When Fun Turns Toxic: A First Look at Aggressive Advertising in Mini-games

Pei Chen[†], Geng Hong[†]✉, Yicheng Qin[†], Huazhe Wang[†], Mengying Wu[†], Min Yang[†]✉, Ziru Zhao[§], Yuanpeng Zhu[§] and Tao Su[§]
[†]*Fudan University,* [§]*vivo Mobile Communication Co., Ltd*
✉ *Co-corresponding authors*

Abstract

Mini-games have emerged as a dominant paradigm within super-app ecosystems, enabling lightweight services like casual games to reach millions of users instantly. While official advertisement interfaces simplify monetization, the ease of integration and insufficient oversight have led to aggressive and potentially deceptive advertising practices, severely degrading the user experience. Aggressive advertising, though not malware, still subverts platform security boundaries by abusing legitimate APIs to bypass auditing, manipulate user interaction, and undermine platform trust, constituting a systemic security risk rather than mere policy violation.

In this work, we conduct the first systematic security analysis of aggressive advertising in mini-games. We analyze platform policies and developer capabilities across nine mini-game platforms, and characterize aggressive advertising behaviors. We further design a scalable detection framework, MAAD, and perform a large-scale measurement across three major platforms, i.e., WeChat, Facebook Instant Games, and Quickgame, revealing that 49.95% of mini-games exhibit aggressive advertising, including cases in highly popular titles with over 100k user reviews. Our analysis further uncovers their disruptive behavioral patterns, such as game-specific triggers, excessive pop-up frequency, and misleading strategies, as well as adversarial bypass techniques. These findings uncover that aggressive advertising constitutes a widespread form of platform abuse enabled by structural blind spots in current enforcement mechanisms. We provide actionable implications for strengthening platform governance, detection, and long-term ecosystem resilience.

1 Introduction

Mini-games, which can be played instantly by clicking on links without downloading and installation, are rapidly rising in popularity and attracting a large number of users. In China, the mini-game market reached approximately USD 5.5 billion in 2024, nearly doubling from the previous year and underscoring its explosive growth [24]. This prosperity is evident

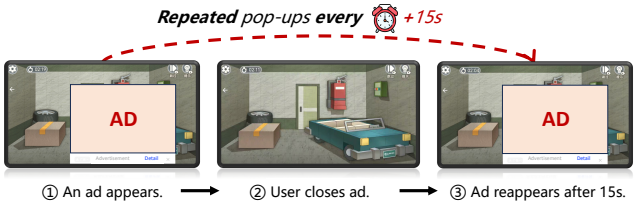

Figure 1: Examples of Aggressive Advertising: Unstoppable Advertising. The user is repeatedly interrupted by advertisement pop-ups every 15 seconds, making it impossible to close the advertisements and continue the game permanently.

as major platforms such as WeChat [60], Facebook [20], and QuickGame [49] have embraced mini-game as a mainstream feature to attract users of their ecosystems.

Unlike app-based native games, which often monetize through paid downloads and/or in-app purchases [3, 26], most mini-games generate income primarily through advertisements [24]. However, the advertisement APIs provided by mini-game platforms give developers fine-grained control over when, where, and how advertisements appear, turning monetization mechanisms into a capability surface that can be misused. For example, Figure 1 shows an aggressive misuse where an interstitial advertisement is triggered in the middle of play, and after the user closes it, a new advertisement returns five seconds later, creating an unstoppable loop that severely disrupts gameplay. Platforms have published advertising guidelines to limit such behaviors. For instance, Facebook Instant Games requires interstitial advertisements to appear only at natural breaks in gameplay rather than interrupting user interactions [45].

Nevertheless, existing work on mini-app ecosystems has concentrated on vulnerabilities or privacy of the mini-app system [39, 43, 50, 67, 71, 73]. Yang et al. [72] provides the first large-scale empirical characterization of evasive malicious mini-apps in the market. They identify an adware family that abuses monetization by fabricating advertisement impressions to defraud advertisers, distinct from the aggres-

sive, user-facing advertising we study. Moreover, existing work primarily targets general mini-apps and thus overlooks the gameplay-specific interaction patterns and development practices unique to mini-game, leaving a gap in understanding the prevalence, characteristics, and detection of aggressive advertising in the mini-game ecosystem.

Our Work. This paper provides the first systematic understanding of aggressive advertising threats in mini-game. Specifically, we analyze the policies specified in the platform to characterize the scope of such behaviors. We examined the advertising policies of the top 15 mini-app marketplaces [73], among which nine support mini-games. Through this analysis, we identify four categories of aggressive advertising, encompassing 14 specific behaviors. In parallel, we studied the advertisement integration process from a developer's perspective. To understand the capabilities available to developers for embedding advertisements into mini-games, we analyzed ad-related APIs across nine platforms and found that controls such as size, position, and close-button options can enable intrusive, repeatable advertising patterns.

Based on this understanding, we design a static analysis framework *MAAD*, to detect and analyze aggressive advertising in mini-games built with Cocos [12], which is the largest market share of mini-games. However, the direct application of existing static analysis techniques in aggressive advertising poses significant challenges. **(C1)** Heavy engine coupling and deeply nested advertising logic make whole-program analysis prohibitively expensive. To address this, we develop a lightweight module extraction and pruning technique that isolates a small set of relevant modules for focused analysis. **(C2)** Cocos's resource management disperses advertising call chains across multiple files and languages, which complicates confirming whether a user event triggers an advertisement. We reverse-engineer the engine's resource decoding and implement an interpreter that reconstructs resource resolution and recovers call chains. **(C3)** Advertising behaviors involve a rich user-interaction context rather than mere advertisement occurrence, which is insufficient to determine aggressiveness. We model *Ad-behavior* with five concise dimensions (*User Event*, *Triggering Condition*, *UI Semantic*, *Advertisement Type*, *Call Path*) and combine static reachability analysis with semantic rules to identify aggressive behaviors.

We evaluated *MAAD* on a ground-truth dataset of 100 manually labeled mini-games, achieving a precision of 94.57%. Over 81% of mini-games were analyzed within 10 minutes each. In a one-month deployment on an anonymous cooperating platform, *MAAD* demonstrates practical utility for platform auditing at scale.

Aggresive Advertising in-the-wild. We apply *MAAD* to conduct a large-scale empirical study across three popular mini-game marketplaces, including WeChat [17], Facebook [20], Quickgame [49]. Across the three platforms, 49.95% of ad-enabled games contained aggressive advertising, most commonly interruptive advertising that forcibly disrupts gameplay. We even identified aggressive advertising in highly popular games with over 100k user reviews, which highlights their broad impact on user experience. Beyond prevalence, we characterized how these games exploit game-specific triggers to ensure advertisement exposure, configure popup intervals that exceed platform restrictions, and adopt diverse misleading strategies to induce unintended clicks.

Furthermore, our measurement reveals the presence of adversarial techniques in aggressive advertising. Developers deliberately attempt to evade platform review, with 94.09% of bypass cases relying on dynamic cloud-controlled triggers that suppress advertising during audits and re-enable it afterward. These strategies are intentionally exploited to undermine single-round or runtime-only audit workflows, thereby compromising the integrity of the review pipeline and exposing a fundamental weakness in platform governance.

We have reported the 456 detected aggressive mini-games to their respective platforms. As of the submission date, two of the three marketplaces had acknowledged the threats, resulting in the removal of 43.32% of the detected games, validating the effectiveness of our detection.

Overall, our findings show that aggressive advertising is not merely a usability concern but a systemic form of capability abuse that threatens platform trust. Addressing these security risks will require stronger multi-stage auditing, more transparent enforcement policies, and closer coordination among platforms, researchers, and regulators to ensure a more resilient mini-game ecosystem.

Contribution. The contributions of this paper are as follows:

- The first systematic security analysis of aggressive advertising behaviors in mini-games, based on platform policies and developer practices, laying the foundation for understanding and regulating this emerging threat.
- An automated static analysis framework that detects aggressive advertising with 94.57% precision, analyzing most games within 10 minutes, enabling scalable and practical auditing for real-world deployment.
- A large-scale empirical measurement across three major platforms, demonstrating that 49.95% of ad-enabled mini-games exhibit aggressive advertising behaviors, exposing a widespread, systemic security threat.

2 Background

2.1 Advertisement in Mini-game

Mini-game. Mini-games are lightweight gaming applications that run within super-app platforms such as WeChat. Compared to traditional mobile games, they require no installation, have smaller resource footprints, and support rapid loading, making them a popular form of casual entertainment distributed through platform-native social sharing.

Architecture. Mini-games execute inside WebGL-based game engines (e.g., Cocos [13], Laya [33]), rather than page-

Table 1: Prohibition of Aggressive Advertising in Mini-game Platform Policies. ✓ indicates the platform prohibits such aggressive advertising in official policies.

Aggressive Advertising Behavior		Mini-game Platform								
Category	**Description of Behaviors**									
Interruptive Advertising	1. Unanticipated mid-game pop-ups disrupting gameplay		✓	✓	✓			✓	✓	✓
	2. Blocking progress until ad completion	✓		✓				✓	✓	
	3. Overlapping ads obscuring functional controls	✓	✓	✓		✓	✓	✓	✓	✓
Hijacking Advertising	1. Functional buttons reprogrammed to trigger ads			✓						
	2. Unexpected clickable areas redirecting to ads			✓					✓	
	3. Hidden or delayed close buttons prolonging ad exposure			✓			✓	✓	✓	
Unstoppable Advertising	1. High-frequency pop-ups within short intervals		✓		✓	✓		✓	✓	
	2. Ads reappearing after closure			✓						
	3. Concurrent display of multiple ads on one screen	✓	✓	✓		✓		✓	✓	
	4. Overlapping multiple ads at one place	✓	✓							
Deceptive Advertising	1. Misleading text/image to attract clicks	✓		✓		✓	✓		✓	
	2. Exploiting habitual tapping to induce mis-clicks	✓						✓	✓	
	3. Reward ads shown without required disclosure	✓		✓						
	4. Visually disguised as non-ad interface elements	✓					✓			

Platforms: WeChat Mini Game, Facebook Instant Game, QuickGame, VK Mini Game, Baidu Mini Game, Alipay Mini Game, TikTok Instant Game, Kuaishou Mini Game, UC Browser Game.

routed mini-app frameworks. As illustrated in Figure 2, the engine mediates all execution: it manages scripts and resources, maintains the rendering and physics loops, and adapts platform services such as networking, payments, and advertising. Application logic is therefore funneled through a heavyweight engine layer that intertwines multiple subsystems. This engine-centric design produces deeply entangled call chains, asset-driven control flow, demanding analysis methods distinct from those for mini-apps.

Advertising-Driven Business Model. Most mini-games adopt a free-to-play model, reflecting the social and low-friction nature of super-app platforms [73]. To support monetization, platforms offer official advertising interfaces, allowing developers to embed advertisements and track impressions for revenue easily. Unlike in traditional app ecosystems, advertisement placement in mini-games is often developer-controlled and loosely enforced, making advertising both a key revenue source and a potential vector for abuse.

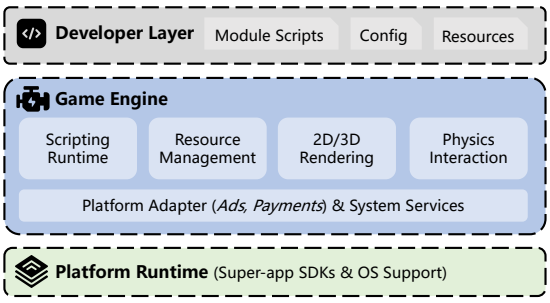

Figure 2: System Architecture of Mini-games. The game engine serves as the core component of the mini-game.

