Artifact Evaluated NDSS Available

ChameleoScan: Demystifying and Detecting iOS Chameleon Apps via LLM-Powered UI Exploration

Hongyu Lin[1(*)], Yicheng Hu[1(*)], Haitao Xu[1(✉)], Yanchen Lu[1], Mengxia Ren[1], Shuai Hao[2], Chuan Yue[3]
Zhao Li[4], Fan Zhang[1] and Yixin Jiang[5]
[1]Zhejiang University, [2]Old Dominion University, [3]Colorado School of Mines
[4]Hangzhou Yugu Technology, [5]Electric Power Research Institute, CSG
{22321068,12521215,haitaoxu,v0n,fanzhang}@zju.edu.cn, shao@odu.edu, chuanyue@mines.edu
mengxiaren@hotmail.com, lzjoey@gmail.com, jiangyx@csg.cn

***Abstract*—Chameleon apps evade iOS App Store review by presenting legitimate functionality during submission while transforming into illicit variants post-installation. While prevalent, their underlying transformation methods and developer-user collusion dynamics remain poorly understood. Existing detection approaches, constrained by static analysis or metadata dependencies, prove ineffective against hybrid implementations, novel variants, or metadata-scarce instances. To address these limitations, we establish a curated dataset of 500 iOS Chameleon apps collected through covert distribution channels, enabling systematic identification of 10 categories of distinct transformation patterns (including 4 previously undocumented variants). Building upon these findings, we present ChameleoScan, the first LLM-driven automated UI exploration framework for reliable Chameleon app verification. The system maintains local decision interpretability while ensuring global detection consistency through its core innovation - predictive metadata analytics, semantic interface comprehension, and human-comparable interaction strategies. Comprehensive evaluation on 1,644 iOS apps demonstrates operational efficacy (9.85% detection rate, 92.59% precision), with findings formally acknowledged by Apple. Implementation and datasets are available at https://github.com/ChameleoScan.**

I. Introduction

With the increasing popularity of mobile apps, malicious actors have increasingly leveraged these platforms to promote and enable illicit services. To achieve wider and more convenient dissemination, they have begun distributing such apps through official channels like the iOS App Store, exploiting its credibility and extensive user base. Despite Apple's rigorous review process, it is not entirely infallible, and illicit apps can still proliferate on the platform [1], [2]. In one year alone, 35,245 fraudulent apps were removed from the App Store [3].

A recent tactic adopted by illicit apps to evade App Store review involves masquerading as legitimate apps during the vetting process and later transforming into their true form (*e.g.*, gambling or pirated apps) after approval, triggered by specific actions. This process typically depends on collusion between developers and users. Developers submit disguised apps to the App Store while sharing transformation methods (*e.g.*, entering a predefined string in a feedback form) through unofficial channels. Users obtain these methods, download the disguised apps from the App Store, and activate the transformation.

We refer to such apps, whose transformation depends on cooperation between developers and users, as *collusion-based Chameleon apps*, though we still use the term *Chameleon apps* for consistency and convenience. This definition is aligned with that in [4], but our study specifically highlights the collusion aspect, which has not been fully explored in prior studies. Beyond this collusion, other key aspects of Chameleon apps, such as their transformation mechanisms and the dissemination channels, remain largely unexamined and poorly understood. Gaining a deeper insight into these mechanisms is crucial for developing effective defensive approaches.

Existing approaches for detecting Chameleon apps exhibit notable shortcomings. Lee *et al.* [5] pioneered research in this domain by developing Chameleon-Hunter, a static analysis tool that detects concealed interfaces by examining semantic discrepancies in user interfaces (UIs) extracted from native code. This approach proves ineffective against hybrid Chameleon apps that incorporate dynamically rendered UI components – a predominant category according to Zhao *et al.* [6]. In response, they introduced Mask-Catcher, which employs a multi-stage detection process: identifying suspicious candidates through inconsistencies between app descriptions and user reviews, analyzing app recommendation relationships, and verifying through binary similarity analysis. However, their research acknowledges that sparse metadata availability and fundamental constraints of binary comparison largely reduce Mask-Catcher's capacity for timely detection of new Chameleon variants.

To bridge these gaps, we systematically investigate Chameleon apps with focused examination of their distribution channels and transformation methods. Our comprehensive analysis of 500 iOS Chameleon apps establishes a verified ground-truth dataset, carefully constructed by monitoring developer dissemination channels and cataloging transformation techniques. This study reveals three fundamental characteristics of Chameleon apps: (1) their transformation methods

[*]These two authors contributed equally as co-first authors.
[✉]Corresponding author: Haitao Xu (haitaoxu@zju.edu.cn).

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.241906
www.ndss-symposium.org

can be consistently identified through metadata examination (encompassing user reviews, BundleIDs, and descriptions) or runtime signals such as pop-up notifications and image ads; (2) these transformation mechanisms typically follow predictable, uncomplicated patterns; and (3) successful transformation invariably produces observable transitions to illicit functionality, with distinctive interface alterations providing definitive visual confirmation of the transformed state.

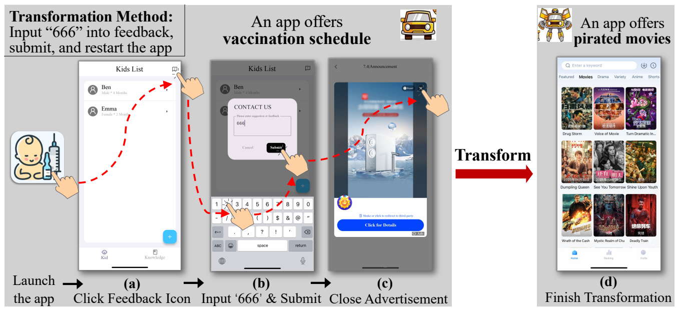
Building on these observations, we propose ChameleoScan, an LLM-powered automated UI exploration system specifically designed to address the limitations of existing LLM-based UI testing approaches [7], [8] when applied to real-world iOS Chameleon apps. The system tackles two primary challenges: (1) accurate recognition and interpretation of UI elements, and (2) fine-grained UI navigation capable of handling dynamic disruptions (*e.g.*, ads, pop-ups) and ambiguous transformation methods.

To address these issues, ChameleoScan employs LLMs with few-shot prompting to infer transformation methods from both app-specific metadata and known transformation intelligence used as exemplars. It then integrates visual and structural UI data, screenshots and view hierarchies, to construct enriched UI representations, enabling precise LLM-driven analysis that discerns both semantically relevant UI elements and disruptive components. Additionally, ChameleoScan uses LLMs to devise human-like exploration strategies and generate executable action sequences to complete the transformation process, while simultaneously detecting functional deviations to confirm the manifestation of Chameleon behavior.

Detection Performance on Ground-Truth Dataset. To assess ChameleoScan's detection efficacy, we established a ground-truth dataset containing 131 functional Chameleon apps and 233 benign apps. The system achieved flawless precision (100%) with 71.76% recall (96.91% for apps containing transformation-related user reviews), demonstrating detection accuracy comparable to professional security audits.

Core Capability Assessment. The system's capabilities were rigorously evaluated across three key dimensions: transformation method inference, UI element recognition, and interference handling. In transformation inference, the top-ranked prediction matched actual implementations in 31.62% of cases, approaching the 38.70% accuracy of manual review-based annotation. In identifying interactive UI elements, the system achieved a precision of 89.36% and a recall of 70.39%. For managing interference elements (*e.g.*, ads, pop-ups), it successfully handled intrusive ads in 85.96% of cases (*e.g.*, by automated dismissal) and responded appropriately to pop-ups (*e.g.*, by auto-granting permissions) in 95.56% of cases.

Real-World Deployment Analysis. When applied to 1,644 unlabeled apps from the App Store, ChameleoScan identified 9.85% (162) as Chameleon variants, with manual verification confirming 92.59% precision. Analysis of the 35 false negatives revealed two primary causes: app unresponsiveness due to lack of maintenance, and failures in inferring transformation methods or identifying semantically meaningful clickable



Transformation Method:
Input "666" into feedback, submit, and restart the app
An app offers vaccination schedule
An app offers pirated movies
Transform
Kids List
CONTACT US
Launch the app
(a) Click Feedback Icon
(b) Input '666' & Submit
(c) Close Advertisement
(d) Finish Transformation

Fig. 1: A typical transformation process of Chameleon apps.

elements. The system demonstrated efficient processing times, averaging 2.43 seconds for transformation inference, 8.16 seconds for UI recognition, 6.66 seconds for action execution, and 5.31 seconds for transformation validation per app.

Comparative Analysis. Benchmarking against Chameleon-Hunter [5] and Mask-Catcher [6] using annotated datasets revealed ChameleoScan's superior performance. Among 223 analyzed Chameleon apps, 46.64% (104) utilized hybrid frameworks (*e.g.*, Flutter or WebView), fundamentally limiting Chameleon-Hunter's detection capacity. While Mask-Catcher failed to detect any review-free Chameleon apps, ChameleoScan identified 96.88% of such cases. Even when reviews were available, Mask-Catcher's effectiveness was compromised by linguistic discrepancies (*e.g.*, English descriptions with Chinese reviews), insufficient suspicious content, or deliberate user review manipulation.

Contributions. Our contributions are summarized as follows:

- We constructed a large dataset of iOS Chameleon apps, conducted the first systematic analysis of their transformation methods, and established a more comprehensive taxonomy of 10 transformation method categories, 4 of which represent novel discoveries not previously documented.
- We developed ChameleoScan, the first LLM-powered automated UI exploration system for detecting iOS Chameleon apps through synergistic integration of App Store metadata and runtime UI evidence. Employing a dynamic, multimodal, few-shot learning approach, ChameleoScan addresses two critical limitations of existing UI testing automation methods: it achieves near-human capability in (1) accurately identifying relevant UI elements while managing distracting pop-ups, and (2) conducting fine-grained exploration under ambiguous task specifications where existing deterministic automation fails.
- ChameleoScan demonstrates robust detection effectiveness through rigorous evaluation on both a verified benchmark dataset and newly released App Store apps. The system significantly outperforms existing state-of-the-art methods, particularly in challenging scenarios involving apps that employ hybrid frameworks or reveal transformation logic solely during runtime without leaving metadata traces.

II. Background and Related Work

We present the background of Chameleon apps and examine prior work in illicit app analysis and mobile task automation.

A. Chameleon Apps

An Example of Chameleon App. Fig. 1 illustrates a typical transformation process of Chameleon apps. A user downloads an iOS app named `Vaccination Schedule`, purported to provide vaccination scheduling services for pediatric care. After acquiring the transformation instruction, "`Enter '666' in feedback field, submit, and restart the app`," the user executes the specified procedure within the app, ultimately transforming it into a pirated movie streaming service.

Lifecycle of Chameleon Apps. The typical lifecycle of Chameleon apps encompasses development, distribution, and transformation phases. Developers initially create and submit these apps to the App Store. They then publicly disclose app information, including download links and transformation methods, through various promotion channels. Users accessing these channels obtain both the download links and transformation instructions. After downloading the app from the iOS App Store, users subsequently apply the acquired transformation methods to reveal the app's true functionality.

B. Illicit iOS App Analysis

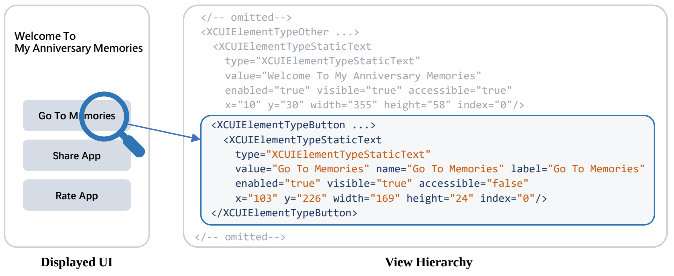
While numerous studies have been conducted on illicit app analysis for Android [9]–[15], there has been limited research focused on iOS apps. PiOS [16] was introduced to detect privacy leaks in iOS apps by analyzing data flows within Mach-O binaries. Lee *et al.* [4] investigated crowdturfing apps, a novel category of malicious apps, and leveraged a view controller graph to detect hidden screens within such apps. Then, Lee *et al.* [5] introduced Chameleon-Hunter, a tool that uses static analysis to examine binary files and UI layouts of apps to identify hidden UIs in Chameleon apps. However, static analysis-based detection methods may suffer from reduced detection accuracy when applied to hybrid apps, which incorporate dynamically rendered UI elements.

Existing research has demonstrated the utility of iOS app metadata, especially user reviews, for privacy and security analysis [17]–[24]. While Mask-Catcher [6] employs metadata including user reviews and App Store recommendations to detect hybrid-mode Chameleon apps, its effectiveness is constrained by several factors. The approach depends on both the availability of user-generated content and the authenticity of "You might also like" recommendations, rendering it ineffective for newly released apps lacking sufficient user feedback. Furthermore, its reliance on known code patterns through similarity analysis prevents identification of novel variants.

Our study centers on the understudied domain of Chameleon app transformation methods, with particular emphasis on their user-collaborative nature. Drawing upon these identified transformation patterns, we introduce an innovative detection system that employs LLM-enhanced mobile automation testing to reliably identify Chameleon apps.

C. Mobile Task Automation

Mobile task automation, aiming to automate multi-step processes within mobile apps, represents a rapidly evolving field. The automation workflow typically comprises 3 key steps:



```
</-- omitted-->
<XCUIElementTypeOther ...>
  <XCUIElementTypeStaticText
    type="XCUIElementTypeStaticText"
    value="Welcome To My Anniversary Memories"
    enabled="true" visible="true" accessible="true"
    x="10" y="30" width="355" height="58" index="0"/>
 <XCUIElementTypeButton ...>
   <XCUIElementTypeStaticText
     type="XCUIElementTypeStaticText"
     value="Go To Memories" name="Go To Memories" label="Go To Memories"
     enabled="true" visible="true" accessible="false"
     x="103" y="226" width="169" height="24" index="0"/>
 </XCUIElementTypeButton>
</-- omitted-->
```

Fig. 2: UI screenshot and view hierarchy (partial) of an app.

UI Representation. UI representation serves as the foundation for task planning. The two most commonly used representations are screenshots [25], [26] and view hierarchies [7], [8], [27], [28]. Screenshots provide visual cues for interpreting UI elements, while view hierarchies offer a structured, tree-like textual representation of UI components and their attributes. An illustrative example showing both a UI screenshot and its corresponding view hierarchy is provided in Fig. 2. Each element typically includes attributes, such as type, position, size, visibility, and interactivity, offering richer context for the decision-making module.

