Automated Software Engineering (2026) 33:30
https://doi.org/10.1007/s10515-025-00575-9

The arts and crafts of android adware across a decade

Chao Wang[1] · Tianming Liu[1] · Yanjie Zhao[1] · Lin Zhang[3] · Xiaoning Du[2] · Li Li[4] · Haoyu Wang[1]

Received: 31 March 2025 / Accepted: 28 October 2025 / Published online: 11 November 2025

Abstract

Adware represents a pervasive threat in the mobile ecosystem, yet its inherent characteristics have been largely overlooked by previous research. This work takes a crucial step towards demystifying Android adware. We present **AdwareZoo**, a comprehensive dataset comprising **15,996** adware samples across **118** distinct families collected from security reports and app repositories. We identify adware family payloads by isolating packages from samples for VirusTotal rescanning, unveiling distinctive patterns in family naming conventions, and exposing the misclassification of legitimate ad networks as adware. Our analysis of payload location strategies reveals that over 30% of adware families employ payloads beyond conventional Java/Kotlin code. Based on our dataset analysis, we conducted a comprehensive Adware Characterization of **92** distinct adware families, revealing diverse implementation patterns and evolving techniques across the mobile ecosystem. To facilitate this analysis, we developed an Adware Characterization Schema that provided a structured taxonomy for systematically classifying the observed behaviors. Our investigation uncovered multiple categories of fraudulent activities, including aggressive ad display techniques, sophisticated click fraud implementations, privacy information leakage, malicious promotion mechanisms, and various persistence and evasion mechanisms employed to avoid detection while maximizing illicit revenue. This research establishes foundations for comprehending the fraudulent and adversarial techniques within the mobile adware landscape and facilitates the development of more robust detection mechanisms against these evolving threats.

Keywords Android adware · Mobile advertising · Android malware dataset · Malware characterization

Chao Wang and Tianming Liu are the co-first authors.

Extended author information available on the last page of the article

1 Introduction

In the domain of cybersecurity, adware is generally defined as software that displays unwanted advertisements (adware 2024). Commonly categorized as greyware (2024) or potentially unwanted applications (PUAs 2024), adware occupies an ambiguous space between benign software and overtly malicious programs like viruses and trojans.

As advertising has become a prevalent revenue model on mobile platforms, mobile adware has emerged as a pervasive threat. A recent report on the mobile malware landscape by Kaspersky indicates that over 40% of detected mobile malware are adware (The mobile malware threat landscape in 2023 2024). Beyond displaying aggressive, unwanted advertisements, mobile adware is known to exhibit various threat-posing behaviors, including stealing privacy information (Zheng and Xu 2015), propagating viruses and trojans (Wu 2020), and committing ad fraud (Xu 2019).

Despite its ubiquity and multifaceted risks, mobile adware has received disproportionately less attention from the research community compared to common mobile malware. Existing research on mobile adware has generally focused on adware identification (Alani and Awad 2022; Suresh et al. 2019) and addressing whether adware should be considered as malware (Gao et al. 2019). Unlike with general Android malware, no previous work has concentrated on demystifying the characteristics of adware. While general malware characterization focuses on overtly malicious behaviors such as data theft, privilege escalation, and system compromise, adware operates in a fundamentally different paradigm. It primarily employs aggressive advertising tactics and fraudulent ad behaviors to generate revenue, blurring the lines between legitimate advertising and malicious activities. This unique operational model necessitates a distinct characterization approach that captures adware-specific behaviors, such as ad fraud techniques, intrusive ad placement strategies, and evasion mechanisms tailored to avoid ad fraud detection systems rather than traditional antivirus engines.

This gap partly originates from **the lack of a dedicated, well-annotated adware dataset**. While mobile malware analysis has made significant strides, with reliable and comprehensive Android malware datasets available such as MalGenome (Zhou and Jiang 2012), Drebin (Arp et al. 2014), AMD (Wei et al. 2017), Rmvdroid (Wang et al. 2019), and Malradar (Wang et al. 2022), these resources have comprehensively elucidated Android malware in terms of their characteristics, behaviors, and implementation details on a family basis, which have provided valuable insights for combating malware. However, there has not been such demystifying work or a dedicated dataset available for adware. The general malware datasets mentioned above contain only a limited number of adware samples that have not been specifically annotated.

To demystify adware, the first critical step is identifying the precise location of malicious code that executes deceptive behaviors. Locating these payloads serves as a fundamental starting point for in-depth analysis, enabling researchers to understand implementation patterns and develop effective detection strategies. Furthermore, a comprehensive and systematic analysis of payloads across different adware families is essential to establish a thorough characterization of their behaviors, techniques,

and evasion mechanisms. During the characterization process, maintaining consistency and reliability in analysis methodologies is also crucial when examining diverse adware samples, particularly when confronting sophisticated obfuscation and code-hardening techniques deployed by adware developers to evade detection.

To address these limitations and fill the existing research gap, our work takes crucial steps in demystifying Android adware by constructing a novel, comprehensive adware dataset (§2.1), identifying adware family payloads (§2.2) within the dataset, and revealing the systematic characterization of adware families (§2.3).

Our dataset construction process involves collecting samples from both security reports and app repositories, implementing adware filtering, and classifying samples into distinct adware families using AVClass (Sebastián and Caballero 2020). We then identify adware family payloads by isolating packages from adware samples for VirusTotal rescanning, while employing feature hashing to overcome obfuscation challenges. This comprehensive methodology resulted in a well-annotated adware dataset **AdwareZoo**, which comprises **15,996** adware APKs across **118** distinct adware families, providing a robust foundation for in-depth adware analysis and future research.

Our analysis revealed significant insights into adware ecosystems. For instance, we uncovered patterns in adware family naming conventions and identified 10 widely-used top ad networks classified as adware, highlighting the pressing need to demystify adware characteristics for better understanding and mitigation of their security threats. We revealed diverse adware payload location strategies, with over 30% adware families using payloads other than traditional Java/Kotlin code. We also identified and characterized several key evasion techniques employed by adware, including package name obfuscation, hiding under trusted package names, utilizing SO library payloads, and dynamic loading of payloads. These findings offer valuable insights for future adware detection and analysis efforts, contributing to the field of mobile security.

Differences with the workshop paper building upon our previous work (Wang et al. 2024), we conduct comprehensive Adware Characterization across numerous families, revealing their sophisticated implementation strategies, evasion techniques, and monetization mechanisms. To enable this systematic analysis, we developed an Adware Characterization Schema that establishes a structured taxonomy for classifying adware behaviors and implementation patterns. Using this schema, excluding those with undetectable payloads and excessive obfuscation, we performed extensive analysis of **92** adware families, including detailed examinations of variants within 6 families (`hiddad`, `hiddenad`, `hiddenads`, `mocen`, `ewind`, and `clicker`) to capture their evolutionary trajectories and behavioral patterns. Critically, our characterization reveals adware-specific patterns that are fundamentally different from general malware. Unlike traditional malware that focuses on system exploitation, adware employs sophisticated monetization strategies through fraudulent advertising practices. Our analysis exposes three distinct click-fraud techniques and various aggressive ad-presentation methods that have not been systematically documented in previous malware studies. These findings demonstrate that adware requires special-

ized analysis frameworks beyond conventional malware characterization approaches and lay the foundation for more effective detection and mitigation methods.

In summary, our work makes the following key contributions:

- We present a comprehensive, well-annotated Android adware dataset **AdwareZoo**, comprising **15,996** adware APKs across **118** distinct adware families. We locate the payload for each adware family, paving the way for future in-depth adware analysis. We make available our dataset to boost related research efforts[1].
- We provide important insights into adware ecosystems, including adware family naming patterns, revealing widely-used ad networks being identified as adware, diverse payload location strategies, and key evasion techniques. These findings contribute significantly to understanding and mitigating adware threats.
- To enable systematic categorization, we developed a structured characterization schema for adware. We conducted an extensive characterization of **92** adware families, revealing behavioral patterns and implementation strategies that are distinct from general Android malware. Unlike traditional malware characterization, which primarily focuses on system-level exploitation, our schema captures adware-specific dimensions such as fraudulent advertising techniques and aggressive monetization strategies targeting ad fraud detection systems. We have open-sourced our characterization schema and results[1], laying a foundation for more effective detection and mitigation approaches.

The remainder of this paper is organized as follows: In §2, we present our comprehensive methodology for dataset construction, payload detection, and adware characterization. Our adware dataset composition is detailed in §3. We then evaluate and analyze payload detection results in §4, followed by adware characterization findings in §5, where we examine fraudulent behavior and adversarial techniques across different families. Finally, we discuss limitations and practical implications in §6, review related work in §7, and offer concluding remarks in §8.

2 Methodology

Our study on Android adware focuses on three critical aspects: dataset construction, payload detection, and adware characterization. In §2.1, we detail the construction process of the **AdwareZoo** dataset. This involves curating a diverse collection of adware samples from multiple sources, including security reports and app repositories. §2.2 introduce our approach for identifying adware payloads, which relies on isolating packages from adware samples for VirusTotal rescanning, laying the foundation for our systematic characterization of adware families. We elucidate the key assumptions underlying this method, our strategies for combating obfuscation, and the payload identification process. §2.3 describes our approach to characterizing

[1] Dataset available at https://github.com/security-pride/AdwareZoo, together with detailed payload detection and characterization results.

adware families through the development of a comprehensive Adware Characterization Schema. We detail the methodical construction of this schema to systematically capture the diverse behaviors and techniques employed by adware. Additionally, we present our carefully designed analysis guidelines that ensure consistency and reproducibility when examining the technical characteristics across different adware families.

2.1 Adware dataset construction

A comprehensive and up-to-date dataset is imperative to facilitate an in-depth analysis of Android adware. While Gao et al. (2019) made a valuable contribution by constructing an adware dataset in 2019 based on VirusTotal labels, this dataset covers only 26 adware families. Hence, we took the initiative to construct a new adware dataset **AdwareZoo**, contributing to future research in this domain.

Typically, there are two primary approaches for curating an Android malware dataset: analyzing security company reports (Cao et al. 2022; Wang et al. 2022), and scrutinizing large app repositories (Wei et al. 2017; Wang et al. 2019). We employ both approaches to construct our dataset, which comprises **15,996** adware APK samples spanning **118** distinct adware families.

2.1.1 Collecting adware from security reports

We first collected Android adware samples from reports from reputable security companies. This process began by crawling reports from a curated list of 19 sites associated with renowned security companies. This list, along with the corresponding crawlers, was sourced from a recent study (Cao et al. 2022), which identified these security companies with Gartner's list of best endpoint security platforms (Gartner 2021). For each site, we utilized the respective crawler to seek out reports related to Android adware with the following keyword sets: [android, mobile, aggressive, security, malware] and [ad, ad fraud, ad library, adware, clicker]. This endeavor resulted in a collection of 2,364 report posts dated from 2008 to 2023. Subsequent filtering excluded reports that either did not mention "android" in their title and body or were not authored in English, leaving us with 797 reports. We further remove those unrelated to Android adware through manual inspection, narrowing down the list to 104 reports. We then attempted to gather the adware samples associated with each report. Therefore, we retained only those reports that included hash indicators of the app (MD5, SHA1, or SHA256) that facilitate app collection. Using the hash indicator, we download the corresponding APK file from VirusTotal (2023) with its premium APIs. This phase yielded 54 reports with 1,410 corresponding APK files.

Sample filtering Our report filtering was solely focused on eliminating reports unrelated to Android adware, instead of directly ascertaining if the described samples were adware or not by simply browsing through the report. Indeed, the definition of adware can be somewhat ambiguous, especially since advertising often serves as a revenue stream for viruses and trojans, leading to challenges in establishing a definitive set of criteria for its identification. We thus resorted to VirusTotal for the filtration

of adware samples following (Gao et al. 2019). Specifically, if a sample was explicitly identified as adware by at least one security vendor on VirusTotal (e.g., **Adware.AndroidOS.Magic.A!c**), we classified it as adware. This procedure resulted in **1,132** APK files from 37 reports being categorized as adware. Of the remaining 17 reports and their corresponding samples, 14 were Dropper Trojans with ad fraud modules, and the remaining 3 were Fakeapp Trojans exhibiting ad behaviors.

2.1.2 Obtaining adware from app repositories via industry partners

Recognizing the limited scope of Android adware samples collected from security reports, we extended our dataset by incorporating adware samples from extensive app repositories. This expansion was facilitated through collaboration with our industrial partner, from whom we acquired an additional **21,018** adware samples. According to their approach, this dataset was curated as follows: Firstly, a comprehensive Android malware dataset was maintained, comprising over 3 million malware APK files collected from Koodous (2023) and Virustotal, covering a decade from 2011 to 2022. Secondly, all malware samples were classified into distinct malware families using the widely adopted AVClass (Sebastián and Caballero 2020), a malware labeling tool that converts labels from various antivirus engines into a single family name for the malware. For each family, a maximum of 300 samples were then randomly selected. Lastly, the same sample filtering procedure as aforementioned was applied to all the selected samples to obtain this adware dataset. We deem this dataset a valuable addition to our research owing to its substantial size, extensive time span, and comprehensiveness in terms of malware family coverage.