2.2 Threat Model

The adversary is a legitimate mini-game developer who aims to increase revenue by manipulating when, where, and how often advertisements appear. Their capabilities are limited to the platform's official advertising APIs, since platforms prohibit other integration channels. They can only invoke permissible calls within their own mini-game by adjusting scripts,i.e., placement, timing, and frequency. They cannot alter advertisement content, forge billing signals, or compromise platform infrastructure.

Adversarial behaviors do not constitute ad-fraud, but they degrade user experience and erode trust, harming to the platform's reputation and ecosystem. We assume that the platform has access to the complete mini-game package during review and can analyze it for potential abuses.

2.3 Platform Policies on Advertising

To clarify the policy boundaries of in-game advertising, we systematically surveyed the top 15 popular mini-app marketplaces [73], 9 of which support mini-game. We collected their advertising policies specific to mini-game, or to mini-apps in general, when no dedicated rules were provided [2, 4, 19, 31, 44, 59, 62, 64, 66].

We examined the policy documents by filtering with the keyword "ad" and manually categorized the relevant clauses into 4 categories of aggressive advertising, with 14 concrete behaviors [1]. Two authors independently performed the anno-

[1]Some super-app platforms may impose content restrictions (e.g., privacy, legality). However, in mini-game advertising, advertisement content is typically controlled by advertisers rather than developers. Content-related violations are thus beyond the scope of this study and were not included in our classification.

tation, which process is provided in Appendix A. The four categories are:

• *Interruptive Advertising*: advertising that forcibly disrupts normal gameplay flow.

• *Hijacking Advertising*: advertising that takes over user operation or navigation in unexpected ways.

• *Unstoppable Advertising*: advertising that cannot be closed permanently or reappears in a short interval.

• *Deceptive Advertising*: advertising that misleads or induces users to click through disguised cues.

As shown in Table 1, every platform prohibits at least one type of interruptive advertising, reflecting a shared consensus that advertising must not obstruct or unduly disrupt core gameplay. Among platforms, WeChat and QuickGame enforce the most comprehensive restrictions, which is consistent with their large user bases and relatively mature compliance systems. In particular, WeChat adopts a graded enforcement mechanism, imposing penalties ranging from warnings and temporary suspension to permanent bans and revenue deductions [70]. In contrast, VK [66] and UC [62] specify only two prohibitions, indicating weaker regulatory coverage.

Notably, although some behaviors remain uncovered in some platform policies, they are still prohibited under higher-level regional laws and regulations. For example, China's Advertising Law against interfering with normal user operations[46], U.S. and EU requirements for "clear and unambiguous" [14, 63], and explicit bans on "deceptive" in China, the United States, and Russia[14, 22, 46]. These principles are vague for concrete advertising behaviors, and no unified industry standard exists. As a result, platform policies differ in both definitions and enforcement. Furthermore, some policies haven't yet covered the category already highlighted in their regional regulations. For example, Russia prohibits deceptive advertising, yet VK's policy does not mention it.

> ⇒ **Takeaway I**: *Platform policies show a clear stance against aggressive advertising and help reveal their behavioral patterns, while the breadth of policies varies widely, with WeChat and QuickGame adopting the most comprehensive restrictions.*

2.4 Developer Capabilities in Integration

Advertisements Integration Process. In a developer's perspective, the normal process of integrating advertisements in mini-game starts with registering a payment account on the advertising platform and creating IDs for different types of advertisements. The advertising platform offers various types of advertisements to adapt to different game scene placements.

Then, developers need to choose an appropriate position to create the advertisement object and set conditions for advertising. Developers have considerable freedom in this process and can establish display relationships based on game progress and user operations.

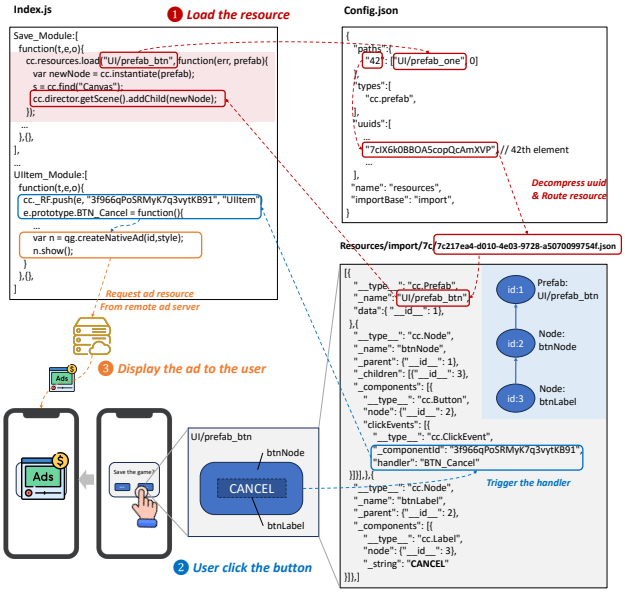

Figure 3: Advertising Integration Progress of an Mini-game. ❶ Initially, the game loads and renders the page resources, with the red line indicating the loading process for button resources. ❷ Next, the user clicks the "CANCEL" button on the mini-game screen, with the blue line indicating the callback control flow after the click. ❸ Finally, the game creates an advertisement instance through the ad-API and requests advertisements from the mini-game platform, displaying them on the screen, as indicated by the yellow line.

After development is completed, a typical process through which an advertisement is displayed to the user is shown in Figure 3. *Step ❶: Resource Loading.* The mini-game uses `cc.resources.load` to load the resource from a specified path, consults the config file to locate the corresponding short-UUID, restores the UUID via Cocos' decoding and decompression, and assembles the final resource path. It then returns to the loading point and renders the resource component in the user's view. *Step ❷: Button Clicking.* When the user clicks a component button, a click event in the resource is triggered. As resources are organized in a tree structure, some prefabs contain `cc.button` sub-nodes whose callback handlers and positions are recorded in their `clickEvents`. The text "CANCEL" on the button conveys semantic information. Parsing the handler in the resource file allows locating the correct callback function. *Step ❸: Advertisement Popping-up.* In the callback, the mini-game invokes `qg.createNativeAd` to request advertising resources from the remote server using the developer's ID, and the advertisement is then presented.

Finally, the advertising platform calculates the developer's earnings based on specific advertising frequencies. Common billing methods in mini-games include Cost-Per-Click (CPC) and Cost-Per-Mile (CPM). CPC requires users to click on the advertisement content, while CPM only requires display. That

is why some developers are trying to increase the pop-ups and set aggressive advertising as much as possible.

Developer Capabilities. Since greater modification freedom may enable developers to craft aggressive advertising, we further analyzed the design of advertisement APIs provided by mini-game platforms to understand which aspects legitimate developers are allowed to modify, which helps us to better analyze the aggressive advertising. By analyzing the ad-related API documentation and implementation examples from the nine platforms that support mini-games, we identified several key parameters in API design that may influence the risk of aggressive advertising, i.e., pop-up size, pop-up position, refresh frequency, and customization.

As shown in Table 2, on most platforms, *banners* allow customizable size and position. While such flexibility increases design freedom, it also enables aggressive practices. Improper parameter choices can result in oversized or centrally placed banners that obscure gameplay elements, or in advertisements overlapping with functional buttons, thereby forcing user attention and inducing accidental clicks. Some platforms (e.g., WeChat and QuickGame) constrain adjustment ranges to mitigate misuse. For example, WeChat requires banner widths no less than 300px, and QuickGame enforces interstitial widths between 720–1080px, while both forbid off-screen rendering [61, 65]. While these safeguards prevent ad-fraud [53], they do not meaningfully restrict aggressive practices. Developers can still exploit the permitted ranges to enlarge banners, intrude on gameplay areas, or stack multiple advertisements, thereby sustaining a wide space for aggressive behaviors.

Additionally, we also identified a platform-specific parameter option on VK, i.e., developers can choose to remove the close button of the advertisement box. To evaluate its practical effect, we randomly sampled 50 VK mini-games that include *banners* and inspected their banner API usage, finding that 18/50 (36%) explicitly set the optional parameter `can_close=false`, rendering banners without a close button and thus depriving users of the ability to dismiss them, and importantly when the parameter was omitted the API's default behavior was `false`. These results demonstrate that this parameter option is dangerous because it enables persistent, non-dismissible banners and therefore poses a concrete risk of developer-driven misuse.

Rewarded videos, in contrast, are uniformly full-screen and non-configurable across platforms. Due to their intrusive nature, most platforms require clear user consent or preconditions for display. As summarized in Table 1, unsolicited reward pop-ups are considered policy violations.

⇒ **Takeaway II**: *Flexible ad-API parameters expand creative freedom but also enable aggressive misuse, making parameter values a critical factor for risk analysis. Unfortunately, existing constraints are insufficient to curb such misuse, and in some cases (e.g., VK's* `can_close=false` *option) even introduce risky defaults enabling non-dismissible advertisements.*

3 Detecting Aggressive Advertising

In this section, we design and implement a static analysis framework, *MAAD* (Mini-game Aggressive Advertising Detector), to detect aggressive advertising in mini-games, targeting Cocos-based games as they constitute the largest share of the ecosystem and offer a representative basis for analysis.

3.1 Challenges & Insights

Before detailing our system design, we summarize the main challenges and our key insights into the detection architecture.

Challenge 1*: How to isolate advertising logic when engine coupling renders whole-program analysis prohibitively expensive.* To facilitate low-code development and optimize loading speed and user interaction, modern game engines adopt modular dynamic loading, physics-based rendering, and other advanced interaction mechanisms. These design choices inevitably increase call-chain depth and tightly couple advertising logic with unrelated functional code, making it difficult for static analysis tools to locate ad-related execution paths directly. As a result, state-of-the-art static analysis methods often become trapped within deeply nested engine structures [77].

Insight 1*: Lightweight module extraction enables focused analysis.* Although the engine codebase is large and complex, ad-trigger logic typically clusters within a limited set of modules, and the inter-module dependency patterns of these modules remain relatively stable. As illustrated in Figure 4, the yellow-highlighted regions represent modules and their dependency relationships. By constructing a dependency graph from explicit module declarations, we can identify and retain only those modules likely to contain advertising behavior, while pruning away unrelated ones. This lightweight extraction substantially reduces the scope of analysis without sacrificing coverage of ad-trigger flows. In practice, we leverage modular dependency analysis to strip out common engine code and apply pruning to eliminate irrelevant modules, thereby avoiding costly whole-program parsing and improving the efficiency of detection.

Challenge 2*: How to trace user events in mini-games where resource management disperses call chains across diverse files and languages.* User-triggered events are often linked to specific mini-game scenarios, but their binding relationships are distributed across multiple files and programming languages. For example in Figure 3, in mini-games, event-binding data may be split between module scripts, configuration files, and resources, with mappings established dynamically depending on the engine's resource management. This distributed, cross-language structure lacks a unified index,

Table 2: Flexibility of API Parameter across Mini-game Platforms. W=Width, H=Height, X=Horizontal, Y=Vertical.

