DEEPEXPLOITOR: LLM-Enhanced Automated Exploitation of DeepLink Attack in Hybrid Apps

Zhangyue Zhang[1], Lei Zhang[2], Zhibo Zhang[2], Yongheng Liu[1], Zhemin Yang[2], Yuan Zhang[2], and Min Yang[2]
1: Fudan University, {zhangyuezhang23, yhliu24}@m.fudan.edu.cn
2: Fudan University, {zxl, zhibozhang19, yangzhemin, yuanxzhang, m_yang}@fudan.edu.cn

***Abstract*—Modern mobile apps widely embed WebView to enable rich and dynamic content, making it an increasingly attractive target for attackers. It is well known that insufficient or improper input validation on WebView-loaded URLs can compromise the entire app or even the underlying system. Among these threats, one of the most critical attack vectors is the DeepLink Attack, which often requires only a single user click to exploit WebView vulnerabilities. Despite the deployment of defense such as URL allowlists, misconfigurations and inconsistent implementations continue to expose apps to exploitation.**

In this paper, we present DEEPEXPLOITOR, the first automated exploit generation framework targeting vulnerabilities exploitable via DeepLink Attack. DEEPEXPLOITOR addresses two key challenges: First, it statically models complex, app-specific routing encapsulation and customized input parsing logic by extracing constraint-related code and resolving them through large language models (LLMs), enabling scalable discovery of valid exploits. Second, it identifies and mutates trusted domains embedded in the app to bypass black-box defenses such as domain-based allowlists. We evaluated DEEPEXPLOITOR on 433 of the most popular Android apps and uncovered 83 zero-day vulnerabilities, including 24 rated as high or critical severity. All findings were responsibly disclosed to affected vendors, with 35 acknowledged to date or assigned CVE/CNVD identifiers.

***Index Terms*—Android, WebView, Deep Link, Automated Exploit Generation, Large Language Model**

I. INTRODUCTION

Nowadays, modern mobile apps prefer to provide rich functionalities through embedded WebView components (e.g., Android's WebView [1] and iOS's WKWebView [2]), gaining benefits from both cross-platform development and native-like interaction experience. However, WebView is becoming a prime target for attackers due to various vulnerabilities, i.e., executing untrusted JavaScript code and exposing native APIs registered by the app [3]–[7]. One critical and stealthy attack often requires only one click from the victim to exploit WebView vulnerabilities and launch phishing or even remote code execution, which we refer to as the DeepLink Attack. For example, TikTok once suffered from such a vulnerability, where a user's account could be hijacked with just one click on a malicious deep link [8]. Considering that over 85% popular apps use WebView [9], this attack poses serious risks to millions of mobile users.

Although defenses such as URL allowlist check [10] have been proposed to prevent such attacks, the occurrence of exploits has never stopped [8], [11], [12]. The reason is that the effectiveness of these defenses is often compromised by insecure allowlist configurations or poor implementation of URL checks. However, modern hybrid apps often adopt app-specific designs like router-based encapsulation and customized parsing logic. While these features improve modularity and flexibility, they introduce implicit control flows and data flows, complicating identification and validation of security checks. These make automated exploitation of DeepLink Attack inherently challenging.

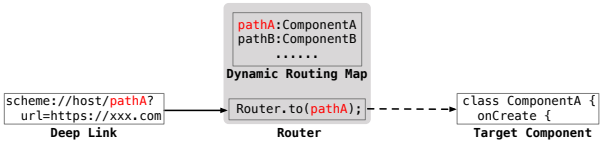

Fig. 1. An example of router-based encapsulation for handling deep link.

To the best of our knowledge, no existing work supports automated end-to-end exploit generation for DeepLink Attack. Exploiting such attack requires crafting both a valid deep link that reaches WebView and a sensitive API invocation. Existing efforts focus on isolated parts of this process. Some proposed heuristic-based fuzzing to construct deep links [6], while others relied on templates or manual modeling to infer API invocations [7], [13], [14]. However, these ad hoc strategies are ineffective in modern hybrid apps due to their app-specific designs. For example, exploiting APIs encapsulated by router often requires resolving the target method name from dynamically constructed routing mapping. These mechanisms introduce implicit flows that obscure the relationship between inputs and API targets. Prior approaches either fail to handle such encapsulation or rely heavily on manual effort and expertise, resulting in poor scalability. Thus, a new approach is needed to drive automated exploit generation in modern hybrid apps, posing the following three challenges:

First, modern apps commonly[1] adopt router-based encapsulation to enhance development flexibility; however, this hides the exploitation targets (i.e., WebView component and APIs), making them difficult to trigger (**C1**). These apps receive external inputs and determine the navigation target using dynamically constructed routing maps. Figure 1 illustrates such a routing mechanism, where the deep link path is resolved through a runtime map and indirectly routed to the target component. As router implementation varies across different

[1]According to our preliminary study in §II, 45% of apps use routers for inter-component navigation, and 60% for WebView API encapsulation, with 75% adopting at least one router-based mechanism.

apps, traditional static analysis cannot directly resolve mapping relation and determine navigation targets. Previous works rely on manually modeling the customized routers of a few apps [7], [15], but these modeling results are not applicable to a wider range of apps.

Second, attacker-controlled inputs are often parsed by customized methods in the target app, which increases the difficulty of generating well-structured exploits (**C2**). Specifically, these exploits are constrained by the expected object types defined in the parsing logic, resulting in nested structures with diverse field values at each layer. Due to the lack of standardized structure in exploits, traditional methods struggle to model them comprehensively. Existing works (i.e., XAWI [6], BridgeScope [13], and APIScope [14]) construct such exploits using heuristic information or templates. However, the heuristic cues do not align with well-structured exploits, nor do the templates fully capture the structural variations. As a result, these approaches cannot effectively construct valid exploits.

Third, the customized defenses are typically diversified and black box, which increase the difficulty of generating realistic exploits (**C3**). A real exploit requires successfully bypassing these defenses, which in turn demands accurate inference of the app's allowlists and the reconstruction of evasive URLs. However, these allowlists are often dynamically loaded at runtime or enforced on the server side, making them inaccessible to the attacker. Prior works (XAWI [6], MEDUSA [7]) only perform template-based URL mutation to bypass URL checks, without considering the necessity of extracting the allowlisted domain. In the absence of a domain allowlist, their mutation strategies struggle to generate URLs capable of bypassing URL checks.

To address these challenges, we present *the first* AEG (Automated Exploit Generation) framework for DeepLink Attack, called DEEPEXPLOITOR (DeepLink Attack Exploit Generator). This framework is based on two key insights. On the one hand, DEEPEXPLOITOR treats router-based encapsulation and customized parsing logic as a constraint reasoning task. It extracts precise dependencies from routing and parsing logic to capture how requests shape the execution path. Then, it uses LLMs to synthesize valid exploits across diverse app designs without relying on fixed templates or heuristics. On the other hand, DEEPEXPLOITOR infers allowlists by identifying hard-coded domains that appear in various functional components of the app, including WebView configurations, intent-filters, and network request destinations, under the observation that allowlisted domains are also used in normal scenarios.

We have designed and implemented DEEPEXPLOITOR and conducted extensive evaluations. We collected 433 most popular apps from two leading Android app stores, Huawei AppGallery [16] and Google Play [17], for evaluation. Our evaluation shows that DEEPEXPLOITOR discovers 83 vulnerable apps, of which exploits were generated for all of them. We verified and responsibly reported all of these vulnerabilities to the corresponding app vendors, and so far 35 of them were acknowledged or assigned CVE/CNVDs. Among these, 24 vulnerabilities were classified as high risk or severe (or with equivalent severity). We combined the state-of-the-art approaches, i.e., XAWI [6] for generating deep links, and BridgeScope [13] and APIScope [14] for generating API invocation. Our evaluation results show that these tools are not able to generate any of the actual exploits.

II. BACKGROUND

In this section, we present a brief survey study of modern Android apps to provide some background knowledge on the router-based encapsulation and customized security check that may exists on the DeepLink Attack path. These findings reveal the wide diversity of WebView-powered features in modern hybrid apps. Motivated by the generalization ability of LLMs, we could first extract each app's custom constraints and then leverage the LLM to synthesize valid exploits at scale.

Methodology. Our methodology contains three semi-automatic steps. First, we manually download a small yet diverse set of widely used apps from Google Play and Huawei AppGallery. These apps are the top 5 most downloaded apps from 10 categories (e.g., news, shopping). Second, we automatically analyze these apps for the usage of WebViews and APIs. For apps meeting this criterion, we then manually determine whether they expose deep links that navigate to WebView, yielding 20 apps. Lastly, we reverse engineer these apps to analyze their encapsulation and countermeasures.

