Cracks in the Walled Garden: Dissecting the Gray-Market of Unauthorized iOS App Distribution via Ad Hoc Sideloading

Yijing Liu
Tsinghua University
BNRist

Yiming Zhang
Tsinghua University
BNRist

Baojun Liu
Tsinghua University
BNRist

Haixin Duan
Tsinghua University
BNRist

Abstract

Apple enforces strict code signing and mandates app distribution through its official App Store. Nonetheless, unauthorized apps still spread through sideloading channels. The Ad Hoc provisioning mechanism, originally designed for developer testing, has emerged as one such channel. It leverages individual developer certificates and user-side signing to enable unauthorized app installations that bypass Apple's app review process. Over time, this practice has evolved into a structured and prevalent gray-market that connects certificate resale, third-party signing tools, and the distribution of unsigned *.ipa* files. In this work, we present the first systematic study of this market, with a specific focus on its integrated service operations in China. Through a user-centric data collection strategy, we identified 3,359 active signing sites for certificate redemption, reverse engineered 12 signing tools, and obtained 8,216 distributed *.ipa* entries. Our analyses uncover a multi-layered certificate circulation model with resale margins up to 3,000% and reveal common tricks that signing tools employ for code signing. Most distributed apps are modified versions of legitimate ones, which leverage dynamic library injection to enable customized features. Such modifications undermine the security protections that both apps and the system provide to users, exposing them to risks such as unauthorized actions, sensitive data exfiltration, and system capability exploitation. Overall, our findings reveal a mature gray-market that erodes iOS's trust model while operating in plain sight, underscoring the need for targeted interventions from multiple stakeholders.

1 Introduction

iOS is designed as a walled garden, where all apps must be signed by Apple and distributed through its official App Store. This trust model is widely regarded as a cornerstone of iOS security. However, it is not a secret that third-party unauthorized apps can still be installed outside the App Store, bypassing Apple's installation control. The practice known as "sideloading" is prevalent in the iOS ecosystem, reportedly involving 18.3% of mobile users worldwide [84]. It attracts users by offering access to apps unavailable in official stores (e.g., porn apps, gambling apps, or region-restricted apps), but also exposes them to increased risks of malware, spyware, and other threats [75, 84]. Thus, Apple has long considered sideloading as a security threat and prohibits its broad use [40].

Despite Apple's restriction, various sideloading methods have continued to emerge, often exploiting Apple's official testing or internal distribution channels. Prior work [21, 81] primarily examined the abuse of in-house distribution, where enterprise developer certificates are misused under the guise of internal app deployment. However, Apple has recognized this prevalent abuse and proactively mitigated it through more frequent certificate revocations [6, 9]. Other channels, including TestFlight abuse [33] and TrollStore installation [10], have also been observed, but their limited persistence and applicability constrain their widespread adoption.

Against this backdrop, a self-signing sideloading method based on Apple's Ad Hoc mechanism has gained increasing adoption, as evidenced by frequent discussions on online forums [32]. Originally designed for testing, the Ad Hoc mechanism permits app installation only on a limited set of registered devices, with each app signed using an individual developer certificate and a provisioning profile bound to device Unique Device Identifiers (UDIDs). In sideloading scenarios, this mechanism is repurposed to enable the installation of arbitrary *.ipa* packages. Users must first register their devices under a developer account and then perform code signing themselves, effectively disguising unauthorized apps as legitimate test builds. Unlike enterprise certificate abuse, which distributes pre-signed apps for direct installation, Ad Hoc sideloading requires users to perform the signing themselves, with each installation bound to a specific device. This peer-to-peer sideloading channel increases stealth, complicating Apple's efforts to detect and mitigate such abuse.

Ad Hoc sideloading enables unauthorized app distribution and relies on three components: a valid developer certificate, a signing tool, and an unsigned *.ipa* file. The requirement for device registration and manual code signing imposes a non-

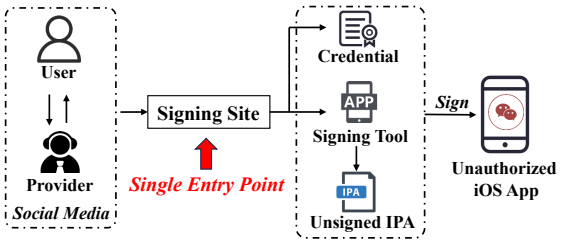


Figure 1: The integrated gray-market of Ad Hoc sideloading.

trivial technical barrier. As a result, third-party services have emerged to supply these components and simplify the sideloading workflow, forming a gray-market that encompasses certificate resale, signing services, and app package (*.ipa* files) distribution. While such services are often fragmented and decentralized globally, the Chinese market is characterized by highly centralized services that integrate all necessary components, further reducing the effort of sideloading for users. Despite sporadic reports [14, 34], illicit activities in the distribution market remain underexplored. In this context, our work presents the first systematic study of Ad Hoc sideloading and its gray-market, demystifying the core components and addressing the following research questions.