2.1.3 Adware family classification

With the two sources combined, we now have a total of **22,150** unique adware APKs differentiated by file hashes. The next step was to obtain a family classification of these adware samples for subsequent analysis. To accomplish this, we turned to AVClass, leveraging its capabilities for malware label aggregation. Given that VirusTotal (2023) is the most comprehensive security platform aggregating multiple antivirus engines, we used the VirusTotal scanning reports, which contain labels from these engines for each adware sample, as input for AVClass. In our pursuit to ensure that our dataset is representative and reflects the prevalence of adware, we implemented an additional filtration step: families with fewer than 20 samples or those for which, on average, less than two security vendors identified their samples as adware were eliminated. However, we made exceptions for seven families which, despite having fewer than 20 samples each, were identified as adware by at least five security vendors on average. They were either associated with popular advertising networks or specific adware behaviors. Given their significance in the adware ecosystem, we decided to retain these families in our dataset. Consequently, our meticulously curated dataset **AdwareZoo** encompasses **15,996** unique APKs spanning **118** distinct families.

2.2 Adware family payload detection

We next attempt to locate the payload for each adware family. In the realm of software security, "payload" typically refers to the component of malware responsible for executing malicious activities. Within the scope of Android adware, a payload is the specific code component facilitating adware behaviors, such as delivering unwanted advertisements or performing ad frauds. Such payload is usually encapsulated in a Java/Kotlin code package within a DEX file (the executable file format in Android). However, payloads may also reside in resource folders as SO files (Android C++ library files accessible with JNI methods) or other executable files (e.g. additional DEXs and JARs) that can be loaded dynamically, a commonly used feature for Android software, especially malware (Poeplau et al. 2014; Xue et al. 2017).

2.2.1 Key assumption

We propose our payload detection approach based on a straightforward and intuitive assumption that, a payload for an Android adware family, if separated and repackaged into a single app for VirusTotal scanning[2], should still be recognized by the antivirus engines as belonging to the same family. Chen et al. (2016) also used a similar approach to identify potentially harmful Android libraries, a huge proportion of which are adware payloads, by separating the library code for scanning. We evaluate this assumption in §4.1.1.

2.2.2 Obfuscation and feature hash

For each adware family, we first obtain all samples belonging to that family, then unpack and de-compile them using ApkTool (2023). As Android's codespace follows Java's package structure, and the payload for an adware family should repeatedly occur in different samples within the family, an intuitive approach would be to rank package names by their occurrence frequency among samples and choose those with higher frequencies as payload candidates for separate scanning. However, adware often employs obfuscation techniques, rendering package names unreliable for differentiating packages. To counter this, we substituted package names with feature hashes for package differentiation, a method adopted from LibRadar (Ma et al. 2016), a widely used tool for detecting third-party libraries in Android apps. The feature hash is calculated based on the frequency of different Android API calls within a package, a metric that is notably resilient to obfuscation[3].

[2] For brevity, we will use "scanning" to refer to "VirusTotal scanning" in this paper.

[3] According to LibRadar (Ma et al. 2016), such feature hashes can uniquely identify different packages, as they observed no instances of two distinct packages sharing the same feature hash within their extensive dataset of one million apps.

2.2.3 Payload identification process

We began by calculating a feature hash for all second-level packages (e.g., com/a) of each sample in the family[4]. Whenever a new feature hash emerges for the family, we extract the corresponding package folder in its entirety (including any code files within), and repackage it into a standalone APK file for subsequent scanning[5].

However, as mentioned earlier, payloads may also be hidden in resource folders as other executable file formats. To address this, we examine the non-DEX components for all samples within the family. For each sample, we unpack the APK file, remove the DEX files within the root directory (those already de-compiled by ApkTool earlier), copy all other folders into a blank APK template, and repackage the template into a new APK for scanning.

Finally, we send our newly generated APKs for scanning and attempted to match the adware family name with the labels yielded by the antivirus engines in the scanning report (e.g., **Ashas** → Android/AdDisplay.**Ashas**)[6]. Whenever there is a match, we deem a payload to be found for that family. For APKs packed with resource folders, if there is a match, we extract all executable files (APKs, DEXs, SOs, JARs, and ZIPs), and pack them one by one for scanning to further locate the exact payload.

2.3 Systematic adware characterization

Adware occupies a unique position in the spectrum of potentially harmful applications, operating in a grey area where it generates revenue through intrusive or malicious advertising practices. A comprehensive analysis of adware characteristics is crucial for gaining a more nuanced understanding of this phenomenon and its implications, as well as for developing targeted mitigation strategies. However, to date, no systematic study has been conducted on adware characteristics, nor has a methodological framework been proposed to facilitate such analysis. While previous research (Cao et al. 2022; Wei et al. 2017; Zhou and Jiang 2012) has established sophisticated characterization schemas for Android malware in general, these frameworks are inherently unsuitable for adware due to its distinct nature.

To address the lack of standardized methodologies for comprehensive adware analysis while ensuring reliability and consistency, we first established an Initial Adware Characterization Schema by learning from security reports, drawing from traditional Android malware models while incorporating adware-specific attributes (in §2.3.1). We then developed static analysis techniques that utilized automated tools to efficiently process large sample sets, identifying suspicious patterns and potential malicious indicators for deeper examination (§2.3.2). Finally, in §2.3.3, we describe our semi-automatic analysis methodology that combines computational tools with

[4] Focusing on level 2 packages strikes an optimal balance between thoroughness and efficiency. Scanning deeper levels, such as com/a/b, would exponentially increase the number of scans required, while level 2 typically provides sufficient payload differentiation.

[5] This is achieved by first creating a blank APK template using Android Studio, copying the folder into the unpacked template, and then repackaging the unpacked template into a new standalone APK file, again leveraging ApkTool.

[6] We also encountered and addressed an aliases problem during the matching process, as detailed in §4.1.1.

human expertise to thoroughly examine complex behaviors that automation alone might miss.

2.3.1 Schema initialization

Designing the characterization schema without prior systematic knowledge presents significant challenges. To overcome this challenge and establish a solid foundation, we leveraged the 37 detailed security reports collected in our dataset (§2.1.1), randomly selecting five samples per report for manual examination. This approach allowed us to methodically map behavioral descriptions from security reports to their corresponding code implementations and gain a preliminary understanding of adware code structure.

From an execution workflow perspective, adware follows patterns similar to malware: it is frequently triggered by specific user interactions or system events, such as timer expirations or screen state changes. Similarly (as with malware), before executing its payload, adware typically performs a series of verification checks and employs various stealth techniques to avoid detection. However, a key distinction is that adware specifically executes advertising fraud behaviors unique to the mobile advertising ecosystem. Based on these findings, we formulated an **Initial Adware Characterization Schema** encompassing three primary dimensions: Survival Mechanisms, Detection Evasion Mechanisms, and Advertising Fraud Mechanisms. This preliminary schema established a foundation that we continuously refined throughout our subsequent manual analysis process.

2.3.2 Static analysis

Through our **Initial Adware Characterization Schema**, we systematically summarized code patterns characteristic of Android adware. We discovered that certain behavior patterns within this schema were amenable to automated or semi-automated detection through static analysis techniques. To facilitate subsequent manual analysis, we developed a static analysis approach leveraging FlowDroid (2023), a sophisticated Android application data flow analysis framework, which enabled us to efficiently identify and flag suspicious code segments across our dataset.

(1) For specific system API call identification, for example, in the Advertising Fraud Mechanisms dimension, adware frequently accesses privileged device information and user data and transmits it to remote servers. This behavior involves identity-related methods such as `getIMEI()`, `getDeviceId()`, `getSubscriberId()`, `getSimSerialNumber()`, `getMacAddress()`, as well as content-access APIs like `getMessageBody()` and application installation queries. By programmatically identifying these signature methods, we established crucial instrumentation points for subsequent manual analysis of potential data exfiltration pathways to advertising networks or tracking infrastructure.

(2) For behaviors requiring data flow analysis, for example, in the Survival Mechanisms dimension, adware frequently employs dynamic registration of system event broadcasts (`registerReceiver()`) or schedules timer-based actions (`setRepeating()`, `postDelayed()`) to ensure it remains active across device states.

Through def-use chain reconstruction and parameter value tracking, we uncovered the temporal patterns and system events leveraged by these mechanisms, revealing how adware implementations strategically schedule their operations to maximize longevity and effectiveness while evading detection.

(3) For heuristic-based pattern recognition, for example, in the Detection Evasion Mechanisms dimension, adware attempts various anti-analysis techniques to evade detection. Given the diverse range of sensitive strings and custom API naming conventions observed (e.g., "generic", "sdk", "google_sdk", and suspicious method nomenclature such as `isEmulator()`, `checkVirtualEnvironment()`, `detectSandbox()`), we employed a heuristic approach by maintaining a comprehensive library of suspicious strings to systematically identify these evasion techniques across our sample set.

For the remaining behavioral characteristics, static analysis alone proved insufficient to detect certain deceptive practices. Notification-based advertisements, for instance, require complex semantic understanding, while applications can legitimately send notifications, determining whether these notifications constitute advertisements necessitates manual verification. Similarly, "Out Of Application" advertisements (where ads appear at inappropriate moments or in unsuitable scenarios) require human judgment to evaluate both the timing and contextual appropriateness of ad displays. These nuanced advertising behaviors highlight the continued importance of expert human analysis alongside automated detection techniques.

2.3.3 Semi-automatic analysis guideline

Before introducing our methodology, we first established approaches to counter anti-analysis techniques employed by malware developers, including code obfuscation, protection mechanisms, and encryption designed to prevent reverse engineering. We utilized jadx (2023) as our primary tool to combat obfuscation techniques, particularly for removing obfuscated naming schemes to improve code readability.

For samples containing jar files, apk files, and native libraries (so files) identified during payload analysis, we incorporated these as integral components of the application code in our analysis workflow. When examining dynamically loaded code, we continued using jadx (2023) for decompilation. In cases where functions were invoked through reflection in dynamically loaded code, we traced the call stack to the corresponding locations within the supplementary code. For additional code in loaded native libraries, we employed IDA (2023) to decompile so files and examined the native library code when the application made native function calls.

To ensure analytical reliability and reproducibility, we established a comprehensive guideline outlining the standardized steps for semi-automatic adware analysis. This guideline serves as our analytical reference framework and maintains methodological consistency across all samples. Our semi-automatic analysis followed these sequential steps:

- Based on payload analysis results, we identified the sources and specific packages of adware family payloads, establishing the foundation for subsequent analysis.
- We then performed static analysis on these identified components to detect key

indicators and suspicious code patterns, which provided initial insights into potential malicious behaviors.
- Using these insights, we systematically examined persistence mechanisms, trigger conditions, and evasion techniques employed by the adware, focusing particularly on how these elements worked together to enable malicious functionality.
- Through detailed code examination, we verified whether samples contained advertising fraud payloads and analyzed their implementation methods and triggering mechanisms.
- To validate our static analysis findings, we conducted dynamic installation and testing when possible, manually adjusting device time, network configurations, and other environmental settings to trigger conditional behaviors. This hands-on approach allowed us to observe execution patterns under various simulated conditions, though some older samples could not be fully tested due to their reliance on decommissioned remote servers.
- Finally, we thoroughly examined all remaining application components to identify any previously undetected adware-related behaviors, ensuring comprehensive coverage of the sample's capabilities.

Whenever we identified new Schema features, we incorporated them into our detection framework and retrospectively examined previously analyzed samples to ensure comprehensive coverage. To minimize bias and validate annotation reliability, two co-authors independently performed manual characterization for each adware family. Their results were then cross-validated by comparing all labeled attributes, including behavioral categories, payload locations, and feature annotations. In cases where discrepancies were observed, both annotators jointly reviewed the corresponding samples to discuss the rationale behind each interpretation. If consensus could not be reached, a third author served as an adjudicator to make the final determination. This iterative cross-validation process not only ensured consistency across families but also allowed continuous refinement of the schema and annotation guidelines throughout the analysis.

3 Dataset overview

This section presents the statistical characteristics of **AdwareZoo**. We begin with the top 20 adware families in §3.1, detailing dataset key metrics and overall statistical features. We then examine the distribution of sample quantities and adware tags across families in §3.2, revealing prevalence patterns and classification metrics. Finally, in §3.3, we analyze adware family naming patterns to uncover insights into vendor classification methodologies.