Router-Based Encapsulation. Modern apps often adopt two types of router, i.e., intent router and API router to perform inter-component navigation and API request navigation. These router hides sensitive components or methods, requires structured and expected input messages, significantly increasing the complexity of generating exploits.

- Intent router. This router handles an Intent message, e.g., a deep link URL, and uses a string to Activity class map to choose which Activity should be started. We found that 9/20 apps adopt such dynamic router rather than conducting the direct inter-component navigation. Note that, some of these routers are open sourced with 14.5k stars, indicating a wild adoption or imitation among apps.
- API router. This router handles API invocation message, e.g., a JSON object contains target API name and parameter values, and uses a string to Java method map to choose which API should be invoked. Note that, such invocation messages are coming from exposed WebView APIs (e.g., method added by addJavaScriptInterface [18]).

Customized Defenses. Modern apps commonly implement two types of defenses against DeepLink Attack—noLoad and noBridge. These strategies are consistent with directions proposed in prior research [19], making them representative of current defense practices.

- NoLoad (14/20). This strategy performs dynamic URL allowlist checks before loading URLs into WebView to prevent untrusted code loading. We find the allowlists can be either hard-coded in apps (4), or dynamically loaded or checked from back-end server (10).

- NoBridge (8/20). This strategy checks if the API caller is privileged, protecting sensitive APIs (e.g., `getToken`) from attacker access. This check can also be performed in back-end server.

However, these defense implementations remain ad hoc and lack standardization, rendering security checks in real-world apps fragile and often flawed, leading to exploitable vulnerabilities. But it is difficult for an adversary to extract the allowlist in a uniform manner, which is the basic data for the url mutation and bypassing check.

Beyond the mechanism itself, the security of the allowlist also depends on the trustworthiness of the domain assets it includes. However, the security of these assets is determined by the website developers, beyond the control of app developers. An attacker can exploit the design flaw created when app developers incorrectly place their security trust in website developers. By investigating the top OWASP web vulnerabilities [20], we identified 4 types of vulnerabilities that can be used for bypassing URL check, including: XSS, Open redirection, subdomain takeover and expired domain.

III. Overview

In this section, we demonstrate a real-world vulnerability discovered by DeepExploitor as a motivating example, and illustrate challenges of generating exploit. After that, we describe how DeepExploitor solves these challenges. Finally, we present our threat model and research scope.

A. Motivating Example

We illustrate a motivating example of exploiting *J******G*, a famous e-Commerce app with more than 580 million users [21], to describe the challenges of automated DeepLink Attack exploit generation. J******G employs WebView for product display, registering rich APIs to support native-like interactions, such as manipulating file system. However, it has a vulnerable URL check in the NoLoad defense and lacks a NoBridge defense, which eventually leads to DeepLink Attack. As a result, a crafted malicious URL can be executed within the WebView, enabling attackers to abuse its sensitive APIs, such as deleting files on the victim's device, ultimately leading to a denial-of-service.

Figure 2 shows the vulnerable code and our generated exploits. As shown in the top-left corner of the figure, the exploit begins with a crafted deep link containing a malicious `url` designed to bypass URL check. A remote attacker can send this URI through the Internet to lure users into clicking it, thereby triggering the attack. Line 19 in the `onCreate` function shows the vulnerable URL check, which relies on a back-end service (`safeCheckURL` at Line 21) to validate whether the domain ends with ".xd.com". However, this check can be bypassed by formatting the `url` as "evil.com://xd.com", which results in the attacker-controlled "evil.com" being loaded in the WebView. After loading this URL, a payload contains exploit code that interacts with sensitive API is executed in the WebView, as shown in the top-right corner of the figure. This leads to the successful invocation of `deleteFile` at Line 38. This exploit presents three distinct challenging tasks:

Challenging Task I: Constructing well-structured deep link. To reach the vulnerable WebView, the attacker must construct a deep link that satisfies app's Intent router (step ❶). In Line 16-17, this involves resolving non-standard navigation via `JXRouter`, which requires identifying the correct `"des"` field value that leads to `WebActivity`. Additionally, the deep link must satisfy intermediate logic conditions, such as setting the category field to `"jump"` in order to trigger `execJump` (Line 9-11). These constraints are derived through multiple layers of parameter extraction (e.g., `getQueryParameter`, `getString`), making the legal exploit hard to infer (**C1** and **C2**). Existing tool XAWI [6] relies on templates and string heuristics, which is insufficient here due to app's customized exploit and logic.

Challenging Task II: Bypassing the black-box security mechanism. To load a malicious URL in WebView, the exploit must bypass app's server-side URL check (step ❷). However, the domain allowlist used for validation is absent from the codebase and is checked remotely via https://un.m.xd.com/safeCheck?to=. This redirection only occurs when the back-end deems the URL safe. Such a black-box design hides the allowlist, which prevents the construction of effective bypass customized defenses (**C3**). Existing approaches like XAWI [6] and MEDUSA [7] use predefined mutation templates, but without access to the actual allowlist, they fail to generate valid URLs that can pass the check.

Challenging Task III: Constructing well-structured API invocations. Exploiting sensitive APIs requires invoking undocumented APIs that involve indirect invocation chains and nested parameter constraints. In step ❸, the adversary must invoke the `callNative` API, which uses a centralized API router to dispatch invocations to methods. The API expects a JSON payload where the `"routerURL"` field specifies the method alias in the format router://[class]/[method] (**C1**), enabling the router to resolve and execute the corresponding native method (Line 36). Furthermore, the payload must satisfy nested structural constraints (**C2**). For example, invoking `deleteFile` requires a `"routerParam"` field containing `"arg"`, which must include `"savePath"` as an array indicating file paths. Previous works like BridgeScope [13] and APIScope [14] rely on templates or heuristics, but fails to handle such indirect mappings and to generate legal invocations.

B. Solution Overview

We present an overview of DeepExploitor using the motivating example in Figure 2. At a high level, DeepExploitor begins by tracing the creation of the router map (i.e., JXRouter at Line 17 and API router at Line 36) to extract registered key-value pairs used for inter-component/API navigation. It then analyzes data-flow paths from external inputs to sensitive operations, such as WebView.loadUrl and native API invocations, to collect conditional statements that enforce structure and semantic constraints. These constraints

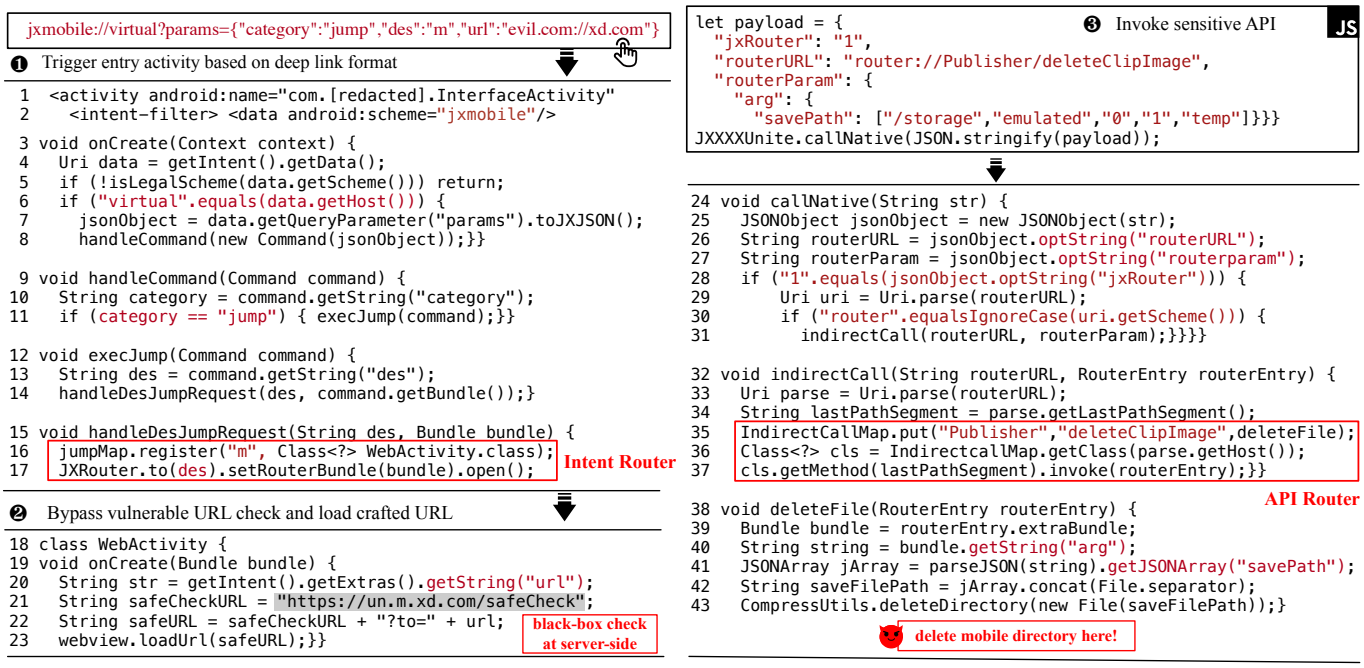

Fig. 2. The vulnerable code and exploit of J******G. The code is simplified to facilitate presentation and protect privacy of the vendor, e.g., removing redundant code and redacting sensitive identifiers.

are passed to LLM, which assists in solving them and synthesizing valid input values. Finally, DEEPEXPLOITOR leverages these values to construct deep link-based exploits capable of bypassing security checks and invoking sensitive API.