Task Planning. LLMs have been prominently leveraged for task planning tasks. Following Wang *et al.*'s pioneering work [29] on conversational UI interactions, numerous systems have adopted LLMs as their central decision-making component [7], [8], [30]–[32]. Researchers have developed advanced prompts to enhance task planning. Notable implementations include AutoDroid [7], which integrates action history into prompts to prevent cyclic behavior, and Guardian [8], which employs action spaces to eliminate redundant or invalid operations.

Agent Execution. Once the operations are selected, agents interact with the mobile interface and execute these actions to complete the task. Despite the demonstrated potential of task automation tools in task planning and semantic understanding, they face two notable limitations. First, the majority of research and datasets are exclusively focused on the Android platform [33], [34], creating a gap in solutions tailored for iOS. Second, existing task automation tools frequently demonstrate diminished robustness when handling complex apps, particularly Chameleon apps characterized by intrusive ads, dynamic interface changes, and overlapping content layers.

III. Preliminary Study

A. Data Collection of Chameleon Apps

The collection of iOS Chameleon apps poses significant challenges. First, identifying such apps based on their descriptions on the App Store or through existing malicious app detection approaches is impractical due to their seemingly legitimate appearance and non-traditional malicious nature. Moreover, the promotion channels for disseminating their transformation methods are often highly covert. To this end, we utilized specialized terminology (*e.g.*, "disguised app" or "covert app") to search for promotion channels on popular platforms, including Telegram [35], WeChat [36], and AppRaven [37],

TABLE I: Dataset statistics for Chameleon apps

Source Channels (#)	Apps Collected	Apps w/ IPA
WeChat Accounts (4)	346	175
Appraven Groups (2)	108	39
Illicit Websites (17)	46	20
Total (23)	**500**	**234 (46.8%)**

where developers frequently share relevant information in irregular, fragmented communications.

We successfully identified 23 promotional channels containing information related to Chameleon apps, including 17 websites discovered through Telegram, 4 official WeChat accounts, and 2 AppRaven chat groups. Beginning in May 2024, we conducted a six-month monitoring campaign, tracking daily updates from these channels and collecting shared content related to Chameleon apps, including App Store download links, hidden functionalities, and transformation methods.

By tracing iOS App Store download links, we obtained installation IPA files for these apps using ipatool-py [38], and collected their metadata in collaboration with an anonymous mobile intelligence provider. This metadata encompassed app names, categories, descriptions, and user reviews. Table I summarizes the collected Chameleon app statistics. Our final dataset consists of 500 iOS Chameleon apps originating from 4 WeChat accounts, 2 Appraven groups, and 17 illicit service websites. We successfully downloaded and manually verified IPA files for 234 apps (46.8%) through expert installation testing, exceeding the previous largest dataset of 180 Chameleon apps [6]. The remaining apps were unavailable for download as they had been removed from the App Store prior to our investigation.

B. Characterizing Chameleon Apps

Leveraging our unique dataset of Chameleon apps, we first analyze their transformation methods, an aspect that has been largely overlooked in existing research. We then examine their functionalities before and after transformation.

*1) **Transformation Methods**:* Transformation methods of Chameleon apps are typically shared by their developers to guide potential users in transforming a seemingly normal app into its true form. Through analysis of our collected samples, we classified them into 10 distinct categories as follows (summarized in Table II):

❶ **Immediate transformation upon app launch.** The transformation occurs instantly upon the app's initial launch, requiring no further user actions.

❷ **Delayed transformation after app launch.** Transformation takes place within a few seconds (*e.g.*, 10) after the app is launched for the first time, frequently accompanied by visual indicators such as "Updating" displayed during the process.

❸ **Transformation triggered by app restart.** The transformation is achieved simply by restarting the app.

❹ **Region-specific transformation.** The transformation is restricted to users located in a specific geographic region, and it is automatically triggered, requiring no user interactions.

❺ **Time-restricted transformation.** The transformation can be completed exclusively during predefined time windows, *e.g.*, between 12 a.m. and 1 a.m., without any user interactions.

❻ **Transformation by interacting with specific UI elements.** Transformation is initiated by tapping certain areas or UI elements for a predefined number of times, *e.g.*, 5 taps on a blank area or 8 taps on a specific button. In some cases, tapping must occur rapidly after app launch, as the tapping area only appears along with the loading bar.

❼ **Transformation after viewing in-app ads.** Transformation is triggered only after watching a certain number of ads (*e.g.*, three) displayed within the app.

❽ **Transformation by selecting a specific option.** Transformation is initiated only after the user selects and submits a particular option from a dropdown menu.

❾ **Transformation by submitting a specific string code.** Transformation occurs when a predefined code (*e.g.*, "520," "ys777," or "Peppa Pig") is entered into a designated text input field, such as a user feedback box or a search box.

❿ **Transformation using a magic string with clipboard permission.** Transformation involves a more complex process, where a magic string is first copied from an external covert distribution channel to the smartphone's clipboard. Upon launching the app and granting clipboard access, the app reads the string to complete the transformation.

As shown in Table II, the two most prevalent formats of transformation methods, "submitting a specific string code" and "tapping a designated area," account for 69.60% (348 cases) and 9.80% (49 cases), respectively. Additionally, 18 Chameleon apps employ hybrid transformation methods that combine two or three types, with none utilizing more than three. For instance, one app requires users to enter "999999" into a calculator interface and press the `calculate` button three times to trigger the transformation.

Based on the user actions required, these 10 distinct transformation methods can be further grouped into four types: (i) *Auto-Transformation*: no action or app restart only. (ii) *Spatiotemporal-based*: out-of-app settings of location or time. (iii) *Click-based*: in-app clicks, *e.g.*, consecutive clicking in a specific region. (iv) *Input-based*: submitting a specific string into a designated field, such as text boxes or clipboard. The specific user actions and associated elements for each transformation type are also provided in Table II. Overall, most transformations require users to perform one or more actions, including `click`, `restart`, `select options`, and `copy`, on elements, such as `button`, `text field`, `image`, `static text`, and `clipboard`. Furthermore, these actions and elements typically follow predictable patterns, with 90% of these transformations being completed in five or fewer steps. These findings highlight that *the transformation methods are often straightforward and predictable*.

Novelty of Our Identified Transformation Methods. Our work establishes a more comprehensive taxonomy of transformation methods, significantly extending and refining the fragmented observations documented in prior studies [5], [6]. Of the 10 transformation methods we identified, four (❷❼❽❿) represent novel discoveries, while four correspond to previously reported transformation types (❶❸❹❾).

TABLE II: Classification of transformation methods employed by Chameleon apps. Parenthetical values denote app counts for each method (with hybrid-method apps counted in all relevant categories). The *Known/Novel* column categorizes methods as Novel (newly discovered), Partial (partially documented), or Reported (previously identified). *Required Actions* and *Operated Elements* detail the necessary user interactions and corresponding interface components respectively.

Category (#)	Description of Transformation Methods (#)	Known/ Novel	Required Actions	Operated Elements
Auto-Transform. (95, 19.0%)	❶ No additional actions are required, simply open the app to complete the conversion. (28, 5.6%)	Reported [6]	N/A	N/A
	❷ Transformation takes place within a few seconds. The app usually displays "Updating" during this time. (20, 4.0%)	Novel	N/A	N/A
	❸ The transformation is achieved simply by restarting the app, without requiring any additional interaction. (47, 9.4%)	Reported [6]	Restart	N/A
Spatiotemporal Based (7, 1.4%)	❹ Require the geographical location in a specific region (*e.g.*, China) (4, 0.8%)	Reported [5]	Out-of-app location sett.	N/A
	❺ Require the time to be within a specific range, such as between 12:00 AM and 1:00 AM. (3, 0.6%)	Partial [5]	Out-of-app time settings	N/A
Click-Based (60, 12.0%)	❻ Tap a blank area or specific control on the page more than a specified number of times. (49, 9.8%)	Partial [6]	Click, Rapid Click, Restart	Static Text, Button, Image (*e.g.*, Backdrop)
	❼ Watch a sufficient number of ads, such as completing the trigger by watching three ads. (10, 2.0%)	Novel	Click	Static Text, Button (*e.g.*, "Close Ads", "Confirm")
	❽ Perform a specific selection operation, such as selecting a specific option in a dropdown menu and tap submission. (1, 0.2%)	Novel	Click, Option Select, Restart	Picker (*e.g.*, dropdown Menu), Button
Input-Based (356, 71.2%)	❾ Enter a specific code in a designated text input field (commonly feedback or search bar) and tap submission. (348, 69.6%)	Reported [6]	Text Input, Click, Restart	Static Text, Button (*e.g.*, "Feedback", "Submit", "!", "+"), Alert
	❿ Copy specific text content to the clipboard, grant clipboard access upon entering the app, then trigger automatically. (8, 1.6%)	Novel	Copy, Click	Static Text, Clipboard

The remaining two methods (❺❻) demonstrate partial correspondence with existing research but are substantially enhanced through our empirical findings. The *Spatiotemporal-Based* method (❺) introduces precise temporal parameters, advancing beyond the vague temporal triggers noted in [5]. Similarly, our *Click-Based* method (❻) extends prior work on location-specific clicking [6] by incorporating multi-click sequences and temporal constraints.

Critically, existing detection methods [5], [6] frequently prove inadequate even for known transformation types (*i.e.*, ❶❸❹❾). This limitation is particularly pronounced when apps employ hybrid frameworks or reveal transformation logic exclusively during runtime without leaving discernible metadata traces. The fundamental dependence of these conventional approaches on static analysis and metadata inherently restricts their effectiveness, underscoring the essential requirement for our dynamic, multimodal methodology as a substantive advance beyond incremental, rule-based enhancements.

*2) **Disclosure of Transformation Methods**:* As established by Zhao *et al.* [6], user reviews often contain valuable insights into app functionality, including subtle clues about their transformation methods. Our examination of 14,692 reviews across 500 Chameleon apps (84.80% of which contained at least one review) demonstrates that both developers and users systematically embed transformation instructions within these reviews. Notably, 48.8% (244) of analyzed apps contained review-disclosed transformation methods, ranging from implicit cues (*e.g.*, "useful: 520") to explicit procedural guidance (observed in 79 cases), *e.g.*, "Click the exclamation mark '!' in the upper-right corner, enter '777ys,' then submit." *These findings substantiate user reviews as a credible source for deriving Chameleon app transformation techniques.*

Beyond user reviews, we investigated whether app metadata, such as BundleID and description, could also reveal transformation methods. Clustering apps by app description revealed patterns: 8 apps disguised as games employed either *Auto-Transformation* or simple *Click-Based* methods, while 13 of 23 apps posing as scientific calculators required inputting specific numeric strings (*e.g.*, "666") into "feedback" sections. Similarly, clustering by BundleID showed that all 8 apps containing "dz.spd" or "hy.buto" domain strings shared the same keyword-based transformation method (submitting "persimmon"). These findings suggest that *apps with similar metadata tend to use identical transformation methods*, possibly due to developer template reuse. Thus, metadata may serve as a predictive indicator, especially when user reviews are scarce.

Additionally, some Chameleon apps disclose transformation methods through run-time information, such as pop-up alerts or image ads. We identified 11 such cases, including one app (BundleID "com.shuidain.baocun") that displayed an alert stating, "Kind reminder: Watch an ad or provide feedback to unlock!". This further underscores the diverse channels through which transformation methods may be revealed.

*3) **Functionalities Before and After Transformation**:* For the 234 Chameleon apps with accessible IPA files, we systematically compared their declared App Store functionalities with actual behaviors uncovered through empirical device testing. Our verification process involved two experts independently examining each app's metadata (particularly app description) followed by execution of documented transformation procedures. This rigorous examination revealed substantial disparities between advertised and actual functionality. Initial UI interfaces faithfully reflected their official descriptions, occasionally containing peripheral elements (*e.g.*, advertisements, settings panels, feedback interfaces) unrelated to core functionality. Following successful transformation, these apps invariably transitioned to illicit operational states (*e.g.*, financial scams, pirated media services, and adult content), with definitive visual confirmation apparent in their modified user interfaces.

Our statistical analysis reveals that these apps primarily masquerade as puzzle games, habit trackers, or productivity tools within the App Store, predominantly classified under Utilities (62.39%), Lifestyle (18.38%), or Entertainment (5.98%). Following transformation, they consistently transition to delivering illicit services including media piracy, gambling platforms, adult content, and copyright-infringing literary access.

*4) **Summary**:* Our investigation yields three key insights about Chameleon apps: (1) their transformation methods can be reliably identified through metadata analysis (including user reviews, BundleIDs, and descriptions) or runtime indicators like pop-up notifications and image ads; (2) these methods are often straightforward and predictable; and (3) successful transformation consistently leads to clearly detectable transitions to illicit operational modes, with distinctive UI alterations serving as unambiguous visual indicators of the transformed state.

IV. ChameleoScan Approach

In this section, we present the threat model underlying our study, then examine the requirements and challenges of automated UI testing, and finally elaborate on ChameleoScan's design and implementation.

A. Threat Model

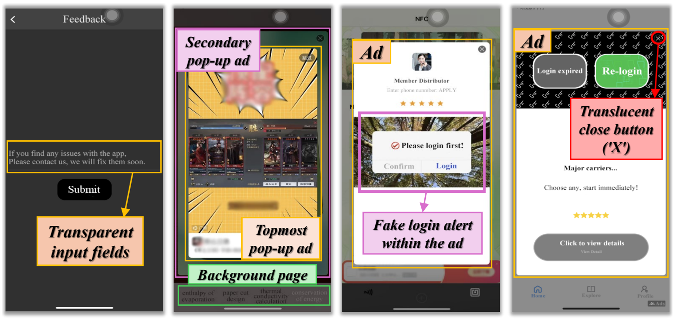
The adversary aims to bypass iOS App Store review protocols by masquerading as a legitimate app during submission. Following the app's release, the adversary may embed subtle clues about its transformation mechanisms within app metadata or runtime pop-up alerts.

This model assumes successful adversary deployment on the App Store, with defenders employing automated detection systems to mitigate Chameleon app proliferation post-release.

B. Requirements and Challenges of Automated UI Testing of Chameleon Apps

To identify the core functionalities of a suspicious app, automated feature-based UI testing [39], [40], which aims to validate an app's primary functions with minimal manual intervention, is essential and desirable. For a suspicious Chameleon app, drawing on prior knowledge, defenders need to automatically generate multi-step UI tests that emulate typical user interactions to trigger the transformation process. These tests generally involve executing the app in a controlled environment, applying various potential transformation methods, observing its responses to specific triggers, and detecting inconsistencies between its declared and actual functionalities.