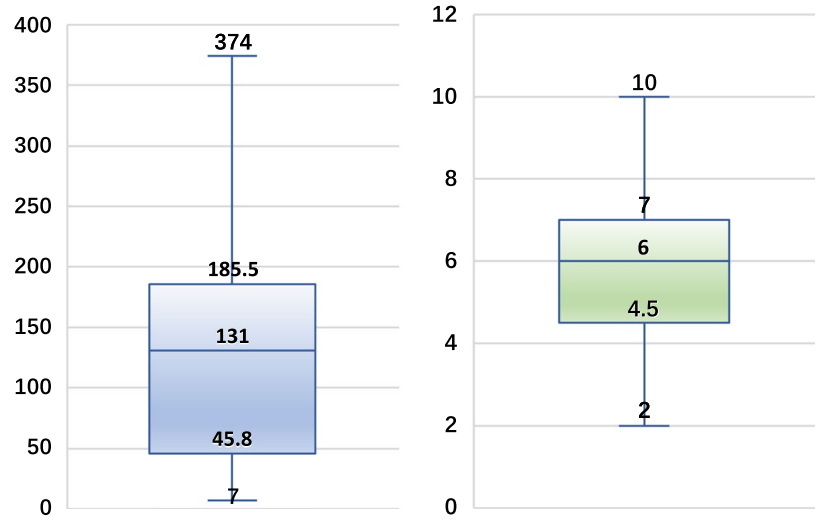
3.1 The top 20 adware families

Table 1 presents a comprehensive overview of the top **20** Android adware families from our **AdwareZoo** dataset, ranked by sample quantity. This table encapsulates key metrics including the number of samples and average adware tag count for each

Table 1 Top 20 adware families in our **AdwareZoo** dataset

No.	Family Name	Sample Quantities	# Adware Tags	Naming Patterns		Payload Location			
				Named After Advertising Networks	Named After Behavior	DEX	Package Name	Resource	Resource Type
1	hiddad	597	4		√	√	android/support/v4/tools	√	.apk
2	dowgin	491	9			√	com/feiwo	√	.img
3	kuguo	374	7	√		√	com/kuguo	√	.so
4	ewind	360	3			√	com/dot	√	.jar
5	domob	309	4.5	√		√	cn/domob		
6	dianle	305	5	√		√	com/dianle		
7	revmob	299	4	√		√	com/revmob		
8	mobcore	297	7	√		√	com/mobcore		
9	cyfin	294	5			√	com/cyjh		
10	airpush	292	7	√		√	zmm0		
11	adviator	291	9	√		√	com/av111411	√	.apk
12	gappusin	281	9	√		√	com/waps		
13	zdtad	269	11			√	com/zdtpay	√	.jar
14	youmi	264	7	√		√	net/youmi		
15	leadbolt	263	5	√		√	com/wpuvbenaeamobfo		
16	plague	253	4			√	net/htfstudio		
17	obtes	247	9			√	ouo		
18	tgpotato	247	3.2			√	com/tgpotato		
19	adwo	241	7	√		√	com/adwo		
20	dasu	238	5			√	com/loader		

family, which will be introduced in §3.2. Additionally, the table provides insights into family naming patterns (§3.3) and critical information on payload location (§4.2). This table serves as a representative showcase of our dataset's structure and content. To facilitate further research, we have made available the complete information for all 118 families in our dataset. While the corresponding samples for each family are too sizable for direct sharing, they are provided as lists of file hashes on our website and are available upon request. Our extensive dataset provides researchers with a rich, well-annotated resource for conducting in-depth analyses of Android adware ecosystems, offering a unique opportunity to explore the intricacies of adware families and their behavioral patterns.



(a) Sample Quantities Distribution (b) Adware Tags Distribution

Fig. 1 Boxplot distributions of sample quantities and adware tags for the 118 adware families

3.2 Distribution of sample quantities and adware tags across families

The 118 adware families within our dataset are each characterized by two key metrics: sample quantity and average number of adware tags. The quantity of samples within each family serves as a proxy indicator of the family's prevalence in the wild, while the average number of adware tags—the mean number of antivirus engines on VirusTotal that identified samples within a family as adware—represents the level of agreement among security vendors. This metric not only reflects the reliability of classifying a family as adware, but also serves as a rough indicator of its potential severity or aggressiveness. A higher number of adware tags suggests stronger agreement among security vendors about the adware nature of the samples, potentially indicating more obvious or problematic behaviors. Conversely, lower numbers might point to more sophisticated evasion techniques or less certainty in classification.

Figure 1a presents a boxplot illustrating the distribution of sample quantities across adware families. The distribution exhibits a notable right-skewed pattern, meaning that while most families have relatively fewer samples, a small number of families possess substantially larger sample sizes, including outliers like the `hiddenads` family with 597 samples and `dowgin` with 491 samples[7]. The median sample size is 131, with a mean of 135.6 samples. The interquartile range spans from 45.8 to 185.5 samples, delineating the range within which the majority of families fall. Figure 1b depicts the distribution of average adware tag numbers across families. This

[7] These extreme outliers are not displayed in the boxplot to maintain visual clarity and focus on the primary distribution.

distribution exhibits a relatively symmetrical and concentrated pattern. The median and mean are both approximately 6 tags (the precise mean being 6.01). Most adware families have between 4.5 and 7 tags. The overall range extends from 2 to 14 tags.

3.3 Adware family naming patterns

In the field of Android malware analysis, researchers typically employ malware labeling tools such as AVClass or Euphony (Hurier et al. 2017) to derive family names (Zhao et al. 2021; Zhu et al. 2020). These family names are aggregated from labels provided by multiple antivirus engines, each adhering to its naming conventions. Consequently, the resulting family names often present a diverse and chaotic landscape, with some names lacking clear semantic meaning. Despite the apparent disorder in the nomenclature of the 118 adware families in our dataset, we have identified two significant patterns in adware family naming conventions:

Named after advertising networks A prominent naming convention we observed involves families being named after advertising networks, suggesting that samples within these families are classified as adware due to their adoption of specific advertising libraries. While it is impractical to identify all such families, particularly those associated with minor ad networks, we have identified at least **37** families following this naming pattern. The naming conventions, however, are not always straightforward. Some are directly named after the advertising company, while others are named based on the advertising library's package name. In some cases, names are derived from specific resource file names such as SO libraries (e.g., Family `Landlord` employs a SO library with a matching name, specifically `liblandlord.so`.). This variety likely reflects the diverse detection methods employed by antivirus companies to identify these adware families.

Interestingly, some families are named after well-known ad networks. By cross-referencing with the top Android ad networks listed on AppBrain (2024), we found **10** such instances, including `Adcolony`, `AppLovin`, `InMobi`, `Mobwin`, `Airpush`, and `Leadbolt`, among others. While we address the reasons for `InMobi`'s identification in §4.2.1, the rationale behind most of these classifications remains unclear. Given the extensive reach of these networks in the mobile advertising market, this situation underscores the urgent need to demystify the characteristics of these adware families. It also calls for greater transparency and regulation in the rapidly evolving landscape of the mobile advertising ecosystem.

Named after behavior We observed another notable naming convention where adware families are designated based on specific advertising behaviors. Our analysis identified at least **6** such families, with their names and corresponding behavioral descriptions presented in Table 2. Unlike families named after ad networks, samples within these families often display a scattered distribution of distinct payloads with diverse package names, indicating the presence of multiple variants within each family. The number of these variants is also included in the table, highlighting the

Table 2 Adware families named after advertising behaviors

Family	# Variants	Behavior Description
Hiddad Hiddenad Hiddenads	21	Invisible or out-of-app ads
Clicker	2	Fraudulent ad clicks generation
Noiconads	N/A	Displaying ads while hiding app icon
Mulad	N/A	Apps adopting multiple ad networks

diversity of implementing similar advertising behaviors in adware.[8] Notably, most of these advertising behaviors are fraudulent or aggressive in nature, underscoring their potential for harm and the critical importance of demystifying their characteristics.

We present a comprehensive, well-annotated Android adware dataset **AdwareZoo**, comprising key information including the number of samples, average adware tag count, and payload location information for each family. We identified two primary naming patterns for adware families: those derived from advertising networks (37+ families) and those based on specific advertising behaviors (6 families). Notably, 10 well-known ad networks are identified as adware, warranting further attention.

4 Payload detection

This section presents the empirical results of our adware payload detection. In §4.1, we evaluate our payload detection approach through a two-step process: first verifying our key assumption with a controlled experiment using ground truth data (§4.1.1), then conducting a manual assessment of detection results across 30 randomly selected adware families (§4.1.2), confirming the effectiveness and accuracy of our methodology. Next, §4.2 presents the findings of our payload detection results, analyzing the payload location patterns across different adware families, revealing distinctive implementation strategies and distribution mechanisms characteristic of each family.

4.1 Payload detection approach evaluation

4.1.1 Verifying the key assumption

We first evaluate our key assumption introduced in §2.2.1, which serves as the basis for our payload detection approach.

Recall that we have successfully collected adware samples from 37 security reports released by reputable security companies (§2.1.1), we noticed that some reports indeed contain or infer the payload location of the described adware sample, which could serve as ground truth for our evaluation experiment. We then ran-

[8] We also provide payload information for each identified variant on our repository.

domly selected ten of the reports that indicated the payload location and selected at most 5 adware samples for each report for the experiment. As a result, 36 samples with the indicated payloads were collected[9]. According to AVClass, they belong to 8 distinct adware families. For each of the samples, its payload is then separated and scanned by VirusTotal. We then simply search for each sample's family name within the labels yielded by the VirusTotal antivirus engines (e.g. **Ashas** → Android/AdDisplay.**Ashas**).

It turns out that, for the 36 samples, we were able to find an exact string match for 29 samples. For the remaining 7 samples, the lack of exact matches can be attributed primarily to discrepancies in family alias naming, notably in the cases of '**Ghostteam**' being referred to as 'Andr/**Ghosttm**-A' and '**Gunpoder**' as 'Trojan:**Gupno**'. This variation in naming across different antivirus engines, based on our further investigation, is quite common. Recognizing the need to address these inconsistencies, we embarked on a dedicated manual alias collection task for each of the 118 adware families from the labels obtained from VirusTotal. Such a manual process is a feasible one-time effort for existing adware families, since aliases are mostly simple rearrangements or minor alterations of the letters in the family names, such as `adviator/davitor`, `adpooh/poohad`, and `kalfere/WalkFree/reefwal`. After incorporating these collected aliases into our family name string matching process (which is also inherited in our payload identification), 35 out of the 36 samples were successfully identified in the VirusTotal re-scanning, validating the original assumption. The only sample that failed is because it only carries a partial payload.

4.1.2 Evaluating payload detection result

To further assess the accuracy of our adware family payload detection, we randomly selected **30** families for verification. For each identified payload within these families, we conducted a manual check to confirm its authenticity as a payload of the family. The process and results are detailed below.

Payloads within DEX files All 30 families were found to have payloads in the form of code packages within DEX files. Among these, **17** families' detected payload packages exhibited a consistent and unique package name that accurately represented the family, validating the correctness of our payload detection. The remaining **13** families showed multiple package names for their detected payload packages, which warrants further investigation. This multiplicity of package names potentially indicated the use of package name obfuscation, a prevalent evasion technique in adware families, as characterized in §4.2.1. To verify this, we manually examined the code structure within these payload packages to determine if they were similar despite different package names, as structural similarities would indicate the same payload under obfuscation. Our analysis revealed that 12 out of the 13 adware families indeed had similar and unique code structures among their respectively detected payloads,

[9] Some reports correspond to less than 5 samples. We collected 45 samples in total. However, based on our manual checking, the remaining 9 samples do not contain the payload indicated by the report and, therefore are excluded from the experiment.

confirming both their use of obfuscation and the effectiveness of our approach in counteracting it, as described in §2.2.2. This only exception was the `Kuguo` family. Some of its identified payloads employ a sophisticated obfuscation technique called package flattening. This method completely disrupts the Java package structure by dispersing all code files across the root directory, making manual verification of payload package similarities challenging. However, we did identify some payloads within this family that were not subjected to such obfuscation, with the package name `com.kuguo` that corresponds to its family name. This partial confirmation allows us to verify at least some correct detections for this family. In conclusion, our evaluation indicates that the precision of the detected adware family payloads within DEX files ranges from **96.7%** to **100%**.

Payloads within resource folders Among the 30 evaluated adware families, **8** were detected to contain payloads within resource folders. Specifically, 3 families had payloads in the form of additional APK files, 1 in additional DEX, 2 in JAR, and 2 in SO files. Notably, these payloads were observed across multiple samples within their respective families, and through decompilation and manual analysis, we also confirmed that these files indeed contained code related to adware behaviors, thus validating their classification as family payloads.

4.2 Adware family payload location result

The distribution of payload locations across adware families offers crucial insights into the implementation strategies preferred by adware developers. Our analysis successfully identified the payload locations for **111** out of **118** adware families within our dataset. For the remaining 7 families we failed to locate payloads, one were due to the adoption of app packing techniques that hinder reverse engineering efforts, another adopted sophisticated obfuscation techniques, namely package flattening, which completely disrupts the Java package structure by dispersing all code files across the root directory, rendering our approach ineffective. Additionally, the “Mulad” family integrates multiple ad networks, resulting in the absence of a fixed payload. Consequently, separate scanning of any individual package within a “Mulad” sample fails to trigger identification as “Mulad”. For the remaining four families, we were unable to locate payloads, and the reasons for this eluded our analysis. We hypothesize that these families may employ payloads dynamically loaded during app runtime, or that antivirus engines identify these families based on cross-package relationships that our approach does not account for.