We now present how DEEPEXPLOITOR addresses the aforementioned challenging tasks. To address Challenging Task I, DEEPEXPLOITOR performs taint analysis on the code paths that handle deep links. Upon locating the JXRouter component, it extracts the mapping between the parameter `"m"` and the target `WebActivity` from Line 16, and propagates the control flow between Lines 17 and 19 for further analysis. Next, to precisely manage inter-component control and data dependencies, DEEPEXPLOITOR constructs an Inter-component Constraint Propagation Graph (ICPG), which maintains all conditional statements that influence the validity of deep link formats. For example, the tainted variable host is obtained through getHost() in Line 6. When the condition equals("virtual") is detected in Line 6, DEEPEXPLOITOR adds the corresponding onCreate() method node to the ICPG. After that, DEEPEXPLOITOR converts the ICPG into a prompt for the LLM. The LLM is used to solve the extracted exploit constraints and generate logically valid deep link exploits.

To address Challenging Task II, DEEPEXPLOITOR first extracts domain strings from the app (e.g., constant URLs like xd.com loaded into WebView), and uses them to infer a candidate allowlist. Then it applies predefined URL-mutation templates to these domains and instruments the WebView event handlers (e.g., `onPageStarted`) to observe which mutations successfully pass the server-side check. In our motivating example, DEEPEXPLOITOR bypasses URL validation by constructing the URI evil.com://xd.com, allowing the attacker-controlled domain to be loaded into WebView.

To address Challenging Task III, DEEPEXPLOITOR begins by loading an inferred allowlisted URL (e.g., https://xd.com /mall) into the WebView and instruments the `callNative` API to capture a real invocation sample. Next, it resolves the API router by identifying the container objects (such as Java Map) used to dispatch API invocations. An LLM assists in recognizing the complete API pool (the IndirectCallMap at Line 35) and identifying sensitive methods for exploitation (i.e., `deleteFile`). DEEPEXPLOITOR then constructs an ICPG by extracting all conditional statements related to API invocation (for example, parsing `"routerURL"` via `optString` at Line 26). It feeds both the ICPG and the captured invocation sample into the LLM to generate the invocation code of `deleteFile`. Finally, by instrumenting API callbacks (e.g., `evaluateJavascript`), DEEPEXPLOITOR captures the execution result and uses the LLM to interpret whether the execution is successful.

C. Threat model

A DeepLink Attack occurs when a victim clicks a malicious deep link, often embedded in a website or message. The link is crafted to launch a target app and deliver a malicious URL as a parameter. If the app's security checks are insufficient or misconfigured, this URL may bypass validation and be loaded into the WebView component. Once loaded, the attacker's webpage gains access to WebView-exposed APIs—without requiring any device permissions. By reverse engineering the app, the attacker can identify sensitive APIs and the constraints on their invocation. These APIs can then be silently triggered via

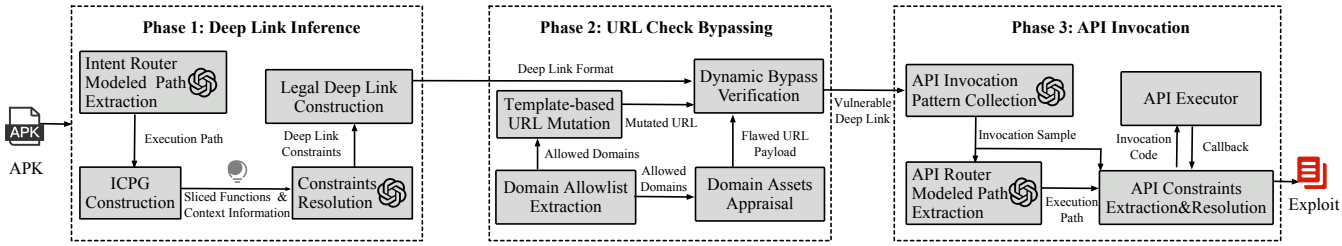

Fig. 3. The Overall Architecture of DEEPEXPLOITOR.

malicious JavaScript, leading to a wide range of consequences, from unauthorized data access to full system compromise.

IV. APPROACH

In this section, we first present the architecture of DEEPEXPLOITOR. Then, we describe three phases respectively.

A. System Architecture

DEEPEXPLOITOR takes an Android app as input and outputs a verified exploit chain that abuses a deep link to invoke sensitive native APIs via WebView. Figure 3 presents the overall architecture of DEEPEXPLOITOR, which consists of three key phases. In Deep Link Inference phase, DEEPEXPLOITOR identifies valid deep link formats that navigate to WebView. In URL Check Bypassing phase, DEEPEXPLOITOR focuses on bypassing noLoad enforcement. In API Invocation phase, DEEPEXPLOITOR generates concrete API invocations to exploit the API exposed in WebView.

B. Phase1: Deep Link Inference

DEEPEXPLOITOR constructs valid deep links to reach WebView in following four steps:

Intent Router Modeled Path Extraction. DEEPEXPLOITOR first reconstructs the customized intent router by identifying instructions related to route registration. By preliminary reverse engineering, we find that most apps use clustered registration calls to map URL paths to activities. We set a clustering threshold of 20 to effectively identify these routing patterns across all evaluated apps while minimizing false positives. For each detected cluster, a representative sample is analyzed with the help of an LLM to isolate relevant instructions and extract the parameters that define routing paths and activity targets.

After modeling the intent router, DEEPEXPLOITOR performs taint analysis to identify execution paths triggered by incoming deep links. DEEPEXPLOITOR identifies activities that declare intent-filters, and uses their lifecycle methods (e.g., `onCreate()`) as entry points for taint analysis. Then DEEPEXPLOITOR employs a flow-sensitive, context-sensitive, and field-sensitive analysis to track how tainted data propagate from deep link parameters to WebView loading operations. Specifically, it marks values derived from `getIntent()` as taint sources, and treats all `loadUrl()` invocations on both the system and custom WebView classes as sinks. During analysis, when inter-component navigation driven by intent routers is encountered, DEEPEXPLOITOR resolves the target component by consulting the reconstructed routing map. For

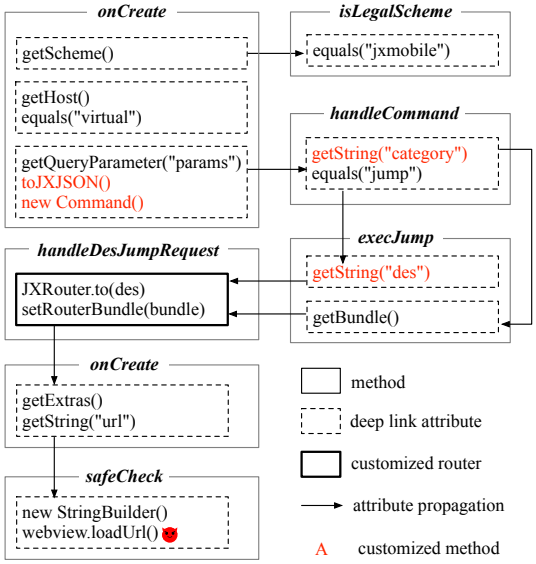

Fig. 4. The simplified ICPG of our motivating example for deep link constraints inference.

conventional navigation patterns (e.g., `startActivity()`), it runs a lightweight data flow analysis on the passed `Intent` object to determine the target. When navigation targets cannot be precisely resolved (e.g., multiple candidate destinations), DEEPEXPLOITOR falls back to naming heuristics and prioritizes components with names containing keywords such as `"web"` or `"h5"`, which are empirically associated with WebView functionality. This strategy improves efficiency by reducing unnecessary propagation into unrelated components.