- **RQ1-Certificate:** How are individual developer certificates traded and abused in the self-signing gray-market?
- **RQ2-Signing Tools:** What tools are employed for code signing, and how do they facilitate unauthorized distribution?
- **RQ3-Distributed Apps:** What types of apps are being distributed, and what security threats do they present?

To address these questions, we focus on integrated gray-market services in the Chinese market, which offer a structured and observable workflow for analysis. We first outline the typical workflow of the integrated gray-market service through a preliminary investigation. As illustrated in Figure 1, users initially contact the provider via social media and then register their devices by submitting a UDID on the "signing site", which serves as the single entry point. The site issues the corresponding signing credential[1] and provides a "signing tool" that both performs the signing process and hosts repositories of unsigned *.ipa* files for download. Guided by this workflow, we adopt a user-centric strategy to collect all the necessary components for the sideloading process. In total, we identified 62 signing sites from social media and expanded this set to 3,359 sites using a passive DNS dataset. From these sites, we collected distributed signing tools and manually reverse-engineered 12 samples using IDA Pro [24]. We further extracted 8,216 *.ipa* entries and downloaded 2,654 app packages from repositories bundled with the signing tools.

We conducted a systematic analysis based on these datasets. For RQ1, we find that the signing sites remain highly active, recording over 75k DNS resolutions in 92 days. Certificate abuse has evolved into a multi-layered industry sustained by organized providers, generating profit margins of up to 3,000%. For RQ2, most third-party signing tools are built on "zsign" [39] for code signing, with some exploiting iOS kernel vulnerabilities to bypass signature verification. For RQ3, the majority of distributed apps are modified versions of legitimate ones, primarily offering VIP unlocking or ad blocking by injecting third-party dynamic libraries. These libraries modify core functions to manipulate memory and intercept traffic, introducing risks such as unauthorized actions, data leakage, and system capability exploitation. Overall, the integrated gray-market substantially lowers the technical barrier for abusing Ad Hoc sideloading and accelerates unauthorized iOS app distribution. This not only harms the economic interests of Apple and official developers but also exposes users to security and privacy threats. Based on our findings, we have reported this abuse scenario to Apple and propose three mitigation strategies targeting different stakeholders.

Contributions. (1) Our work presents the first in-depth study of iOS Ad Hoc sideloading and its gray-market, identifying diverse service entry points and distributed unauthorized apps through a user-centric data collection approach. (2) Our analysis reveals the illicit circulation of iOS certificates and the technical design of signing tools, offering insights to support iOS governance and interventions. (3) In addition, we characterize common modification practices and security risks of the distributed apps, providing a factual basis for app vendors to prevent tampering. Collectively, our work improves the understanding of unauthorized iOS distribution and highlights potential points of intervention at the ecosystem level.

2 Background

iOS enforces a tightly controlled app distribution model based on certificate validation [26], with the App Store as the primary official channel for public access. Despite these restrictions, sideloading, which refers to installing apps outside the App Store, continues to be widely used [49]. The practice circumvents Apple's intended distribution controls and destroys the integrity of the iOS trust model. Although Apple introduced alternative app marketplaces in the European Union in March 2024 to comply with the Digital Markets Act (DMA) [5], this remains at an early stage, and sideloading continues to be prohibited in other regions [40]. This section provides an overview of the iOS certificate-based signing mechanism and examines several widely used sideloading techniques, as a prelude to our study.

2.1 iOS Certificates and Distribution

Certificate-Based Code Signing in iOS. Digital certificates play a central role in governing the iOS apps lifecycle. To develop and distribute iOS applications, developers must first

[1]The credential consists of a developer certificate and a provisioning profile. Within this ecosystem, providers often use "certificate" to denote both, as the certificate is the core asset and the profile is issued alongside.