We categorized the 111 families based on their payload locations, as illustrated in Fig. 2. Our analysis reveals that an overwhelming majority of **107** families (96.4%) contained payloads within their DEX files[10], representing the most prevalent approach. Furthermore, **36** families (32.4%) housed their payloads in local resources. Notably, **32** families (28.8%) exhibited a hybrid strategy, integrating payloads in both Java code and local resources. This significant proportion of hybrid implementations

[10] This refers to the main classes*.dex files within the root directory of the APK. DEX files found in resource folders are categorized under ”Resource” as they are loaded dynamically.

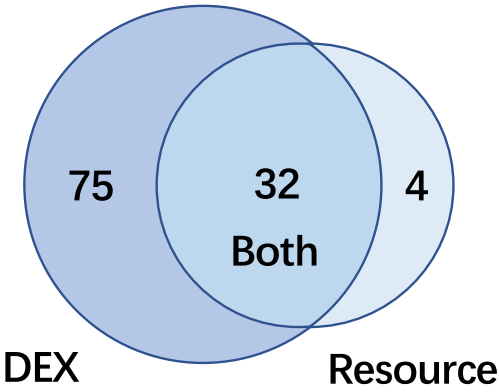


Fig. 2 Distribution of payload locations

indicates a trend towards more sophisticated and potentially more resilient adware architectures.

4.2.1 Payloads within DEX files

More than 90% adware families incorporate their payloads within the DEX files of the APK. This means they are implemented in Java/Kotlin, aligning with common Android development practices. For these families, we also identified the most frequently recurring package names associated with their payloads, as exemplified in Table 1. However, upon closer examination, several noteworthy findings reveal that the nature of these payloads is more complex than anticipated.

Package name obfuscation As mentioned in §2.2.2, adware payloads often obfuscate their package names to evade detection. We hence characterize this phenomenon. Recall that we identify payloads in the form of feature hashes, which are calculated from the adware samples' second-level packages (e.g., com/a). For each feature hash identified as payload, we locate all packages with this hash across the adware family's samples and compare their package names. Our analysis revealed that **31** families exhibit different package names for payloads with identical feature hashes, providing concrete evidence of payload package name obfuscation. Notably, many of these obfuscated names convey minimal semantic information, likely an attempt to obscure their origin, such as the advertising network company behind them. This approach starkly contrasts with common ad networks, which often use explicit SDK package names like `"com/facebook/ads"` or `"com/amazon/device/ads"`. For instance, consider the "leadbolt" adware family. we observed its obfuscated payloads `"com/qzmpeyumuvalcpblod"` and `"com/dscevemibbpwdfxe"` sharing the same feature hash. These package names, however, offer no meaningful indications about the payload's functionality or origin.

Payload version diversity within families Next, we investigated whether multiple different feature hashes exist for payloads with the same package name within a family, indicating the presence of different payload versions. Our analysis identified **47** families exhibiting multiple payload versions for the same payload package name.

A particularly intriguing case is the `inmobi` family, which is named after a prominent yet not infamous Indian multinational mobile advertising network (inmobi 2024), an uncommon situation for adware families. Within our dataset, we identified four distinct versions of its payload. Further investigation revealed that these payloads are a batch of early InMobi SDK versions dating back to 2013, which exploited a then-existing Android WebView vulnerability to access sensitive privileged information such as photos, SMS, and even make phone calls (javascript-interfaces 2022). This vulnerability has been fixed since Android 4.2 (JavascriptInterface 2024), and InMobi has long ceased such practices. Only a fraction of their SDK versions that exploited this vulnerability were identified as adware.

This case demonstrates that even renowned advertising networks have, at specific periods in their development history, adopted practices that could be perceived as privacy-invasive or potentially malicious. It highlights the complexity of the mobile advertising ecosystem and the challenges advertising libraries face in balancing functionality with user privacy.

Hidden in common packages In addition to package name obfuscation, our investigation uncovered another prevalent strategy employed by adware payloads to evade detection: placing themselves under directories of widely used and trusted packages, such as `com.google` or `android`. This tactic likely aims to gain the trust of security personnel or exploit the whitelists of certain virus detection engines. Our analysis revealed that **17** adware families employed this sophisticated method, strategically positioning their payloads under common package names. Some families went even further, distributing their code across multiple common packages. Figure 3 illustrates the distribution of common packages under which these adware families hide their payloads.

4.2.2 Payloads within resource folders

Our analysis revealed that **36** adware families carry payloads located within app resource folders. Of these, **8** families utilized SO files (Android C++ library files). These files are notably more challenging to analyze statically (secret-inside-of-so-file 2024) compared to DEX files, making them an effective vehicle for adware payloads

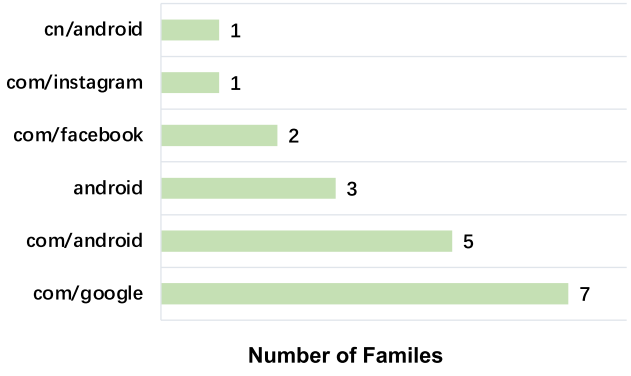


Fig. 3 Common packages adware payloads hide under

to conceal their intended behaviors. However, their development complexity likely contributes to their relatively low prevalence in our dataset, suggesting that creating SO libraries remains a hurdle for many adware payload developers.

Dynamic loading emerged as another prevalent evasion technique. This method typically involves loading JAR, DEX, or APK files during runtime to circumvent antivirus software checks. Our investigation uncovered **14** families with JAR and DEX files in their resource folders. Furthermore, **17** families contained additional APK payloads, with 6 of these families obfuscating the file extensions of the APKs to further evade detection.

Intriguingly, we identified **7** families where merely uploading the Manifest files resulted in an adware flag. This finding suggests that some antivirus engines adopted semantic checking of the Manifest file for detection.

Our analysis of adware family payload location reveals diverse payload implementation strategies, with 96.4% using DEX files, 32.4% utilizing resource folders, and 28.8% adopting a hybrid approach. We identified key evasion techniques including package name obfuscation, hiding under trusted package names, and employing SO files and dynamic loading. These findings provide a crucial understanding of adware development strategies, offering valuable insights for future adware detection and analysis.

5 Adware characterization

After constructing a comprehensive Android adware dataset and locating payloads for numerous adware families, this section provides an in-depth characterization across these families while uncovering the ”arts and crafts” that define their deceptive practices and monetization tactics. To this end, we first present an overview of our **Adware Characterization Schema** and offer a comprehensive characterization across all analyzed adware families (§5.1). We then examine each component in detail: Adware Survival Mechanisms (§5.2), Detection Evasion Mechanisms (§5.3), Fraudulent Ad Presentation Forms (§5.4), and Fraudulent Ad Behavioral Patterns (§5.5). Through this systematic analysis, we reveal how adware combines sophisticated techniques to maximize revenue while evading security measures.

5.1 Adware characterization schema

Recalling the schema construction methodology described in §2.3, we leveraged 37 security reports to gain a preliminary understanding of adware code structure. This initial schema provided the foundation for the characterization mechanisms and was continuously refined throughout our manual analysis process. Based on this methodology, we analyzed all adware families included in our dataset. However, due to our inability to locate payload locations in certain samples, heavily obfuscated code in some samples, or the dependency on remote servers for complete payload delivery in others, we successfully analyzed **92** families in total. Notably, for 6 families with multiple variants—`hiddad`, `hiddenad`, `hiddenads`, `mocen`, `ewind`, and

clicker, we conducted separate analyses for each variant to capture their distinct characteristics.

As depicted in Fig. 4, our proposed **Adware Characterization Schema** provides a comprehensive framework for analyzing modern adware families. The schema categorizes adware behaviors into 3 primary dimensions: survival mechanisms, detection evasion techniques, and advertising fraud mechanisms. In this context, advertising fraud refers to deceptive practices through either fraudulent ad presentation (such as aggressive advertisements continuously appearing on the home screen) or fraudulent ad behaviors (such as simulating user clicks on advertisements in the background without user awareness). Each dimension captures distinct strategies employed by adware to maintain persistence, avoid detection, and generate illicit revenue.

Overview of adware characterization distribution Our comprehensive analysis encompassed **92** adware families and **30** distinct variants, revealing patterns in how modern adware operates.

As illustrated in Fig. 5, the distribution of techniques across our characterized dimensions shows that privacy information leakage (74 families) dominates fraudulent behaviors due to its implementation simplicity, while system event listeners (58 families) serve as the primary persistence mechanism. Detection evasion techniques are distributed relatively evenly across four main approaches: App Environment Detection (22 families), Execution Timing Selection (19 families), Icon Disguise (17 families), and Device Environment Detection (15 families). Click fraud techniques appear less frequently (9 families total across Java, JavaScript, and network traffic implementations), representing more complex attack vectors requiring advanced implementation skills. Various forms of Out-of-app advertising (including browser, interstitial, and overlay variants) also constitute a significant portion of fraudulent presentation techniques, as they are highly favored by attackers for their effectiveness in generating illicit revenue.

Distribution patterns in top adware families Due to space constraints, we present the top 20 families that received the highest number of "adware" classifications from security vendors in VirusTotal scans, as shown in Table 3.

We found that 7 families implement 5 or more techniques across categories, demonstrating sophisticated adware capabilities. 9 families employ both survival and evasion mechanisms, showing a strategic focus on maintaining presence despite detection attempts. Only 5 families (ashas, kalfere reefwal, walkFree, gappusin) implement techniques across all four major categories (fraudulent advertising forms, fraudulent advertising behaviors, survival mechanisms, and evasion mechanisms), representing the most comprehensive adware toolkits. Regarding specific techniques, 16 families utilize system event monitoring for persistence, 7 families leverage app environment checking for evasion, and 13 families collect private information. These patterns suggest an ecosystem where sophisticated adware increasingly combines multiple techniques to maximize both effectiveness and resilience.

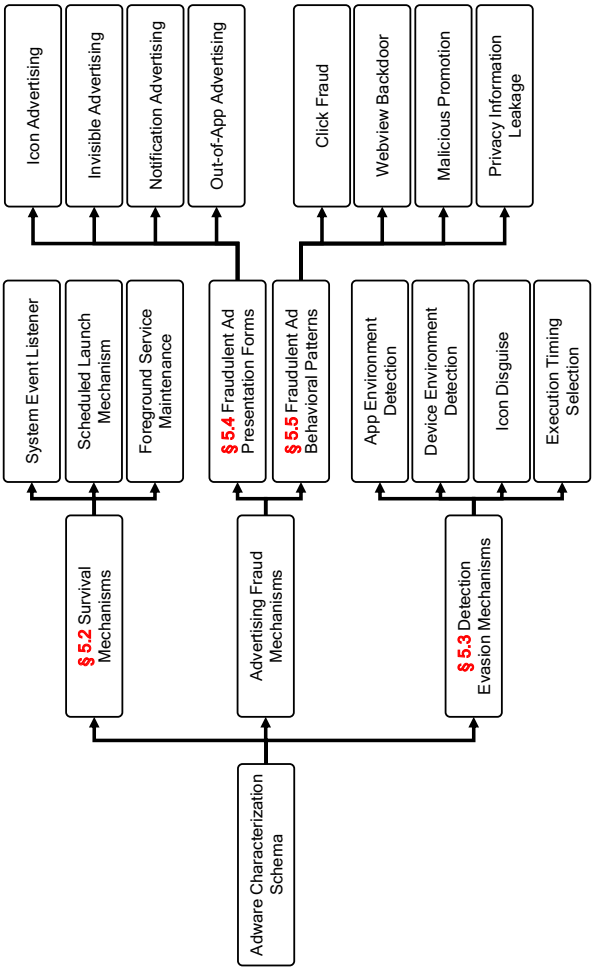


Fig. 4 Adware characterization schema

Advertising Fraud Mechanisms	Families
Fraudulent Ad Presentation Forms	42
Notification Advertising	26
Icon Advertising	25
Out-of-App - Interstitial Advertising	17
Out-of-App - Browser Advertising	19
Invisible Advertising	6
Out-of-App - Overlay Advertising	5
Fraudulent Ad Behavioral Patterns	82
Privacy Information Leakage	74
Malicious Promotion	30
Webview Backdoor	14
Click Fraud - Java	4
Click Fraud - JavaScript	3
Click Fraud - Network Traffic	2

Detection Evasion Mechanisms	Families
Device Environment Detection	15
Device	8
Battery Status	7
Network	5
Location	3
Debug	2
App Environment Detection	22
Time After Installation	13
Wakelock	11
Special Permission	10
Icon Disguise	17
Execution Timing Selection	19

Survival Mechanisms	Families
Scheduled Launch Mechanism	30
Foreground Service Maintenance	6
System Event Listener	58
Boot	36
Screen On / Off	27
Package Add / Remove	27
Network	12
Charge	6
Call Status	4

Fig. 5 Distribution of mechanism implementations across adware families

Table 3 Characterization of the top 20 families with the highest average number of adware classifications by security vendors