Inter-component Constraints Propagation Graph Construction. DEEPEXPLOITOR builds an ICPG to capture constraints and contextual information related to deep link parameters. Along previously identified execution paths, it records how tainted values are constrained by program logic. For example, an invocation like `startsWith()` indicates a prefix constraint on a specific deep link component. Methods containing such constraint-related expressions are added into the ICPG. Note that, this taint analysis is also field-sensitive, which traces tainted values stored in object fields and propagated across methods via field access. The ICPG captures how deep link components are validated across multiple methods and enables accurate constraint resolution in the next step.

Deep Link Constraints Resolution. For each ICPG method that involves tainted values, DEEPEXPLOITOR decompiles

Task
You are given a Java code that handles an incoming deep link and eventually navigates to a WebView.
Infer ONLY constraints provable from the code for:scheme, host, path, and query.

Inputs
Decompiled code: <CODE>
Taint to deep link conponent mapping(immediately following the code): <MAPPING>

Reasoning
1. Associate each tainted value with its deep link component via invocation(e.g., getHost()).
2. Find conditional checks applied to tainted values (e.g., equals) and translate them into constraints on the mapped deep link component.
3. If not provable, leave the field empty (see contract below).
4. Record the query key whose value flows into WebView.loadUrl, and any query key–value constraints enforced by the code.

Output (JSON only; micro-contract)
- scheme/host/path: arrays of {t: "equals|prefix|suffix", v: "<str>"}; unknown = [].
- query: { required: [<key>], constraints: { <key>: {t: "equals|url|enum|prefix|suffix", v|vs: ...} } }.
- url_param_key: string; unknown = "".
- No extra keys. No prose.

Fig. 5. Prompt templates of resolving deep link constraints. For space reasons, the figure presents a condensed schema.

it into Java code through JADX [22], exposing high-level semantics and easing LLM reasoning. To make the LLM explicitly link each tainted variable to its deep link component, DEEPEXPLOITOR attaches a mapping before each method. Specifically, it lists the method's tainted parameters/fields, and indicates their corresponding deep link components via the invocation sequence (e.g., param str: [getIntent(), getHost()]). We then construct a prompt that includes the decompiled code and this mapping as inputs. The prompt requests CoT reasoning to lower the model's reasoning burden and improve consistency. The output format is constrained by a JSON Schema [23] (Structured Outputs [24] when supported). A prompt example is shown in Figure 5 with a condensed schema. To curb hallucinations, the prompt requires code-grounded evidence by quoting invocations from the shown code when forming constraints. For robustness, DEEPEXPLOITOR validates every output against the schema and performs diagnostic retries with validator feedback by appending error to the prompt (e.g., missing field); after bounded retries, non-conformant outputs are recorded with empty fields.

Legal Deep Link Generation. DEEPEXPLOITOR combines the inferred component values to generate candidate deep links. These links are dynamically triggered via `adb` [25] commands to simulate user clicks and test whether they are handled by the app. To monitor the loading behavior, DEEPEXPLOITOR instruments the WebView event handlers (i.e., `onPageStarted`, `onLoadResource`, and `onPageFinished`) to determine whether the URLs are successfully loaded. Based on these observations, DEEPEXPLOITOR identifies deep links that load URLs into WebView.

C. Phase2: URL Check Bypassing

DEEPEXPLOITOR mutates deep links to bypass potential URL checks in following three steps:

Domain Allowlist Extraction. We design an algorithm to identify domains configured in the allowlist of an app. Specifically, DEEPEXPLOITOR performs data flow analysis on the source code of a given app. As illustrated in algorithm 1, DEEPEXPLOITOR first identifies all WebView load statements (e.g., invocations to `loadUrl`) in the app, backtraces each loadpoint to its URL source, extracts the domain if the URL is statically resolvable, and accumulates them into set S_{dom} (Lines 8–11). Next, DEEPEXPLOITOR parses AndroidManifest.xml to collect every android:host declared under `<intent-filter>` entries, and merges these domains into S_{dom} as well (Lines 12–13). Then, it scans the app code for network-request invocations (e.g., `OkHttpClient`), extracts any string-literal URLs, derives their domains, and adds them to S_{dom} (Lines 14–17). Note that DEEPEXPLOITOR intentionally extracts only hard-coded domains and does not attempt to recover runtime-formed strings, which provides a lightweight basis for allowlist extraction. Furthermore, since domain matching is often broad, e.g., using the `endswith()` API, DEEPEXPLOITOR extend the S_{dom} to E_{dom} (Line 12-15) through subdomain finder. Through this process, DEEPEXPLOITOR yields a comprehensive and accurate approximation of the domains that might be included in the allowlist.

Algorithm 1: Domain allowlist inference

```
   Input: hybrid app source code s
   Output: Inferred allowlist domains E_dom
1  Initialize S_dom, E_dom as empty sets;
2  for each WebView instance in s do
3      loadPoints ← collectLoadPoints(WebView instance);
4      for each point in loadPoints do
5          url ← backtraceURL(point);
6          S_dom ← S_dom ∪ {extractDomain(url)};
7      end
8  end
9  manifestDomains ← parseIntentFilter(AndroidManifest.xml);
10 for each networkRequest in collectNetworkRequests(s) do
11     urlLiteral ← extractLiteralURL(networkRequest);
12     if urlLiteral is not empty then
13         S_dom ← S_dom ∪ {extractDomain(urlLiteral)};
14     end
15 end
16 S_dom ← S_dom∪manifestDomains;
17 for each domain in S_dom do
18     subdomains ← subdomain-finder(domain) ;
19     E_dom ← E_dom ∪ subdomains;
20 end
21 return E_dom
```

Deep Link Mutation. After inferring allowlisted domains, DEEPEXPLOITOR performs template-based URL mutation and allowlisted domain assets appraisal to bypass URL check and load crafted URL in WebView. Specifically, we collect a template for URL mutation by collecting and analyzing academic and non-academic resources [7], [26]–[28]. Then DEEPEXPLOITOR performs template-based mutations on the allowlisted domains. In addition, DEEPEXPLOITOR appraises allowlisted domain assets to identify vulnerabilities that enable code injection, as discussed in §II. Specifically, DEEPEXPLOITOR leverages several popular web vulnerability detection tools to scan these type of vulnerabilities, and constructs cor-

responding URLs that exploit the discovered vulnerabilities.

Dynamic Bypass Verification. Finally, DEEPEXPLOITOR verifies whether the mutated URLs can bypass security checks at runtime. It instruments WebView event handlers function and monitors their invocation during URL loading to observe whether mutated URLs are successfully loaded. Then, DEEPEXPLOITOR embeds these URLs into syntactically valid deep links and dynamically triggers them to confirm the existence of exploitable deep links.

D. Phase3: API Invocation

DEEPEXPLOITOR generates exploit code to invoke sensitive APIs in following four steps:

API Invocation Pattern Collection. To identify valid API invocation, DEEPEXPLOITOR collects real-world invocation samples through dynamic analysis. Some allowlisted URLs invoke APIs for logging or analytics, and these invocations embed valuable parameter structures that guide constraint resolution. DEEPEXPLOITOR loads such URLs in WebView and instruments the `addJavascriptInterface` method to collect class and method names of registered APIs. Based on the extracted class and method names, DEEPEXPLOITOR crawls web pages under allowlisted domains and performs keyword matching to locate URLs potentially triggering API calls. Next, DEEPEXPLOITOR instruments these APIs and dynamically loads the identified URLs in WebView to collect API invocation traces. If crawler does not locate potential API invocations, DEEPEXPLOITOR still tries collecting samples by loading allowlisted domains. These traces support subsequent constraint resolution. If no such samples are found, DEEPEXPLOITOR falls back to program analysis and LLM inference to synthesize valid API invocations.

API Router Modeled path Extraction. Modern hybrid apps often encapsulate WebView-exposed APIs behind a centralized API router. To resolve these API routers, DEEPEXPLOITOR analyzes the underlying routing logic and extracts the full API router map using a hybrid approach. First, DEEPEXPLOITOR statically analyzes control flow to locate container objects (e.g., Map) to extract candidate API router maps. These containers are then instrumented, and DEEPEXPLOITOR trigger previously verified deep link to retrieve their runtime contents. To distinguish actual API router maps from unrelated containers, DEEPEXPLOITOR uses LLM-based analysis. Specifically, because candidate API pools exhibit patterns (e.g., meaningful method names), DEEPEXPLOITOR slices these candidates and submits them to the LLM to identify true router maps.

Once the method pool is extracted, DEEPEXPLOITOR proceeds to identify sensitive targets for exploitation. Since some apps perform noBridge security checks, e.g., validating the current URL before API execution, targeting security-critical methods allows DEEPEXPLOITOR to assess whether such defenses are bypassed. To reduce analysis overhead, DEEPEXPLOITOR maintains a curated list of sensitive method names (e.g., `installApp`, `getUserInfo`) compiled via manual inspection guided by prior work [7]. DEEPEXPLOITOR filters candidate methods via name matching, and then confirms their reachability through control flow analysis.