Table 1: Key features of different Apple developer accounts.

	Individual	Company	Enterprise
Annual Fee	$99	$99	$299
Require D-U-N-S	NO	Yes	Yes
Team Collaboration	No	Yes	Yes
App Store	Yes	Yes	NO
In-House	NO	NO	Yes
Ad Hoc	Yes	Yes	NO

enroll in the Apple Developer Program, and Apple issues a digital certificate that binds the developer's identity to a public key. Developers also need to generate a provisioning profile that links the certificate to an App ID, authorized devices (if applicable), and a set of entitlements, which declare permissions that define app capabilities such as push notifications or background execution. During compilation, the app is signed using the developer's private key, and the resulting signature, certificate, and provisioning profile are embedded into the app bundle. Upon installation, iOS verifies the digital signature to ensure the app originates from an authorized developer and has not been tampered with. It also enforces the constraints encoded in the provisioning profile, including entitlements and device limitations, ensuring that only explicitly permitted capabilities are granted. Through this process, iOS code signing enforces authenticity, integrity, and strict control over distribution and execution privileges.

Certificate Types and Distribution Models. Apple provides two main types of certificates for code signing: "Development Certificate" for testing on registered devices, and "Distribution Certificate" for external deployment. Obtaining a Distribution Certificate requires enrolling in the Apple Developer Program [8], which provides three account types: Individual ($99/year), Company ($99/year), and Enterprise ($299/year). As summarized in Table 1, account type determines the distribution certificate and the supported distribution methods. Individual and Company accounts support App Store distribution as well as limited external testing through TestFlight and Ad Hoc, while Enterprise accounts are restricted to in-house deployment without App Store review.

• *App Store Distribution*: This is the default public release model. Apps must be signed with a distribution certificate and submitted for Apple's review. Upon approval, they are published on the App Store and made available to the public.

• *TestFlight (Beta) Distribution*: TestFlight [35] is Apple's official beta testing platform. It allows developers to upload pre-release builds, invite testers, and collect feedback in real time. Each uploaded build is available for testing for up to 90 days. Testers participate by installing the TestFlight app and accessing the test version via an invitation link.

• *Ad Hoc Distribution*: This method supports limited external testing by allowing app installation on devices whose UDIDs are pre-registered in the provisioning profile. The profile controls app access on registered devices and is valid for one year. Apple permits up to 100 device registrations per developer account annually [41], with the first 10 approved instantly and later ones requiring 24–72 hours for review.

• *In-House Distribution*: This method is intended for internal app deployment within an organization. It allows companies to distribute apps directly to employees without App Store review or device registration, with installation typically via download links or mobile device management (MDM).

Table 2: Common sideloading methods.

Method	Mechanism	Validity
In-House	Enterprise Developer Certificate	1 year
AltStore	Free Apple Developer Account	7 days
TestFlight	Official Beta distribution	90 days
Jailbreak	Root access via iOS vulnerabilities	Permanent
TrollStore	iOS CoreTrust exploit	Permanent
WebClip	Web app shortcut	Permanent
Ad Hoc	Individual Developer Certificate	1 year

2.2 Known Sideloading Methods

Among all official iOS distribution methods, only the App Store supports public releases, while others are limited to internal or testing use. These channels impose strong constraints but have nevertheless been exploited to enable sideloading. Based on prior studies [50, 81] and public sources [36], we listed the common sideloading approaches in Table 2.

• **Abuse of In-House Distribution.** In-house distribution relies on enterprise certificates, and has been widely misused for public app distribution [21, 81]. They can be used to sign arbitrary apps and enable large-scale distribution without requiring App Store approval. By trusting the associated provisioning profile, users can install these signed apps, making the device appear to belong to the authorized organization. This method was once common but is now tightly regulated [6, 9]. Nevertheless, users can leverage a custom DNS configuration to block access to Apple's certificate verification endpoints (e.g., *ocsp.apple.com*, *crl.apple.com*), thereby bypassing revocation checks and allowing continued use of apps signed with revoked enterprise certificates.

• **Repurposing of Official Testing Channels.** Apple's testing-oriented mechanisms have been repurposed for sideloading. Free developer accounts (registered with Apple IDs) issue short-lived development certificates that allow users to self-sign *.ipa* files for installation. This process can be carried out through Apple's Xcode or third-party utilities such as AltStore [3], Sideloadly [53], and i4Tool [74]. However, this method is restricted to three apps per device and each certificate expires after seven days, requiring frequent re-signing. TestFlight has likewise been misused to distribute unautho-

rized apps, as its review is less strict than the App Store's.