Advertisement in Mini-game			Mini-game Platform								
Ad Type	**Typical Appearance**	**API Parameter**									
Banner	Thin strip at bottom	Pop-up Size	W	–	–	–	–	W	W	W/H	W/H
		Pop-up Position	X/Y	–	Y	Top/Bottom	–	X/Y	X/Y	X/Y	X/Y
		Refresh Frequency	≥30s	–	–	–	–	≥30s	≥30s	≥30s	-
		Customization	–	–	–	Disable close	–	–	–	–	–
Interstitial	Centered popup	Pop-up Size	W	–	W	–	–	–	–	–	–
		Pop-up Position	X/Y	–	X/Y	–	–	–	–	–	–
		Refresh Frequency	≥30s	–	–	–	–	–	–	–	–
Box Portal	Sidebar Icon Grid	Pop-up Size	W/H	–	–	–	–	–	Grid count	–	–
		Pop-up Position	X/Y	–	Y	–	–	–	X/Y	–	–
		Refresh Frequency	≥30s	–	–	–	–	–	–	–	–
Reward Video	Full-screen video	–	–	–	–	–	–	–	–	–	–

Platforms: WeChat Mini Game, Facebook Instant Game, QuickGame, VK Mini Game, Baidu Mini Game, Alipay Mini Game, TikTok Instant Game, Kuaishou Mini Game, UC Browser Game.

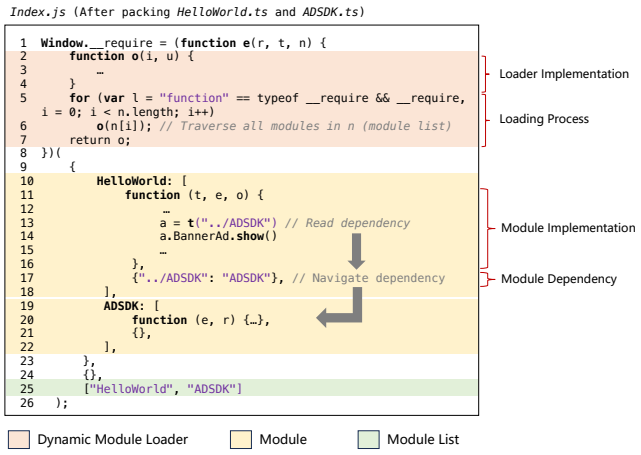

Figure 4: Module Management in Cocos Mini-games. The modules are traversed by dependencies at runtime.

making it challenging for static analysis to trace the complete call chain.

Insight 2*: An interpreter reconstructs dispersed traces, inspired by reverse-engineering.* By reverse-engineering how the Cocos engine parses and links resource files, we gain insights into the conventions and rules governing cross-language references. While call chains of events are fragmented across multiple languages, resource file naming conventions, configuration fields, and function signatures typically follow recognizable patterns, e.g., UUID's compression and decompression. Leveraging this understanding, we exploit cross-file pattern matching and recover mappings between configurations and scripts, making it possible to rebuild the user events without executing the program.

Challenge 3*: How to analyze aggressive advertising behaviors that depend on rich user interactions rather than mere occurrence.* Aggressive advertising behaviors in mini-games often depend on multiple interacting factors that extend beyond what a pure call-chain can capture. As shown in Table 1, most advertising behaviors involve explicit trigger conditions, such as time intervals, user interactions, or state-dependent checks, while certain behaviors like deceptive advertising additionally require semantic analysis about whether the triggered interfaces are misleading. Moreover, as illustrated in Table 2, aggressive behavior may also arise from specific modifications to API parameters. These complex advertising behavior cannot be holistically expressed within a call-graph–only framework, revealing a fundamental modeling gap in representing multi-condition aggressive advertising.

Insight 3*: Multi-dimensional modeling enables unified detection of aggressive advertising.* Despite the diversity of aggressive advertising, the factors driving aggressive advertising behaviors are often localized and structurally regular within the codebase. Trigger conditions typically appear in event handlers as predicate checks (e.g., time checker), deception cues can be approximated by analyzing textual or layout features in the triggered interface, and API-parameter deviations from platform defaults can identify driven behaviors. By modeling these heterogeneous cues in a unified predicate–resource–parameter representation, we can bridge the gap between structural reachability and behavioral semantics.

To operationalize this observation, we enrich the formal behavioral model capable of capturing heterogeneous cues from multiple sources. Specifically, each (potential) advertisement is represented as an *Ad-behavior*, formally defined as:

$$\langle \textit{User Event}, \textit{Triggering Condition}, \textit{User Interface Semantic}, \textit{Advertisement Type}, \textit{Call Path} \rangle \quad (1)$$

Here, the first three elements describe the user-triggered event, the logical state at the time of triggering, and the user-interface context in which the advertisement appears. The *Advertisement Type* denotes the display format (e.g., banner, interstitial), while the *Call Path* records the sequence of invoked functions,

enabling differentiation of behavior variants.

For example, the behavior illustrated in Figure 3, where a cancel button unexpectedly triggers an interstitial pop-up, can be expressed as:

$$\langle\, click\ button,\ null,\ "CANCEL",\ interstitial,\ [UI/Prefab\_btn,\ BTN\_Cancel,\ createNativeAd]\,\rangle \tag{2}$$

This tuple-based representation unifies control-flow structure, logic condition, and semantic context into a single analytical unit, closing the identified modeling gap for multi-condition aggressive advertising behavior.

3.2 Design Architecture

We design and implement the static analysis framework *MAAD* for mini-games to detect the *Ad-behaviors* from game files. It takes a packaged mini-game as input and outputs the aggressive *Ad-behaviors* report. Figure 5 shows our workflow.

Stage I unpacks the mini-game, then applies module-, function-, and polyfill-level pruning to strip away engine code and produce lightweight analysis targets. Based on *Ad-behavior*, Stage II enhances context by reconstructing user events, extracting trigger conditions, and analyzing UI semantics. These elements are integrated into an interaction graph, from which structured *Ad-behaviors* are extracted. Stage III evaluates the extracted *Ad-behavior* against platform rules, ultimately flagging aggressive advertising practices.

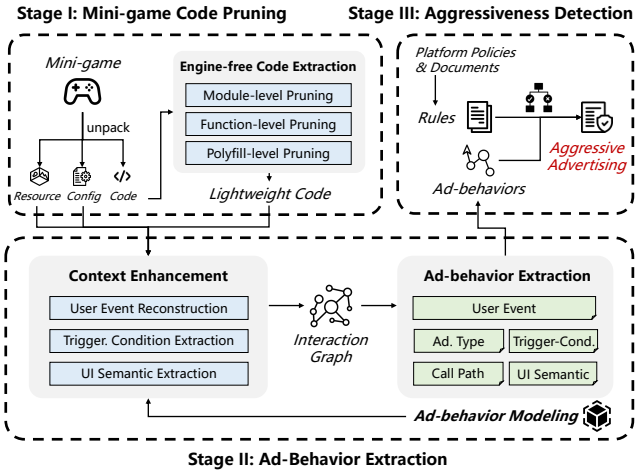

Figure 5: The Three-Stage Workflow of *MAAD*.

3.3 Stage I: Mini-game Code Pruning

In mini-games, game engines help developers easily construct mini-game programs. However, their high coupling and engine code complexity also impose significant costs on static analysis. To address this challenge, we apply a dedicated code pruning pipeline that simplifies the game code to improve analysis efficiency. Our design centers on three pruning techniques, each applied at a different code granularity: module-level dependency pruning, function-level reachability pruning, and polyfill-level nesting pruning.

Module-level Dependency Pruning. Mini-game projects often include a wide range of modules that are not relevant to advertising behavior, such as those supporting physics interaction or animation rendering. Analyzing these modules not only increases resource usage but also introduces noise to downstream analysis. Fortunately, module dependencies are explicitly declared in the engine's management structure. As illustrated in Figure 4, a module (e.g., `Helloworld`) invokes functions defined in external modules (line 13-14), while the actual dependency is indexed as the last argument in the module (line 17). This indexing pattern enables reliable extraction of inter-module relationships. Based on this structure, we construct a global module dependency graph, where nodes represent modules and edges denote declared dependencies. By identifying modules that invoke advertisement APIs as entry points, we retain their reachable subgraphs and discard all others, thereby reducing the codebase while preserving dependency integrity.

Function-level Reachability Pruning. Beyond module-level filtering, we further reduce the code by identifying functions involved in ad-related execution paths. Building a complete function call graph is prohibitively expensive, so we adopt an approximate backward traversal approach. Starting from known advertisement APIs, we iteratively locate their callers based on text-level matching and expand the reachable set until no new relevant functions are found. This method effectively captures the advertising execution logic while minimizing analysis overhead.

Polyfill-level Nesting Pruning. To ensure compatibility with ES5-only platforms such as WeChat [18], mini-game engines automatically insert polyfill functions to emulate ES6+ features. These polyfills, particularly those simulating inheritance, introduce deeply nested and redundant structures that burden static analysis. For example, the polyfill method of the extends method in ES6, simply iterates through all classes in the file for the parent class and assigns them to the child class every time. In cases of multiple inheritance, when invoking the child class, this recursive call of the polyfill method will significantly consume computational resources. As a solution, we identify these polyfill methods, remove the declarations and related function calls of polyfill methods, and implement a copy of the parent class method with an order of topological sorting. This sorting ensures that the parent class being read has already been processed, avoiding recursive inheritance and thereby reducing complexity.

3.4 Stage II: Ad-Behavior Extraction

While Stage I prunes engine code into lightweight analysis targets, this is insufficient for characterizing aggressive advertising behaviors. Such behaviors depend not only on advertisement occurrences but also on multiple dimensions, as modeled in Section 3.1. Our key insight is that these heterogeneous signals, though scattered across configurations, scripts, and layouts, can be systematically reconstructed and integrated. Stage II develops this graph through three analyses—user event reconstruction, triggering condition extraction, and UI semantic extraction—from which structured *Ad-behaviors* are ultimately derived.

User Event Reconstruction. To faithfully capture how user interactions in mini-games evolve into advertising behaviors, *MAAD* reconstructs complete interaction-to-advertisement pathways. Building on our reverse-engineering of the Cocos resource parsing process (Insight 2), we leverage conventions in file naming, configuration fields, and function signatures to enrich the call graph and merge dispersed execution flows into a coherent representation.

Concretely, *MAAD* maps callback handlers to their control scripts across heterogeneous files and programming languages. For each event type, we construct a dedicated *entry* function and use identifier relationships recovered from configuration files as the mapping hub to reconnect fragmented logic. All reconstructed *event entries* are then consolidated under a designated *main entry*, yielding a unified call that faithfully reflects advertising behaviors. Specifically, we model four categories of *event entries* that capture user-triggered interactions potentially leading to advertisements: (i) *Button Clicking.* User interactions through explicit button clicks, typically specified in configuration files by the callback function names of button components. (ii) *Component State Switching.* Changes in the lifecycle states of in-game components, such as `onLoad`, `onEnable`, and `onDestroy`. (iii) *Screen Touching.* General screen-level interactions, that are modeled similarly to clicks, but operate at the global screen level rather than being tied to a specific component. (iv) *Object Collision.* Physical interactions between game objects, as supported by the engine's physics module, which represent a game-specific form of event triggering.

Triggering Condition Extraction. A plain call graph captures reachability but fails to account for the runtime predicates that actually gate advertising triggers. Such conditions are crucial, since aggressive behaviors often arise not from the mere presence of an ad-API call, but from how frequently or under what circumstances it is invoked. We therefore annotate each candidate path with trigger conditions extracted from AST/IR in two forms: (i) general predicates recovered from control-flow constructs (e.g., if/else, switch/case), attached as branch guards on edges; and (ii) time-related predicates obtained from scheduler/delay APIs (e.g., `schedule(a,t)`, `setTimeout(a,t)`), encoded as temporal constraints on the trigger node. These annotations are stored on the path and propagated into the *Ad-behavior*.