API Parameters Constraints Collection and Resolution. To construct valid API invocations, DEEPEXPLOITOR resolves input constraints with the similar LLM workflow used for deep link constraints. DEEPEXPLOITOR builds an ICPG that models how parameters are parsed and constrained. The ICPG, together with a compact mapping from tainted parameters/-fields to their sources, is provided to the LLM as the prompt input. In the reasoning stage DEEPEXPLOITOR additionally provides an API invocation sample to anchor inference, and the prompt explicitly requires the model to output a concrete API invocation in schema-constrained JSON. This example-guided formulation reduces the complexity of reasoning and improves the accuracy of generation, as corroborated by our ablation study in §VI-C1.

However, some samples contain verbose parameters that may mislead the LLM. To mitigate this, DEEPEXPLOITOR prunes irrelevant arguments based on two heuristics: (1) string values exceeding a predefined length threshold (i.e., 30 characters), and (2) keys commonly used in logging or analytics (e.g., `pageId`, `deviceInfo`). This preprocessing removes non-essential content while preserving structural constraints, allowing the LLM to focus on inferring meaningful dependencies and generating valid invocation code.

Invocation Code Execution and Verification. Finally, DEEPEXPLOITOR embeds the generated JavaScript code into an HTML page via a `<script>` block. The page is then deployed to a controlled server and loaded into the target app's WebView via a vulnerable deep link. Upon execution, the `evaluateJavaScript` callback returns the execution result. DEEPEXPLOITOR instruments this callback to collect runtime feedback, which is then analyzed by the LLM to verify whether the invocation succeeded or failed due to invalid parameters. If errors occur, the callback often reveals diagnostic messages (e.g., "error: invalid parameters"), which are used to refine constraints and regenerate corrected payloads. The refinement process enables DEEPEXPLOITOR to iteratively correct invocation errors, and once a sensitive API is successfully executed, it confirms that all runtime defenses are bypassed, signifying a complete DeepLink-based exploit.

V. IMPLEMENTATION

We implemented DEEPEXPLOITOR with more than 10,000 lines of new Java code and several Python code, which are available in an repository[2]. Specifically, we implement taint analysis module based on Soot [29]. We also perform apk decoding using Apktool [30] during static analysis. We decompile bytecode to Java using JADX [22]. To evaluate allowlist domain assets, we use Xray [31] for detecting websites vulnerable to XSS and open redirect, Aquatone [32] for identifying domains vulnerable to subdomain takeovers, and whois [33] for detecting expired domains. In API invocation, we utilize Frida [34] to perform instrumentation. When using

[2]https://github.com/yatooimh/DeepExploitor

TABLE I
VULNERABILITY DETAILS OF 15 POPULAR HYBRID APPS.

Description: Considering ethical reasons, we withhold the names of real-world apps.

#	Application	Category	Downloads	Version	URL Payload	API Consequence	Status
1	OPPO App*****	Tools&Utilities	600M+	11.20.0	http.evil.com://safe.com	Malicious app installation silently	Confirmed
2	t**p.com	Travel&Local	10M+	7.96.2	open redirection	Remote code execution	Confirmed
3	M**tuan	Lifestyle	25.2B+	12.8.404	safe.com(expired domain)	Steal phone number, token	Confirmed
4	D**u	Lifestyle	5.1B+	5.29.0	javascript://	Phishing to steal user credentials	CNVD Assgined
5	X***hongshu	Lifestyle	24.2B+	8.19.0	open redirection	Phishing to steal user credentials	Confirmed
6	T**ll	Shopping	4.8B+	15.34.2	evil.com://safe.com	Manipulate sensitive information	Confirmed
7	I**eFish	Shopping	9B+	7.16.20	evil.com\@safe.com	Manipulate sensitive information	Confirmed
8	C**na Unicom	Comminucate	2.8B+	10.7.1	evil.com	Arbitrary theft user device files	CNVD Assigned
9	Y**ku	Entertainment	13.7B+	11.0.92	safe.com.evil.com	Manipulate sensitive information	Confirmed
10	T**piaopiao	Movies&TV	200M+	11.9.6	evil.com\@safe.com	Manipulate courier logistics information	Confirmed
11	D***ping	Lifestyle	10.2B+	11.0.13	%0ajavascript://safe.com	Steal phone number, token	Confirmed
12	L***da	Shopping	500M+	7.51.0	evil.com://safe.com	Phishing to steal user credentials	Confirmed
13	H**wei Health	Health&Fitness	30.7B+	14.0.10	open redirection	Manipulate user token	CNVD Assigned
14	G**ek	Travel&Local	100M+	4.80.4	https:/safe.com?r=	Steal account name, token	CVE Assigned
15	G**de Maps	Map&Navigate	16.9B+	13.19.1	javascript://	Phishing to steal user credentials	Confirmed

LLMs, we use Ollama [35] to run local models downloaded from Hugging Face [36]. For commercial LLM services, we invoke OpenAI [37] and DeepSeek [38] APIs.

VI. EVALUATION

We evaluated DEEPEXPLOITOR with the following research questions:

RQ1: *How effective is* DEEPEXPLOITOR *in detecting and exploiting zero-day vulnerabilities in real-world apps?*

RQ2: *Compared with previous works, by how much has* DEEPEXPLOITOR *improved its capability in exploiting DeepLink Attack?*

RQ3: *How does each component of* DEEPEXPLOITOR *contribute to its overall performance?*

A. RQ1: Exploiting Real-world apps

Dataset and Environment Setup. We collect apps from Google Play [17] and Huawei AppGallery [16], as they are widely used and represent both global and Chinese app ecosystems. Specifically, we select the top 20 most downloaded apps from each of 10 categories on both Google Play and Huawei AppGallery. We also add pre-installed apps from the top 5 smartphone manufacturers (e.g., OPPO AppStore) into dataset, resulting in a total of 433 apps. Next, we develop an automated static scanner to identify hybrid apps by detecting WebView usage and JS bridge registration, yielding 402 apps for evaluation. Our evaluation is conducted on a Linux machine with 32G memory, 13th Gen Intel(R) Core(TM) i7-13700, running Ubuntu 22.04. The Android apps are executed on a OnePlus 6T device running Android 11 (H_2OS 10.0.11, build number ONEPLUS A6010_41_210716). After generating exploits for each app, we manually triggered the deep links and executed the generated API invocations to verify the authenticity and consequences of the exploits.

Results Overview. During automated exploit generation on 402 hybrid apps, DEEPEXPLOITOR generated 83 end-to-end exploits (20.6%). Specifically, it generated 136 legal deep links (33.8%) that load the URL into WebView. In the generation of API invocations, it succeeded for 83 of those 136 apps (61.0%). Of the 83 exploits, DEEPEXPLOITOR collected API invocation samples for 62 apps; in 56 cases the sample directly enabled exploits generation, 6 were non-essential, and the remaining 21 exploits were generated without invocation samples. The 83 vulnerabilities span 56 unique vendors; 35 have been acknowledged or assigned CVE/CNVDs, and 24 are rated high/critical, underscoring the substantial impact of these security flaws. Table I lists 15 popular apps that are found to be vulnerable to DeepLink Attack. Among these apps, the result that invokes sensitive APIs varies from each other. For example, in OPPO App***** and t**p.com, the adversary can abuse sensitive APIs to achieve severe consequences, including malicious app installation silently or remote code execution. In C**na Unicom and J******G, the adversary can arbitrarily steal or delete file directories on the victim device. In 6 apps, the adversary can manipulate sensitive information from victims, such as deleting courier delivery details. In addition, 41 apps can lead to sensitive user information leakage (e.g., phone numbers and login tokens).

For these apps vulnerable to DeepLink Attack, We also classified the root cause of the identified vulnerabilities. Specifically, 22 cases stem from insecure URL check logic implementation (e.g., unsafe URL parsers). In 18 apps, the allowlist contains vulnerable domain assets, enabling attackers to perform code injection. The remaining 43 apps lack any form of URL check, allowing attackers to directly exploit without the need to bypass URL check logic.