Shorthand Explanation	Fraudulent Ad Presentation Forms	Out Of App ad	OOA Browser ad (B) OOA Overlay ad (O) OOA Interstitial ad (I)
	Fraudulent Ad Behavioral Patterns	Click Fraud	Java Type (J) JavaScript Type (S) Network Traffic Type (T)
	Survival Mechanisms	System Event Listener	Screen (S) Boot (B) Call Status (C) Network (N) Battery (BA) Package (P)

id	Family Name	Fraudulent Ad Presentation Forms				Fraudulent Ad Behavioral Patterns				Survival Mechanisms			Detection Evasion Mechanisms			
		Notification Ad	Invisible Ad	Icon Ad	Out Of App Ad	Click Fraud	Webview Backdoor	Info Leak	Malicious Promotion	Scheduling	Foreground Service	System Event Listener	Icon Disguise	Execution Timing	Device Environment Detection	App Environment Detection
1	noiconads				B/I					√		S/B	√	√		√
2	zdtad			√	B/I					√		S/B/P/N		√		√
3	apofer								√			B/P				
4	appoffer								√			B/P				
5	adviator							√				B/N				√
6	callflakes						√	√				C/P				
7	davitor							√				B/N				√
8	dowgin							√								
9	fictus							√					√			
10	gappusin			√			√	√	√			P/N				
11	kalfere	√			B			√	√			B/P		√		√
12	reefwal	√			B			√	√			B/P		√		√
13	suad							√		√		S/B		√		
14	Tekya					J						S/B				
15	walkFree	√			B			√	√			B/P		√		√
16	wkload								√							

Table 3 (continued)

Shorthand Explanation	Fraudulent Ad Presentation Forms	Out Of App ad	OOA Browser ad (B) OOA Overlay ad (O) OOA Interstitial ad (I)
	Fraudulent Ad Behavioral Patterns	Click Fraud	Java Type (J) JavaScript Type (S) Network Traffic Type (T)
	Survival Mechanisms	System Event Listener	Screen (S) Boot (B) Call Status (C) Network (N) Battery (BA) Package (P)

id	Family Name	Fraudulent Ad Presentation Forms				Fraudulent Ad Behavioral Patterns				Survival Mechanisms			Detection Evasion Mechanisms			
		Notification Ad	Invisible Ad	Icon Ad	Out Of App Ad	Click Fraud	Webview Backdoor	Info Leak	Malicious Promotion	Scheduling	Foreground Service	System Event Listener	Icon Disguise	Execution Timing	Device Environment Detection	App Environment Detection
17	ashas			√	I			√		√		S	√	√	√	√
18	floodad							√	√			C				
19	gomunc	√			B							S				
20	gunpoder							√							√	

The table is organized according to four major analytical dimensions in our schema: (1) Fraudulent Ad Presentation Forms, (2) Fraudulent Ad Behavioral Patterns, (3) Survival Mechanisms, and (4) Detection Evasion Mechanisms. Each subcolumn lists the corresponding behavioral features identified in the top 20 adware families

Distribution patterns in adware family variants The Table 4 presents 30 variants across 6 adware families, revealing significant technical diversity both within and between families. Analysis shows that 21 variants (70%) collect private information, while 18 (60%) implement system broadcast monitoring for persistence. Interstitial ads appear in 17 variants (56.7%), making them the most common advertising type. For evasion, app condition checking (13 variants) and execution timing (12 variants) are most prevalent.

Within families, `hiddad` family exhibits the greatest technical diversity with its 12 variants employing different combinations of techniques, suggesting ongoing evolution of this particularly adaptable family. The `mocen` family demonstrates specialization, with all 4 variants focusing primarily on click fraud implementations. The `hiddenad` family shows consistency in information collection (100% of variants) but varies in persistence mechanisms.

Cross-family analysis reveals that sophisticated variants typically combine techniques from at least three categories, with `Hiddad.Poseidon`, `Hiddad.Scyll`, and `HiddenAds.I` among the most technically complex implementations. Only 5 variants (16.7%) implement techniques across all four major categories (fraudulent advertising forms, fraudulent advertising behaviors, survival mechanisms, and evasion mechanisms), indicating that comprehensive capability remains relatively uncommon even among established adware variants.

Behavioral differences across adware families From the perspective of the top adware families, as summarized in Table 3, several representative examples illustrate distinct behavioral characteristics. NoIconAds, for instance, employs scheduled tasks and system-event listeners such as Screen (S) and Boot (B) events to trigger its payload activities. By disguising its launcher icon, it displays out-of-app advertisements (OOA Ads) from unknown sources, effectively concealing its origin. Tekya similarly monitors Screen (S) and Boot (B) events but focuses on performing Java-type click-fraud operations, simulating user interactions to generate fraudulent ad clicks. These examples highlight that among adware families, the triggering logic and payload behavior vary notably.

From the variant-level perspective, as shown in Table 4, families such as HiddenAd, HiddAd, and HiddenAds share a common reliance on event listeners or timers to initiate malicious ad activities. They typically perform different forms of out-of-app advertising, including OOA Browser (B), OOA Overlay (O), and OOA Interstitial (I) ads, or generate Network Traffic (T) type click-fraud by automatically sending simulated click traffic. In contrast, Clicker and Mocen adopt Java-type (J) or JavaScript-type (S) click-fraud schemes, while Ewind combines multiple behavioral strategies encompassing both ad delivery and traffic generation.

Overall, these analyses demonstrate that adware families exhibit diverse payload-triggering mechanisms, ad-presentation strategies, and fraudulent behaviors, each showing distinct family-specific signatures.

Table 4 Characterization of 30 adware variants representing 6 adware families

Shorthand Explanation	Fraudulent Ad Presentation Forms		Out Of App ad			OOA Browser ad (B) OOA Overlay ad (O) OOA Interstitial ad (I)										
	Fraudulent Ad Behavioral Patterns		Click Fraud			Java Type (J) JavaScript Type (S) Network Traffic Type (T)										
	Survival Mechanisms		System Event Listener			Screen (S) Boot (B) Call Status (C) Network (N) Battery (BA) Package (P)										
id	Family Name	Variant Name	Fraudulent Ad Presentation Forms				Fraudulent Ad Behavioral Patterns				Survival Mechanisms			Detection Evasion Mechanisms			
			Notification Ad	Invisible Ad	Icon Ad	Out Of App Ad	Click Fraud	Webview Backdoor	Info Leak	Malicious Promotion	Scheduling	Foreground Service	System Event Listener	Icon Disguise	Execution Timing	Device Environment Detection	App Environment Detection
1	ewind	Ewind.Judy		√		I	J/S	√	√		√		S/B/P				
2		Ewind.ehewea				B		√	√	√	√		S/B/N				
3		Ewind.h	√	√	√	B/I			√		√		S/B/P	√	√		√
4	clicker	Clicker.Turkish Clicker		√			S		√					√			
5		Clicker.New Clicker	√	√		B	S				√	√	B		√	√	√
6	hiddad	Hiddad.BZ	√	√		I							S			√	
7		Hiddad.Skinner				I			√		√					√	√
8		Hiddad.HidenAd				B/I			√		√	√	S/B	√	√		
9		Hiddad.HRXJA				B/I			√		√		S	√	√	√	√
10		Hiddad.Poseidon				I	T		√		√	√	S/B/C	√	√	√	√
11		Hiddad.AdBeauty				B/I					√		S	√	√		√
12		Hiddad.Scylla		√	√	I	T		√		√	√	S		√	√	
13		Hiddad.Ghostclicker				I	J		√		√				√	√	
14		Hiddad.AIK		√			J/T				√		B	√			
15		Hiddad.DR				B/I		√	√		√	√	S/N/BA	√		√	
16		Hiddad.E				B		√		√	√		S/N/				
17		Hiddad.HP				I							S				

Table 4 (continued)

Shorthand Explanation	Fraudulent Ad Presentation Forms		Out Of App ad			OOA Browser ad (B) OOA Overlay ad (O) OOA Interstitial ad (I)										
	Fraudulent Ad Behavioral Patterns		Click Fraud			Java Type (J) JavaScript Type (S) Network Traffic Type (T)										
	Survival Mechanisms		System Event Listener			Screen (S) Boot (B) Call Status (C) Network (N) Battery (BA) Package (P)										
id	Family Name	Variant Name	Fraudulent Ad Presentation Forms				Fraudulent Ad Behavioral Patterns				Survival Mechanisms			Detection Evasion Mechanisms			
			Notification Ad	Invisible Ad	Icon Ad	Out Of App Ad	Click Fraud	Webview Backdoor	Info Leak	Malicious Promotion	Scheduling	Foreground Service	System Event Listener	Icon Disguise	Execution Timing	Device Environment Detection	App Environment Detection
18	hiddenad	Hiddenad.AD			√				√	√	√		S/N/BA				
19		Hiddenad.BF	√		√				√	√	√		S/P	√		√	
20		Hiddenad.DZ				B/I			√		√	√	S/B	√			
21		Hiddenad.J				I			√		√		B	√			
22		Hiddenad.JO			√	O/B/I			√		√		S/BA		√	√	√
23		Hiddenad.QY				I					√		S/P	√	√		
24		Hiddenad.UP				I					√	√	S				
25	hiddenads	Hiddenads.AAC		√		B			√				S/B	√	√		
26		HiddenAds.I	√		√	I			√	√	√	√	P	√	√		√
27	mocen	Mocen.Click					S		√	√				√		√	
28		Mocen.Haken Clicker					J				√		B/P/N				
29		Mocen.BoostClicker					J			√				√			
30		Mocen.Bear Clicker		√			S										

Comparison with existing malware schemas Previous malware characterization research, such as Cao et al. (2022), has briefly mentioned ad-related behaviors, summarizing them under general ad abuse categories such as aggressive advertising, hidden ads, and automated ad clicks. In these studies, ad-related behaviors were treated as a subcategory within broader malware analysis, covering only limited forms of advertising abuse. These works provided valuable early insights into adware behaviors but primarily focused on describing functional phenomena rather than developing a systematic behavioral framework or examining detailed implementation mechanisms.

In contrast, our schema provides a comprehensive and structured framework specifically tailored to adware, capturing its monetization driven nature through two complementary analytical dimensions: Fraudulent Ad Presentation Forms and Fraudulent Ad Behavioral Patterns. The presentation forms dimension characterizes how adware delivers illicit advertisements through notification, invisible, out of application, and icon based techniques, each implemented through distinct technical mechanisms designed to maximize ad exposure. The behavioral patterns dimension focuses on actively executed malicious behaviors that manipulate advertising logic and user interactions, encompassing three distinct click fraud techniques (Java type, JavaScript type, and network traffic type) as well as privacy violations, WebView backdoors, and malicious promotion behaviors.

Together, these two dimensions reveal the technical structure and implementation features of adware. Our schema uncovers adware's unique operational model centered on deceptive monetization and targeted evasion, and establishes a specialized analytical foundation that extends beyond existing general malware frameworks.

5.2 Survival mechanisms

Adware, like other types of malware, requires persistence mechanisms to ensure continued operation on infected devices. These survival mechanisms enable adware to restart after device reboots, recover from system-initiated termination, and maintain operation despite battery optimization measures. Our analysis reveals three primary mechanisms widely employed across adware families to achieve longevity.

5.2.1 System event listener

System Event Listeners leverage Android's broadcast receiver system to monitor system-wide events. This mechanism enables adware to automatically restart by registering for specific system broadcasts such as device boot completion, application installations, screen unlocks, or network connectivity changes. When these events occur, the system automatically activates the registered components, allowing adware to resume operation without requiring explicit user interaction.

This approach is particularly effective because it integrates with Android's core functionality, making it difficult to disable without affecting legitimate applications. The broadcast system was designed to allow applications to respond to sys-

tem events, but adware exploits this mechanism to ensure persistent operation across device states and user activities.

5.2.2 Scheduled launch mechanism

Scheduled Launch Mechanisms utilize Android's job scheduling subsystems to periodically activate adware components. By registering recurring tasks through JobScheduler, WorkManager, or AlarmManager APIs, adware can ensure regular execution intervals regardless of user activity. These scheduling APIs allow applications to specify timing constraints, network conditions, and device states under which tasks should execute.

While originally designed to optimize battery usage by coordinating background tasks, these scheduling mechanisms provide adware with reliable activation windows. The system handles the complexity of determining optimal execution times, potentially waking devices from sleep states to ensure scheduled tasks complete, thereby guaranteeing adware operation at regular intervals.

5.2.3 Foreground service maintenance

Foreground Service Maintenance represents one of the most resilient survival techniques. Android differentiates between background and foreground processes, applying aggressive battery optimization to the former while allowing the latter more operational freedom. By establishing a foreground service with an associated notification, adware gains priority treatment from the system resource manager.

Foreground services are designed for user-facing operations that should continue regardless of application focus state, such as music playback or navigation. Adware exploits this designation to maintain continuous operation, as the system will attempt to preserve foreground services even under memory pressure. Additionally, foreground services can implement restart mechanisms that trigger when the system terminates them, creating a persistent execution environment that significantly enhances adware longevity.