User Interface Semantic Extraction. Paths alone do not capture how an advertisement is presented at trigger time, yet such presentation details are often central to determining aggressiveness. To recover this information, we reconstruct the UI context for each candidate path by linking the trigger node to its corresponding scene and component resources. In particular, we record two cues: (i) textual content on the clickable component and its child components, and (ii) layout position, distinguishing default placements from explicitly specified coordinates that may indicate manipulative placement, such as misalignment or stacking. These attributes are attached to the path and propagated into the *Ad-behavior*, ensuring the later detection of semantics.

Ad-Behavior Extraction. By combining user events, triggering conditions, and UI semantics, we enrich the call graph into a behavior-aware interaction graph. A depth-first search from *main entry* to ad-API calls yields structured *Ad-behaviors*, each annotated with attributes for subsequent detection.

3.5 Stage III: Aggressiveness Detection

After Stage II yields structured *Ad-behaviors* for each mini-game, however, whether these *Ad-behaviors* are aggressive remains undetermined. Building on our *Ad-behavior* model, we treat each *Ad-behavior* as a unified analytical unit with multiple information. On top of this abstraction, we design a rule-based detector that evaluates *Ad-behaviors* against constraints distilled from platform policies, thereby bridging structural reachability with behavioral semantics.

Specifically, the detector translates advertising policies from nine major mini-game platforms (Table 1) into executable patterns over *Ad-behaviors*, encoding platform parameter setups (Table 2). We enforce standards that are explicitly prohibited by each platform, while applying a unified default threshold when no explicit rule is provided. When handling unstoppable advertising behaviors, platform policies specify two interval thresholds, 30s (Facebook [20], QuickGame [49]) and 60s (TikTok [47], Kuaishou [31]). To better capture aggressive behaviors, we adopt the more conservative 30-second threshold. The detector performs in three steps: (1) Loading key behavior elements from the extracted Ad-behavior (e.g., Ad type, function parameters); (2) Comparing these elements against predefined thresholds derived from policy (e.g., popup intervals); (3) Combine multiple conditions to infer whether the *Ad-behavior* constitutes an aggressive category (e.g., Ad type plus short interval indicates unstoppable advertising). The multi-condition rules for each aggressive advertising category are detailed in Appendix B.1. Beyond structural constraints, we augment the detector with semantic signals to capture inducive or misleading UI text, enabling the detection of deceptive advertising behaviors. To this end, we fine-tuned an XLM-RoBERTa-base classifier to identify in-

ducive intent in UI Semantic. Details of the classifier's design and evaluation are provided in Appendix B.2.

4 Evaluation

In this section, we evaluate *MAAD* in terms of its effectiveness measured by precision and recall, its efficiency under different pruning strategies, and its practical performance in large-scale deployment.

4.1 Dataset & Implementation

Dataset. To support the effectiveness evaluation, we manually constructed a dataset of 371 *Ad-behaviors*. Specifically, we obtained 100 real-world mini-games from an anonymous official audit department. These cases originated from user complaints between January and December 2023 and had been validated by auditors, providing a realistic and representative basis for ground-truth aggressive behaviors.

For each mini-game, we conducted manual behavior labeling: we interacted with the game for five minutes to trigger advertisements on physical devices, applied dynamic instrumentation to ad APIs to capture runtime logs and stack traces, and then performed step-by-step reverse engineering along the corresponding call paths to trace parameter flows. This process produced 371 labeled *Ad-behaviors*, which were independently cross-checked by three researchers to ensure consistency and accuracy.

Implementation. We implemented *MAAD* for the Cocos engine. In *MAAD*, we use Esprima [27] to convert JavaScript code to the AST. In order to unify the ES6+ syntax in the code into ES5, we use swc [48] to downgrade ES6+ JavaScript code. We chose WALA [8] for generating call graphs due to its superior performance among evaluated tools, with comparative results reported in Appendix C. We performed all the experiments on an Ubuntu server with 224 GB of memory and an Intel Xeon CPU with 32 cores.

4.2 Effectiveness Evaluation

We ran *MAAD* on all 100 mini-games in the dataset. As a result, *MAAD* detected a total of 29,584 *Ad-behaviors*, with a median of 66 and an average of 295.84. It is worth noting that, we observed that one mini-game outputted more than 10,000 *Ad-behaviors*, resulting in the average of *Ad-behaviors* being quite larger than the median (295.84 *vs.* 66). After manually verifying the result, we found that it was not a false positive. The large number of them comes from different buttons in multiple scenes using the same callback, which can be traversed and triggered during the dynamic analysis.

Precision. We checked the correctness of *Ad-behaviors* by manual verification. Specifically, we randomly sampled 10 *Ad-behaviors* from each mini-game (selecting all *Ad-behaviors* if fewer than 10) to avoid clustering within a single advertising type, resulting in a total of 903 *Ad-behaviors*. Then, for each sampled *Ad-behaviors*, we manually reverse-engineered mini-games and verified whether *User Event*, *Triggering Condition*, *User Interface Semantic*, *Advertisement Type*, and *Call Path* match to *MAAD* output, confirming that 854/903 *Ad-behaviors* are correct, with a precision of 94.57%. Per-category statistics are summarized in Table 3.

Table 3: Precision and Recall Across Four Categories

Category	Precision	Recall
Interruptive Advertising	432 / 454 (95.15%)	219 / 255 (85.88%)
Hijacking Advertising	146 / 149 (97.98%)	56 / 72 (77.77%)
Unstoppable Advertising	253 / 277 (91.33%)	32 / 39 (82.05%)
Deceptive Advertising	23 / 23 (100.00%)	3 / 5 (60.00%)
Overall	854 / 903 (94.57%)	310 / 371 (83.55%)

For false positives (FPs), there are two main factors. The first factor is dynamic parameter passing in JavaScript. In mini-game development, many functions use dynamic parameters to perform similar tasks, which makes it challenging for static analysis to predict exact runtime behavior accurately. For instance, frontend parameters at runtime control the game level being played. However, static analysis tools, aiming for completeness, connect all possible function parameters, resulting in non-existent *Ad-behaviors*. In this case, the unstoppable category is more affected, because it depends on evaluating parameter values against thresholds. When its decisive dynamic parameters fail to be extracted, and the static default values exceed the threshold, the behavior is incorrectly reported as a false positive. The second factor is polymorphism and inheritance. To ensure compatibility across different platforms, many handlers employ a polymorphic design with a base class and multiple subclasses, each overriding methods for platform-specific logic. When multiple platform handlers occur in more than one stage of advertisement triggering, we fail to force the tool to always choose the same platform subclass in different handlers.

Recall. We compare the outputs with the dynamically *labeled behaviors* in Section 4.1. We manually verified that there are 310/371 *labeled behaviors* as true positive outputs, with a recall of 83.55%, which indicates the capability of detecting the truly recorded *Ad-behaviors* (also see Table 3).

For false negatives (FNs), the primary reasons include the inherently dynamic dispatch of JavaScript, obfuscation techniques, and the integration of third-party plugins. Specifically, we have observed an integration of third-party plugins based on a state machine, which makes the tool miss the necessary state-switching callback. Moreover, static analysis of JavaScript has the obvious disadvantage of missing dynamic call edges, particularly in non-trivial usages of indirect calls such as `bind` or `apply`. This issue affects the hijacking category more noticeably, because hijacking advertising relies on

button-click handlers, and developers may implement using indirect calls. In addition, our inducive detection is text-based, so two ground-truth cases using image-based inducement cannot be captured and are counted as FNs.

4.3 Efficiency Evaluation

We tested the runtime efficiency of the *MAAD*. The time overhead for analyzing each mini-game was measured as the duration of the whole processing. Additionally, we conducted an ablation study to evaluate the impact of each pruning strategy in Section 3.3 on the time overhead.

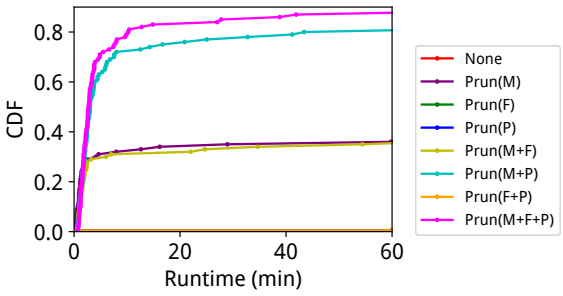

Figure 6: Ablation Runtimes on Different Pruning Strategies. This figure presents a comprehensive Cumulative Distribution Function (CDF) analysis of execution times when using different combinations of Pruning strategies, encompassing Module-level Pruning (M), Function-level Pruning (F), and Polyfill-level Pruning (P). None represents not applying any of the above strategies. In this Figure, None, Prun(F), Prun(P), Prun(F+P) remain at 0 (all timeout within 60 min).

As shown in Figure 6, the bottom line is the baseline analysis, which represents the WALA analysis of mini-game source codes, excluding any additional processing. Each pruning strategy performs better than the baseline. When we perform all strategies, more than 87% of mini-games can be analyzed within 60 minutes, and more than 81% of mini-games can be analyzed within 10 minutes, demonstrating the efficiency of our approach. It is noticeable that Polyfill Pruning is the most efficient pruning strategy. One reason is that the polyfill methods, which are generated by mini-games packagers, typically involve extensive traversal and deeply nested assignment operations, introducing substantial time overhead to static analysis.

Furthermore, we also consider the potential effectiveness loss during efficiency optimization. Polyfill-level Pruning (P)) Rewrites functions without altering the control logic, thus won't introduce effectiveness loss. We admit that in cases where JavaScript includes reflection calls, relevant functions may be mistakenly removed during Function-level Pruning (F). We previously discussed this type of false negative (FN) in RQ1, noting that it affects only a small number of cases (3/61). Consequently, an application using all of the strategies will provide optimal analytical performance.

4.4 Large-Scale Deployment

We validated the practicality of *MAAD* by deploying it on over 1.6K real-world Cocos mini-games from a cooperating platform. The tool identified aggressive behaviors in more than half of the cases, demonstrating its scalability and effectiveness (see Appendix D for details).

5 Real-World Measurement

Building upon the evaluation of our tool, we now conduct a large-scale measurement study to examine the prevalence and characteristics of aggressive advertising in real-world mini-games. To this end, we collected mini-game packages from popular super-app marketplaces. From the nine platforms analyzed earlier, we selected WeChat Mini Game, Facebook Instant Games, and QuickGame, as they host the largest mini-game ecosystems with each exceeding 100 million monthly active users (MAU) [11, 21, 58]. We then crawled the *Game* category of these platforms using the open-source tool *miniCrawler* [74]. This process yielded 6,769 mini-games in total, from which we further extracted 2,076 Cocos-based mini-games (1,593 from WeChat, 312 from Facebook, and 171 from QuickGame). Based on this dataset, the remainder of this section provides an overview of the advertising landscape, a detailed analysis of the distribution and behavioral patterns of aggressive advertising, and a summary of our findings reported to platforms.