FP analysis. We group the 10 false positives into three patterns. Seven are *immediate receipt acknowledgments*: upon invocation, the JS bridge returns a success-like callback that merely confirms receipt; parameter validity is not yet established, and subsequent execution fails (e.g., due to missing or ill-formed values), yielding no observable side effect. Two cases are *validation-only confirmations*: the callback indicates that argument parsing/format checks passed, yet the later execution path is blocked (e.g., by domain allowlists), so no side effect occurs. The remaining one reflects *an unmet prerequisite in a multi-step invocation*: the sensitive API requires a prior state-establishing invocation (e.g., creating a DB record); invoking only the terminal API returns a non-error callback but yields no effect. All cases were manually

verified and excluded from the final results. Apart from this, as DEEPEXPLOITOR exploits DeepLink Attack by dynamically triggering deep links and invoking APIs, this dynamic validation process can eliminate the vast majority of false positives. For instance, during ICPG construction in deep link inference, some redundant strings may be identified as potential constraint values. However, since DEEPEXPLOITOR dynamically triggers the constructed deep link, FP can be effectively avoided in the final results.

FN analysis. Exhaustively inspecting all unexploited apps is impractical due to the reverse engineering effort. We therefore sample 50 popular apps unexploited by DEEPEXPLOITOR and conduct manual verification. We identified 7 exploitable cases, which are classified as four *static-related* FNs and three *runtime-related* FNs. For static-related FNs, one app handles the deep link using Java reflection, preventing resolution of target callee and blocking our taint analysis at this site; after manually modeling this reflection call, DEEPEXPLOITOR generates a working exploit. Another app implements a branch-based (`if/else`) router instead of a mapping, which prevents router mapping extraction and downstream constraint resolving. In the remaining two apps, DEEPEXPLOITOR does not extract representative API invocation samples for parameters with cross-field dependencies; without such examples, the LLM cannot instantiate the required constraints into a valid exploit. For runtime-related FNs, the 5-minute time limit for domain asset crawling leaves one vulnerable domain asset unvisited in one app. Strong anti-instrumentation checks in two apps prevent dynamic confirmation, even DEEPEXPLOITOR employs bypass strategies (e.g., Florida [39]). Other potential FN sources include server-fetched fields missed by static analysis, dynamic code loading, and heavily obfuscated routing logic.

Runtime Overhead. In this experiment, we evaluated the efficiency of DEEPEXPLOITOR in generating exploits for apps in the dataset. On average, DEEPEXPLOITOR took about 48h to generate exploit for all vulnerable apps, averaging 35min18s per app. The main time consumption of the tool lies in taint analysis and domain assets appraisal, which take an average of 16min8s and 8min49s per app, respectively. Although these two processes make DEEPEXPLOITOR slightly more time-consuming than previous heuristic-based approaches, they significantly improve the accuracy of exploit construction. By precisely extracting and resolving constraints for the exploit, DEEPEXPLOITOR generates results that satisfy the required nested structures and diverse field values at each layer. Additionally, domain assets appraisal helps broaden the attack surface, enabling more effective bypassing of URL checks.

B. RQ2: Comparison with previous work

We compare the effectiveness of DEEPEXPLOITOR with the state-of-the-art approaches across the entire dataset.

Baseline Setup. Currently, no approach can automatically generate exploits of DeepLink Attacks, which involve constructing both legal deep links and corresponding API invocations. XAWI [6] proposed a fuzzing tool to generate legal deep link. BridgeScope [13] and APIScope [14] proposed strategies to generate API invocation. Therefore, we adopt XAWI+BridgeScope and XAWI+APIScope as our baselines.

TABLE II
COMPARISON BETWEEN DEEPEXPLOITOR AND BASELINES.

Approaches	TP	FP	FN	Prec(%)	Recall(%)
`XAWI+BridgeScope`	7	31	83	18	8
`XAWI+APIScope`	5	10	85	33	6
`DeepExploitor`	83	10	7	89	92

We first implement XAWI [6]'s strategy for deep link construction by statically extracting heuristic strings from apps and applying its fuzzing strategy. For BridgeScope [13], we implement its heuristic strategy, which leverages parameter types, API names and semantics of API. For APIScope [14], we extracts JSON-based invocations via static analysis and uses predefined templates to instantiate parameters, randomly shuffling them to create multiple invocations. As these baselines do not support fully automated exploit generation, we substitute only deep link and API generation parts in DEEPEXPLOITOR, keeping the rest intact to preserve automation.

Ground Truth Construction. Ideally, comparing each approach's exploit generation effectiveness requires enumerating all vulnerabilities in the dataset, which is infeasible [40]–[43]. Therefore, to ensure a fair comparison, we followed the widely used method [40], [41], [44] of constructing a ground truth. Specifically, we aggregate all vulnerabilities exploited by DEEPEXPLOITOR or baselines in our dataset, which consists of 83 vulnerabilities. We also include the 7 additional vulnerabilities identified during our FN analysis. All the vulnerabilities involved in the ground truth are carefully manually confirmed and tested with exploits, ensuring that they are indeed real vulnerabilities. In all, our ground truth consists of 90 vulnerabilities.

Results. Table II shows the results of comparison between DEEPEXPLOITOR and baselines. For apps in dataset, XAWI+BridgeScope and XAWI+APIScope generate 7 and 5 legal exploits, respectively. Overall, DEEPEXPLOITOR surpasses baselines by 86% in and 84% in recall. Baselines struggle to generate valid deep links when exploits deviate from fixed templates or involve structural constraints not captured by heuristics. For example, the deep link for D**u is [redacted]://m.poizon.com/router/web/BrowserPage?loadUrl=. Under XAWI's strategy, it can only generate links like [redacted]://m.poizon.com/?url= As a comparison, DEEPEXPLOITOR successfully generates legal deep links in 136 apps.

When generating API invocations, baselines fail to effectively resolve structural and field constraints of API parameters in modern hybrid apps, which often contain nested structures and diverse field values. For example, Listing 1 shows the invocation code for exploiting `getLocation` in M**tuan. The parameter of this API contains many constraints. Its parameter must start with `js://_` and is treated as a URL. The method name and JSON parameters are extracted from queryParameter, and `getLocation` further requires four fields

TABLE III
ABLATION STUDY FOR FOUR VARIANTS OF DEEPEXPLOITOR

Approaches	TP	FN	Recall(%)
DE-NoIRouter	46	44	51
DE-NoIndirect	22	68	24
DE-NoSample	27	63	30
DeepExploitor	83	7	92

with strict values (e.g., `type = "GCJ02"`). BridgeScope and APIScope cannot effectively handle these nested structure and various field values in M**tuan. In contrast, DEEPEXPLOITOR models API router and combine program analysis with LLM to identify and resolve these constraints.

```
JSAPI.invoke("js://_?method=getLocation&args="
  + encodeURIComponent(JSON.stringify({
    mode: "instant",
    timeout: 15000,
    type: "GCJ02",
    raw: false
})));
```

Listing 1. API invocation for exploiting getLocation in M**tuan. For ethical reasons, we have decided to withhold the names of API

C. RQ3: Ablation Study

In this research question, we conduct two ablation studies. We evaluate how each core modules of DEEPEXPLOITOR contributes to exploit generation in §VI-C1. Then we evaluate whether the LLM ability influences the effectiveness of DEEPEXPLOITOR in §VI-C2.

1) Variants Setup: We systematically evaluate the contribution of each core module within DEEPEXPLOITOR by incrementally removing them to assess their impact on performance. Specifically, we constructed three variants of DEEPEXPLOITOR. The detailed configurations are listed below.

- DE-NoIRouter: We remove intent router modeling and directly perform taint analysis to track execution path.
- DE-NoIndirect: We remove API router modeling and directly construct the ICPG to extract parameter constraints.
- DE-NoSample: When generating API invocations, we remove the invocation sample from the prompt.

Table III presents the comparison results of DEEPEXPLOITOR and its four variants against the entire dataset. Experimental results show that DEEPEXPLOITOR performs best, with a total of 83 exploitable apps detected, far exceeding DE-NoIRouter (46), DE-NoIndirect (22) and DE-NoSample (27). The experimental results prove that the model of intent router and indirect call router is essential for determining the execution path. Invocation samples also effectively reduce the complexity of generating the API invocation for LLM.

2) Influence of LLM capability: In this part, we evaluate whether the capability of LLM influences the process of automated exploit generation. We conduct experiments on 30 randomly selected apps through five different LLMs, including commercial APIs (e.g., ChatGPT-4o, DeepSeek-v3) and three locally deployed LLMs (e.g., qwen2.5:7b, gemma2:27b and Llama3:70b). When DEEPEXPLOITOR works with these different LLMs, all of the selected models can generate effective exploits in 83 apps. Even DEEPEXPLOITOR works with a 7b model can generate legal deep links and API invocations, which reflects the generality of the proposed methodology.