This process of UI testing, which requires determining the next action based on the responses to previous UI actions, inherently constitutes a sequential planning problem [29]. Despite the promising potential of LLMs for sequential planning



(a) Unable to recognize transparent input fields solely by vision (b) Unable to differentiate underlying background pages from foreground overlay windows (c) Unable to resist the interference from deceptive interactive elements in pop-up ads (d) Unable to detect translucent close buttons ('X') in ads

Fig. 3: Demonstrated limitations in vision-based UI recognition.

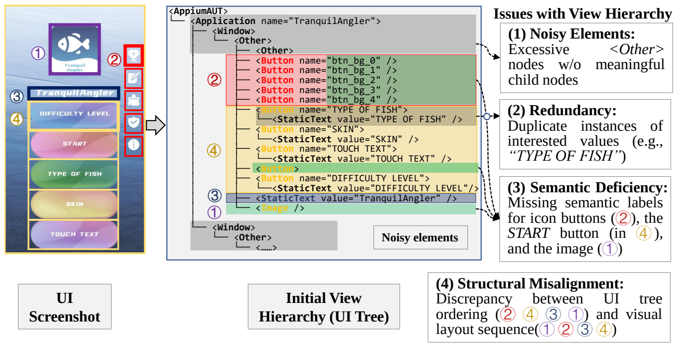
problems, state-of-the-art LLM-based UI testing approaches [7], [8] encounter the following challenges when applied to our iOS Chameleon app dataset.

Difficulty with UI Element Recognition. Accurate UI element recognition is essential for automated UI testing, yet reliably identifying elements from screenshots alone remains problematic. Key challenges include deceptive elements (*e.g.*, intrusive pop-ups), low-contrast elements blending with backgrounds, and ambiguous elements lacking clear boundaries (*e.g.*, borderless text fields or minuscule ad close buttons)—issues particularly prevalent in Chameleon apps with poor UI design.

Even leading vision-based automation systems demonstrate significant limitations when handling such complex UI scenarios. Fig. 3 demonstrates several problematic interface scenarios in Chameleon apps where VisionTasker [25], a leading vision-based automation system, exhibits unreliable performance. The tool systematically misinterprets transparent input fields as static text components (a). When ad pop-ups obscure the underlying content, it erroneously classifies partially visible background elements as interactive due to insufficient view hierarchy context (b). Additionally, it cannot reliably differentiate deceptive interface elements (*e.g.*, promotional images mimicking login alerts) from authentic controls (c). The system also consistently overlooks small, translucent close buttons ('X') within ad content (d).

Issues with View Hierarchies. While view hierarchies (*a.k.a.*, UI trees) complement screenshots by providing structural metadata (*e.g.*, bounding boxes, hidden attributes), they exhibit essential limitations, as illustrated in Fig. 4 showing both the screenshot and corresponding UI tree of "Tranquil Angler", a purported fishing app that transforms into a pirated video streaming service at midnight.

- **Noisy Elements**: The hierarchy contains excessive nodes (*e.g.*, empty `<Other>`) that complicate processing by LLMs.
- **Redundancy**: Critical attributes appear redundantly across nodes (*e.g.*, `TYPE OF FISH` in both parent and child elements), necessitating data consolidation.
- **Semantic Deficiency**: Critical interactive elements (functional icons ②, `START` button ④, image ①) lack functional annotations, impeding automated analysis.
- **Structural Misalignment**: The hierarchy's node sequence



```
<AppiumAUT>
└── <Application name="TranquilAngler">
    ├── <Window>
    │   └── <Other>
    │       ├── <Other>
    │       ├── <Button name="btn_bg_0" />
    │       ├── <Button name="btn_bg_1" />
    │       ├── <Button name="btn_bg_2" />
    │       ├── <Button name="btn_bg_3" />
    │       ├── <Button name="btn_bg_4" />
    │       ├── <Button name="TYPE OF FISH">
    │       │   └──<StaticText value="TYPE OF FISH" />
    │       ├── <Button name="SKIN">
    │       │   └──<StaticText value="SKIN" />
    │       ├── <Button name="TOUCH TEXT">
    │       │   └──<StaticText value="TOUCH TEXT" />
    │       ├── <Button>
    │       ├── <Button name="DIFFICULTY LEVEL">
    │       │   └──<StaticText value="DIFFICULTY LEVEL"/>
    │       ├── <StaticText value="TranquilAngler" />
    │       └── <Image />
    └── <Window>
        └── <Other>
            └── <......>
```

(1) Noisy Elements: Excessive *<Other>* nodes w/o meaningful child nodes

(2) Redundancy: Duplicate instances of interested values (e.g., *"TYPE OF FISH"*)

(3) Semantic Deficiency: Missing semantic labels for icon buttons (②), the *START* button (in ④), and the image (①)

(4) Structural Misalignment: Discrepancy between UI tree ordering (② ④ ③ ①) and visual layout sequence(① ② ③ ④)

Fig. 4: Example UI screenshot and corresponding view hierarchy, annotated with identified view hierarchy issues.

(②④③①) contradicts the visual layout (①-④).

Fine-Grained UI Navigation. Effective automated UI navigation that demands precise action control also poses challenges:

- **Disruption of Distracting Elements**: Chameleon apps often include ads, which constitute external distracting content rather than the app's own functionality. A robust UI testing framework must resist such disturbances by identifying and handling these ad elements. Otherwise, testing may be disrupted, either stalling due to unclosed ads or deviating to unintended pages, ultimately leading to test failure. Similarly, pop-up alerts (*e.g.*, permission requests) from the app or the iOS system may impede testing and necessitate additional handling mechanisms. Conventional ad-blocking techniques are frequently insufficient for addressing obfuscated ads or custom pop-up interfaces.
- **Identification of Repetitive vs. Required Actions**: LLMs often produce repetitive actions, which are typically mitigated by maintaining an `action history` to track executed operations. However, this approach performs poorly for Chameleon apps, where the transformation process may legitimately require identical actions on different pages (*e.g.*, a `Next` button). Consequently, LLMs may misclassify essential actions as redundancies or overlook cross-page repetitions. Therefore, distinguishing between truly repetitive and contextually required actions is crucial.
- **Ambiguous Task Specifications**: Transformation methods, which serve as tasks for UI automation, often suffer from vagueness and incompleteness (*e.g.*, a user review may mention only a keyword "666" without indicating the specific input location). This renders them unsuitable for direct decomposition into executable subtasks. Hence, the LLM must intelligently formulate exploration strategies to infer page-specific actionable steps that collectively contribute to achieving the overall objective.

C. ChameleonScan *Design*

We propose ChameleonScan, a LLM-powered automated UI exploration system designed for efficient Chameleon app detection. The system addresses the challenges outlined in §IV-B by providing accurate UI element recognition and comprehension, precise detection and handling of distracting elements, and human-comparable granularity in UI exploration.

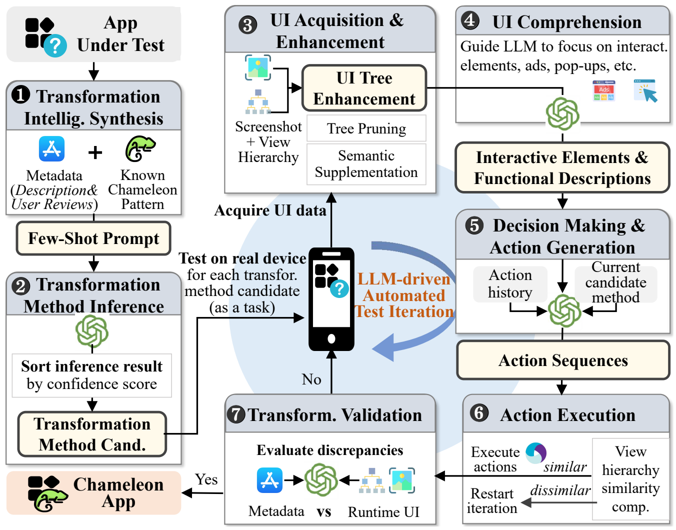

App Under Test
❶ Transformation Intellig. Synthesis
Metadata (Description& User Reviews)
Known Chameleon Pattern
Few-Shot Prompt
❷ Transformation Method Inference
Sort inference result by confidence score
Transformation Method Cand.
Chameleon App
Yes
❸ UI Acquisition & Enhancement
Screenshot + View Hierarchy
UI Tree Enhancement
Tree Pruning
Semantic Supplementation
Acquire UI data
Test on real device for each transfor. method candidate (as a task)
LLM-driven Automated Test Iteration
No
❹ UI Comprehension
Guide LLM to focus on interact. elements, ads, pop-ups, etc.
Interactive Elements & Functional Descriptions
❺ Decision Making & Action Generation
Action history
Current candidate method
Action Sequences
❻ Action Execution
Execute actions
similar
dissimilar
Restart iteration
View hierarchy similarity comp.
❼Transform. Validation
Evaluate discrepancies
Metadata vs Runtime UI

Fig. 5: The workflow of ChameleonScan.

The operational workflow of ChameleonScan is depicted in Fig. 5. When analyzing a suspicious app, the system executes the following core components (or steps) to ascertain its Chameleon classification. **(❶) Transformation Intelligence Synthesis.** Leveraging known patterns from established Chameleon apps, the framework systematically extracts transformation-relevant indicators from app metadata (including BundleID, descriptions, and user reviews), which are then processed by LLMs to derive the app's functionality profile (*i.e.*, the app's intended core features, behaviors, and expected user interactions) for subsequent analysis. **(❷) Transformation Method Inference.** By integrating documented Chameleon transformation exemplars with target app metadata into carefully crafted few-shot prompts, the system employs LLMs to generate probabilistically ranked transformation methods for empirical validation. Then, the system performs iterative evaluation of all candidate transformation methods (steps ❸-❼ constituting a single evaluation round) until conclusive determination of the app's Chameleon status is achieved. **(❸) UI Data Acquisition and Enhancement**. An iOS instrumentation agent executes the app binary, applies each transformation method candidate (*e.g., wait 5 seconds after launch*), automatically captures UI snapshots along with the corresponding structural view hierarchy (subsequently enhanced) upon UI state transitions. **(❹) UI Comprehension.** The system employs LLM-based analysis to process both visual and structural UI representations, enabling the identification, classification, and semantic annotation of meaningful interface elements. **(❺) Decision Making and Action Sequence Generation.** The LLM formulates exploration strategies and generates executable interaction sequences by synthesizing current UI understanding, historical action context, and transformation objectives. **(❻) Action Execution.** The mobile agent sequentially executes generated actions, dynamically capturing new interface states (screenshots and view hierarchies) for iterative analysis. **(❼) Transformation Validation.** The system performs comparative analysis of each newly rendered UI's screenshots and structural hierarchies against declared specifications, evaluating functionality discrepancies to both validate transformation occurrences and conclusively determine Chameleon app classification.

We selected GPT-4o as the underlying LLM for the core analytical components of ChameleoScan due to its advanced multimodal reasoning capabilities and native support for structured output generation (at the time of writing), both essential for our Chain-of-Thought implementation. Note that ChameleoScan itself is model-agnostic and can be adapted to utilize other LLMs with comparable capabilities.

The subsequent section details the system's operational methodology, employing the `Vaccination Schedule` app (Fig. 1) as a representative case study, with full procedural details provided in Appendix A.

```
<instruction>
    Given {app metadata} (including BundleID,
    description, and user reviews), infer potential
    transformation methods. Respond with a list of
    methods, each accompanied by a confidence score
    and a brief rationale. Provide no additional
    output.
</instruction>
<output_format>
    Return a list where each item is a JSON object
    with the following fields: "method" (string), "
    confidence" (float between 0-1), and "rationale"
     (1-3 sentences). Adhere strictly to the
    exemplar structure provided in <examples>.
</output_format>
<examples>
    1. Input: [metadata_1]; Output: [transform.
    method_1], [confidence_1], [reasoning_1]}
    2. Input: [metadata_2]; Output: [transform.
    method_2], [confidence_2], [reasoning_2]}
    ...
    n. Input: [metadata_n]; Output: [transform.
    method_n], [confidence_n], [reasoning_n]}
</examples>
```

Listing 1: Prompt template for transform. method inference.

❶ *Transformation Intelligence Synthesis*

The identification of transformation methods in potential Chameleon apps is complicated by their concealed promotion mechanisms. As evidenced in our preliminary study (§III-B2), metadata analysis, encompassing app BundleID, descriptions, and user reviews, proves effective for two key reasons. First, such metadata frequently contains unintentional revelations from either developers or users. Second, apps sharing similar metadata characteristics tend to employ comparable transformation methods, likely stemming from common developer practices or template replication. This correlation proves particularly valuable when analyzing suspicious apps with limited user reviews, as known Chameleon apps with analogous metadata can provide critical insights into their potential transformation techniques. Also, during the metadata analysis process, LLMs generate the app's functionality profile, establishing the baseline for detecting semantic discrepancies between observed UIs and expected core functionality.

To enable automated extraction of transformation methods from preprocessed metadata, we employ a few-shot prompting approach, utilizing both known transformation methods and corresponding metadata as exemplars. Our structured prompt template is illustrated in Listing 1.

❷ *Transformation Method Inference*

Providing the aforementioned prompts to an LLM generally produces a limited set (typically n=2 or n=3) of tuples structured as {*transformation method*, *confidence score*, *reasoning*}. As shown in Table II, a transformation method may include parameters such as post-launch delay, time windows, geographic constraints, click locations, click counts, input strings and their designated input fields, though not all fields are always present. The *confidence score* (higher when app metadata strongly support the method) represents the estimated probability of a potential method being correct, while *reasoning* provides supporting evidence from metadata analysis.

The LLM-derived transformation methods are typically app-specific, whereas some app-agnostic methods exist. Notably, *Auto-Transformation* methods (§III-B1), representing roughly 20% of documented cases, activate automatically upon app launch/restart. To optimize detection efficiency, we prioritize these app-agnostic methods before proceeding the LLM-suggested approaches in descending order of confidence scores.

Continuing with the `Vaccination Schedule` case, an excerpt of its accessible metadata is provided as follows.

App Name: Vaccination Schedule
Category: Utilities
BundleID: com.qiu.okw.opg.yimiao
Description: Provides detailed vaccination schedules, reminders, and records to help parents monitor and know their child's immunizations.
User Reviews (112 in total):
(1) The app is useful, but have too many ads.
(2) App update fails, cannot unlock!!!!!
(3) Enter "666" does not work? What is code?
(4) Contact Us, 666
(5) Feedback is exclamation mark, in the right corner!
(6) 666 666 this app is 666
... (106 additional reviews)

When processed by the LLM, this metadata yields three probable transformation methods. The model appears to associate the frequently occurring term "666" with potential input requirements, given its prevalence in both the metadata and other established Chameleon patterns, while references to "Feedback" and "Contact Us" suggest probable input locations.