5.1 Aggressive Advertising Landscape

Applying *MAAD* to the collected dataset, we observed that aggressive advertising behaviors are highly prevalent across the three major platforms. As shown in Table 4, 49.95% of ad-enabled mini-games contained at least one type of aggressive advertising, underscoring developers' strong incentive to maximize impressions. Among the categories, *interruptive advertising* was the most common: its triggering is tied to engine lifecycle functions, allowing seamless embedding into gameplay and guaranteeing that advertisements will be displayed in a controlled manner. For aggressive advertising behaviors that are not explicitly specified in certain platform policies, we apply a unified detection criterion. Our result shows that, although such behaviors appear at lower rates compared to categories explicitly prohibited by platforms, they are not absent across the platforms, e.g., Hijacking Advertising in WeChat (6.85%) and Facebook (4.82%). This indicates that the absence of explicit policy constraints does not imply the absence of risk, but rather reveals an emerging aggressive advertising behavior that may not yet be fully recognized or addressed by current platform policies.

Table 4: Real-world Landscape of Aggressive Advertising across WeChat, Facebook, and QuickGame. Detection follows platform-specific advertising policies, except where explicitly marked.

Mini-game Platform	Interruptive Ads	Hijacking Ads	Unstoppable Ads	Deceptive Ads	Aggressive-Ad / With-Ad
WeChat Mini-game	289 (43.07%)	46 (6.85%)†	65 (9.69%)†	7 (1.04%)	315 / 671 (46.94%)
Facebook Instant Game	63 (43.45%)	7 (4.82%)†	29 (20.00%)	0 (0%)†	80 / 145 (55.17%)
QuickGame	55 (55.55%)	16 (16.16%)	17 (17.17%)	8 (8.08%)	62 / 99 (62.62%)
Total	407 (44.48%)	69 (7.54%)	111 (12.13%)	15 (1.64%)	457 / 915 (49.95%)

† Marked categories not explicitly specified in platform policies. We detected using a unified cross-platform behavioral definition.

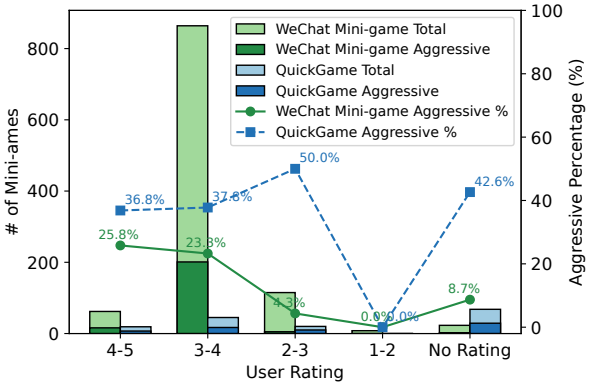

Figure 7: Distribution and Prevalence of Aggressive Advertising across User Rating Intervals in WeChat and QuickGame.

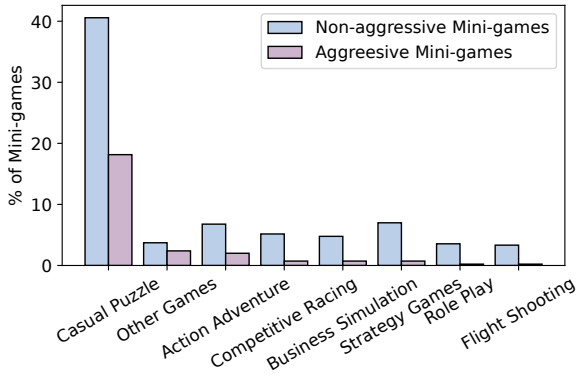

Figure 8: Distribution of Game Category among Aggressive and Non-aggressive Mini-games.

⇒ **Takeaway III**: *49.95% of ad-enabled mini-games exhibit aggressive advertising behaviors, dominated by interruptive advertising.*

5.2 Profiling Aggressive-Ad Mini-games

Beyond measuring overall prevalence, we further investigate the characteristics of mini-games with aggressive advertising, aiming to uncover their commonalities and distributional preferences.

Game Popularity. To examine the relationship between popularity and aggressive advertising, we collected user ratings from WeChat and QuickGame marketplaces. Since WeChat adopts a five-star scale while QuickGame uses percentage scores, we normalized both into a five-point scale.

Figure 7 illustrates the prevalence of aggressive advertising across rating intervals. In QuickGame, lower-rated games exhibit notably higher aggressive advertising rates (50.0% in the 2–3 rating), indicating that aggressive advertising strongly undermines user experience. By contrast, WeChat mini-games display that high-rated games contain the most aggressive advertising rates (25.8% in the 4–5 rating), suggesting that even high-rated mini-games are willing to compromise user experience with aggressive advertising, alarming in the ecosystem.

Notably, among these mini-games, we identified eight mini-games with more than 100k user ratings that still contained aggressive advertising. Given their large user base, the impact of such practices is substantial. Reviews of six of these popular mini-games explicitly mentioned advertising, most often expressing dissatisfaction with the excessive frequency of advertising and a wish for its reduction.

⇒ **Takeaway IV**: *Aggressive advertising substantially affects user experience: they drive down ratings in QuickGame and persist even in highly popular games with massive user bases (over 100k ratings), where users frequently complain about excessive advertising.*

Game Categories. We further analyzed the distribution of aggressive advertising across different game categories, covering eight types: action adventure, role-playing, competitive racing, flight shooting, business simulation, strategy, casual puzzle, and others. Figure 8 presents the comparison between aggressive and non-aggressive mini-games.

The results show that *casual puzzle* games dominate both in absolute number and in aggressive advertising prevalence. This trend can be explained by the lower development cost and broader audience appeal of *casual puzzle* games, which make them an attractive target for developers to maximize advertising revenue. Other categories, such as *action adventure* and *competitive racing*, also contain aggressive advertising,

but their proportions are substantially smaller.

> ⇒ **Takeaway V**: *Aggressive advertising is most concentrated in casual puzzle games, reflecting developers' preference for low-cost, high-reach games as vehicles for intrusive advertising.*

Developer Clusters. We also examined the distribution of aggressive advertising mini-games across developers. For WeChat and QuickGame platforms, company-level information was obtained from the official mini-game information pages. After excluding unknown and individual developers, we identified 126 companies in total, among which 40 had published more than one mini-game containing aggressive advertising. This finding suggests that aggressive advertising is not confined to isolated cases but tends to recur within particular organizational clusters.

In addition, we observed substantial code-level similarities among aggressive advertising mini-games. To measure code duplication between any two mini-games, we used PMD-CPD [1], an open-source copy–paste detection tool that identifies duplicated code fragments across programs using token-based analysis. We identified 18 groups comprising 42 games with highly similar or even identical code, spanning 31 companies. Two recurring patterns emerged: (i) different companies publishing nearly identical code, which may indicate either shared developer control across multiple firms or the adoption of common prebuilt templates; and (ii) single companies repeatedly releasing nearly identical games with only minor modifications (e.g., renaming), thereby amplifying their presence in the marketplace.

> ⇒ **Takeaway VI**: *Aggressive advertising mini-games are not isolated incidents but cluster within certain companies, with evidence of code reuse and template-driven replication that amplify their ecosystem-wide impact.*

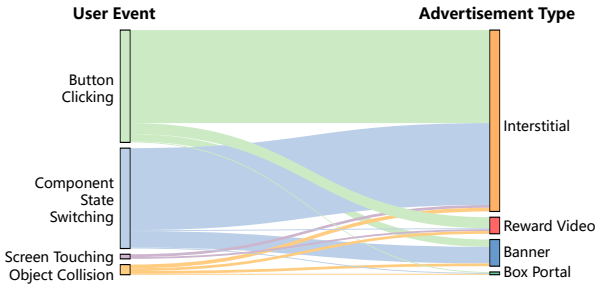

Figure 9: Relation of User Events and Advertisement Types in Mini-games with Aggressive Advertising.

5.3 Behavioral Patterns of Aggressive-Ad

Beyond profiling games and developers, we further examine the specific behavioral patterns through which aggressive advertising manifests in mini-games. Our analysis focuses on four aspects: the trigger selection, the pop-up frequency, the misleading strategies, and the bypass techniques. These dimensions provide a fine-grained view of how aggressive advertising is operationalized in practice.

Trigger Selection. To understand how aggressive advertising are triggered, we investigated the relationship between the triggering user events and the types of advertisements in aggressive *Ad-behavior* chosen by the developers, as illustrated in Figure 9. Overall, *interstitial* and *rewarded-video* advertisements are the most frequently adopted formats. Both occupy the screen centrally or in full, effectively interrupting user operations and disrupting gameplay. In terms of triggering methods, developers most often relied on *component state switching* and *button clicks*, with interstitials strongly associated with the former and rewarded videos with the latter.

Beyond these dominant patterns, we observed that *object collisions* represent a unique, game-specific triggering mechanism. Because collisions are inherently tied to player actions, this approach is both covert and easily activated. Advertisements often appeared precisely at critical gameplay moments, covering control areas and inducing accidental clicks. In addition, we identified cases where advertisements were triggered solely by *screen touching*. This approach is particularly disruptive: even an unintentional tap that does not correspond to actual gameplay may forcibly invoke an ad, severely degrading the user experience.

> ⇒ **Takeaway VII**: *Developers exploit game-specific mechanisms such as object collisions and screen touches to covertly trigger advertisements, maximizing their inevitability while severely degrading user experience.*

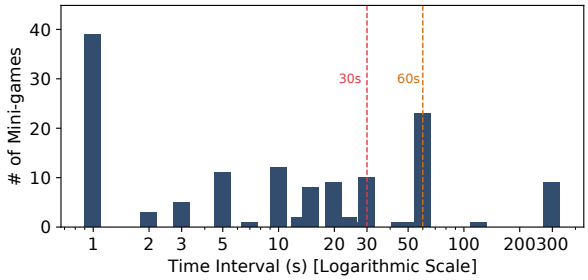

Figure 10: Distribution of Advertisement Pop-up Intervals (Log Scale). The red/orange lines denote platform policy thresholds: 30s on Facebook and QuickGame, and 60s on TikTok and Kuaishou.

Pop-up Frequency. We next examined how frequently ag-

gressive advertising is configured to appear. The interval here refers to the elapsed time between consecutive pop-ups, with shorter intervals indicating more aggressive behavior. From the parameters of interval functions (e.g., `schedule(a,t)`, `setTimeout(a,t)`), we extracted developer-defined time intervals and plotted their distribution in Figure 10.

We further compare them with two explicit platform policies: Facebook and QuickGame mandate at least 30 seconds between advertising, while TikTok and Kuaishou require a minimum of 60 seconds. Our results show that the majority of mini-games using interval functions violate these requirements, with 67.39% shorter than Facebook's 30-second threshold. This practice forces players to face recurrent interruptions at a much higher frequency than allowed, severely degrading the gameplay experience.

> ⇒ **Takeaway VIII**: *67.39% of aggressive advertising mini-games configured pop-up intervals shorter than the 30s required by Facebook, severely disrupting user experience through overly frequent interruptions.*

Misleading Strategies. We identified two major forms of misleading strategies that developers employ to increase advertising engagement. The first is *hijacking*, where normal functional buttons are repurposed to trigger advertisements. In particular, the most frequently hijacked buttons were "share", "start", and "score", all of which are integral to the mini-game experience and thus naturally attract user clicks. By binding advertisements to these functions, developers ensure that players encounter advertising during core interactions.

The second form is *textual inducement*, where developers deliberately craft misleading textual cues to lure users into clicking advertisements. Typical keywords include "free" and "double rewards", often combined with urgency-evoking modifiers such as "now" or "limited timing", and reinforced with exclamation marks to heighten the sense of immediacy. This combination of reward-oriented wording and urgent tone maximizes the likelihood of inducing unintended clicks.