VII. CASE STUDY

To further demonstrate how DeepLink Attack influence real-world apps, we illustrate two interesting cases:

Case #1: Remote Code Execution in Tp.com.**

This example demonstrates a vulnerability that combines DeepLink Attack and path traversal vulnerability on t**p.com, leading to remote code execution. T**p.com is a famous online travel agency with more than 400 million users. Let us describe how this vulnerability occurs.

T**p.com accepts deep links of the format ctripglobal://[redacted]?url=, which loads the `url` parameter into a WebView. The app registers several sensitive APIs for WebView; however, it lacks customized access control to restrict unsafe invocations, allowing attackers to abuse them. One particularly dangerous API is `downloadData`, which accepts three parameters: `url`, `pageUrl`, and `dirSavePath`. It creates a directory based on the parameters `pageUrl` and `dirSavePath`. Then it downloads a file from the link specified by the `url` parameter to the created directory. However, this API contains a path traversal vulnerability: By prepending several instances of `../` to the `dirSavePath` parameter, an attacker can direct file downloads to arbitrary locations on the device. Even worse, t**p.com stores a `.so` file in its files folder. By carefully crafting parameters and invoking the `downloadData` API, an attacker can overwrite this .so file, thereby achieving arbitrary code execution.

Case #2: Privileged back-end service exploit in Tll.**

This example demonstrates how a DeepLink Attack can abuse sensitive APIs to exploit the privileged back-end services of the T**ll mobile app, a shopping platform with over 500 million users. An attacker can abuse its sensitive APIs to access privileged back-end service and manipulate private information (e.g., express logistics data).

The T**ll app accepts the deep link t**ll://[redacted]?url= and loads the `url` parameter in a WebView. Before loading a URL, T**ll performs a domain allowlist check. However, it relies on the java.net.URI parser, which incorrectly treats parsing failures as safe. Since this parser fails to handle backslashes, an attacker can bypass the check using a URL like https://evil.com\safe.com. The app exposes a sensitive WebView API, `nativeCall`, implemented via an API router. Its first parameter specifies the method name. A notable example, `[redacted].send`, manipulates user data on A**baba's high-privilege back-end servers using subsequent parameters. By crafting the second parameter, an attacker can steal or delete express package data—including logistics info, addresses, and phone numbers—posing a serious privacy risk.

VIII. DISCUSSIONS

Mitigations of DeepLink Attack. DeepLink Attack stems from unsafe, customized URL checks. Hybrid apps lack a

unified access control framework for WebView APIs, leading developers deploy ad-hoc mechanisms that introduce vulnerabilities. In the short term, we have proposed remedial measures for the vulnerable apps identified by DEEPEXPLOITOR and reported these issues to respective developers. In the long term, hybrid apps need a standardized security check framework to avoid logic errors inherent in customized implementations. Additionally, vulnerabilities in domain assets that facilitate code injection can also lead to DeepLink Attack. Developers should regularly evaluate trusted domain assets to identify and remediate vulnerabilities that might trigger code injection. As a solution, DEEPEXPLOITOR can assist by scanning allowlisted domain assets from four distinct attack surfaces.

Ethics Concerns. During evaluation, we used our own devices and accounts to avoid impacting developers' back-end servers. In domain asset evaluation, we took care not to compromise the back-end servers. For example, when using Xray to scan XSS vulnerabilities, we focused only on reflected XSS, ensuring no impact on server databases [45]. When assessing the security implications of sensitive APIs, we relied solely on the author's personal accounts and devices to avoid causing potential harm to developers and other app users. Upon identifying vulnerable apps, we responsibly reported these vulnerabilities to all developers. For vendors lacking a public security response center, we notified them by email and provided a 45-day window to fix the vulnerabilities.

Threats to Validity and Limitations. We summarize the primary threats to validity and the limitations of DEEPEXPLOITOR. A major threat to internal validity stems from bias in re-implementing prior works. To reduce such threat, we adhere closely to the methodologies described in their papers, and use them solely to replace corresponding modules in DEEPEXPLOITOR to preserve end-to-end automation. For external validity, our sampling of popular apps across ten categories may introduce bias. We mitigate this via cross-store sampling (Google Play and Huawei AppGallery) and a stratified design that selects the top 20 apps per category. For conclusion validity, LLM outputs are inherently stochastic; results may vary across runs. To mitigate this threat, we fix the sampling temperature to 0.

In addition, although DEEPEXPLOITOR has generated many exploits, several limitations remain. First, our static analysis implementation may prematurely truncate taint flows in corner cases, thereby undercounting reachable sinks. Second, adversarial defenses such as anti-instrumentation can disable dynamic tracing. Third, when collecting API invocation samples, JS obfuscation may prevent the crawler from locating representative instances, reducing exploit-synthesis coverage. Fourth, DEEPEXPLOITOR may not handle designs not surfaced in our preliminary study (e.g., branch-based routers). Finally, when synthesizing API invocations, our oracle primarily relies on callback-centric semantic cues, which may miss behaviors outside callbacks and lead to false positives. Empirically, these issues were infrequent in our evaluation (10 FPs and 7 FNs in §VI-A), and thus the overall impact is limited.

IX. RELATED WORK

A. WebView Security

WebView has been studied by many research works [3]–[7], [15], [19], [46]–[49]. Hassanshahi et al. [5] proposed the Web-to-App Injection attack, requiring another app to send an Intent containing a crafted hyperlink. The authors also proposed W2AIScanner to detect it. Han et al. [7] proposed MEDUSA Attack, which leverage in-app QR code readers to inject code and exploit sensitive APIs through manual work. Zhang et al. [15] proposed identity confusion in WebView-based apps. Different from these works, we focus more on proposing an Automated Exploit Generation for DeepLink Attack.

B. Automated Exploit Generation

There are a few prior works having explored automatic exploit generation (AEG) for Android apps. Garcia et al. [50] presented LetterBomb, which is the first approach for automatically generating exploits of Android apps at their ICC interface. Yang et al. [46] proposed EOEDroid, which can automatically analyze event handlers, detect exploitable functionalities, and generate exploit code. Different from these works, we present the first automatic exploit generation framework for DeepLink Attack, where LLMs assist in modeling customized router encapsulation and constructing valid exploit inputs.

C. LLM for Vulnerability Detection

Recent works have proposed various method to utilize LLM for vulnerability detection [51]–[60]. For example, Li et al. [51] utilized LLM to prune false positive when detecting UBI bugs through static analysis. Wang et al. [52] leveraged LLM to handle various natural language processing tasks while fuzzing IoT devices. Sun et al. [60] proposed an approach for smart contract logic vulnerability detection, which utilized LLM to match candidate vulnerable functions based on code-level semantics. Different from these works, we present the first AEG for DeepLink Attack enhanced with LLM.

X. CONCLUSION

In this paper, we design and implement the first framework, called DEEPEXPLOITOR, to automatically detect and exploit DeepLink Attack enhanced with LLM. DEEPEXPLOITOR discovers 83 zero-day vulnerabilities, with 24 of them considered high or severe risk. We responsibly disclosed all the vulnerabilities to their vendors and 35 of them acknowledged.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful comments. This work was supported in part by the National Natural Science Foundation of China (62102093, U2436207, 62172105, 62202106, 62302101, 62172104, 62102091, 62472096, 62402114, 62402116). Min Yang is the corresponding author and a faculty of Shanghai Institute of Intelligent Electronics & Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China. Yuan Zhang was supported in part by the Shanghai Pilot Program for Basic Research - FuDan University 21TQ1400100 (21TQ012).

REFERENCES

[1] (2025) Webview. [Online]. Available: https://developer.android.com/reference/android/webkit/WebView

[2] (2025) Wkwebview. [Online]. Available: https://developer.apple.com/documentation/webkit/wkwebview

[3] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on webview in the android system," in *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011, pp. 343–352.

[4] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation," in *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, 2014, pp. 66–77.

[5] B. Hassanshahi, Y. Jia, R. H. Yap, P. Saxena, and Z. Liang, "Web-to-application injection attacks on android: Characterization and detection," in *Computer Security–ESORICS 2015: 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II 20*. Springer, 2015, pp. 577–598.

[6] T. Li, X. Wang, M. Zha, K. Chen, X. Wang, L. Xing, X. Bai, N. Zhang, and X. Han, "Unleashing the walking dead: Understanding cross-app remote infections on mobile webviews," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 829–844.

[7] X. Han, Y. Zhang, X. Zhang, Z. Chen, M. Wang, Y. Zhang, S. Ma, Y. Yu, E. Bertino, and J. Li, "Medusa attack: Exploring security hazards of {In-App}{QR} code scanning," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4607–4624.