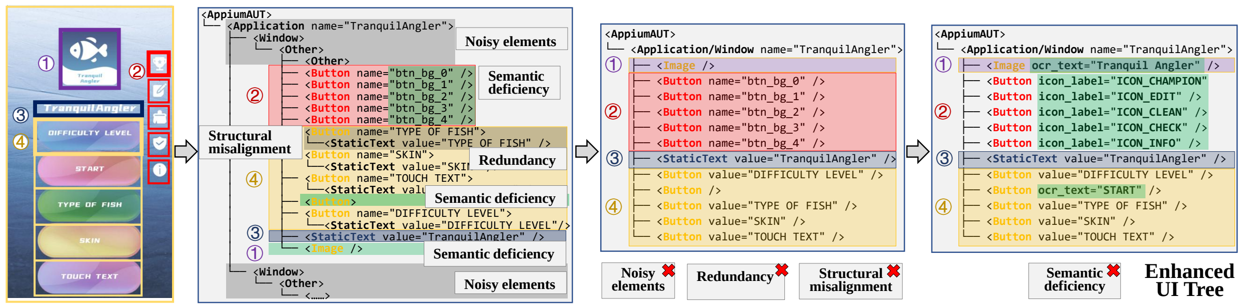
Possible transformation methods with confidence scores and rationale.
(1) Click "Contact Us", input "666" & submit feedback (0.9) - [reasoning]
(2) Restart the app and wait for update for three times (0.4) – [reasoning]
(3) Watch ads before clicking the "Close Ads" button (0.2) – [reasoning]

Furthermore, the LLM generates the app's *functionality profile* from this metadata, producing the following summary:

This app assists parents in managing pediatric vaccination schedules with features like a dashboard, multi-child profiles, editable schedules, vaccine info, and record logging. It offers reminders, regional settings, and in-app guidance. User reports indicate the presence of full-screen and pause-triggered ads.

❸ *UI Data Acquisition and Enhancement*

For each inferred transformation method, an iOS agent executes the app binary while simultaneously capturing both a visual screenshot and its corresponding UI hierarchy. To overcome the four principal constraints of UI trees discussed



```
<AppiumAUT>
└── <Application name="TranquilAngler">
    ├── <Window>
    │   └── <Other>
    │       ├── <Other>
    │       ├── <Button name="btn_bg_0" />
    │       ├── <Button name="btn_bg_1" />
    │       ├── <Button name="btn_bg_2" />
    │       ├── <Button name="btn_bg_3" />
    │       ├── <Button name="btn_bg_4" />
    │       ├── <Button name="TYPE OF FISH">
    │       │   └──<StaticText value="TYPE OF FISH" />
    │       ├── <Button name="SKIN">
    │       │   └──<StaticText value="SKI
    │       ├── <Button name="TOUCH TEXT">
    │       │   └──<StaticText valu
    │       ├── <Button>
    │       ├── <Button name="DIFFICULTY LEVEL">
    │       │   └──<StaticText value="DIFFICULTY LEVEL"/>
    │       ├── <StaticText value="TranquilAngler" />
    │       └── <Image />
    └── <Window>
        └── <Other>
            └── <......>
```

```
<AppiumAUT>
└── <Application/Window name="TranquilAngler">
    ├── <Image />
    ├── <Button name="btn_bg_0" />
    ├── <Button name="btn_bg_1" />
    ├── <Button name="btn_bg_2" />
    ├── <Button name="btn_bg_3" />
    ├── <Button name="btn_bg_4" />
    ├── <StaticText value="TranquilAngler" />
    ├── <Button value="DIFFICULTY LEVEL" />
    ├── <Button />
    ├── <Button value="TYPE OF FISH" />
    ├── <Button value="SKIN" />
    └── <Button value="TOUCH TEXT" />
```

```
<AppiumAUT>
└── <Application/Window name="TranquilAngler">
    ├── <Image ocr_text="Tranquil Angler" />
    ├── <Button icon_label="ICON_CHAMPION"
    ├── <Button icon_label="ICON_EDIT" />
    ├── <Button icon_label="ICON_CLEAN" />
    ├── <Button icon_label="ICON_CHECK" />
    ├── <Button icon_label="ICON_INFO" />
    ├── <StaticText value="TranquilAngler" />
    ├── <Button value="DIFFICULTY LEVEL" />
    ├── <Button ocr_text="START" />
    ├── <Button value="TYPE OF FISH" />
    ├── <Button value="SKIN" />
    └── <Button value="TOUCH TEXT" />
```

(a) UI Screenshot **(b) Initial View Hierarchy (UI Tree)** **(c) Tree Pruning** **(d) Semantic Supplementation**

Fig. 6: View hierarchy (UI tree) enhancement procedure demonstrated using the "Tranquil Angler" app.

in §IV-B, we implement two critical improvements, *tree pruning* and *semantic supplementation*, prior to processing the interface data through LLMs for enhanced interpretation. Fig. 6 demonstrates this refinement process using the "Tranquil Angler" app as a case study, with Fig. 6(b) specifically depicting the UI tree limitations originally shown in Fig. 4.

Tree Pruning. This enhancement primarily involves:

- **Removing Noisy Elements**: Our approach leverages the observation that semantically meaningful nodes predominantly reside at leaf positions in UI trees. To enhance LLMs' focus on these critical elements, we systematically eliminate noise by removing irrelevant UI components (*i.e.*, those with zero dimensions, off-screen positioning, or empty containers) along with non-critical attributes (which functions merely as a structural container indicator rather than denoting actionable or meaningful interface elements) from the raw UI tree.
- **Reducing Redundancy**: We then perform recursive consolidation of single-child nodes with their parents, merging elements with identical frames or overlapping types while preserving distinct attributes (*e.g.*, retaining unique values like `TYPE OF FISH`). This method maintains strict element-to-visual alignment between the UI tree and screenshot representation, effectively minimizing structural redundancy.
- **Correcting Structural Misalignment**: We reorganize the node structure into a systematic left-to-right, top-to-bottom grid layout according to their absolute coordinates in the UI screenshot, thereby resolving structural inconsistencies.

Semantic Supplementation. Despite pruning, LLMs may still misinterpret UI elements with insufficient semantic context, particularly unlabeled icons and images. Table III documents instances where LLMs misinterpret functional icons' purposes without proper semantic augmentation. To mitigate this, we propose targeted semantic enrichment:

TABLE III: Functional icon examples requiring semantic augmentation for accurate LLM interpretation.

Icon	Ground-truth Interpretation
⚙	**Settings icon**, directing to a comment submission interface.
⊗	**Close button**, specifically designed for a particular ad.
☺	**Smile icon**, denoting user profile w/ an embedded contact option.
ⓘ	**Information query icon**, intended to access the feedback page.

- **For Icons**, we identify functional ones typically situated at leaf nodes of view hierarchies. Candidate icons are defined as leaf nodes with minimal descriptive attributes and proper geometric shapes (circular or square), as demonstrated in Table III. Using the RICO Semantics dataset [41], we implement a lightweight *EfficientNet-B0* model [42] to generate semantic annotations (*e.g.*, `X` mark, `SETTINGS`, `MAGNIFYING_GLASS`), which are incorporated as node attributes to enhance icon interpretability.
- **For Images**, we employ PaddleOCR [43] to extract embedded text from bounding boxes. The refined UI tree undergoes traversal to locate the optimal matching node (*i.e.*, the smallest enclosing frame), with extracted text appended as supplementary attributes (*e.g.*, `ocr_text="Tranquil Angler"` in Fig. 6(d)).

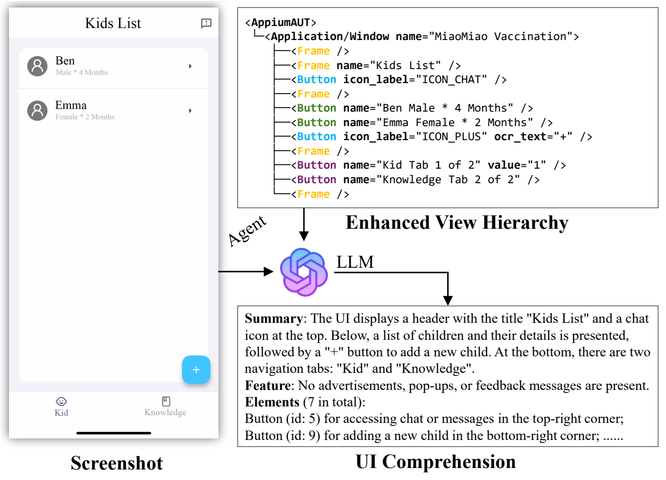
❹ *UI Comprehension*

The visual UI screenshots and enriched view hierarchies are subsequently analyzed by LLMs for interface understanding. As outlined in §IV-B, combining both screenshots and view hierarchies for UI representation helps mitigate challenges stemming from dynamic element disruptions and lack of semantic labeling for functional icons.

```
<instruction>
    Given the {view hierarchy in XML} and a {runtime
     screenshot}, perform comprehensive UI analysis
    by: (1) establishing visual correspondence
    between hierarchy elements and screenshot
    regions while filtering non-matching or obscured
     components; (2) classifying all UI elements as
    either non-interactive (e.g., labels, static
    content) or interactive (e.g., buttons,
    clickable icons, input fields). Pay special
    attention to promotional content, transient
    dialogs, and system messages, with documentation
     of their dismissal mechanisms. Present the
    final analysis as a structured inventory of UI
    elements without additional commentary.
</instruction>
<output_format>
    Return a list where each item is a JSON object
    representing a single UI element with the
    following fields: "type" (string), "state" (
    string), "text" (string), and "description."
</output_format>
```

Listing 2: Prompt template for UI comprehension.

The *UI Comprehension* process generates a structured representation of all visible UI elements—including their



```
<AppiumAUT>
 └─<Application/Window name="MiaoMiao Vaccination">
    ├──<Frame />
    ├──<Frame name="Kids List" />
    ├──<Button icon_label="ICON_CHAT" />
    ├──<Frame />
    ├──<Button name="Ben Male * 4 Months" />
    ├──<Button name="Emma Female * 2 Months" />
    ├──<Button icon_label="ICON_PLUS" ocr_text="+" />
    ├──<Frame />
    ├──<Button name="Kid Tab 1 of 2" value="1" />
    ├──<Button name="Knowledge Tab 2 of 2" />
    └──<Frame />
```

Summary: The UI displays a header with the title "Kids List" and a chat icon at the top. Below, a list of children and their details is presented, followed by a "+" button to add a new child. At the bottom, there are two navigation tabs: "Kid" and "Knowledge".
Feature: No advertisements, pop-ups, or feedback messages are present.
Elements (7 in total):
Button (id: 5) for accessing chat or messages in the top-right corner;
Button (id: 9) for adding a new child in the bottom-right corner; ......

Fig. 7: The main interface of `Vaccination Schedule`, its enhanced view hierarchy, and the output of UI comprehension.

categories, states, and semantic meanings—with particular emphasis on disruptive components such as pop-up ads. The prompt template for this process appears in Listing 2.

Fig. 7 presents the optimized view hierarchy of the `Vaccination Schedule` app alongside the LLM-generated interpretation of its main interface.

❺ *Decision Making and Action Sequence Generation*

Leveraging current UI comprehension outputs and historical action data, the LLM systematically generates executable action sequences to accomplish the target task (*i.e.*, a potential transformation method to be verified). To address the UI navigation challenges outlined in §IV-B while ensuring accurate and adaptive automation, we implement a structured Chain-of-Thought reasoning approach, guiding LLMs through the following 6 phases, as depicted in Fig. 8.

(1) **Task Progress Evaluation**: Assesses current advancement toward the main goal by analyzing unresolved prerequisites (*e.g.*, incomplete input fields, pending steps) and correlates them with historical logs to generate progress reports.

(2) **Repetitive Action Analysis**: Analyzes action history to flag recurring sequences (>3 iterations) via pattern analysis, identifying necessary repetition, or triggering alternative path generation or strategy adjustments aligned with the primary task objective to avoid stagnation.

(3) **Exploration/Completion Strategy Selection**: Determines workflow prioritization (direct task progression vs. UI exploration) based on real-time interface responsiveness, task urgency, and historical success rates of similar actions.

(4) **Context-Specific Subtask Identification**: Decomposes overarching goals into atomic subgoals (*e.g.*, `Complete login form`) by mapping available UI elements (buttons, input fields) to task requirements, ensuring alignment with the main objective.

(5) **Action Generation**: Produces executable commands (clicks, inputs) bound to element IDs, with enforced completeness verification (including form field population checks and target element validation) before execution.

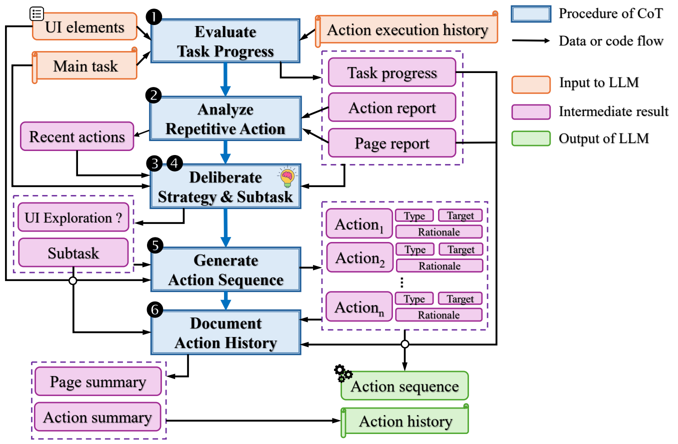


Fig. 8: Chain-of-thought reasoning process for decision making and action sequence generation.

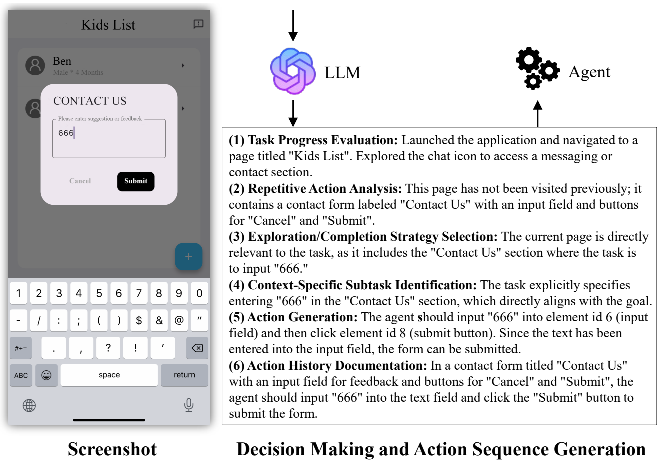
(6) **Action History Documentation**: Maintains a dual-format action log comprising human-readable UI page semantics (*e.g.*, `Settings menu with toggle switches`) and action summaries (*e.g.*, `click [Submit] to navigate to dashboard`), supporting real-time auditability and retrospective analysis. Successful executions are archived in the historical record.

