceTitle wx.chooseInvoice, wx.chooseContact wx.chooseLicensePlate wx.getClipboardData
Sinks	Navigate	wx.navigateToMiniProgram wx.openEmbeddedMiniProgram wx.navigateBackMiniProgram wx.postMessageToReferrerMiniProgram
	Request	wx.request, wx.sendSocketMessage SocketTask.send, wx.createTCPSocket TCPSocket.write, wx.createUDPSocket UDPSocket.send, UDPSocket.write wx.uploadFile

C. Scope of MINNIE’s Concolic Engine.

TABLE XI: Scope of MINNIE’s concolic engine. Byte-preserving operations support precise symbolic constraints; non-byte-preserving transformations retain taint but are explored concretely. Frequencies are measured across approximately 6.63M string operation call sites in 10,625 miniapps.

Category	Operations	Frequency
Byte-preserving	concat, substring, substr, slice, startsWith, endsWith, includes, indexOf, lastIndexOf, charAt, padStart, padEnd, trim, property access	76.55%
Non-byte-preserving	replace, replaceAll, split, match, matchAll, search, new RegExp / literals	23.45%