5.3 Detection evasion mechanisms

Adware implements sophisticated evasion techniques to avoid detection by security tools, researchers, and users. These mechanisms help adware remain undetected, prolonging its operation on infected devices and complicating analysis efforts. Our research identified four primary evasion strategies employed across adware families.

5.3.1 App environment detection

App Environment Detection involves monitoring the application's operational context to determine when it's safe to execute malicious behaviors. Adware examines various application-specific conditions before activating its core functionality. This typically includes verifying whether requested permissions have been approved before executing sensitive operations, which prevents premature exposure of malicious intent.

Many samples implement installation time checks, deliberately delaying malicious behavior for a predetermined period after installation to evade time-based analysis and establish a reputation as benign. Additionally, adware often verifies WakeLock acquisition before proceeding with critical operations, ensuring the system won't terminate the app during key activities. This comprehensive approach allows adware to establish operational prerequisites before executing suspicious behaviors, reducing the likelihood of detection during initial analysis periods or when operating under constrained permissions.

5.3.2 Device environment detection

Device Environment Detection techniques evaluate the broader device environment to identify potential analysis scenarios. Advanced adware examines multiple device characteristics to determine whether it's operating on a genuine user device. This includes sophisticated emulator and virtual machine detection through hardware identifiers, sensor behavior analysis, or identification of virtualization artifacts. Many samples implement IP address and geolocation verification, activating only in specific geographic regions or when connected to particular network types to avoid researcher environments. Power state monitoring is also common, with adware altering its behavior based on battery level or charging status to avoid detection during forensic analysis, which often occurs in laboratory environments with devices connected to power sources. These techniques enable adware to distinguish between real user devices and analysis environments, selectively executing malicious functionality only in scenarios deemed safe for operation.

5.3.3 Icon disguise

Icon Disguise techniques manipulate application visibility to reduce user awareness and complicate removal attempts. Sophisticated adware frequently removes its application icons from launcher interfaces while maintaining background operation, preventing casual discovery by device owners. When icons remain visible, adware often employs icon obfuscation using legitimate-appearing icons or names to blend with system applications, creating confusion about the source of unwanted behavior. Many implementations also prevent the application from appearing in the device's recent applications interface, eliminating a common method users employ to identify and terminate problematic apps. By reducing visibility across these multiple vectors, adware can operate without drawing user attention, decreasing the likelihood of manual detection and removal efforts while maintaining persistent execution.

5.3.4 Execution timing selection

Execution Timing Selection strategically determines when and how frequently adware behaviors execute. Rather than displaying advertisements immediately or continuously, sophisticated adware modulates display frequency to avoid overwhelming users, which might prompt investigation and removal. Operation postponement is widely implemented, with malicious behavior execution delayed to establish a period

of apparently benign operation that builds user trust. Many samples also vary execution patterns, altering the timing and sequence of operations to evade pattern-based detection systems that look for consistent behavioral signatures. This approach helps adware balance monetization goals against detection risk by introducing variability that complicates both static and dynamic analysis techniques. The randomization and contextual awareness of these timing strategies represent an evolutionary response to increasingly sophisticated security tools designed to identify suspicious application behavior.

5.4 Fraudulent advertising presentation forms

The effectiveness of adware largely depends on its ability to deliver advertisements in ways that maximize user exposure while minimizing detection and removal. Our analysis reveals a diverse ecosystem of fraudulent ad presentation techniques that go beyond legitimate advertising practices. These techniques are specifically designed to circumvent platform protections, deceive users, and generate revenue regardless of user experience.

Fraudulent ad presentation represents a significant evolution beyond traditional advertising models, employing technical mechanisms to force ad visibility, manipulate user interaction, and extract value without respecting user consent or platform guidelines. These presentation forms vary in their implementation complexity, visual obtrusiveness, and effectiveness at generating fraudulent revenue. By understanding these presentation mechanisms, developers and security researchers can better identify and mitigate the impact of adware across the Android ecosystem.

5.4.1 Notification advertising

Notifications in Android refer to a type of information prompt that the system or application sends to the user, which usually consists of elements such as icons, titles, and content and is displayed in the notification bar. But malicious developers often abuse the notification bar for advertising purposes (Liu et al. 2019).

We systematically installed identified adware samples on test devices and manually triggered their activation conditions by adjusting device settings, time parameters, and network configurations. This hands-on approach allowed us to directly observe how these applications exploit the notification mechanism. Our experiments revealed that notification bar advertisements typically employ sophisticated social engineering techniques to display deceptive content and entice users to click on them. We documented numerous instances where notifications mimicked system alerts, security warnings, or urgent messages to increase click-through rates. By tracing the complete interaction flow, we observed that once users click these ads, they are redirected to a browser to open promotional pages or, in more severe cases, trigger a drive-by download attack.

5.4.2 Invisible advertising

In digital advertising, an impression constitutes the successful rendering of an advertisement to users, serving as a fundamental metric for quantifying ad exposure and effectiveness. Application developers have financial incentives to maximize impression counts, as advertiser payments are often directly correlated with impression volume.

This economic model creates a fundamental tension: while increased ad frequency generates higher revenue, excessive visible advertisements significantly degrade user experience, potentially leading to application abandonment. To circumvent this limitation, sophisticated adware developers implement mechanisms to render advertisements invisible to users while still triggering impression counting mechanisms on advertising platforms. This technique enables the generation of substantial fraudulent impression revenue while maintaining normal user experience.

Through our analysis, we identified two primary technical approaches for implementing invisible ad fraud:

(1) Manipulation of view visibility attributes: Developers exploit the Android View system by explicitly setting the visibility property of ad components to either *View.INVISIBLE* or *View.GONE* after impression registration has occurred. For example, our examination of the `Hiddad.Scylla` family revealed systematic implementation of this technique, where ad views are momentarily loaded to register impressions before being programmatically hidden from user perception.

(2) Implementation of imperceptible view dimensions: An alternative approach involves configuring ad view components with dimensions that render them effectively invisible (typically 0×0 or 1×1 pixels). Our analysis of the `Ewind.Judy` family documented sophisticated implementations where 1×1-sized WebView instances systematically load advertiser-specified URLs and execute JavaScript code, triggering impression counts while remaining imperceptible to users.

Significantly, our research indicates that invisible ad implementations frequently serve as a foundation for more complex click fraud operations. These implementations typically enable JavaScript execution within invisible WebViews by activating the *setJavaScriptEnabled* property. Furthermore, many samples register custom *JavascriptInterface* bridges, establishing bidirectional communication channels that allow JavaScript functions to invoke registered Java methods annotated with *@JavascriptInterface*. The detailed implementation mechanisms and implications of these JavaScript-based fraud techniques are comprehensively examined in §5.5.1.

5.4.3 Out of application advertising

Out-of-Application (OOA) advertising refers to the delivery of promotional content to users when the originating application is not actively running in the foreground. This technique deliberately circumvents the conventional application lifecycle boundaries to maximize ad exposure, resulting in disrupted user experience and generating fraudulent revenue streams for developers. Through our comprehensive analysis, we identified three distinct technical implementations of OOA advertising:

(1) OOA interstitial advertising Interstitial advertisements occupy substantial screen real estate (typically full-screen or half-screen) and are displayed to users in a pop-up window during the use of an application, which may interrupt the user's experience. Malicious developers deploy these high-impact ad units outside their originating application to generate more ad revenue. The technical implementation relies on invoking the *startActivity()* method from an out-of-application context, configured with the *FLAG_ACTIVITY_NEW_TASK* flag. This configuration enables the ad container to manifest as a separate activity outside the application's UI hierarchy.

In our sample analysis, `ashas` demonstrated particularly aggressive behavior, immediately displaying a half-screen advertisement to the user upon unlocking their smartphone, regardless of whether the application is currently running. Another notable variant, `Hiddad.HRXJA`, implements a time-based triggering mechanism that displays OOA advertisements every 15 minutes, and immediately closes the advertisement in the callback after it is loaded, causing a strange flickering phenomenon.

(2) OOA overlay advertising Android floating window is a type of window that allows overlaying on top of other application interfaces. Starting from Android 6.0, the permission to request floating window, *SYSTEM_ALERT_WINDOW*, has been classified as a special permission and requires runtime permission request during application runtime.

Our technical analysis identified multiple sophisticated implementations of this approach. `fakeadblocker` periodically introduces circular ad views on the user's screen, allowing them to be dragged and clicked for redirection to a browser. It even continues displaying ads persistently when the phone is in sleep mode. On the other hand, `solid` mimics the lockscreen interface and overlays advertisements on top of it. Disguised as an antivirus software, `solid` requests the *SYSTEM_ALERT_WINDOW* permission and listens for the user's screen off and screen on actions to display advertisements.

(3) OOA browser advertising This implementation leverages system-level intent mechanisms to spawn browser instances displaying advertising content outside the application context. Specifically, developers utilize *Intent("android.intent.action.VIEW", Uri.parse(url))* to open a browser and display the ad page, which may be a normal website with advertising content, and the purpose is to attract traffic and increase the revenue of website developers. Alternatively, it may be their own ad inventory, which often contains fraudulent content and increases the revenue of malicious adware developers.

Our analysis of the `gomunc` sample revealed sophisticated monitoring of the *Intent.ACTION_USER_PRESENT* event, which occurs when the user unlocks the device, and displays advertisements by opening a browser. This strategic timing ensures maximum visibility by triggering browser-based advertisements precisely when users initiate device interaction sessions.

5.4.4 Icon advertising

Android shortcuts are typically presented in the form of application icons, and clicking on them directly leads to specific screens, performs specific actions, or triggers specific functionalities within an application. This system component is designed to enhance user experience by providing rapid access to frequently used application features.

Malicious developers systematically exploit this legitimate Android feature to generate fraudulent advertising revenue. Our analysis revealed a sophisticated technique combining icon-based advertising with deliberate concealment of the application's primary icon from the device launcher. This implementation creates a deceptive user experience where shortcuts appear to represent standalone applications, but actually function as advertising delivery mechanisms. When users interact with these deceptive shortcuts, they are redirected to webpages populated with high-density advertising content. A particularly insidious aspect of this implementation is that removal of these shortcuts has no impact on the persistence of the host application, allowing continued exploitation even after users attempt remediation.

In our analysis, `Hiddenads.I` demonstrated an advanced implementation leveraging the VirtualApp framework to establish a sophisticated exploitation chain. This sample installs secondary applications as plugins within the host application's sandbox environment and generates corresponding shortcuts on the device home screen. When users interact with these shortcuts, the malware launches the associated plugin application within its containerized sandbox, allowing developers to fraudulently monetize both the installation metrics and subsequent activation events of these concealed applications.

5.5 Fraudulent advertising behavioral patterns

In this section, we analyze the distinct behavioral signatures we identified across our dataset, revealing how adware orchestrates deceptive interactions with advertising networks, manipulates user engagement metrics, and circumvents existing detection mechanisms to maximize fraudulent monetization while minimizing visibility to both users and security solutions.

5.5.1 Click fraud

In the digital advertising ecosystem, developers generate revenue not only from impression-based metrics but also through user click interactions, with the latter commanding significantly higher compensation rates. This economic incentive has led malicious actors to implement sophisticated click fraud mechanisms to generate artificial interactions, thereby obtaining illicit profits. Click fraud represents the most consequential form of advertising abuse due to its direct financial impact on advertisers and profound disruption of the advertising ecosystem's integrity. Through our systematic analysis, we have identified three distinct technical implementations of click fraud.

(1) Java type click fraud At the Java code level, malicious developers implement programmatic simulations of legitimate user interactions with advertisement elements. This technique typically involves two critical components: identifying advertising view elements within the application's view hierarchy, and subsequently triggering synthetic click events against these elements. The identification of advertisement view elements presents significant technical challenges, as most advertising SDKs employ aggressive obfuscation techniques and dynamic loading mechanisms to protect their view instances. We observed a spectrum of approaches to overcome these protections:

`Hiddad.Ghostclicker` implements a relatively rudimentary approach by first obtaining the root view instance of the current window through *findViewById()*, and subsequently performing depth-first traversal operations to identify WebView instances within the hierarchy, as illustrated in Fig. 6.

However, as advertising libraries have strengthened their defensive countermeasures, the class names of advertising views have become increasingly obfuscated, and the implementation types have diversified beyond simple WebView instances. In response, `Tekya` employs a more sophisticated methodology. This variant first performs runtime monitoring to capture the obfuscated function signatures specific to AdMob banner advertisements. Subsequently, it obtains the application's root view instance and executes targeted traversal operations to identify elements matching these captured signatures. Notably, these complex view identification and interaction operations are implemented within native code (SO libraries) to evade detection mechanisms operating at the Java layer.