Such intelligent decision-making capability facilitates dynamic workflow adjustments and emulates human-like interaction by circumventing unresponsive components, preventing repetitive operations, and restarting apps to eliminate persistent overlays. Furthermore, it promotes active exploration in the face of ambiguous tasks by systematically examining all remaining actionable elements while avoiding previously attempted actions, thereby ensuring an exhaustive traversal of each UI interface to comprehensively evaluate all potential solutions. The prompt template for LLM interaction is shown in Listing 3.

```
<instruction>
    Given the {current UI elements}, {action
    execution history}, and {primary task}, employ
    structured reasoning to: (1) identify the most
    appropriate immediate sub-task, and (2)
    formulate a coherent, goal-oriented action
    sequence. Prioritize handling urgent UI
    components (e.g., advertisements, pop-up dialogs
    ), avoid redundant operations, and ensure all
    proposed actions maintain contextual relevance,
    operational feasibility, and explicit
    justification. Present the analysis as
    structured output without additional commentary.
</instruction>
<output_format>
    Return a single JSON object containing the
    following fields: "summary" (string), "
    history_analysis" (string), "subtask" (string),
    and "action_plan" (a list of JSON objects, each
    containing "action" and "rationale" strings).
</output_format>
```

Listing 3: Prompt template for decision making and action sequence generation.

Fig. 9 demonstrates the module's processing of the `Vaccination Schedule` app's current UI page.



Kids List
Ben
Male * 4 Months
CONTACT US
Please enter suggestion or feedback
666
Cancel
Submit
LLM
Agent
Screenshot
Decision Making and Action Sequence Generation

(1) Task Progress Evaluation: Launched the application and navigated to a page titled "Kids List". Explored the chat icon to access a messaging or contact section.
(2) Repetitive Action Analysis: This page has not been visited previously; it contains a contact form labeled "Contact Us" with an input field and buttons for "Cancel" and "Submit".
(3) Exploration/Completion Strategy Selection: The current page is directly relevant to the task, as it includes the "Contact Us" section where the task is to input "666."
(4) Context-Specific Subtask Identification: The task explicitly specifies entering "666" in the "Contact Us" section, which directly aligns with the goal.
(5) Action Generation: The agent should input "666" into element id 6 (input field) and then click element id 8 (submit button). Since the text has been entered into the input field, the form can be submitted.
(6) Action History Documentation: In a contact form titled "Contact Us" with an input field for feedback and buttons for "Cancel" and "Submit", the agent should input "666" into the text field and click the "Submit" button to submit the form.

Fig. 9: The *Decision Making and Action Sequence Generation* module's output for `Vaccination Schedule`'s current UI.

ChameleoScan evaluates both page visitation history and task relevance before generating executable instructions involving: (1) locating the "Contact Us" form, (2) entering "666" in the specified input field, and (3) submitting the form - thereby guiding the iOS agent's subsequent operations.

❻ *Action Execution*

The agent then sequentially executes generated actions to complete an operational cycle. To address challenges posed by dynamic elements (see §IV-B), we implement adaptive execution strategies beyond direct action execution.

The temporal gap between UI data acquisition and the eventual execution of actions may introduce *transient UI components* (*e.g.*, pop-ups) that invalidate initial analyses. Such cases require restarting the cycle from data acquisition to accommodate the updated interface context. Nonetheless, certain apps inherently include dynamic elements (*e.g.*, image carousels) that are neither obstructive nor urgent. To prevent such benign dynamics from disrupting analysis and decision-making, we incorporate a similarity comparison mechanism between view hierarchies captured before and after execution. This mechanism determines whether a structural change justifies a cycle restart. Specifically, (1) view hierarchies with a tree edit distance below 3 are treated as similar, thereby tolerating moderate dynamics in fixed-position components; and (2) view hierarchies with a Jaccard index exceeding 0.8 for all texts and labels are also regarded as similar, accommodating content changes in visually stable frames. If either condition is met and all intended action targets remain present, the UI is deemed valid for continued execution.

These thresholds were empirically derived through manual inspection of apps exhibiting dynamic content or transient UI components. We examined 20 pairs of user interfaces captured before and after dynamic UI changes (*e.g.*, countdowns, pop-ups), computing text-level Jaccard similarity coefficients. We determined an optimal operating range (0.77-0.84) through Youden index analysis [44], and then selected 0.8 as the threshold that balances tolerance for benign content variations against sensitivity to genuine UI changes. However, for apps with minimal textual content (*e.g.*, auto-scrolling galleries, icon-heavy layouts), text-based similarity proves inadequate. Through iterative testing, we determined that a tree edit distance threshold of 3 effectively accommodates minor structural modifications (equivalent to 1-2 leaf-node operations) while reliably detecting significant layout transformations.

Furthermore, for *disruptive elements* with consistent structures (*e.g.*, system permission dialogs, ads with identifiable close buttons), we implement predefined handlers for immediate dismissal upon detection, significantly improving both system responsiveness and operational efficiency. The framework additionally incorporates recovery protocols (*e.g.*, app foregrounding, keyboard dismissal) to address potential operational errors. Unrecoverable failures, including app crashes or hardware incompatibilities, are classified as detection failures.

❼ *Transformation Validation*

Our preliminary investigation (§III-B4) reveals that successful transformation consistently converts Chameleon apps into an illicit operational mode, with unambiguous UI evidence revealing hidden functionalities. To detect such transformations, ChameleoScan employs a structured four-phase LLM-guided framework that systematically analyzes discrepancies between an app's runtime behavior and its *functionality profile* derived from App Store metadata:

(1) **Functionality Consistency Analysis**: LLMs assess alignment between observed runtime behaviors and declared functionalities, flagging significant mismatches as positive detections while confirming compliant pages as negative.
(2) **Non-Indicative Element Filtering**: LLMs identify and exclude interface elements unrelated to core functionality or transformation indicators (*e.g.*, ads, settings panels), automatically designating such screens as negative detections.
(3) **Evidence-Based Reasoning**: LLMs must provide explicit, observable UI evidence (*e.g.*, "hidden mode" toggle exists) for all determinations to mitigate hallucination issues.
(4) **Confidence-Based Scoring**: LLMs assign standardized discrepancy scores (0.0-1.0 scale), where scores $\geq$0.8 confirm transformation, $\leq$0.5 indicate compliance, and intermediate scores flag cases requiring further scrutiny. These scores function exclusively as decision-making aids.

The implementation utilizes few-shot learning with carefully selected exemplars in a structured prompt framework, combining App Store metadata with runtime UI evidence (screenshots and extracted text labels). ChameleoScan maintains rigorous evidence-based validation by strictly tethering all determinations to observable interface characteristics. The prompt template is provided in Listing 4.

```
<instruction>
    Given the {runtime behavior} and {app metadata},
     determine whether the observed app runtime
    behavior clearly contradicts or deviates from
    the app's declared functionality. Base your
    assessment exclusively on visible UI evidence,
    citing specific textual or visual indicators to
    support your conclusions. Additionally, flag
    instances where advertisements or transient
    elements substantially obstruct interface
    evaluation. Provide no additional output.
</instruction>
<output_format>
    Return a list containing a single JSON object
    with the following fields: "determination" (
    string), "confidence" (float between 0-1), and "
    rationale" (1-3 sentences). Adhere strictly to
    the exemplar structure provided in <examples>.
</output_format>
<examples>
    1. Input: [runtime behavor_1], [metadata_1];
    Output: [determination_1], [confidence_1], [
    reasoning_1]}
    2. Input: [runtime behavor_2], [metadata_2];
    Output: [determination_2], [confidence_2], [
    reasoning_2]}
    ...
    n. Input: [runtime behavor_n], [metadata_n];
    Output: [determination_n], [confidence_n], [
    reasoning_n]}
</examples>
```

Listing 4: Prompt template for transformation validation.

To address apps that disclose transformation methods solely through runtime indicators, we augment the few-shot examples with observed behavioral patterns, enabling effective identification and extraction of such methods. When explicit cues such as "unlock hidden mode" are detected, the *Transformation Validation* module returns an immediate positive result. For implicit instructions like "Click the submit button 3 times to proceed," the system records the cue in the action history and adjusts the exploration path in subsequent iterations.

We demonstrate CHAMELEOSCAN's workflow through the `Vaccination Schedule` case study, documenting the complete detection pipeline from initial metadata analysis and transformation inference (❶–❷) to iterative UI processing cycles encompassing acquisition, comprehension, decision-making, action execution, and validation (❸–❼). The full procedural sequence is detailed in Appendix A (Fig. 11–14).

V. EVALUATION

We assess CHAMELEOSCAN's effectiveness by investigating three critical research questions:

- **RQ1:** How does it perform overall on the *KNOWN* dataset of apps collected in the preliminary study (§III)?
- **RQ2:** How does it perform on core tasks essential to automated detection on the *KNOWN* dataset?
- **RQ3:** How effectively does it identify Chameleon apps in practical deployments using an *UNKNOWN* dataset?

A. Experiment Setup

Data Collection. To evaluate CHAMELEOSCAN, we constructed two distinct datasets: *KNOWN* and *UNKNOWN*. **(1) *KNOWN* dataset.** We began by selecting a pool of popular apps from the App Store based on reputable developers, high popularity, and compatibility with our test devices. From this collection, 300 apps were randomly sampled and downloaded. Following metadata inspection and manual testing on real devices, 233 apps spanning 26 categories were verified as benign (*e.g.*, TikTok, TestFlight). Together with the 234 Chameleon apps described in §III-A, they form the *KNOWN* dataset. For tests requiring execution on real devices, a functional subset of 131 active Chameleon apps was utilized, excluding those that became non-operational between collection and evaluation periods. **(2) *UNKNOWN* dataset.** The dataset was constructed through continuous monitoring of newly released App Store apps throughout December 2024. From 23,416 initially collected apps, quality-control filtering preserved 3,253 entries demonstrating active user adoption (non-zero downloads). Subsequent compatibility testing confirmed 1,644 apps as device-testable, distributed across 39 categories, with *Utility* emerging as the dominant category at 16.85% representation.

TABLE IV: CHAMELEOSCAN's detection efficacy on *KNOWN*

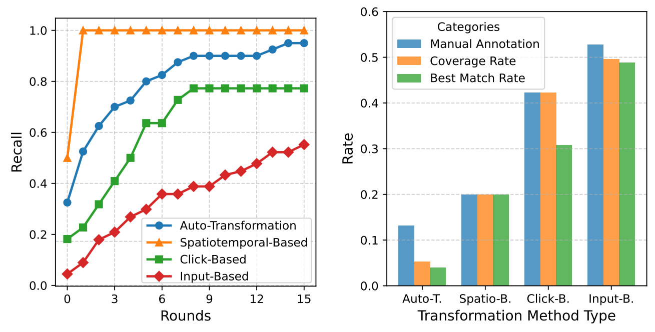
Type	AR	Recall (%)	Recall′ (%)
Auto-Transformation	2.58	95.00	100.00
Spatiotemporal-Based	0.50	100.00	100.00
Click-Based	2.81	77.27	100.00
Input-Based	4.18	55.22	92.50
Total	**3.45**	**71.76**	**96.91**

Experiment Environment. The iOS agent in CHAMELEOSCAN was developed through integration of established open-source tools: Frida [45] provided application lifecycle management and runtime context manipulation, Appium [46] (via XCUITest) enabled comprehensive UI hierarchy analysis including hybrid-rendered components (Flutter and WebView implementations), and ZXTouch [47] facilitated precise system-level interaction simulation (including rapid touch sequences and text input). libimobiledevice [48] managed device communication and application operations (installation/uninstallation processes and screenshot capture), with system parameter modifications (temporal and geolocation data) executed through SSH command-line operations. All testing occurred on an iPhone XR (iOS 14.4.1) under controlled laboratory conditions.

B. RQ1: Performance on KNOWN Dataset

We conducted end-to-end testing on the *KNOWN* dataset. Table IV shows CHAMELEOSCAN's detection efficacy across 3 metrics: *recall*, *precision*, and *average round (AR)* denoting the mean execution iterations needed per app for detection.

Overall, CHAMELEOSCAN achieves a recall of 71.76% and a precision of 100%. Among the four Chameleon app types, *Auto-Transformation* and *Spatiotemporal-Based* apps attain high recall rates of 95% and 100%, respectively. In contrast, *Click-Based* and *Input-Based* apps show lower recall, likely due to the increased complexity in identifying transformation triggers and completing the associated transformation task automatically. The limited false negatives in *Auto-Transformation* detection predominantly occur when apps launch with full-screen advertisements containing obfuscated close buttons that resist reliable automated recognition. To evaluate robustness under optimized conditions, we further assessed recall (denoted as Recall′ in Table IV) on a subset of 97 apps whose transformation behaviors can be triggered solely using user review information by professional security auditors. In this



(a) Overall Recall Curves **(b)** Accuracy of Method Inference

Fig. 10: Performance evaluation of ChameleoScan: (a) recall curves across interaction rounds by Chameleon app type, (b) accuracy of transformation method inference by app type.

setting, ChameleoScan achieves 100% recall for *Click-Based* apps and 92.50% for *Input-Based* apps, demonstrating that its automation and auditing capabilities closely approach human-level performance.

To demonstrate efficiency, we plotted recall across interaction rounds, shown in Fig. 10a. Notably, 70.00% of *Auto-Transformation* and 100% of *Spatiotemporal-Based* apps are successfully identified within 3 rounds. The remaining *Auto-Transformation* apps require more rounds due to intrusive ads with obfuscated close buttons not present in the view hierarchy. The other two Chameleon types show more gradual recall growth. Nevertheless, among detected apps, 14 out of 17 *Click-Based* and 24 out of 37 *Input-Based* apps are identified within 6 rounds. On average, ChameleoScan requires only 2.26 rounds per app (3.45 for Chameleon and 1.59 for benign).

C. RQ2: Task-wise Performance

We analyze ChameleoScan's performance across three representative tasks: (i) inferring transformation methods, (ii) recognizing and semantically understanding UI elements, and (iii) handling interference and dynamic UI changes.