> ⇒ **Takeaway IX**: *Misleading strategies exploit either functional hijacking of buttons or deceptive textual cues, causing users to click advertisements unintentionally.*

Bypass Techniques. Since super-app platforms have already established review mechanisms requiring mini-games to pass audits before release, aggressive advertising games face a high risk of being detected and rejected. To retain aggressive advertising while still securing approval and revenue, some developers deliberately adopt technical bypass strategies.

We analyzed the trigger conditions in the measurement. Based on whether the condition depends on data from network API requests, there are two categories: *static* and *dynamic*.

• *Static Bypass.* Developers embed hard-coded conditions that deliberately postpone or restrict advertising display. Typical examples include predefined time delays, disabling advertising during the first game load, requiring a large number of interactions before triggering advertisements (count-based triggers), or randomizing ad-related parameters to obscure advertising presentation. These make it difficult for short-duration or single-run audits to capture aggressive behaviors.

• *Dynamic Bypass.* Developers determine advertising behavior using values fetched from network API requests (cloud control), such as IP-based gating, server-side flags, time windows, or user/device signals. This approach is more flexible because multiple predicates can be combined (for example, region match and not-first-run and interaction-count,$\geq k$), and the logic can be updated remotely without resubmitting the game, which makes reproduction during audits challenging.

Across all bypass cases we analyzed, 94.09% relied on dynamic, network-dependent triggers. The dominance of cloud-controlled gating, together with its multi-condition flexibility and server-side mutability, exposes a challenging weakness in current review workflows.

> ⇒ **Takeaway X**: *Developers employ diverse bypass strategies, with 94.09% relying on dynamically controlled triggers, reflecting the limits of single-round or runtime-only audits in effectively curbing such abuses.*

5.4 Reporting and Feedback

Finally, we reported the aggressive advertising mini-games identified on the three platforms back to the respective providers. Both the WeChat and QuickGame teams confirmed the issues and expressed appreciation for our disclosure. In contrast, Facebook did not provide a response before the submission of this paper. By the time of submission, 43.32% of the reported mini-games had already been taken down by the platforms, demonstrating both the practical value of our detection results and the seriousness with which platforms regard aggressive advertising violations.

6 Discussion

Ecosystem-Level Security Implication. This work represents the first systematic study of aggressive advertising behaviors in the mini-game ecosystem. Our findings show that aggressive advertising is not an isolated usability issue but a systemic threat shaped by diverse platform policies and developer practices. Prior studies on policy inconsistency have mainly focused on privacy protection [68, 73] and content moderation [51], while the terms of use governing in-game advertising have received far less attention. Through a comprehensive policy analysis and real-world measurement, we find that inconsistencies across platforms and uneven enforcement create gaps that allow aggressive advertising to persist

at scale. These gaps further lead to practical consequences. Platforms lack a clear baseline for enforcement, developers lack best practices for compliance, and users face inconsistent experiences across platforms and unclear boundaries of their rights. By revealing these issues, our study and tool aim to make aggressive advertising more visible and to support healthier ecosystem development.

Evolving Game of Online Deception. Aggressive advertising in mini-games represents an evolving form of online deception [41, 42] rather than a simple misuse of platform mechanisms. Our study characterizes the current landscape of such deceptive behaviors and presents a configurable, scalable detection framework. In adversarial settings, platform enforcement and developer behavior follow an arms-race dynamic [6, 53], leading to more evasive and adaptive forms of deception that fixed rule sets cannot fully capture. This suggests the need for iterative refinement of detection mechanisms to address the long-term dynamics of online deception.

Technical Scalability. *MAAD* can be easily scaled and also offers insights for future technical improvements. First, the rule system is configurable. Thresholds differ across platforms, e.g., 60 seconds on TikTok. *MAAD* encodes these values as configurable parameters, allowing adaptation. Second, *MAAD* is platform-extensible. We have already implemented prototypes on six platforms (WeChat, Facebook, QuickGame, Alipay, Baidu, and VK). With appropriate preprocessing, the framework can be ported to additional platforms with minimal engineering effort. Third, *MAAD*'s ideas, e.g., bundle-aware pruning and event-entry reconstruction, can reduce bundle-ingrated overhead and enable hybrid static–dynamic analyses on broader JavaScript program analysis.

Limitations. *MAAD* is bounded by Cocos-centric design and static analysis challenges, including dynamic parameter passing, JavaScript polymorphism, and obfuscation as noted in Section 4.2. Limited by our sample size and the fact that dynamic interaction cannot exhaustively trigger all behaviors, some aggressive behaviors, particularly deceptive advertising, have lower support numbers, which may introduce potential bias in the following evaluation and measurement. Our measurement covers three major platforms, though broader coverage may reveal additional patterns. We focus on textual inducement, and we admit that MAAD can not fully cover all kinds of deception, i.e., image-based or multimodal inducements are out of scope. To enhance deceptive behavior detection, future work may leverage dark pattern detection techniques from the UI domain [9, 42]. Finally, *MAAD* relies solely on static analysis, so that certain behaviors that depend on real-time rendering or visual interaction properties, such as advertising overlapping, may require hybrid techniques to achieve reliable detection.

7 Related Work

Game Security. Research on game security mainly addresses game vulnerabilities, bot detection, and cheating. For vulnerabilities, Tao et al. [57], Zuo and Lin [77] identify payment security flaws in gaming platforms, Liu et al. [37] expose misuse of cloud gaming resources via malicious injections, and Macklon et al. [40] detect visual bugs in web-based games. Numerous studies [30, 35, 54, 56, 76] target game bots in online and mobile environments. For cheating, Bethea et al. [5] proposes a symbolic-execution method, Bursztein et al. [6] uncovers map hacking with a defensive strategy, and others [10, 29] leverage deep learning to detect cheating in first-person shooters. In summary, prior work mainly addressed vulnerabilities and cheating in games, while our study is the first to examine aggressive advertising in mini-games.

Mini-app Ecosystem Security. In the mini-app ecosystem, prior research has explored diverse security issues spanning vulnerabilities, API abuses, cross-mini-app attacks, malware, and privacy leakage. Early work examined flaws, such as resource-management vulnerabilities in app-in-app systems [39] and identity confusion problems [73]. Subsequent studies revealed API-related risks, including undocumented interfaces [69], cross-platform discrepancies [68], and their exploitation in WeChat and other super apps. Researchers also demonstrated attack vectors, such as access-control flaws and phishing [39], race conditions [74], and cross mini-app request forgery [71]. Meanwhile, another line of work focuses on malicious or privacy-invasive behaviors, including large-scale characterization of mini-app malware [72], credential leakage through mini-app ecosystems [50], and privacy leaks tracked via taint-analysis frameworks like WeMint [43], MiniTracker [34], and TaintMini [67]. Existing mini-app analyzing frameworks face a significant challenge in parsing the mini-game due to its integration with gaming engine. *MAAD* offers an effective solution to mini-game ecosystem.

Advertisement Security. The abuse of advertising for profit has spurred extensive research on advertisement security. One line of work focuses on ad fraud. Springborn and Barford [53] analyze impression fraud via PPV networks, DECAF [36] detects ads violating network policies, and others [7, 15, 28, 75] address click fraud, including large-scale click farms [55]. Another line of work focuses on malicious advertisements. Liu et al. [38] conduct a comprehensive study of malicious advertisement content. Son et al. [52] discloses the issues that advertisers utilize malicious advertisements to infer sensitive user information by accessing external storage. Other research examines deceptive content and dark patterns [9, 41, 42], which identify misleading ads or manipulative UI designs based on textual, visual, and layout cues, focusing on user-visible content or interface structures. In contrast, our work analyzes advertising abuse at the mini-game program level, capturing how deceptive or aggressive ads are enforced through event-driven logic beyond UI.

8 Conclusion

This paper presented the first systematic investigation of aggressive advertising in the mini-game ecosystem. Through the implementation of *MAAD* and our large-scale measurement study, we revealed both the prevalence and characteristic patterns of such behaviors across major platforms. Our findings highlight the feasibility of static analysis for large-scale auditing and underscore the need for sustained technical and regulatory efforts to mitigate aggressive advertising as a systemic threat to ecosystem integrity.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments, which improved the quality of the paper. This work was supported by the New Generation Artificial Intelligence-National Science and Technology Major Project (No. 2025ZD0123204). Min Yang is a faculty of Shanghai Institute of Intelligent Electronics & Systems and Engineering Research Center of Cyber Security Auditing and Monitoring, and Shanghai Collaborative Innovation Center of Intelligent Visual Computing, Ministry of Education, China.

Ethical Considerations

In conducting this work, we paid close attention to the ethical dimensions of our methodology and its potential impact. In particular, we considered five stakeholder groups that may be impacted: (i) mini-game players, (ii) mini-game developers, (iii) mini-game platforms, (iv) research community, and (v) our own research team. For each group, we discuss the guiding ethical principles, the potential harms, the mitigations we applied, and our rationale for publishing.

(i) Mini-game Players. *Principles:* Following *Beneficence* and *Respect for Persons*, we aim to protect players from intrusive advertising and respect their privacy. *Harms:* Players may face degraded gameplay, frustration, or unintended costs; misuse of research artifacts could worsen these risks. *Mitigations:* We do not collect personal data, account information, or behavioral logs. Our analysis is limited to static packages and reported only in aggregate, ensuring no individual player is implicated. Moreover, our findings are framed to highlight protective implications rather than to provide misuse instructions. *Decision:* Publishing increases transparency and informs stronger enforcement, ultimately benefiting players while keeping risks minimal.

(ii) Mini-game Developers. *Principles:* Guided by *Justice* and *Respect for Persons*, we avoid stigmatizing developers and distinguish systemic issues from individual misconduct. *Harms:* Developers could face reputational damage or indirect economic effects if platforms tighten policies. *Mitigations:* All major mini-game platforms (e.g., WeChat, Facebook Instant Games) state in their developer agreements that submitted games are subject to compliance and safety reviews by the platform. Developers are required to submit a full mini-game package, including scripts, resources, and metadata, for pre-release review and publication. Our detection pipeline treats all samples equally and operates without accessing developer-identifying information. All analyzed packages were obtained from public sources, ensuring compliance with platform policies and preventing undue exposure of individual developers. *Decision:* We remove all developer personally identifiable information in disclosure. *MAAD* can also assist developers in self-assessing their mini-games for compliance.

(iii) Mini-game Platforms. *Principles:* Consistent with *Respect for Law and Public Interest*, *Beneficence*, and *Justice*, we acknowledge platforms' duty to protect users and the need for fair evaluation. *Harms:* Platforms may face reputational or regulatory pressure, and technical details could be abused if misused. *Mitigations:* We conducted responsible disclosure through official channels prior to publication, notifying platforms of risky parameter settings and default behaviors. In addition, we reported the aggressive mini-games identified in our measurement study (Section 5) back to the respective providers. Both the WeChat and QuickGame teams confirmed the issues and expressed appreciation for our disclosure, while Facebook did not respond before submission. By the submission deadline, 68.86% of the reported mini-games had already been removed by the platforms, demonstrating both the practical value of our detection results and the seriousness with which platforms regard aggressive advertising violations. In presenting our results, we use neutral framing that emphasizes ecosystem-wide challenges rather than attributing fault to a specific platform. We also deliberately omit implementation details that could facilitate adversarial exploitation. *Decision:* While disclosure may create short-term concerns, it strengthens compliance and user protection in the long run.