Further advancing this technique, `Hiddad.Scylla` implements a behavioral mimicry approach by recording the coordinates and timing patterns of legitimate user click interactions during normal application usage. These interaction patterns are subsequently replayed against identified advertisement elements, creating synthetic engagements that more effectively simulate authentic user behavior patterns.

```
01.    public static webView m2201d(view view) {
02.       if (view instanceof webview) {
03.          return (webview) view;
04.       }
05.       if (view instanceof ViewGroup) {
06.          int childcount = ((viewGroup) view).getchildCount();
07.          for (int i = 0; i < childCount; i++) {
08.             webView d = m22o1d(((viewGroup)view).getchildAt(i));
09.             if (d != null) {
10.                return d;
11.             }
12.          }
13.       }
14.       return null;
15.    }
```

Fig. 6 Depth-first traversal for WebView detection in `Hiddad.Ghostclicker`

Upon successful identification of advertising view elements, adware must simulate convincing user click actions. Through our analysis, we have identified three distinct implementation methodologies for generating these synthetic interactions:

- **dispatchTouchEvent** Method: This technique is commonly employed to simulate touch events on advertising elements. Malicious developers programmatically construct *MotionEvent* objects with precisely defined coordinates and action types, then dispatch these synthetic events to the advertising view's *dispatchTouchEvent()* method without user awareness or consent. Our analysis of `Hiddad.AIK` revealed sophisticated implementation where the malware constructs paired *MotionEvent* objects to simulate complete interaction sequences—including both initial touch (*ACTION_DOWN*) and subsequent release (*ACTION_UP*) actions. These synthetic interaction events are then dispatched to the WebView instance using the *dispatchTouchEvent()* method, creating a complete simulation of legitimate user engagement.
- **sendPointerSync** Method: Unlike *dispatchTouchEvent()*, which targets specific view instances, the *sendPointerSync()* method belongs to the *Instrumentation* class and is designed to send touch events directly to the currently active window without requiring a specific view object reference. This approach enables the simulation of user click events at arbitrary screen coordinates, providing greater flexibility for fraudulent interactions. `Mocen.Haken Clicker` employs reflection techniques to invoke the *sendPointerSync* method, passing programmatically generated *MotionEvent* objects to simulate user click interactions at coordinates where advertising elements are displayed.
- **ProcessBuilder** Approach: *ProcessBuilder* is a Java class designed for creating and managing operating system processes. Through this interface, external commands can be executed from within a Java application context, with the ability to capture command execution results. In Android development, *ProcessBuilder* is frequently utilized for system-level operations, including shell command execution or launching external applications. `Hiddad.Scylla` implements a particularly sophisticated implementation by first recording legitimate user interaction patterns during normal application usage. Subsequently, it leverages *ProcessBuilder* to execute shell commands—specifically using the *input tap x y* instruction—to distribute synthetic click events that replicate previously captured user behavior patterns, creating highly convincing click fraud that closely mimics natural interaction patterns.

(2) Java script type click fraud WebView is a component provided by Android for displaying web content. It is built on the WebKit engine and is capable of loading and rendering various types of web content, including HTML, CSS, and JavaScript. This versatile rendering capability creates a vector for sophisticated click fraud implementations.

JavaScript-based click fraud demonstrates a strong correlation with the invisible advertisement techniques discussed in §5.4.2. The implementation methodology follows a consistent pattern: malicious developers first instantiate a WebView compo-

nent with dimensions or visibility attributes that render it imperceptible to users. Subsequently, they load both advertising content and specialized JavaScript code into this concealed WebView instance. During execution, the injected JavaScript code programmatically simulates user interaction events by automatically triggering click events on advertisement elements, thereby generating synthetic click traffic. This technique enables malicious developers to systematically accumulate fraudulent revenue through fabricated engagement metrics without any legitimate user interaction.

A representative example is `Clicker.Turkish Clicker`, which employs a particularly sophisticated implementation methodology. During runtime, this variant dynamically downloads both specialized JavaScript code and a collection of target URLs. As illustrated in Fig. 7, the malware systematically loads each target URL within an invisible WebView container and executes the downloaded JavaScript. This script methodically identifies and programmatically interacts with all clickable elements present on the rendered webpage, including embedded advertisements. This approach generates substantial volumes of synthetic click traffic, directing fraudulent revenue to both the website operators and the advertising networks while maintaining complete invisibility to the device user.

(3) Invalid traffic type click fraud Generating false traffic through simulated user operations and network requests represents a direct but readily detectable methodology for implementing click fraud.

- **Directly generate fake traffic.** This approach involves the programmatic creation of simulated user operations and network request patterns to generate fraudulent clicks and impressions. Implementation typically leverages distributed networks or zombie device networks to scale the impact while distributing the origination points. `Hiddad.AIK` implements sophisticated protocol manipulation by

```
01.    function fireEvent(e, n) {
02.       var i = e;
03.       if (document.createEvent) {
04.           var t = document.createEvent("MouseEvents");
05.           t.initEvent(n, !0, !1), i.dispatchEvent(t)
06.       } else
07.       document.createEventObject && i.fireEvent("on" + n)
08.    }
09.
10.    for (var links = document.getElementsByTagName("a"), \
       elmalar = null, i = 0; i0) {
11.       fireEvent(document.links[i], "mouseover"),
12.       fireEvent(document.links[i], "mousedown"),
13.       fireEvent(document.links[i], "click");
14.       break
15.    }
```

Fig. 7 Invisible WebView-based click fraud mechanism in `Clicker.Turkish Clicker`

systematically modifying the identifiers embedded within advertising request packets. This variant programmatically forges request signatures that mimic legitimate user-initiated ad interactions and establishes persistent communication channels to advertising servers. Through these channels, it continuously transmits fraudulent advertising interaction requests, effectively deceiving the advertising delivery infrastructure and generating illicit revenue through synthetic engagement metrics.

- **Disguising as other application bids.** Our analysis revealed a more sophisticated technique exploiting the economic fundamentals of digital advertising marketplaces. Advertisers typically allocate premium budgets for advertisement placement within popular applications with high user engagement metrics. Malicious developers exploit this valuation differential by programmatically misrepresenting their application identity within the advertising ecosystem. These implementations provide falsified application identifiers to advertising networks, creating the deceptive impression that advertisements will be displayed within premium, high-engagement application contexts. `Hiddad.Scylla` exemplifies this approach by systematically spoofing ”bundle ID” or ”app ID” parameters during advertising network communications. This sophisticated deception manipulates both advertisers and advertising technology platforms into misclassifying the application’s marketplace position, resulting in either inappropriate ad placement decisions or inflated bid prices for advertisement display opportunities within the fraudulent application context.

5.5.2 Privacy information leakage

User information leakage represents one of the most pervasive compliance violations identified in our analysis of adware samples. These unauthorized data collection activities extend far beyond the legitimate information requirements for advertising functionality, constituting significant privacy infractions.

Our technical analysis revealed systematic collection of personally identifiable information (PII) through various programmatic interfaces, including sensitive identifiers such as phone numbers, International Mobile Equipment Identity (IMEI) values, and persistent device identifiers. Beyond these direct personal identifiers, we observed comprehensive harvesting of contextualized user data, including internet browsing patterns, comprehensive inventories of installed applications, and detailed hardware configuration information including CPU specifications and performance characteristics. This exfiltrated data serves multiple commercial exploitation vectors within the adware ecosystem. Primary utilization includes the construction of sophisticated user tracking profiles that persist across application boundaries and installation cycles.

5.5.3 Webview backdoor

Our analysis identified numerous instances where developers have implemented backdoors within the WebView components used for advertising content delivery, with the majority of these implementations being embedded within advertising

libraries. These backdoors primarily facilitate remote control interaction capabilities, introducing significant security vulnerabilities to the host device.

- **shouldOverrideUrlLoading Exploitation.** *shouldOverrideUrlLoading()* is a method in a custom WebViewClient that intercepts URL requests when loading web pages in WebView. Malicious developers systematically override this method to establish command-and-control channels, parsing specially formatted URLs as command instructions from remote servers to perform privileged operations on the device. Our analysis of `AdColony` revealed a particularly concerning implementation where the library defined specialized URL parsing methods within *shouldOverrideUrlLoading()*. This implementation enabled a range of sensitive operations including initiating phone calls, sending text messages, and capturing device screenshots—all triggered through remotely delivered URL commands without user awareness or consent.
- **JavaScriptInterface Exploitation.** WebView provides the *JavaScriptInterface* feature, which creates a bidirectional communication bridge allowing JavaScript code to invoke Java methods. Malicious developers exploit this capability by registering custom *JavaScriptInterface* implementations that expose privileged system functions to potentially untrusted JavaScript code delivered through advertising content. In early versions of the `InMobi` advertising library, our analysis documented the registration of extensive *JavaScriptInterface* capabilities, exposing sensitive system functions including *MakeCall()*, *SendEmail()*, *SendSMS()*, and *getGalleryImage()*. This implementation effectively grants remote advertising content the capability to access privileged device functionality without appropriate security controls or user consent mechanisms.

5.5.4 Malicious promotion

Promoting applications constitutes a standard component of legitimate advertising ecosystems; however, our analysis identified numerous adware implementations that employ non-compliant promotion methodologies to generate fraudulent revenue streams.

These malicious promotion techniques can be categorized into several distinct implementation patterns:

`BetterAd` implements particularly egregious behavior by programmatically initiating application downloads in background processes without any user notification or consent mechanisms. This silent installation attempt circumvents fundamental platform security controls and user agency regarding installed software components. Conversely, `Apofer` and `Dianle` implement coercive promotion strategies through manipulative user experience design. These implementations enforce engagement with offerwall functionality as a prerequisite for accessing core application capabilities. Users are systematically required to download and install third-party applications to earn virtual currency or points, which subsequently become mandatory for accessing the advertised features or content that motivated the initial application installation. This implementation creates artificial barriers to application functionality that can only be circumvented through forced interaction with promoted content.

`BoostClicker` and `Hiddenads.I` implement a particularly sophisticated approach by exploiting plugin functionality for application promotion. These variants programmatically install incentive-based applications from mobile advertising platforms directly into a plugin virtual environment, circumventing standard installation procedures and associated security controls. The implementation subsequently creates unauthorized shortcuts for each downloaded application on the device's home screen, presenting them as legitimate installations while actually launching them through a plugin-based execution environment. This technique enables the promotion of applications without triggering normal installation security prompts or visibility within the system's application management interfaces, effectively bypassing both user consent mechanisms and platform security controls designed to maintain installation transparency.

6 Discussion

6.1 Limitation

Our research makes contributions to the Android security field through the construction of a comprehensive adware dataset, precise localization of adware payloads, and characterization of diverse adware families revealing their implementation strategies, evasion techniques, and monetization mechanisms. Despite these advances, several limitations in our methodology warrant acknowledgment:

Firstly, while our dataset is extensive, collecting adware from both security reports and app repositories, it may not encompass all existing adware families given the vast scale of the Android app ecosystem.

Secondly, Our payload analysis method, which relies on app package structure and rescanning of isolated packages using VirusTotal, may be ineffective in certain scenarios, including: a) advanced obfuscation techniques such as package flattening, b) app packing methods that conceal code structure, c) dynamically loaded payloads that are not present in the static APK, d) families identifiable only through inter-package relationships. Fortunately, these limitations collectively affected only 7 families for which we were unable to detect payloads.

Thirdly, in our evaluation of payload detection results (Section 4.1.2), while we attempted to verify the identified payloads through dynamic testing, the wide time span of our dataset posed challenges. Many adware samples' remote communication servers had already ceased operations, impeding our ability to observe their real-time behavior. To mitigate this, we pivoted to conducting thorough static analysis on the adware samples to ensure the highest possible accuracy within these constraints.

Lastly, our research methodology involves significant manual analysis components, particularly during schema construction and behavior classification. This manual component introduces the potential for inconsistencies or oversights dependent on analyst expertise. To mitigate this issue, two co-authors independently conducted cross-validation of all manual annotations, systematically reviewing and verifying each other's classifications. In cases of disagreement, a third author adjudicated the final decision. This collaborative verification process effectively reduced individual

bias and enhanced the overall consistency and reliability of our analysis. Despite these limitations, the study establishes a robust foundation for advancing the understanding and mitigation of adware.

6.2 Practical implications for adware detection and defense

Our study provides practical implications for both industry practitioners and research communities working on Android security and malware detection. Our work comprehensively characterizes the behavioral and structural patterns commonly observed in adware. While individual API calls or system features may also appear in benign applications, our analysis reveals adware-specific behavioral combinations that produce distinct discriminative signals. We also identify several defining characteristics, including fraudulent advertising behaviors such as click fraud and impression fraud, aggressive advertisement presentation mechanisms such as out-of-application displays and notification hijacking, as well as persistence and evasion techniques. These characteristics, whether manifested independently or jointly, enable practitioners to differentiate adware from both benign advertising-supported applications and other categories of malware. Although our primary contribution lies in systematic characterization rather than detection system design, the identified behavioral patterns and structural indicators can guide future detection research.

For app store operators and antivirus vendors, our dataset and schema provide a more organized representation of adware behaviors that can facilitate consistent understanding and integration across platforms. The structured taxonomy promotes more coherent labeling practices and supports information exchange between different detection systems. By offering detailed behavioral and functional descriptions for each adware family, the dataset helps supplement existing adware repositories, particularly for families with incomplete or missing annotations. This organized representation can assist vendors in correlating detection results, improving interpretability across tools, and enhancing automated classification performance.