Transformation Method Inference. We analyze the transformation method inference results for 234 Chameleon apps and 233 benign apps in the *KNOWN* dataset. On average, ChameleoScan infers 0.71 transformation methods per Chameleon app and 0.08 per benign app. Fig. 10b presents the coverage and best match rates, defined as the proportion of apps where the predicted method with the highest confidence aligns with the actual transformation method, across the four Chameleon app types, compared with manual annotation results. For *Auto-Transformation* and *Spatiotemporal-Based* apps, both manual and automated recognition accuracies are relatively low, as these simpler strategies are rarely mentioned explicitly in user reviews. In contrast, for *Click-Based* and *Input-Based* apps, the inference module demonstrates strong coverage and best match rates, closely approximating manual performance (42.3%/30.77% vs. 42.3% for *Click-Based*, and 49.61%/48.82% vs. 52.76% for *Input-Based*). Additionally, the module successfully decodes obfuscated cues, *e.g.*, deriving "5201080" from "⑤2o①o8o."

An ablation study removing BundleID and description metadata reveals 12 failure cases where transformation methods cannot be inferred. These apps exclusively employ *Input-Based* methods requiring magic string codes (*e.g.*, "666"), yet provide no relevant cues present in user reviews or runtime behavior. By contrast, with complete metadata, ChameleoScan successfully learns these codes from few-shot exemplars by identifying latent metadata patterns (*e.g.*, shared BundleID substrings like "zw.sz" or similar app functionalities), confirming the critical value of metadata components.

UI Element Recognition and Comprehension. With the *UI Data Acquisition and Enhancement* module, ChameleoScan effectively reduces noise in the view hierarchy and enriches it with essential semantic cues. Specifically, the average token length of the view hierarchy is reduced from 7,892 to 2,874, while each UI page is supplemented with an average of 2.09 OCR-recognized texts and 2.28 icon classification results.

TABLE V: Component Impact Analysis for UI Element Recognition and Comprehension: Evaluation of tree pruning ($Pruning$), image semantic augmentation (SS_{img}), and icon semantic enhancement (SS_{icon}) on recognition precision (P), recall (R), and accuracy (Acc) across 927 XML-formatted interface pages from the *KNOWN* dataset.

Modules	UI Recognition			UI Comprehension
	P (%)	R (%)	Acc (%)	Acc (%)
{}	85.50	63.21	74.19	87.26
{ *Pruning* }	87.04	65.45	76.02	87.10
{ *Pruning, SS_img* }	88.42	68.26	77.58	89.39
{ *Pruning, SS_icon* }	87.84	66.38	77.05	91.85
{ *Pruning, SS_img, SS_icon* }	**89.36**	**70.39**	**78.94**	**92.54**

To evaluate how these enhancements improve the LLM's ability (specifically, the *UI Comprehension* module in §IV-C) to identify interactive UI elements, we manually assessed 927 different UI pages encountered during app testing on the functional ground-truth dataset. Our analysis focused on two categories: (i) frequently observed UI elements related to transformation methods, such as buttons, icons, and text fields (as listed in Table II), and (ii) potential interference elements, including ads and pop-ups. Based on this, we conducted an ablation study to evaluate the contribution of each submodule, as presented in Table V. ChameleoScan achieves an overall precision of 89.36% and a recall of 70.39% in detecting key UI elements, and enables the LLM to accurately infer the potential functionalities of 92.54% of those elements. Additionally, we quantitatively measured recall improvements for key element types: icon recall increased from 57.22% to 77.45%, and text field recall from 45.30% to 56.10%. These results mark a significant enhancement over the baseline configuration without the proposed submodules, forming a solid foundation for downstream components while validating the effectiveness of our design choices.

Interference Handling and Dynamic UI Management. ChameleoScan demonstrates robust handling of both distracting elements (*e.g.*, ads, pop-ups) and dynamic interface changes during task execution. System evaluation across the *KNOWN* dataset yields a total of 4,415 UI pages observed, with manual analysis identifying distracting elements in 19.21% of cases (848 pages) affecting 39.19% of tested apps. Overall, ChameleoScan achieved 96.70% precision and 97.03% recall in identifying these elements, and further managed to resolve 85.96% of intrusive ads (*e.g.*, via dismissal, countdown awaiting) and 95.56% of pop-up instances (*e.g.*, through automated permission granting or dialog interpretation). Without the proposed mitigation strategies, the system either fails to resolve such interference or requires substantially extended processing time, confirming the practical effectiveness of our architectural decisions.

Furthermore, the *Action Execution* module's dynamic element handler (§IV-C) detected 309 instances of significant pre-execution UI modifications. By automatically discarding outdated UI pages and re-extracting latest layouts, the system maintained reliable task completion across dynamic transitions.

D. RQ3: Performance on UNKNOWN Dataset

To assess RQ3, we examined ChameleoScan's detection efficacy and runtime performance using the real-world *UNKNOWN* dataset, validating its ability to identify Chameleon apps in previously unseen environments. The system categorized all tested apps into either Chameleon apps or inconclusive cases, as summarized in Table VII.

Detection Results. Out of 1,644 *UNKNOWN* apps, ChameleoScan identified 162 (9.85%) as Chameleon apps, with manual verification confirming a precision of 92.59%. Among the 12 false positives, 8 resulted from ambiguous app descriptions or interference from promotional content, leading to misinterpretation. The remaining 4 were caused by misclassification of legitimate UIs as suspicious.

For each of the remaining 1,482 inconclusive apps, we manually inspect their app metadata, runtime UI behavior, and whether they appeared in known Chameleon promotion channels (Table I). This analysis revealed 35 missed Chameleon apps (*i.e.*, false negatives), 20 resulting from apps no longer being maintained, which led to unresponsive or blank pages after transformation attempts, and 15 due to failures in inferring transformation methods or identifying semantically meaningful clickable elements. These findings indicate that, under ideal conditions, ChameleoScan could achieve even higher recall and identify suspicious apps in real-world settings.

TABLE VI: Resource and Cost Analysis of ChameleoScan on *UNKNOWN* dataset (1,644 apps).

Module / Submodule	**Avg. Time**	**#Tokens**	**Cost**	**#Count**
Transformation Method Inference	2.43 s	2,949	0.79 ¢	1
Environment Setup and App Installation	9.23 s	–	–	1
App-agnostic Triggering Attempts	22.16 s	–	–	1
UI Data Acquisition and Enhancement				
Acquisition	5.46 s	–	–	3.41
Enhancement	1.70 s	–	–	3.41
UI Comprehension	8.16 s	3,744	1.17 ¢	3.41
Decision Making and Action Sequence Generation	4.49 s	2,612	0.90 ¢	3.35
Action Execution	6.66 s	–	–	3.35
Transformation Validation	5.31 s	2,881	0.79 ¢	3.40
Per App	133.99 s	27,686	10.48 ¢	–

Runtime Efficiency and Cost Analysis. Among the 162 Chameleon apps detected, 92.59% (150 apps) were identified within 3 interaction rounds. The remaining 7.41% (12 apps) required an average of 7.67 rounds due to more complex transformation logic. We further conducted a resource and cost analysis as illustrated in Table VI. On average, ChameleoScan required 2.43 seconds for transformation inference, 8.16 seconds for UI comprehension, 4.49 seconds for decision making, 6.66 seconds for action execution, and 5.31 seconds for transformation validation. Each app took approximately 133.99 seconds to test, including 9.23 seconds for app installation and 22.16 seconds for app-agnostic transformation attempts, with a token count of 27,686 (around $0.1). This observed timeframe aligns with the predefined 2-minute limit of execution (typically 3–4 rounds) for cases without inferred transformations, as most *Auto-Transformation* apps complete within this period. These results highlight ChameleoScan's efficiency, scalability, and cost-effectiveness for large-scale deployment.

TABLE VII: ChameleoScan's detection on *UNKNOWN*

Category	**#Apps (Correct)**	**AR**	**Avg. Time (s)**	**P (%)**
Chameleon App	162 (150)	0.89	81.33	92.59
Inconclusive	1,482 (1,447)	2.13	139.74	97.53
Total	**1,644 (1,597)**	**2.01**	**133.99**	**97.14**

Insights from Newly Discovered Chameleon Apps. Analysis of newly detected Chameleon apps yielded noteworthy insights.

- **Emerging Illicit Services**: While known Chameleon apps predominantly facilitate gambling, pornography, and content piracy, newly identified variants predominantly promote game account trading platforms and financial services.
- **Developer Attribution Patterns**: Among 128 newly discovered instances, a prevalent pattern involved automatic redirection to external websites upon launch. Notably, developer naming conventions frequently disclosed actual functionality, for example, `com.ImgPixerNwa` (masquerading as a calculator but redirecting to an e-wallet) bore the developer designation "imtoken wallet app bitpie tpwallet tokenpocket TP lMT." Subsequent keyword analysis ("gamble,""wallet") across 23,416 apps in the *UNKNOWN* dataset identified 153 additional apps associated with prohibited services, confirming developer metadata as a reliable detection signal.

These results highlight ChameleoScan's effectiveness in detecting novel Chameleon apps under real-world conditions, including those utilizing previously unobserved code patterns or evasion techniques. *The findings were responsibly reported to Apple, which acknowledged the issues and expressed interest in further dialogue regarding our study.* While most reported apps have since been removed from the App Store, we cannot definitively attribute these takedowns to our disclosure.

E. Detection Performance Without User Reviews

While user reviews offer valuable transformation cues, ChameleoScan achieves reliable detection through multimodal analysis without exclusive reliance on them. As highlighted in §III-B4, the system leverages multiple alternative indicators including app metadata (BundleID and descriptions), runtime interface patterns (*e.g.*, pop-up notifications, ads), and predefined transformation sequences. This approach successfully identified 23 of 44 review-deficient apps in the *KNOWN* dataset, covering all four major transformation categories.

F. ChameleoScan *vs. Chameleon-Hunter & Mask-Catcher*

We evaluate ChameleoScan in comparison with two closely related tools, Chameleon-Hunter [5] and Mask-Catcher [6], based on a manually annotated dataset.

Comparison with Chameleon-Hunter. Prior work by Zhao *et al.* [6] demonstrated Chameleon-Hunter's limited effectiveness against hybrid-framework Chameleon apps. In our evaluation using the *KNOWN* dataset (234 apps), we successfully decrypted and analyzed the Mach-O binaries of 223 apps, revealing their UI framework implementations. The analysis identified 89 apps (39.91%) utilizing Flutter for UI rendering, 76 exclusively and 13 combining Flutter with native UI components. Furthermore, 16 apps (7.17%) employed WebView to deliver illicit services through dynamically embedded URLs rather than hardcoded. Collectively, 104 apps (46.64%) relied on hybrid frameworks (Flutter or WebView) for either legitimate or illicit functionality, significantly impairing Chameleon-Hunter's detection efficacy.

We further evaluated Chameleon-Hunter on a subset of our newly discovered Chameleon apps, which define static UI pages and view controllers in the binary but immediately dismantle them during early callback flows (*e.g.*, `willConnectToSession`), redirecting users to external websites via Safari. As the destination URLs are dynamically fetched from CloudKit at runtime, without any hard-coded traces in the binary, such designs completely evade the static analysis techniques employed by Chameleon-Hunter.

Comparison with Mask-Catcher. To compare with the performance of Mask-Catcher, we gathered metadata for 185 manually labeled Chameleon apps from the *UNKNOWN* dataset (57 containing user reviews and 128 review-free instances), along with 172 apps recommended under the "You Might Also Like" section. Given the unavailability of original models and threshold configurations, we reimplemented its core logic based on published source code.

Mask-Catcher successfully flagged 46 apps as suspicious, predominantly through detecting semantic inconsistencies between app descriptions and user reviews (44 out of 46). However, this detection was exclusively limited to apps containing user reviews, with none of the 128 review-free apps being flagged. Although Mask-Catcher can infer suspicious apps based on recommendation relationships, only 19 apps in our dataset exhibited such connections (including 5 without reviews), of which merely 8 were flagged as suspicious.

Mask-Catcher may fail to detect Chameleon apps even when relevant user reviews are available. The failure to detect 11 such apps can be attributed to the following factors: 3 cases involved language discrepancies between app descriptions and reviews (*e.g.*, English descriptions paired with Chinese reviews), disrupting semantic alignment; 2 cases contained insufficient quantities of suspicious reviews; and 4 cases resulted from erroneous filtering of potentially suspicious content (*e.g.*, repeated numerical patterns like "777"). The remaining 2 cases involved coordinated campaigns flooding apps with irrelevant positive reviews (*e.g.*, "TV, works great"), effectively drowning out genuine user feedback and subverting review-based analysis. These findings highlight how strategic obfuscation tactics and coordinated user behavior undermine Mask-Catcher.

In Mask-Catcher, the identification of suspicious apps requires additional analysis using BinDiff [49]. While we did not replicate Mask-Catcher's full classifier pipeline, we examined the distribution of maximum similarity scores between 58 suspicious apps from our *UNKNOWN* dataset and 79 labeled Chameleon apps from the *KNOWN* dataset. The results show that 94.83% of suspicious apps had a maximum similarity score below 0.5, with 36.21% falling below 0.25. A control group of benign apps exhibited a comparable pattern, with 30.00% scoring below 0.25 and all scoring below 0.5. These findings indicate that many suspicious apps are not mere repackaged versions but are likely independently developed or significantly restructured. Importantly, ChameleoScan was still able to detect them effectively, demonstrating strong generalization beyond known code patterns.

VI. Discussion

Hallucination Mitigation. To address potential LLM hallucinations, we have incorporated multiple strategies: (i) transformation behaviors undergo dual-validation protocols before final determination to minimize false positives; (ii) the LLM is explicitly prompted to verify action preconditions during Chain-of-Thought reasoning processes; and (iii) each operational iteration incorporates code-level verification (*e.g.*, UI element identifier validation) and operational safeguards (*e.g.*, app foreground management or keyboard dismissal). While occasional false positives may still occur due to factors like deliberately obfuscated ads or low-quality app content, our approach achieves 97.14% precision on the UNKNOWN dataset (as documented in Table IV), demonstrating effective hallucination management.

Evasion Resistance. Our methodology builds upon two key empirical observations: Chameleon apps predominantly employ straightforward transformation mechanisms frequently revealed through app metadata or runtime interface elements. While adversaries could theoretically employ evasion tactics such as extended delay mechanisms, geo-location restrictions, server-side control, generic descriptions, and review manipulation, these strategies inherently conflict with fundamental characteristics of Chameleon apps. Extended delays compromise user engagement, strict geo-fencing limits audience reach, server-side control increases operational complexity, and comprehensive review sanitization remains practically infeasible due to platform constraints. To achieve scalability, attackers must maintain app discoverability, inevitably generating detectable signals across user reviews, runtime behaviors, and network communications, precisely where our dynamic analysis excels.