(iv) Research Community. *Principles:* Following *Beneficence*, *Justice*, and *Respect for Law and Public Interest*, we aim to advance understanding while ensuring fair and responsible access. *Harms:* Excessive detail could enable evasion, while overly restricting artifacts may hinder reproducibility. *Mitigations:* We strike a balance by releasing curated artifacts that support reproducibility without enabling abuse. Publicly released materials include the source code of *MAAD*, anonymized test cases, and documents. For platform-specific rule-matching components, we provide a packaged decision system rather than exposing raw patterns, preventing adversaries from directly extracting patterns for targeted evasion. Sensitive details—such as adversarial case studies—are withheld or abstracted. This approach complies with the conference's Open Science requirements while mitigating dual-use risks. *Decision:* Selective disclosure maximizes academic value while minimizing misuse.

(v) Research Team. *Principles:* In line with *Accountability*, *Beneficence*, and *Justice*, we commit to rigorous and trans-

parent research. *Harms:* Risks include reputational damage if labeling is biased or if data provenance is unclear. *Mitigations:* To mitigate these risks, three independent researchers labeled portions of the dataset, with disagreements resolved through consensus to minimize individual bias. Clear labeling guidelines were developed in advance, and the process was documented to ensure reproducibility. All test data was collected exclusively from publicly available sources (open-source crawler *MiniCrawler* [74]), ensuring compliance with platform policies and avoiding any unauthorized access. The annotated data used for evaluation was anonymized and restricted to the minimum necessary scope, preventing any unnecessary exposure of developers or users. *Decision:* Careful annotation, transparency, and exclusive use of public data make our process ethically sound and its dissemination beneficial to both research and society.

Open Science

In accordance with the USENIX Security Open Science policy, we provide the following artifacts necessary to evaluate our contributions:

(i) Source code of *MAAD*: The full implementation of our static analysis framework, including mini-game code pruning, *Ad-behavior* extraction, and aggressiveness detection modules.

(ii) Anonymized test cases: A collection of representative mini-games from six different platforms, curated and anonymized to demonstrate the cross-platform compatibility of *MAAD*.

(iii) Documentation: Detailed materials that describe the workflow of *MAAD*, the configuration of the experimental environment, a step-by-step usage guide, and explanations of the output file formats.

All artifacts are available at the following link: https://doi.org/10.5281/zenodo.18227703. Sensitive or proprietary content (e.g., raw user traffic, unmodified platform submissions) has been excluded to prevent potential misuse.

References

[1] Pmd copy/paste detector (cpd). https://pmd.github.io/pmd/pmd_userdocs_cpd.html, 2025.

[2] Alipay Open Platform. Mini program advertising rules, 2024. URL https://opendocs.alipay.com/rules/0a9d84.

[3] Apple Inc. In-app purchase, 2025. URL https://developer.apple.com/in-app-purchase/.

[4] Baidu Smart Program. Operations specification – violative advertising behavior, 2024. URL https://smartprogram.baidu.com/docs/operations/specification/.

[5] Darrell Bethea, Robert A Cochran, and Michael K Reiter. Server-side verification of client behavior in online games. *ACM Transactions on Information and System Security (TISSEC)*, 14(4):1–27, 2008.

[6] Elie Bursztein, Mike Hamburg, Jocelyn Lagarenne, and Dan Boneh. Openconflict: Preventing real time map hacks in online games. In *2011 IEEE Symposium on Security and Privacy*, pages 506–520. IEEE, 2011.

[7] Chenhong Cao, Yi Gao, Yang Luo, Mingyuan Xia, Wei Dong, Chun Chen, and Xue Liu. Adsherlock: efficient and deployable click fraud detection for mobile applications. *IEEE Transactions on Mobile Computing*, 20(4): 1285–1297, 2020.

[8] IBM T.J. Watson Research Center. Wala: T.j. watson libraries for analysis, 2012. URL https://github.com/wala/WALA.

[9] Jieshan Chen, Jiamou Sun, Sidong Feng, Zhenchang Xing, Qinghua Lu, Xiwei Xu, and Chunyang Chen. Unveiling the tricks: Automated detection of dark patterns in mobile applications. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23)*, pages 114–133, 2023. doi: 10.1145/3586183.3606783.

[10] Minyeop Choi, Gihyuk Ko, and Sang Kil Cha. {BotScreen}: Trust everybody, but cut the aimbots yourself. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 481–498, 2023.

[11] Cocos. Oppo & vivo games of the year announced, cocos developers win big, 2021. URL https://www.cocos.com/en/post/oppo-vivo-games-of-the-year-announced-cocos-developers-win-big.

[12] Cocos. How to make a great game with cocos creator. https://www.cocos.com/en/post/b9cf16c6bc32c1f73bd7a762c6dc43e0, 2024.

[13] Cocos Technology Co., Ltd. Cocos creator: A free, open-source, cross-platform game engine, 2024. URL https://www.cocos.com/en/creator.

[14] Federal Trade Commission. Advertising and marketing, 2025. URL https://www.ftc.gov/business-guidance/advertising-marketing.

[15] Jonathan Crussell, Ryan Stevens, and Hao Chen. Madfraud: Investigating ad fraud in android applications. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 123–134, 2014.

[16] cwi swat. Field-based Call Graph Construction for JavaScript. https://github.com/cwi-swat/javascript-call-graph, 2014.

[17] WeChat Developers. Game engine best practices for wechat mini games. https://developers.weixin.qq.com/minigame/dev/guide/best-practice/game-engine.html, 2024.

[18] Weixin Developers. Weixin mini program code compilation. https://developers.weixin.qq.com/miniprogram/dev/devtools/codecompile.html, 2024.

[19] Douyin Open Platform. Mini game operation norms, 2024. URL https://developer.open-douyin.com/docs/resource/zh-CN/mini-game/operation1/norms/norms#167418a1.

[20] Facebook. Facebook instant-games docs. https://www.facebook.com/fbgaminghome/developers/instant-games, 2024.

[21] Facebook Gaming. Changes to the instant games platform, 2020. URL https://www.facebook.com/fbgaminghome/blog/changes-to-the-instant-games-platform/.

[22] Russian Federation. Federal law "on advertising" no. 38-fz of march 13, 2006, 2025. URL https://www.consultant.ru/document/cons_doc_LAW_58968/f67f81c57fdcdacc2643d19d59369f7e185e1156/.

[23] S. Fink and J. Dolby. *WALA–The TJ Watson Libraries for Analysis*, 2012. URL https://github.com/wala/WALA.

[24] Game Publishing Committee (GPC) of the China Audio-Video and Digital Publishing Association and Gamma Data (CNG). China game industry report 2024 (official release), 2025. URL https://www.cgigc.com.cn/details.html?id=08dd2ada-5934-4680-892e-63760092eef9&tp=news.

[25] GitHub. *CodeQL*, 2023. URL https://codeql.github.com/.

[26] Google LLC. Google play's billing system, 2025. URL https://developer.android.com/google/play/billing.

[27] Ariya Hidayat. Esprima: Ecmascript parsing infrastructure for multi-purpose analysis, 2024. URL https://esprima.org/.

[28] Md Shahrear Iqbal, Md Zulkernine, Fehmi Jaafar, and Yuan Gu. Fcfraud: Fighting click-fraud from the user side. In *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*, pages 157–164. IEEE, 2016.

[29] Md Shihabul Islam, Bo Dong, Swarup Chandra, Latifur Khan, and Bhavani Thuraisingham. Gci: A gpu-based transfer learning approach for detecting cheats of computer game. *IEEE Transactions on Dependable and Secure Computing*, 19(2):804–816, 2020.

[30] Ah Reum Kang, Huy Kang Kim, and Jiyoung Woo. Chatting pattern based game bot detection: do they talk like us? *KSII Transactions on Internet and Information Systems (TIIS)*, 6(11):2866–2879, 2012.

[31] Kuaishou Open Platform. Mini game ad authorization specification, 2024. URL https://open.kuaishou.com/miniGameDocs/operation/specification/ad-auth.html.

[32] Mathias Rud Laursen, Wenyuan Xu, and Anders Møller. Reducing static analysis unsoundness with approximate interpretation. *Proceedings of the ACM on Programming Languages (PACMPL)*, 4(PLDI):194:1–194:24, 2024.

[33] Layabox Technology Co., Ltd. Layaair: Open-source 2d/3d engine for high-performance mini-games, 2023. URL https://layaair.com/.

[34] Wei Li, Borui Yang, Hangyu Ye, Liyao Xiang, Qingxiao Tao, Xinbing Wang, and Chenghu Zhou. Minitracker: Large-scale sensitive information tracking in mini apps. *IEEE Transactions on Dependable and Secure Computing*, 2023.

[35] Wenbin Li, Xiaokai Chu, Yueyang Su, Di Yao, Shiwei Zhao, Runze Wu, Shize Zhang, Jianrong Tao, Hao Deng, and Jingping Bi. Fingformer: Contrastive graph-based finger operation transformer for unsupervised mobile game bot detection. In *Proceedings of the ACM Web Conference 2022*, pages 3367–3375, 2022.

[36] Bin Liu, Suman Nath, Ramesh Govindan, and Jie Liu. {DECAF}: Detecting and characterizing ad fraud in mobile apps. In *11th USENIX symposium on networked systems design and implementation (NSDI 14)*, pages 57–70, 2014.

[37] Guannan Liu, Daiping Liu, Shuai Hao, Xing Gao, Kun Sun, and Haining Wang. Ready raider one: Exploring the misuse of cloud gaming services. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1993–2007, 2022.

[38] Tianming Liu, Haoyu Wang, Li Li, Xiapu Luo, Feng Dong, Yao Guo, Liu Wang, Tegawendé Bissyandé, and Jacques Klein. Maddroid: Characterizing and detecting devious ad contents for android apps. In *Proceedings of The Web Conference 2020*, pages 1715–1726, 2020.

[39] Haoran Lu, Luyi Xing, Yue Xiao, Yifan Zhang, Xiaojing Liao, XiaoFeng Wang, and Xueqiang 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*, pages 569–585, 2020.

[40] Finlay Macklon, Mohammad Reza Taesiri, Markos Viggiato, Stefan Antoszko, Natalia Romanova, Dale Paas, and Cor-Paul Bezemer. Automatically detecting visual bugs in html5 canvas games. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–11, 2022.

[41] S. M. Hasan Mansur, Sabiha Salma, Damilola Awofisayo, and Kevin Moran. Aidui: Toward automated recognition of dark patterns in user interfaces. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2023. doi: 10.1109/ICSE48619.2023.00166.

[42] Arunesh Mathur, Gunes Acar, Michael J. Friedman, Elena Lucherini, Jonathan Mayer, Marshini Chetty, and Arvind Narayanan. Dark patterns at scale: Findings from a crawl of 11k shopping websites. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW): 1–32, 2019.

[43] Shi Meng, Liu Wang, Shenao Wang, Kailong Wang, Xusheng Xiao, Guangdong Bai, and Haoyu Wang. Wemint: Tainting sensitive data leaks in wechat mini-programs. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1403–1415. IEEE, 2023.

[44] Meta Platforms, Inc. *Audience Network Interstitial Ads*, 2023. URL https://developers.facebook.com/docs/audience-network/guides/ad-formats/interstitial.

[45] Meta Platforms, Inc. Instant games playbook. https://developers.facebook.com/resources/IG-Playbook-EN.pdf, 2024.

[46] People Republic of China. Advertising law of the people's republic of china, 2021. URL https://www.samr.gov.cn/zw/zfxxgk/fdzdgknr/fgs/art/2023/art_5474cf75173c45d6a0379730fb4e8d97.html.