[8] (2025) Vulnerability in tiktok android app could lead to one-click account hijacking. [Online]. Available: https://www.microsoft.com/en-us/security/blog/2022/08/31/vulnerability-in-tiktok-android-app-could-lead-to-one-click-account-hijacking/

[9] (2025) Recently popular android apps. [Online]. Available: https://www.appbrain.com/apps/popular/

[10] (2025) Webviews – unsafe uri loading. [Online]. Available: https://developer.android.com/privacy-and-security/risks/unsafe-uri-loading

[11] (2025) Technical advisory: Xiaomi 13 pro code execution via getapps dom cross-site scripting (xss). [Online]. Available: https://www.nccgroup.com/us/research-blog/technical-advisory-xiaomi-13-pro-code-execution-via-getapps-dom-cross-site-scripting-xss/

[12] (2025) Galaxy store applications installation/launching without user interaction. [Online]. Available: https://ssd-disclosure.com/ssd-advisory-galaxy-store-applications-installation-launching-without-user-interaction/

[13] G. Yang, A. Mendoza, J. Zhang, and G. Gu, "Precisely and scalably vetting javascript bridge in android hybrid apps," in *Research in Attacks, Intrusions, and Defenses: 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18–20, 2017, Proceedings*. Springer, 2017, pp. 143–166.

[14] C. Wang, Y. Zhang, and Z. Lin, "Uncovering and exploiting hidden apis in mobile super apps," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2471–2485.

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

[16] (2025) Huawei appgallery. [Online]. Available: https://consumer.huawei.com/cn/mobileservices/appgallery/

[17] (2025) Google play. [Online]. Available: https://play.google.com/store/games

[18] (2025) Addjavascriptinterface. [Online]. Available: https://developer.android.com/reference/android/webkit/WebView#addJavascriptInterface(java.lang.Object,%20java.lang.String)

[19] M. Georgiev, S. Jana, and V. Shmatikov, "Breaking and fixing origin-based access control in hybrid web/mobile application frameworks," in *NDSS symposium*, vol. 2014, 2014, p. 1.

[20] (2025) Owasp top 10. [Online]. Available: https://owasp.org/Top10/

[21] (2025) volume of jingdong users. [Online]. Available: https://pdf.dfcfw.com/pdf/H3_AP202209021577967454_1.pdf

[22] (2025) Jadx. [Online]. Available: https://github.com/skylot/jadx

[23] (2025) Json schema. [Online]. Available: https://json-schema.org/docs

[24] (2025) Structured model outputs. [Online]. Available: https://platform.openai.com/docs/guides/structured-outputs

[25] (2025) Android debug bridge (adb). [Online]. Available: https://developer.android.com/tools/adb

[26] (2025) Url format bypass. [Online]. Available: https://book.hacktricks.wiki/en/pentesting-web/ssrf-server-side-request-forgery/url-format-bypass.html

[27] J. Reynolds, A. Bates, and M. Bailey, "Equivocal urls: Understanding the fragmented space of url parser implementations," in *European Symposium on Research in Computer Security*. Springer, 2022, pp. 166–185.

[28] Z. Zhang, L. Zhang, Z. Zhang, G. Hong, Y. Zhang, and M. Yang, "Misdirection of trust: Demystifying the abuse of dedicated url shortening service," in *32st Annual Network and Distributed System Security Symposium, NDSS*, 2025.

[29] (2025) soot. [Online]. Available: https://soot-oss.github.io/soot/

[30] (2025) Apktool. [Online]. Available: https://apktool.org/

[31] (2025) xray. [Online]. Available: https://github.com/chaitin/xray

[32] (2025) Aquatone. [Online]. Available: https://github.com/michenriksen/aquatone#installation

[33] (2025) whois. [Online]. Available: https://www.kali.org/tools/whois/

[34] (2025) Frida. [Online]. Available: https://frida.re/

[35] (2025) Ollama. [Online]. Available: https://ollama.com/

[36] (2025) Hugging face. [Online]. Available: https://huggingface.co/

[37] (2025) Openai api. [Online]. Available: https://openai.com/api/

[38] (2025) Deepseek api. [Online]. Available: https://api-docs.deepseek.com/

[39] (2025) Florida. [Online]. Available: https://github.com/Ylarod/Florida

[40] F. Liu, Y. Shi, Y. Zhang, G. Yang, E. Li, and M. Yang, "Mocguard: Automatically detecting missing-owner-check vulnerabilities in java web applications," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2025, Conference Proceedings, pp. 10–10.

[41] F. Liu, Y. Zhang, T. Chen, Y. Shi, G. Yang, Z. Lin, M. Yang, J. He, and Q. Li, "Detecting taint-style vulnerabilities in microservice-structured web applications," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2025, Conference Proceedings, pp. 934–952.

[42] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A {state-aware}{black-box} web vulnerability scanner," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 523–538.

[43] E. Güler, S. Schumilo, M. Schloegel, N. Bars, P. Görz, X. Xu, C. Kaygusuz, and T. Holz, "Atropos: Effective fuzzing of web applications for {Server-Side} vulnerabilities," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4765–4782.

[44] E. Trickel, F. Pagani, C. Zhu, L. Dresel, G. Vigna, C. Kruegel, R. Wang, T. Bao, Y. Shoshitaishvili, and A. Doupé, "Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities," in *2023 IEEE symposium on security and privacy (SP)*. IEEE, 2023, pp. 2658–2675.

[45] (2025) Xray documentation. [Online]. Available: https://docs.xray.cool/tools/xray/BasicIntroduction

[46] G. Yang and J. Huang, "Automated generation of event-oriented exploits in android hybrid apps," in *Proc. of the Network and Distributed System Security Symposium (NDSS'18)*, 2018.

[47] P. T. Lau, "Static detection of event-driven races in html5-based mobile apps," in *Verification and Evaluation of Computer and Communication Systems: 13th International Conference, VECoS 2019, Porto, Portugal, October 9, 2019, Proceedings 13*. Springer, 2019, pp. 32–46.

[48] M. Diamantaris, S. Moustakas, L. Sun, S. Ioannidis, and J. Polakis, "This sneaky piggy went to the android ad market: Misusing mobile sensors for stealthy data exfiltration," in *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, 2021, pp. 1065–1081.

[49] J. Hu, L. Wei, Y. Liu, and S.-C. Cheung, "ωtest: Webview-oriented testing for android applications," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 992–1004.

[50] J. Garcia, M. Hammad, N. Ghorbani, and S. Malek, "Automatic generation of inter-component communication exploits for android applications," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 661–671.

[51] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Assisting static analysis with large language models: A chatgpt experiment," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 2107–2111.

[52] J. Wang, L. Yu, and X. Luo, "LLMIF: augmented large language model for fuzzing iot devices," in *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 2024, pp. 881–896. [Online]. Available: https://doi.org/10.1109/SP54263.2024.00211

[53] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society, 2024. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/large-language-model-guided-protocol-fuzzing/

[54] D. Wang, G. Zhou, L. Chen, D. Li, and Y. Miao, "Prophetfuzz: Fully automated prediction and fuzzing of high-risk option combinations with only documentation via large language model," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, B. Luo, X. Liao, J. Xu, E. Kirda, and D. Lie, Eds. ACM, 2024, pp. 735–749. [Online]. Available: https://doi.org/10.1145/3658644.3690231

[55] G. Deng, Y. Liu, V. M. Vilches, P. Liu, Y. Li, Y. Xu, M. Pinzger, S. Rass, T. Zhang, and Y. Liu, "Pentestgpt: Evaluating and harnessing large language models for automated penetration testing," in *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, D. Balzarotti and W. Xu, Eds. USENIX Association, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/deng

[56] Asmita, Y. Oliinyk, M. Scott, R. Tsang, C. Fang, and H. Homayoun, "Fuzzing busybox: Leveraging LLM and crash reuse for embedded bug unearthing," in *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, D. Balzarotti and W. Xu, Eds. USENIX Association, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/asmita

[57] D. Liu, Z. Lu, S. Ji, K. Lu, J. Chen, Z. Liu, D. Liu, R. Cai, and Q. He, "Detecting kernel memory bugs through inconsistent memory management intention inferences," in *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, D. Balzarotti and W. Xu, Eds. USENIX Association, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/liu-dinghao-detecting

[58] X. Ma, L. Luo, and Q. Zeng, "From one thousand pages of specification to unveiling hidden bugs: Large language model assisted fuzzing of matter iot devices," in *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, D. Balzarotti and W. Xu, Eds. USENIX Association, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/ma-xiaoyue

[59] C. Ye, Y. Cai, and C. Zhang, "When threads meet interrupts: Effective static detection of interrupt-based deadlocks in linux," in *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, D. Balzarotti and W. Xu, Eds. USENIX Association, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/ye

[60] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu, "Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.