Potential countermeasures include multi-epoch cross-regional testing for delayed or geo-restricted transformations, network traffic analysis for remote configuration detection, review-graph examination for off-platform promotion patterns, and optional human verification for ambiguous cases. These approaches strategically target modifiable adversary behaviors while leveraging immutable constraints including usability requirements, scalability needs, and the fundamental necessity to communicate transformation methods to end users.

Novelty and Advancement. Our work establishes a systematic taxonomy of transformation methods, an area previously underdeveloped. Unlike prior studies, we provide precise operational definitions for these methods and empirically examine their prevalence in the real world.

Existing detection methods [5], [6] frequently prove inadequate even against established transformation types, particularly when apps employ hybrid frameworks or expose transformation logic exclusively during runtime without metadata indicators. The dependence of these approaches on static analysis and metadata fundamentally constrains their effectiveness in such scenarios, highlighting the necessity of our dynamic, multimodal methodology as a substantive advancement beyond incremental, rule-based improvements.

Furthermore, our work addresses two distinct challenges previously unresolved by existing approaches. First, conventional methods demonstrate insufficient capability for accurately identifying relevant UI elements and efficiently managing distracting interface components within our problem context. Second, we resolve the unique challenge of executing fine-grained exploration under ambiguous task specifications, where transformation methods typically lack the precise input parameters required for deterministic automation.

Limitations. Although dynamic analysis offers distinct advantages for detecting instant-transformation apps, ChameleoScan's effectiveness depends critically on accessible app metadata and runtime UI information. The LLM integration, while enabling advanced analysis, introduces substantial computational overhead that may limit scalability in production environments. While technically adaptable to Android through agent substitution, this portability remains unevaluated given the current absence of verified Android Chameleon instances.

VII. Conclusion

In this study, we introduce a comprehensive dataset of 500 iOS Chameleon apps, enabling systematic identification of 10 categories of distinct transformation patterns (including 4 novel variants). We further propose ChameleoScan, a novel LLM-driven automated UI exploration framework that integrates predictive metadata analysis with human-par interaction strategies for reliable Chameleon app verification. Rigorous evaluation confirms the system's detection efficacy and operational practicality, demonstrating significant advancements in automated mobile interface analysis.

VIII. Ethical Considerations

We follow the ethical guidelines set forth in the Menlo Report [50], maintaining a careful balance between potential risks and research benefits. No financial incentives were provided to Chameleon app developers during our consultations on evasion techniques. While downloading and activating hidden functionalities may yield minimal financial benefit for developers, we consider such incidental gains negligible relative to the broader positive impact of this work, which aims to improve understanding and mitigation of Chameleon apps within the research community.

Acknowledgments

This work was supported in part by the National Key R&D Program of China (No.2023YFB3106800) and the National Natural Science Foundation of China (No.62272410).

References

[1] "Porn apps disguised as learning apps on China's iOS App Store," TechNode, https://technode.com/2023/10/08/porn-apps-disguised-as-learning-apps-on-chinas-ios-app-store.

[2] "Porn platform poses as a snack delivery app on iOS," LINE TODAY, https://today.line.me/hk/v2/article/DPeGZV.

[3] "2023 App Store transparency report," https://www.apple.com/legal/more-resources/docs/2023-App-Store-Transparency-Report.pdf.

[4] Y. Lee, X. Wang, K. Lee, X. Liao, X. Wang, T. Li, and X. Mi, "Understanding iOS-based crowdturfing through hidden UI analysis," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 765–781.

[5] Y. Lee, X. Wang, X. Liao, and X. Wang, "Understanding illicit UI in iOS apps through hidden UI analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2390–2402, 2019.

[6] Y. Zhao, L. Yu, Y. Sun, Q. Liu, and B. Luo, "No source code? no problem! demystifying and detecting mask apps in iOS," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 2024, pp. 358–369.

[7] H. Wen, Y. Li, G. Liu, S. Zhao, T. Yu, T. J.-J. Li, S. Jiang, Y. Liu, Y. Zhang, and Y. Liu, "Autodroid: LLM-powered task automation in Android," in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, 2024, pp. 543–557.

[8] D. Ran, H. Wang, Z. Song, M. Wu, Y. Cao, Y. Zhang, W. Yang, and T. Xie, "Guardian: A runtime framework for LLM-based UI exploration," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 958–970.

[9] Z. Chen, L. Wu, Y. Hu, J. Cheng, Y. Hu, Y. Zhou, Z. Tang, Y. Chen, J. Li, and K. Ren, "Lifting the grey curtain: Analyzing the ecosystem of Android scam apps," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 4, pp. 3406–3421, 2024.

[10] Y. Hu, H. Wang, Y. Zhou, Y. Guo, L. Li, B. Luo, and F. Xu, "Dating with Scambots: Understanding the Ecosystem of Fraudulent Dating Applications," *IEEE TDSC*, 2018.

[11] Z. Chen, J. Liu, Y. Hu, L. Wu, Y. Zhou, Y. He, X. Liao, K. Wang, J. Li, and Z. Qin, "DeUEDroid: Detecting Underground Economy Apps Based on UTG Similarity," in *ACM ISSTA*, 2023.

[12] G. Hong, Z. Yang, S. Yang, X. Liaoy, X. Du, M. Yang, and H. Duan, "Analyzing Ground-truth Data of Mobile Gambling Scams," in *IEEE S&P*, 2022.

[13] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*. Association for Computing Machinery, 2012, p. 281–294.

[14] Y. Gao, H. Wang, L. Li, X. Luo, G. Xu, and X. Liu, "Demystifying Ilegal Mobile Gambling Apps," in *Proceedings of the Web Conference*, 2021, pp. 1447–1458.

[15] Y. Liu, Y. Zhang, B. Liu, H. Duan, Q. Li, M. Liu, R. Li, and J. Yao, "Tickets or privacy? understand the ecosystem of chinese ticket grabbing apps," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 5107–5124.

[16] M. Egele, C. Krügel, E. Kirda, and G. Vigna, "PiOS: Detecting privacy leaks in iOS applications," in *Network and Distributed System Security Symposium*, 2011.

[17] D. Kong, L. Cen, and H. Jin, "Autoreb: Automatically understanding the review-to-behavior fidelity in android applications," in *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, 2015, pp. 530–541.
[18] D. C. Nguyen, E. Derr, M. Backes, and S. Bugiel, "Short text, large effect: Measuring the impact of user reviews on android app security & privacy," in *2019 IEEE symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 555–569.
[19] M. Hatamian, J. Serna, and K. Rannenberg, "Revealing the unrevealed: Mining smartphone users privacy perception on app markets," *Computers & Security*, vol. 83, pp. 332–353, 2019.
[20] C. Tao, H. Guo, and Z. Huang, "Identifying security issues for mobile applications based on user review summarization," *Information and Software Technology*, vol. 122, p. 106290, 2020.
[21] H. Gao, C. Guo, G. Bai, D. Huang, Z. He, Y. Wu, and J. Xu, "Sharing runtime permission issues for developers based on similar-app review mining," *Journal of Systems and Software*, vol. 184, p. 111118, 2022.
[22] R. Wang, Z. Wang, B. Tang, L. Zhao, and L. Wang, "Smartpi: Understanding permission implications of android apps from user reviews," *IEEE Transactions on Mobile Computing*, vol. 19, no. 12, pp. 2933–2945, 2019.
[23] H. Wu, W. Deng, X. Niu, and C. Nie, "Identifying key features from app user reviews," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 922–932.
[24] Y. Hu, H. Wang, T. Ji, X. Xiao, X. Luo, P. Gao, and Y. Guo, "Champ: Characterizing undesired app behaviors from user comments based on market policies," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 933–945.
[25] Y. Song, Y. Bian, Y. Tang, G. Ma, and Z. Cai, "Visiontasker: Mobile task automation using vision based ui understanding and llm task planning," in *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, 2024, pp. 1–17.
[26] A. R. Sereshkeh, G. Leung, K. Perumal, C. Phillips, M. Zhang, A. Fazly, and I. Mohomed, "Vasta: a vision and language-assisted smartphone task automation system," in *Proceedings of the 25th international conference on intelligent user interfaces*, 2020, pp. 22–32.
[27] M. D. Vu, H. Wang, J. Chen, Z. Li, S. Zhao, Z. Xing, and C. Chen, "Gptvoicetasker: Advancing multi-step mobile task efficiency through dynamic interface exploration and learning," in *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, 2024, pp. 1–17.
[28] L. Zhang, S. Wang, X. Jia, Z. Zheng, Y. Yan, L. Gao, Y. Li, and M. Xu, "Llamatouch: A faithful and scalable testbed for mobile ui automation task evaluation," *arXiv preprint arXiv:2404.16054*, 2024.
[29] B. Wang, G. Li, and Y. Li, "Enabling conversational interaction with mobile ui using large language models," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–17.
[30] T. Huang, C. Yu, W. Shi, Z. Peng, D. Yang, W. Sun, and Y. Shi, "Prompt2task: Automating UI tasks on smartphones from textual prompts," *ACM Trans. Comput.-Hum. Interact.*, 2025.
[31] S. Lee, J. Choi, J. Lee, M. H. Wasi, H. Choi, S. Ko, S. Oh, and I. Shin, "Mobilegpt: Augmenting llm with human-like app memory for mobile task automation," in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, ser. ACM MobiCom '24, 2024, p. 1119–1133.
[32] Y. Liu, P. Li, Z. Wei, C. Xie, X. Hu, X. Xu, S. Zhang, X. Han, H. Yang, and F. Wu, "InfiGUIAgent: A multimodal generalist GUI agent with native reasoning and reflection," 2025. [Online]. Available: https://arxiv.org/abs/2501.04575
[33] J. Zhang, J. Wu, Y. Teng, M. Liao, N. Xu, X. Xiao, Z. Wei, and D. Tang, "Android in the zoo: Chain-of-action-thought for GUI agents," 2024. [Online]. Available: https://arxiv.org/abs/2403.02713
[34] W. Li, W. Bishop, A. Li, C. Rawles, F. Campbell-Ajala, D. Tyamagundlu, and O. Riva, "On the effects of data scale on ui control agents," in *Advances in Neural Information Processing Systems*, vol. 37, 2024, pp. 92 130–92 154.
[35] "Telegram: A new era of messaging." https://telegram.org.
[36] "Wechat: Connecting a billion people with calls, chats, and more." https://web.wechat.com.
[37] "Appraven: Apps gone free," https://appraven.net.
[38] "IPATool-py: Download ipa easily," https://github.com/NyaMisty/ipatool-py.
[39] Y. Li, J. He, X. Zhou, Y. Zhang, and J. Baldridge, "Mapping natural language instructions to mobile UI action sequences," *ACL*, 2020.
[40] J.-W. Lin, N. Salehnamadi, and S. Malek, "Test automation in open-source android apps: A large-scale empirical study," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1078–1089.
[41] S. Sunkara, M. Wang, L. Liu, G. Baechler, Y.-C. Hsiao, Jindong, Chen, A. Sharma, and J. Stout, "Towards better semantic understanding of mobile interfaces," 2022. [Online]. Available: https://arxiv.org/abs/2210.02663
[42] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," 2020. [Online]. Available: https://arxiv.org/abs/1905.11946
[43] "PaddleOCR: Awesome multilingual OCR toolkits." https://github.com/PaddlePaddle/PaddleOCR.
[44] W. J. Youden, "Index for rating diagnostic tests," *Cancer*, vol. 3, no. 1, pp. 32–35, 1950.
[45] "Frida: A world-class dynamic instrumentation toolkit," https://frida.re/.
[46] "Appium: Cross-platform automation framework." https://github.com/appium/appium.
[47] "ZXtouch: iOS automation framework," https://github.com/xuan32546/IOS13-SimulateTouch.
[48] "libimobiledevice: A cross-platform foss library to communicate with ios devices natively." https://libimobiledevice.org/.
[49] "Bindiff: Quickly find differences and similarities in disassembled code," https://zynamics.com/bindiff.html.
[50] M. Bailey, D. Dittrich, E. Kenneally, and D. Maughan, "The Menlo Report," *IEEE Security & Privacy*, vol. 10, no. 2, 2012.

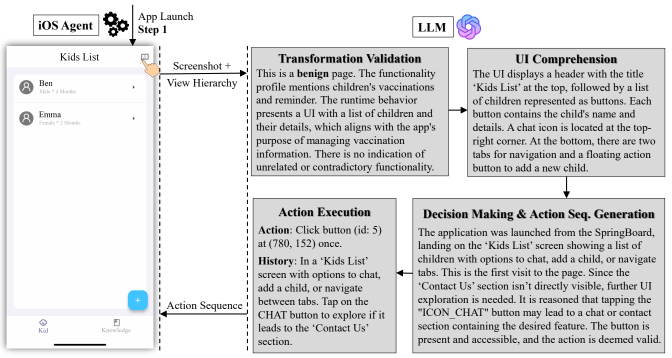
APPENDIX

A. Detection Workflow of CHAMELEOSCAN *on the* `Vaccination Schedule` *App*

The detection methodology of CHAMELEOSCAN is examined through a comprehensive case study of the `Vaccination Schedule` app (Fig. 1), demonstrating the system's standard analytical pipeline.

The process initiates with metadata acquisition (App ID 6747711377), retrieving critical artifacts including app name, category, BundleID, description, and 112 user reviews (as shown in §IV-C ❶). Through *Transformation Method Inference*, the LLM identifies significant patterns - particularly the recurrent token "666" alongside interface cues like "Contact Us", "Feedback", and "Exclamation Mark." Employing contextual prompts with established Chameleon exemplars, the system generates prioritized hypotheses, with the top prediction (H) proposing a transformation sequence "Open the Contact Us dialog → input the magic string '666' → submit." This hypothesis directs the subsequent interface exploration process, structured as four distinct detection cycles (Cycles 1-4).

Initial Verification Cycle. Fig. 11 demonstrates the first validation cycle, initiated through cold launch instrumentation (❸) that subsequently acquires both visual and structural UI representations (❸). The *Transformation Validation* module (❼) performs initial assessment by contrasting the observed "Kids List" interface with declared vaccination scheduling functionality, finding no transformation indicators. Subsequent *UI Comprehension* (❹) extracts semantic components including navigation elements and interactive controls, notably identifying a prominent chat/feedback icon. Guided by hypothesis H, the decision engine (❺) prioritizes chat icon interaction as the most probable verification path, with the *Action Execution* module (❻) executing this minimal-cost transition. This cycle exemplifies the system's core verification methodology, where metadata-derived hypotheses direct interface exploration.