• **Certificate-Free Sideloading.** Beyond official certificates, some methods bypass Apple's signature checks entirely. Jailbreaking exploits iOS vulnerabilities to gain root access, enabling unrestricted app installation. While effective, it introduces severe security risks [23, 72] and is impractical for most users. Similarly, TrollStore leverages CoreTrust vulnerabilities to install unsigned apps with persistent validity [10]. This method is attractive for sideloading but is limited to older iOS versions. WebClips provide another lightweight way to bypass the iOS signature checks. By adding home-screen shortcuts that launch predefined URLs, WebClips can deliver web-based interfaces that mimic native app behavior without requiring installation or authentication. This ease of deployment makes them attractive to malicious actors [50], and they have been widely abused to distribute gambling, fraud, and pornographic content [52]. It is easy to deploy but offers limited functionality since it is not a native app

Although various sideloading methods exist, each has significant drawbacks. As another Apple official testing channel, Ad Hoc distribution has been repurposed and has gained prominence as a practical sideloading method. Although supported by both company and individual developer accounts, it is predominantly enabled through the latter due to minimal registration barriers. With such certificates, users can self-sign and install arbitrary *.ipa* files on registered devices, and this distribution channel is the primary focus of our study.

3 Gray-Market of Ad Hoc Sideloading

Apple's Ad Hoc provisioning is originally designed to support legitimate app testing on registered devices. When repurposed for unauthorized iOS app distribution, this mechanism gives rise to a gray-market ecosystem.

Key Actors. Ad Hoc sideloading requires three main components: a developer certificate, a signing tool, and the target *.ipa* file. In the gray-market, these resources correspond to three key actors, transforming a testing mechanism into a sideloading channel for unauthorized app distribution.

• *Certificate providers.* Instead of enrolling in Apple's official developer program, users obtain developer certificates through third-party providers that operate in violation of Apple's developer agreements [7]. Representative providers include Kravasign [29] and Applep12 [11].

• *Signing service providers.* In place of the official signing process via Xcode, third-party services such as IPASign [27] and AppTesters [12] enable users to perform code signing through graphical interfaces, avoiding the complexity of manual configuration and command-line operations.

• *Unauthorized app providers.* This group comprises unofficial developers and underground distributors who supply unsigned *.ipa* files through unofficial app stores (e.g., iPAStore [28]) and community platforms (e.g., Discord [19]).

Integration into Single-Entry Services. Conventionally, curious users have to obtain the required components from multiple actors for sideloading. Without centralized guidance, this fragmented workflow imposes considerable complexity, particularly for non-technical users. The technical barrier gives rise to integrated services. Although the underlying gray-market actors operate on a global scale, such integrated services are most prominently observed in China. In this model, a single provider unifies the roles of all three actors into a one-stop service accessed through a single entry point and offers step-by-step guidance to users. These providers are typically resellers rather than original app developers, aggregating app resources and bundling them with certificate resale to deliver end-to-end solutions to users. This streamlines installation and fosters the maturation of the gray-market. Moreover, through promotion on mainstream social media, these services advertise their ability to provide customized apps (e.g., modified WeChat), thereby gaining public visibility and expanding their potential user base.

Typical Workflow. The integrated model offers a practical perspective for studying the self-signing gray-market. To better understand its structure, we conducted an initial exploration from the end-user perspective. Specifically, we actively interacted with a sample of vendors found on major social media (i.e., RedNote [38]), and followed their installation procedure to experience the full process. Based on this, we summarized a typical operational workflow in Figure 2.

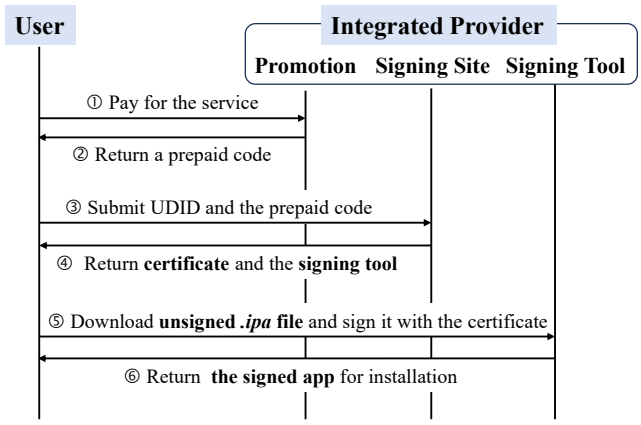


Figure 2: Workflow of the integrated self-signing service.