For machine learning–based malware detectors, the dataset serves as a high-quality benchmark resource. The well-annotated corpus contains detailed labeling of malicious behaviors, functional characteristics, and payload locations for each adware family. Its fine-grained annotations and verified family-level labels support supervised training and evaluation of adware classifiers under both static and dynamic analysis settings. The structured annotations also enable targeted feature extraction and representation learning, allowing researchers and practitioners to directly associate observable behavioral patterns with their corresponding code-level implementations.

7 Related work

Android malware dataset The development of Android malware datasets has been crucial for security research, with pioneering efforts like MalGenome (Zhou and Jiang 2012) in 2011, which relied on security reports to collect 49 malware families. This was followed by more comprehensive collections such as AMD (Wei et al.

2017), Drebin (Arp et al. 2014), RmvDroid (Wang et al. 2019), MalRadar (Wang et al. 2022), and Cao et al. (2022). These datasets have significantly contributed to the advancement of Android security research and defense mechanisms. However, these datasets primarily focus on conventional malware, limiting their applicability for comprehensive adware analysis. While other researchers have developed specialized datasets for specific malware categories like piggybacking (Bai et al. 2019), ransomware (Chen et al. 2017), and banking malware (Li et al. 2017), there remains a significant gap in datasets specifically tailored for adware research. Our work filled this gap with a comprehensive, well-annotated Android adware dataset across 118 distinct adware families.

Adware While our study is not the first to investigate Android adware, to the best of our knowledge, we are the pioneering effort to demystify adware through systematic classification and payload analysis. Previous research has primarily focused on adware identification using various machine learning techniques. For instance, Suresh et al. (2019) conducted classification experiments on a dataset comprising seven adware families, while Alani and Awad (2022) employed machine learning models for adware identification using 79 features extracted from network traffic of 250 adware samples across 5 families. However, these studies typically utilized limited datasets, lacking comprehensiveness and failing to locate or analyze the payloads themselves.

Research on ad libraries, while not specifically focused on adware, has contributed significantly to our understanding of the adware ecosystem. For instance, Li et al. (2016) conducted a comprehensive study, identifying 240 ad libraries by analyzing approximately 1.5 million apps from Google Play. In a related effort, Zhang et al. (2020) investigated 4,957 potentially malicious third-party libraries, which included various ad libraries. Their research delved into critical aspects such as repackaging techniques and potentially harmful behaviors associated with these libraries. Although these studies were not exclusively dedicated to adware analysis, they form an integral part of adware research by providing insights into the broader landscape of mobile advertising and its potential misuse.

Gao et al. (2019) made a notable contribution to adware research by attempting to determine the maliciousness of adware through an analysis of VirusTotal's malicious labels. Their findings revealed that up to 50% of samples within an adware family could be flagged as malicious, providing valuable insights into the potential harm of adware. However, their study was limited to 26 pre-known adware families and did not provide a dataset specifically dedicated to comprehensive adware analysis.

In summary, prior research has contributed to various aspects of adware study, including identification techniques, ad library analysis, and maliciousness assessment. However, these efforts have often been limited in scope, focusing on small sets of known adware families or specific behaviors. Notably absent from previous work is a comprehensive, well-annotated dataset alongside a structured taxonomy schema for systematic behavior classification. Our research addresses these gaps through three key contributions: (1) developing a large-scale adware dataset, (2) locating and analyzing adware payloads, and (3) conducting characterization of diverse adware

families to reveal their implementation strategies, and evasion techniques, and monetization mechanisms. This multifaceted approach significantly enhances understanding of adware implementation strategies and provides a structured framework for future research in analyzing and mitigating mobile adware threats.

8 Conclusion

This study represents a significant stride in understanding and addressing the pervasive threat of Android adware through the construction of **AdwareZoo**, a robust, well-annotated dataset of **15,996** adware samples across **118** distinct families. Our three-fold contribution—comprehensive dataset construction, systematic payload identification, and characterization of diverse adware families has unveiled critical insights into the technical implementation patterns underlying deceptive advertising practices, privacy violations, and fraudulent monetization techniques employed by adware developers. By methodically revealing how different adware families employ these mechanisms, our work not only fills a crucial gap in mobile security research but also establishes a foundation for more effective detection systems and regulatory frameworks. The complexity and prevalence of adware revealed through our analysis underscores the pressing need for continued research to demystify these threats and develop more robust mitigation strategies to protect users in the mobile ecosystem.

Author Contributions Conceptualization: All Authors; Methodology: Chao Wang, Tianming Liu, Haoyu Wang; Writing - original draft preparation: Chao Wang, Tianming Liu; Writing - review and editing: All Authors; Project administration and funding acquisition: Haoyu Wang; Supervision: Xiaoning Du, Li Li, Haoyu Wang.

Funding This work was supported by the National Natural Science Foundation of China (grant No.62072046) and the Key R&D Program of Hubei Province (2023BAB017, 2023BAB079).

Data Availability The comprehensive AdwareZoo repository, containing the complete dataset, payload analysis results, and detailed characterization of adware families, is publicly available at https://github.com/security-pride/AdwareZoo.

Declarations

Ethical Approval Not applicable since there are no human and/or animal studies included in this paper.

Competing interests Non-financial interests: Author Li Li is an Editor for the Special Issue on Advances in Mobile App Analysis and Systems in the ASE Journal.

References

Alani, M.M., Awad, A.I.: Adstop: Efficient flow-based mobile adware detection using machine learning. Comput. Secur. **117**, 102718 (2022)

Apktool. (2023). https://github.com/iBotPeaches/Apktool

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., Siemens, C.: Drebin: Effective and explainable detection of android malware in your pocket. In: Ndss, vol. 14, pp. 23–26 (2014)

Bai, C., Han, Q., Mezzour, G., Pierazzi, F., Subrahmanian, V.: Dbank: Predictive behavioral analysis of recent android banking trojans. IEEE Trans. Dependable Secure Comput. **18**(3), 1378–1393 (2019)
Cao, M., Ahmed, K., Rubin, J.: Rotten apples spoil the bunch: An anatomy of google play malware. In: Proceedings of the 44th International Conference on Software Engineering, pp. 1919–1931 (2022)
Chen, K., Wang, X., Chen, Y., Wang, P., Lee, Y., Wang, X., Ma, B., Wang, A., Zhang, Y., Zou, W.: Following devil's footprints: Cross-platform analysis of potentially harmful libraries on android and ios. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 357–376. IEEE (2016)
Chen, J., Wang, C., Zhao, Z., Chen, K., Du, R., Ahn, G.-J.: Uncovering the face of android ransomware: Characterization and real-time detection. IEEE Trans. Inf. Forensics Secur. **13**(5), 1286–1300 (2017)
Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. ACM sigplan notices 49(6), 259–269 (2014)
Gao, J., Li, L., Kong, P., Bissyandé, T.F., Klein, J.: Should you consider adware as malware in your study? In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 604–608. IEEE (2019)
Gartner Magic Quadrant for Endpoint Protection Platforms. (2021). https://www.gartner.com/en/documents/4001307
Hurier, M., Suarez-Tangil, G., Dash, S.K., Bissyandé, T.F., Le Traon, Y., Klein, J., Cavallaro, L.: Euphony: Harmonious unification of cacophonous anti-virus vendor labels for android malware. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 425–435. IEEE (2017)
IDA. (2023). https://hex-rays.com/ida-free/
inmobi. (2024). https://www.inmobi.com/
jadx. (2023). https://github.com/skylot/jadx
JavascriptInterface. (2024). https://developer.android.com/reference/android/webkit/JavascriptInterface
javascript-interfaces. (2022). https://securityqueens.co.uk/android-attack-javascript-interfaces-and-webviews/
Koodous. (2023). https://koodous.com/
Li, L., Bissyandé, T.F., Klein, J., Le Traon, Y.: An investigation into the use of common libraries in android apps. In: 2016 IEEE 23Rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, pp. 403–414. IEEE (2016)
Li, L., Li, D., Bissyandé, T.F., Klein, J., Le Traon, Y., Lo, D., Cavallaro, L.: Understanding android app piggybacking: A systematic study of malicious code grafting. IEEE Trans. Inf. Forensics Secur. **12**(6), 1269–1284 (2017)
Liu, T., Wang, H., Li, L., Bai, G., Guo, Y., Xu, G.: Dapanda: Detecting aggressive push notifications in android apps. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 66–78. IEEE (2019)
Ma, Z., Wang, H., Guo, Y., Chen, X.: Libradar: fast and accurate detection of third-party libraries in android apps. In: Proceedings of the 38th International Conference on Software Engineering Companion, pp. 653–656. ACM (2016)
Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., Vigna, G.: Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In: NDSS, vol. 14, pp. 23–26 (2014)
Sebastián, S., Caballero, J.: Avclass2: Massive malware tag extraction from av labels. In: Annual Computer Security Applications Conference, pp. 42–53 (2020)
secret-inside-of-so-file. (2024). https://medium.com/@jinwei.han93/secret-inside-of-so-file-c073c8375c63
Suresh, S., Di Troia, F., Potika, K., Stamp, M.: An analysis of android adware. J. Comput. Virol. Hacking Techn. **15**, 147–160 (2019)
The mobile malware threat landscape in 2023. (2024). https://securelist.com/mobile-malware-report-2023/111964/
Virustotal. (2023). https://www.virustotal.com/
Wang, C., Liu, T., Zhao, Y., Zhang, L., Du, X., Li, L., Wang, H.: Towards demystifying android adware: Dataset and payload location. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops, pp. 167–175 (2024)
Wang, H., Si, J., Li, H., Guo, Y.: Rmvdroid: Towards a reliable android malware dataset with app metadata. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp. 404–408. IEEE (2019)

Wang, L., Wang, H., He, R., Tao, R., Meng, G., Luo, X., Liu, X.: Malradar: Demystifying android malware in the new era. Proc. ACM Meas. Anal. Comput. Syst. **6**(2), 1–27 (2022)
Wei, F., Li, Y., Roy, S., Ou, X., Zhou, W.: Deep ground truth analysis of current android malware. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 252–276. Springer (2017)
Well Known Ad-Networks. (2024). https://www.appbrain.com/stats/libraries/ad-networks
What is a adware. (2024). https://www.techtarget.com/searchsecurity/definition/adware
What is a greyware. (2024). https://www.malwarebytes.com/glossary/greyware?srsltid=AfmBOoqKBDEZ2hWoLZpLcj8j0FySWgliAuicMgFJ-cj49wkQJuuMwcLg
What is PUAs. (2024). https://en.wikipedia.org/wiki/Potentially_unwanted_program
Wu, L.: Fake Android Apps Communicate For Malware, Ad Fraud. (2020). https://www.trendmicro.com/en_us/research/20/b/malicious-apps-on-google-play-communicate-with-trojans-install-malware-perform-mobile-ad-fraud.html
Xu, E.: Adware Posing as 85 Photography and Gaming Apps on Google Play Installed Over 8 Million Times. (2019). https://www.trendmicro.com/en_us/research/19/h/adware-posing-as-85-photography-and-gaming-apps-on-google-play-installed-over-8-million-times.html
Xue, Y., Meng, G., Liu, Y., Tan, T.H., Chen, H., Sun, J., Zhang, J.: Auditing anti-malware tools by evolving android malware and dynamic loading technique. IEEE Trans. Inf. Forensics Secur. **12**(7), 1529–1544 (2017)
Zhang, Z., Diao, W., Hu, C., Guo, S., Zuo, C., Li, L.: An empirical study of potentially malicious third-party libraries in android apps. In: Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks, pp. 144–154 (2020)
Zhao, Y., Li, L., Wang, H., Cai, H., Bissyandé, T.F., Klein, J., Grundy, J.: On the impact of sample duplication in machine-learning-based android malware detection. ACM Trans. Softw. Eng. Methodology (TOSEM) **30**(3), 1–38 (2021)
Zheng, C., Xu, Z.: New Android Malware Family Evades Antivirus Detection By Using Popular Ad Libraries. (2015). https://unit42.paloaltonetworks.com/new-android-malware-family-evades-antivirus-detection-by-using-popular-ad-libraries/
Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: 2012 IEEE Symposium on Security and Privacy, pp. 95–109. IEEE (2012)
Zhu, S., Shi, J., Yang, L., Qin, B., Zhang, Z., Song, L., Wang, G.: Measuring and modeling the label dynamics of online {Anti-Malware} engines. In: 29th USENIX Security Symposium (USENIX Security 20), pp. 2361–2378 (2020)

Authors and Affiliations

Chao Wang[1] · Tianming Liu[1] · Yanjie Zhao[1] · Lin Zhang[3] · Xiaoning Du[2] · Li Li[4] · Haoyu Wang[1]

✉ Haoyu Wang
haoyuwang@hust.edu.cn

Chao Wang
chaowang_@hust.edu.cn

Tianming Liu
tmliu@hust.edu.cn

Yanjie Zhao
yanjie_zhao@hust.edu.cn

Lin Zhang
zhanglin@cert.org.cn

Xiaoning Du
Xiaoning.Du@monash.edu

Li Li
lilicoding@ieee.org

[1] Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China

[2] Faculty of Information Technology, Monash University, Melbourne, Australia

[3] The National Computer Emergency Response Team/Coordination Center of China (CNCERT/CC), Beijing, China

[4] School of Software, Beihang University, Beijing, China