Transformation Validation
This is a **benign** page. The functionality profile mentions children's vaccinations and reminder. The runtime behavior presents a UI with a list of children and their details, which aligns with the app's purpose of managing vaccination information. There is no indication of unrelated or contradictory functionality.

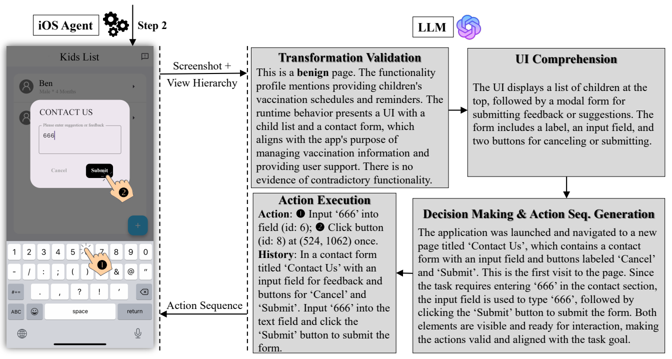
UI Comprehension
The UI displays a header with the title 'Kids List' at the top, followed by a list of children represented as buttons. Each button contains the child's name and details. A chat icon is located at the top-right corner. At the bottom, there are two tabs for navigation and a floating action button to add a new child.

Decision Making & Action Seq. Generation
The application was launched from the SpringBoard, landing on the 'Kids List' screen showing a list of children with options to chat, add a child, or navigate tabs. This is the first visit to the page. Since the 'Contact Us' section isn't directly visible, further UI exploration is needed. It is reasoned that tapping the "ICON_CHAT" button may lead to a chat or contact section containing the desired feature. The button is present and accessible, and the action is deemed valid.

Action Execution
Action: Click button (id: 5) at (780, 152) once.
History: In a 'Kids List' screen with options to chat, add a child, or navigate between tabs. Tap on the CHAT button to explore if it leads to the 'Contact Us' section.

Fig. 11: ChameleoScan's detection pipeline on the `Vaccination Schedule` app (**Cycle 1**).

Secondary Verification Cycle. Fig. 12 depicts the subsequent cycle following chat icon activation. The appearing modal dialog undergoes rigorous validation, confirming absence of transformation indicators as the "Kids List" background interface maintains functional consistency with the declared vaccination scheduling purpose. Then the *UI Comprehension* module precisely identifies interactive components including the text input field and submission controls. The decision engine, constrained by H's requirements, generates a precise two-action sequence: text input followed by submission. This conservative approach exemplifies the system's task-first interaction principle, prioritizing hypothesis validation over exploratory actions that might obscure transformation detection.



Transformation Validation
This is a **benign** page. The functionality profile mentions providing children's vaccination schedules and reminders. The runtime behavior presents a UI with a child list and a contact form, which aligns with the app's purpose of managing vaccination information and providing user support. There is no evidence of contradictory functionality.

UI Comprehension
The UI displays a list of children at the top, followed by a modal form for submitting feedback or suggestions. The form includes a label, an input field, and two buttons for canceling or submitting.

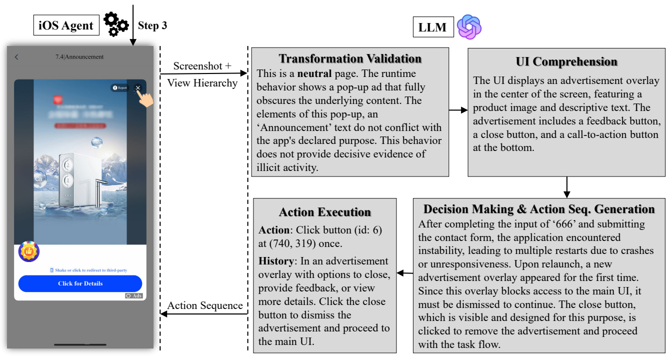
Decision Making & Action Seq. Generation
The application was launched and navigated to a new page titled 'Contact Us', which contains a contact form with an input field and buttons labeled 'Cancel' and 'Submit'. This is the first visit to the page. Since the task requires entering '666' in the contact section, the input field is used to type '666', followed by clicking the 'Submit' button to submit the form. Both elements are visible and ready for interaction, making the actions valid and aligned with the task goal.

Action Execution
Action: ❶ Input '666' into field (id: 6); ❷ Click button (id: 8) at (524, 1062) once.
History: In a contact form titled 'Contact Us' with an input field for feedback and buttons for 'Cancel' and 'Submit'. Input '666' into the text field and click the 'Submit' button to submit the form.

Fig. 12: ChameleoScan's detection pipeline on the `Vaccination Schedule` app (**Cycle 2**).

Third Verification Cycle. Following dialog submission (Fig. 13), the interface exhibits transient instability before displaying a full-screen ad. The *Transformation Validation* module classifies this as non-indicative noise, while *UI Comprehension* accurately detects both the ad frame and its dismissal control ("X"). Applying predefined recovery protocols, the system executes minimal intervention - a single tap on the close button - to restore analyzable interface states, demonstrating its resilience against transient obstructions.

Final Verification Cycle. Post-ad dismissal (Fig. 14), the revealed interface demonstrates fundamental domain divergence: vaccination-related elements are entirely replaced by



Transformation Validation
This is a **neutral** page. The runtime behavior shows a pop-up ad that fully obscures the underlying content. The elements of this pop-up, an 'Announcement' text do not conflict with the app's declared purpose. This behavior does not provide decisive evidence of illicit activity.

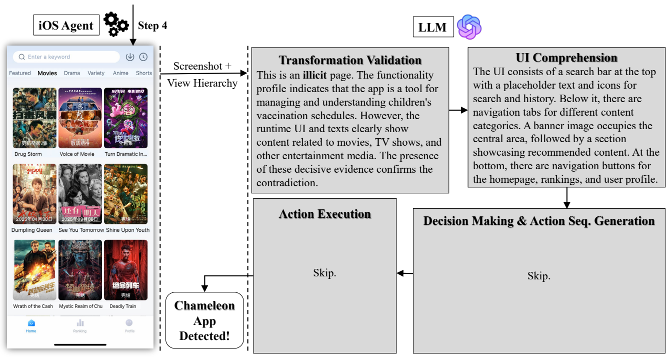
UI Comprehension
The UI displays an advertisement overlay in the center of the screen, featuring a product image and descriptive text. The advertisement includes a feedback button, a close button, and a call-to-action button at the bottom.

Decision Making & Action Seq. Generation
After completing the input of '666' and submitting the contact form, the application encountered instability, leading to multiple restarts due to crashes or unresponsiveness. Upon relaunch, a new advertisement overlay appeared for the first time. Since this overlay blocks access to the main UI, it must be dismissed to continue. The close button, which is visible and designed for this purpose, is clicked to remove the advertisement and proceed with the task flow.

Action Execution
Action: Click button (id: 6) at (740, 319) once.
History: In an advertisement overlay with options to close, provide feedback, or view more details. Click the close button to dismiss the advertisement and proceed to the main UI.

Fig. 13: ChameleoScan's detection pipeline on the `Vaccination Schedule` app (**Cycle 3**).

media streaming components. Through comprehensive semantic alignment analysis, the *Transformation Validation* module quantifies significant functional disparity, conclusively identifying Chameleon behavior. This final assessment completes the detection cycle, with the system terminating upon definitive classification.



Transformation Validation
This is an **illicit** page. The functionality profile indicates that the app is a tool for managing and understanding children's vaccination schedules. However, the runtime UI and texts clearly show content related to movies, TV shows, and other entertainment media. The presence of these decisive evidence confirms the contradiction.

UI Comprehension
The UI consists of a search bar at the top with a placeholder text and icons for search and history. Below it, there are navigation tabs for different content categories. A banner image occupies the central area, followed by a section showcasing recommended content. At the bottom, there are navigation buttons for the homepage, rankings, and user profile.

Decision Making & Action Seq. Generation
Skip.

Action Execution
Skip.

Fig. 14: ChameleoScan's detection pipeline on the `Vaccination Schedule` app (**Cycle 4**).

Summary. ChameleoScan demonstrates a logically coherent workflow from metadata analysis to final classification. Initial pattern analysis generates hypothesis H through few-shot learning, establishing an optimized verification path. Subsequent interface exploration proceeds through methodical validation cycles: targeted navigation to potential transformation points (Cycle 1), precise execution of predicted activation sequences (Cycle 2), resilient handling of interface obstructions (Cycle 3), and conclusive semantic divergence assessment (Cycle 4).

This architecture exemplifies ChameleoScan's core innovation - synergistic integration of predictive metadata analysis, semantic interface understanding, and human-comparable interaction strategies. Each module maintains both local justifiability (via immediate observational evidence) and global consistency (with overarching detection objectives), resulting in a robust, interpretable verification framework for Chameleon apps.

APPENDIX A
ARTIFACT APPENDIX

This appendix describes the artifact associated with this paper.

A. Description & Requirements

The artifact implements CHAMELEOSCAN, a framework for automated detection of iOS Chameleon Apps through LLM-powered semantic UI analysis and dynamic interaction. The system (i) operates real iOS apps on a jailbroken device, (ii) collects screenshots and runtime behavioral data, and (iii) employs LLMs to intelligently execute tasks (i.e., transformation methods) and determine whether an app exhibits "chameleon" behavior (i.e., concealed or prohibited features not disclosed during normal onboarding). The artifact includes all necessary software components to deploy the framework: complete source code, README documentation, automation scripts, trained models, and configuration files.

1) How to access: The full artifact—including source code (with README and LICENSE), trained models, datasets, cached examples, and this appendix—will be archived as a static package on Zenodo (DOI: 10.5281/zenodo.17540091). A GitHub mirror is also provided for convenient access: https://github.com/ChameleoScan.

2) Hardware dependencies: The experimental results reported in the paper were obtained in the environment described below. Reproducing the framework requires a comparable setup. (Note that simply running the framework to inspect cached results does not require physical device interaction.)

- A jailbroken iPhone device running iOS 14.x
- A macOS computer for building WebAgentDriver
- A Windows PC (recommended) for executing the framework

3) Software dependencies: The artifact requires both iPhone-side and Host PC-side software environments:

iPhone environment

- Jailbroken iOS device with Cydia and AppSync (for installing decrypted IPA files)
- OpenSSH, Frida, AFC2 and ZXTouch (formerly IOS13-SimulateTouch) tweaks installed.
- WebAgentDriver deployed as an Appium XCUITest Driver on the device.

Host PC environment (tested on Windows 10/11)

- Runtime: Python 3.11+ and Node.js 22+
- iTunes or Apple Mobile Device Service (for device connection)
- A valid OpenAI API key for GPT-4o access

The iPhone must be connected to the macOS PC during WebAgentDriver compilation, and to the host PC during framework execution.

4) Benchmarks:

- Transformation methods and metadata for 500 known Chameleon apps (`dataset/Dataset_Known.csv`)
- 234 Chameleon apps and 1,644 unknown iOS apps used for evaluation (Due to storage limitations, not all IPA files are included. A subset of IPAs and complete metadata for the unknown dataset are provided in `dataset/IPA` and `dataset/Dataset_Unknown.csv`, respectively. Additional decrypted IPA files are available upon request.)
- Full source code of the framework (`framework/code`)
- A pre-trained EfficientNet-B0 model on the RICO-Semantics dataset (`dataset/model/best_model.pth`)
- Cached examples of Chameleon app detection processes (`framework/code/cached`)

B. Artifact Installation & Configuration

The following steps verify that the archive is complete and interpretable. These do not involve physical device operation.

- Unpack the archive.
- Create a Python 3.11 virtual environment and install packages from `requirements.txt`.
- Run python `benchmarking.py`, which loads data from `cached` folder.
- In the benchmarking UI, open each provided app to inspect:
  - The screenshots and view hierarchies captured
  - Per-screen and per-step analysis generated by LLM
  - Task plans and each action performed
  - App metadata and review summaries (in JSON format)

This procedure confirms: (i) the types of data collected by the framework; (ii) the framework's reasoning process; and (iii) the rationale behind benign versus chameleon classifications. No additional setup is required for this validation.

Readers wishing to conduct live experiments with our benchmarks may follow the procedure below (detailed in `framework/code/README.md`):

[backgroundcolor=gray!10, linecolor=gray!50, roundcorner=4pt] **iPhone device setup**

1. Install Cydia and AppSync, or sign in with an Apple ID.
2. Install OpenSSH, Frida and ZXTouch tweak.
3. Build and deploy WebAgentDriver.
4. Connect the device to the host PC.

Host PC setup (Windows as example)

1. Install Python 3.11+ and Node.js 22+.
2. Install iTunes or Apple Mobile Device Service.
3. Set up a Python Virtual Environment and install `requirements.txt`.
4. Set up Appium by installing dependencies listed in `package.json`.
5. Resolve any tidevice XCUITest startup issues if encountered.
6. Update the default bundle ID in `xcuitest.bat` to match your deployed instance.
7. In `app.py`, set `DEVICE_VERSION`, `DEVICE_UDID` (and optionally `STATUS_BAR_PIXELS` for status bar height) to match your device.
8. Replace `YOUR_API_KEY_HERE` in `gpt_cls.py` with your valid OpenAI API key.
9. Train an EfficientNet-B0 model on the rico-semantics dataset, and save it as `best_model.pth`.

Running the framework

1. Put your IPA file, app store metadata and user reviews into the `cached` directory (see examples).

2. Add target Bundle IDs to `BATCH_TASKS.txt`.

3. Run the batch files `appium.bat`, `tiproxy.bat`, and `xcuitest.bat`, or manually launch these services.

4. Run `app.py` to begin testing.

5. Execute `benchmarking.py` at any time to review results.

C. Experiment Workflow

For readers conducting live experiments, CHAMELEOSCAN automatically executes the following workflow to detect and analyze potential Chameleon behaviors:

1) **App Setup and Metadata Analysis.** The system installs the target app on an instrumented iOS device, analyzes its metadata, and uses LLMs to construct a functional profile of the app.
2) **Transformation Method Prediction.** Using few-shot prompts constructed from known Chameleon exemplars, the framework infers possible transformation methods for experimental validation.
3) **Dynamic Execution.** The agent sequentially executes each predicted method. Throughout execution, the system records screenshots, view hierarchies, UI element recognition outputs, performed actions, and Chameleon validation outcomes for every UI state.
4) **Result Output.** The process concludes when a detection condition is satisfied (e.g., positive classification or reaching the execution round limit). The framework outputs the final Chameleon detection decision alongside supporting evidence.

This workflow generates a structured detection record for each analyzed app, comprising the predicted transformation methods, intermediate UI comprehension results, executed actions, and the final classification.