The signing site serves as the single entry point to the integrated gray-market and is promoted on social media to attract users. The process begins when the user pays for the self-signing service (①) and receives a prepaid code along with a guideline (②). Following the guideline, the user submits the code and device UDID to the signing site to redeem a signing credential (③). The credential, consisting of a *.p12* certificate and a provisioning profile bound to the UDID, is then returned to the user (④). The site also delivers a designated signing tool pre-signed with the redeemed credential.

This tool not only performs code signing but also embeds a built-in repository of unsigned *.ipa* files, allowing users to download arbitrary unofficial apps (⑤). Finally, the credential is used to sign the selected apps and install them directly on the registered device, completing the sideloading (⑥).

4 User-Centric Data Collection Method

Unauthorized iOS apps distributed via Ad Hoc sideloading are not publicly accessible, making direct web crawling [81] ineffective. Guided by the operational workflow of the integrated services, we adopt a user-centric, stepwise strategy to collect relevant data in the gray-market. Starting from social media platforms, we emulate typical user behavior to identify widely promoted signing sites, and then collect signing tools from these sites. The tools typically include pre-configured software repositories, allowing us to use them as a pivot to download the distributed apps. An overview of the collection results is shown in Figure 3. Our analysis focuses on the Chinese market, where integrated services are highly prevalent and sufficiently consolidated, allowing for representative analysis. Moreover, such services are not limited to China, and our work contributes to a broader understanding of Ad Hoc sideloading (see Section 8.2 for further discussion).

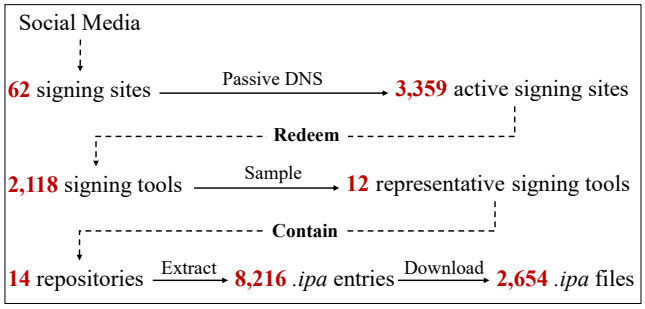


Figure 3: The overview of data collection.

4.1 Discovering Signing Sites

Signing sites serve as the entry point to the self-signing gray-market. We begin our analysis by collecting such sites, which form the foundation for subsequent stages of the study.

Original Collection from Social Media. We selected RedNote (Xiaohongshu) [38] as our primary data source for identifying self-signing service providers, as it is one of the most active text-based social media platforms in China. Using keywords such as "iOS certificate" and "customized V (WeChat)", we manually inspected posts and comments to extract publicly advertised signing sites. Coverage was expanded via snowball sampling [62] on related tags (e.g., "mobile customization", "Apple certificate"). Through this process, we identified a total of 62 distinct signing sites.

Reseller-Based Model. During our collection, we observed that many signing sites shared a common domain pattern, typically user-defined subdomains under fixed second-level domains (SLDs) (e.g., *rlyzzyq.11wink.cn*, *fffyyu.11wink.cn*). Front-end code inspection show that sites with the same suffix cluster communicate with identical backend endpoints for certificate redemption, which we term as "certificate sites". Most certificate sites adopt a "reseller-based" deployment model (see Figure 11 in Appendix) that simplifies portal creation and enables large-scale replication. Resellers create custom portals by registering subdomains under shared SLDs, where each prefix must contain at least four alphanumeric characters and exclude reserved terms such as "cert", "sign", or "udid". This design accounts for the recurring SLDs across signing sites, as each SLD corresponds to a reseller service cluster. We therefore generalize this domain pattern to enable large-scale discovery of additional signing sites.