[47] Douyin Open Platform. Interstitial ad notice, 2025. URL https://developer.open-douyin.com/docs/resource/zh-CN/mini-game/develop/api/javascript-api/ads/interstitial-ad/interstitial-ad-notice.

[48] SWC Project. swc: A super-fast compiler written in rust, 2024. URL https://github.com/swc-project/swc.

[49] Quickgame. Quickgames docs. https://minigame.vivo.com.cn/documents, 2024.

[50] Yizhe Shi, Guangliang Yang, Zhemin Yang, Yifan Yang, Min Yang, Kangwei Zhong, and Xiaohan Zhang. The skeleton keys: A large scale analysis of credential leakage in mini-apps. In *Proceedings of the 32nd Network and Distributed Systems Security Symposium (NDSS)*, 2025.

[51] Mohit Singhal et al. Sok: Content moderation in social media, from guidelines to enforcement and research to practice. In *2023 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2023.

[52] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. What mobile ads know about mobile users. In *NDSS*, 2016.

[53] Kevin Springborn and Paul Barford. Impression fraud in on-line advertising via {Pay-Per-View} networks. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 211–226, 2013.

[54] Yueyang Su, Di Yao, Xiaokai Chu, Wenbin Li, Jingping Bi, Shiwei Zhao, Runze Wu, Shize Zhang, Jianrong Tao, and Hao Deng. Few-shot learning for trajectory-based mobile game cheating detection. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 3941–3949, 2022.

[55] Suibin Sun, Le Yu, Xiaokuan Zhang, Minhui Xue, Ren Zhou, Haojin Zhu, Shuang Hao, and Xiaodong Lin. Understanding and detecting mobile ad fraud through the lens of invalid traffic. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 287–303, 2021.

[56] Jianrong Tao, Jiarong Xu, Linxia Gong, Yifu Li, Changjie Fan, and Zhou Zhao. Nguard: A game bot detection framework for netease mmorpgs. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 811–820, 2018.

[57] Jianrong Tao, Jianshi Lin, Shize Zhang, Sha Zhao, Runze Wu, Changjie Fan, and Peng Cui. Mvan: Multi-view attention networks for real money trading detection in online games. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2536–2546, 2019.

[58] TechNode. Wechat mini-program games hit 500 million monthly users, 2025. URL https://technode.com/2025/06/26/wechat-mini-program-games-hit-500-million-monthly-users-pc-usage-surges/.

[59] Tencent. Ads guidelines of weixin minigames. https://wxa.wxs.qq.com/wxadtouch/upload/t2/533fyowz_d7c30427.pdf.

[60] Tencent. Weixin mini games docs. https://developers.weixin.qq.com/minigame/dev/guide, 2024.

[61] Tencent. Weixin mini game banner ad api documentation. https://developers.weixin.qq.com/minigame/en/dev/guide/open-ability/ad/banner-ad.html, 2024.

[62] UC Mini Game Platform. Mini game operating specifications, 2024. URL https://minigame.uc.cn/design/operating#QA4j4.

[63] European Union. Article 26 — digital services act (advertising on online platforms), 2022. URL https://www.eu-digital-services-act.com/Digital_Services_Act_Article_26.html.

[64] VIVO. Guidelines for app verification on the developers platform. https://developer.vivo.com/doc/detail?id=68.

[65] Vivo. Vivo mini game interstitial ad api documentation. https://minigame.vivo.com.cn/documents/#/api/ad/custom-ad, 2024.

[66] VK Developers. Mini apps rules – advertising, 2024. URL https://dev.vk.com/en/mini-apps-rules#5.1.%20Advertising.

[67] Chao Wang, Ronny Ko, Yue Zhang, Yuqing Yang, and Zhiqiang 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)*, pages 932–944. IEEE, 2023.

[68] Chao Wang, Yue Zhang, and Zhiqiang Lin. One size does not fit all: Uncovering and exploiting cross platform discrepant {APIs} in {WeChat}. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6629–6646, 2023.

[69] Chao Wang, Yue Zhang, and Zhiqiang Lin. Uncovering and exploiting hidden apis in mobile super apps. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2471–2485, 2023.

[70] WeChat Ads. Mini game publisher operation policy, 2024. URL https://ad.weixin.qq.com/docs/239.

[71] Yuqing Yang, Yue Zhang, and Zhiqiang Lin. Cross miniapp request forgery: Root causes, attacks, and vulnerability detection. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 3079–3092, 2022.

[72] Yuqing Yang, Yue Zhang, and Zhiqiang Lin. Understanding miniapp malware: Identification, dissection, and characterization. In *Proceedings 2025 Network and Distributed System Security Symposium. San Diego, CA, USA*, 2025.

[73] Lei Zhang, Zhibo Zhang, Ancong Liu, Yinzhi Cao, Xiaohan Zhang, Yanjun Chen, Yuan Zhang, Guangliang Yang, and Min Yang. Identity confusion in {WebView-based} mobile app-in-app ecosystems. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1597–1613, 2022.

[74] Yue Zhang, Bayan Turkistani, Allen Yuqing Yang, Chaoshun Zuo, and Zhiqiang Lin. A measurement study of wechat mini-apps. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 5(2):1–25, 2021.

[75] Zhen Zhang, Yu Feng, Michael D Ernst, Sebastian Porst, and Isil Dillig. Checking conformance of applications against gui policies. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 95–106, 2021.

[76] Sha Zhao, Junwei Fang, Shiwei Zhao, Runze Wu, Jianrong Tao, Shijian Li, and Gang Pan. T-detector: A trajectory based pre-trained model for game bot detection in mmorpgs. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 992–1003. IEEE, 2022.

[77] Chaoshun Zuo and Zhiqiang Lin. Playing without paying: Detecting vulnerable payment verification in native binaries of unity mobile games. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3093–3110, 2022.

A Policy Annotation Process

Our policy annotation process began by retrieving all official policy documents [2, 4, 19, 31, 44, 59, 62, 64, 66]. We then searched for clauses containing the keyword "ad" (and its variants such as advertisement or advertising). Each clause was independently reviewed by two authors and labeled if it explicitly described developer obligations or restrictions on advertising behavior.

Next, we grouped semantically similar restrictions into candidate clusters (*e.g.,* statements prohibiting mid-game pop-ups or auto-triggered interstitials were grouped together). We then organized these clusters according to the primary user-facing consequence of the aggressive ad, which resulted in four broad categories: *Interruptive Advertising*, *Hijacking Advertising*, *Unstoppable Advertising*, and *Deceptive Advertising*. Each category further encompasses several concrete behaviors, yielding a total of 14 behaviors in our taxonomy.

B Implementation Details of Stage III

B.1 Multiple-Conditions Inference Rules

Here, we summarize the major inference logic for the four aggressive advertising categories.

Interruptive Advertising. An *Ad-behavior* is inferred as interruptive advertising when its *User Event* is a scene lifecycle transition (e.g., `onEnable`, `onLoad`), and its *Advertisement*

Type is an *interstitial*. These advertisements occur before the user initiates any action and disrupt normal gameplay. An *Ad-behavior* is also inferred as interruptive advertising when the *User Event* is activated through *object collision*, because such unexpected pop-ups interrupt the user's intended interactions.

Hijacking Advertising. An *Ad-behavior* is inferred as hijacking advertising when its *User Event* is a *button clicking*, its *User Interface Semantic* contains functional gameplay keywords, and its *Advertisement Type* is *interstitial*. Ads triggered by `screen touching` are also inferred as hijacking, as they intercept any touch interaction regardless of user intent. Furthermore, if the extracted ad parameters show that `can_close` is `false`, which prevents users from dismissing the ad, the *Ad-behavior* is inferred as hijacking.

Unstoppable Advertising. An *Ad-behavior* is inferred as unstoppable advertising when the value of *Triggering Condition* shows repeated activation at high frequency. If the interval between consecutive triggers is shorter than 30 seconds, we treat it as unstoppable advertising. We also detect an unstoppable pattern where closing a *banner* is followed by a timer, and the timer re-displays the same banner after a short delay. As shown in *Call Path*, this forms a loop that unstoppably forces the advertisement back into view.

Deceptive Advertising. An *Ad-behavior* is inferred as deceptive advertising when its *User Event* is a *button clicking* and its *User Interface Semantic* induces user interaction by implying a non-advertising reward, while the click directly triggers an advertisement. This semantic–behavioral mismatch between the implied action and the actual advertising outcome defines deceptive advertising. For example, if users click a button labeled "Click to get rewards" or "Get double coins", an advertisement is shown. The button disguises an ad click as a reward action, which constitutes deceptive advertising.

B.2 Inducive Text Classifier

For inducive-text detection, we use a binary classifier built on a fine-tuned XLM-RoBERTa-base model (278M parameters, 201 tensors) to identify bilingual inducive expressions in mini-game UI elements. Its multilingual pretraining enables a unified representation of English and Chinese text, avoiding separate monolingual models and improving robustness to mixed-language interfaces. We distilled representative inducive patterns from real mini-games and augmented them with LLM-generated phrases under custom part-of-speech rules. The final dataset contains 34,479 positive and 20,879 negative samples. With an 8:1:1 train–validation–test split, the classifier achieved 99.8% precision on the test set, confirming its effectiveness in supporting Stage III.

C Call Graph Tool Selection

We employed static analysis tools to generate the call graph. We evaluated four widely used JavaScript call graph construction frameworks, namely *WALA*[23], *CodeQL*[25], *ACG*[16], and *Jelly*[32], and manually verified their precision. As shown in Table 5, all of these tools are able to produce *Ad-behaviors*, yet their performance characteristics differ substantially. WALA achieved the highest accuracy, Jelly demonstrated the shortest analysis time, and ACG identified the largest number of *Ad-behaviors* due to its field-based static analysis. However, ACG also produced a significant number of false positives, as it fails to differentiate identically named properties across distinct objects, thereby introducing numerous incorrect call edges. Although CodeQL exhibited reasonable accuracy, its database-centric approach to code analysis is not well-suited for global data flow analysis in mini-games, as it incurs considerable performance overhead. Based on these findings, we ultimately selected WALA as the call graph construction tool in *MAAD*, since it provides sound, propagation-based analysis for JavaScript programs.

Table 5: The Performance Comparison of CG Tools

CG Tool	Runtime		*Ad-behaviors*	
	Avg. (s)	Me. (s)	Avg. (#)	Prec. (%)
WALA	212.90	50.22	295.84	94.57%
CodeQL	209.75	72.58	68.47	75.59%
ACG	89.13	12.89	2,941.65	37.33%
Jelly	55.77	6.01	576.70	67.54%

D Large-scale Deployment

To evaluate the practicality of *MAAD* in realistic scenarios, we deployed it on a large-scale real-world dataset from an anonymous cooperating platform containing Cocos mini-game packages submitted for pre-release review in March 2024. We first excluded corrupted or invalid files, leaving 1,613 Cocos mini-games. Under the same experimental setting with 8 threads, *MAAD* completed the analysis of all mini-games in 16.3 hours.

From this deployment, *MAAD* successfully identified aggressive advertising behaviors in 877 mini-games (54.37%). Notably, interrupting pop-ups was the most prevalent type, appearing in 35.17% of all mini-games, which highlights the widespread adoption of intrusive advertising strategies. Among those mini-games, the median number of aggressive *Ad-behavior* is 52, with more than 25% holding at least 100 aggressive *Ad-behaviors*. These findings not only provide a concrete picture of the current mini-game advertising ecosystem but also demonstrate that *MAAD* is both efficient and effective in supporting platform-scale auditing tasks, thereby validating its real-world applicability.