Passive DNS Expansion. Leveraging the domain pattern identified, we performed large-scale expansion using passive DNS data, which records historical DNS resolutions and enables retrospective analysis of domain usage. Given our regional focus, we collaborated with 114DNS, a major Chinese DNS provider processing 600 billion queries per day [80]. Using signing sites identified from social media as seeds, we extracted 50 unique SLDs and compiled them into a regular expression that captures the reseller-based subdomain pattern. We applied this pattern to passive DNS logs from March to May 2025, identifying 8,358 candidate signing sites with their daily aggregated resolution results, including domain (fqdn), resolution results (rdata), and query counts (request_cnt). To ensure accuracy, we probed these domains and archived the source code. Domains lacking UDID collection functionality were excluded, as device registration is required for signing sites. This filtering yielded 3,359 active signing sites associated with 38 SLDs, fewer than the 50 SLDs initially identified, as some domains had no active subdomains during data collection. Based on this dataset, we identified all the corresponding certificate sites and analyzed the provenance of the distributed certificates. Detailed results are presented in Section 5.

4.2 Sampling Signing Tools

The signing tool is the first unofficial app in the self-signing chain and provides the core signing functionality. These tools are typically distributed through signing sites and require payment for access. However, sites discovered via passive DNS often lack identifiable vendor information, making it difficult to associate them with specific sellers for payment. In addition, ethical constraints prevented us from conducting large-scale purchases. Consequently, we adopted a sampling strategy to collect as many distinct tools as possible.

Although we could not obtain all signing tools directly, we extracted their names from front-end source code and identified 2,118 unique tool names across 3,359 active sign-

ing sites. Most sites offer at least one custom-branded tool, whose names and attributes are configured by resellers under the reseller-based deployment model in certificate sites (see Figure 11 in Appendix). In addition, two mature products, *All-in-One Sign* [2] and *Easy Sign* [20], are frequently offered alongside custom-branded tools across signing sites. These two names dominate the frequency distribution, together accounting for 48.59% of all occurrences and appearing substantially more often than any other. Based on these observations, our sampling strategy was designed to capture both the dominant tools and a diverse selection of custom-branded tools customized from different certificate sites.

To implement this strategy, we revisited vendors initially identified from social media in Section 4.1 and filtered those whose signing sites appeared in our final dataset. From this filtered set, we selectively contacted a subset of vendors and purchased a limited number of signing tools, with the goal of capturing diversity across tools rather than achieving full coverage. In total, we acquired 12 signing tools. We further performed reverse engineering using IDA Pro [24], with a particular focus on their customized signing implementations. Detailed results are presented in Section 6.

4.3 Accessing Distributed Apps

Signing tools not only preform code signing, but also bundle repositories of unsigned *.ipa* apps for users to download. These repositories are integrated into signing tools via URLs, which return a JSON manifest containing metadata for multiple apps, including name, version, description, and download link (see Figure 9 in Appendix). In total, we identified 14 repositories, with some tools linking to multiple sources.

From the extracted repositories, we collected metadata for 8,216 unofficial IPA entries, of which 2,660 contained direct download links. Entries without links typically require an additional payment to unlock access. Notably, the distribution of these links was highly centralized. A single domain (*pan.iosr.cn*) hosted 1,220 apps and appeared across multiple repositories. In total, we identified 43 distinct hosting domains. Over half (25/43) used Alist [1], an open-source indexing service that facilitates large-scale, unauthenticated distribution via third-party cloud storage.

Subsequently, we accessed each extracted download link to retrieve the corresponding *.ipa* files. Paid entries requiring additional unlocking were excluded, as freely accessible samples were sufficient for our analysis. Due to link expiration and network inaccessibility, we successfully downloaded 2,654 *.ipa* files. To characterize these apps, we first analyzed their declared functionality and intended use based on the metadata. We then performed reverse engineering to examine implementation details and potential security risks, with a focus on injected dynamic libraries (*.dylib* files). Further analysis and results are presented in Section 7.

5 Illicit Circulation of iOS Certificates

In this section, we begin with the collected accessible signing sites to examine the misuse of iOS developer certificates within the self-signing gray-market (RQ1). By analyzing the roles and economic incentives of different actors, we reveal key characteristics of this illicit certificate abuse.

5.1 Practices of Signing Sites

Usage Scale. Our analysis reveals a substantial self-signing gray-market, with 3,359 active signing sites generating 75,371 DNS resolutions over 92 days (averaging 228.3 sites and 1,508.2 resolutions daily). The temporal trend suggests both stability and expansion, as shown in Figure 4, with new subdomains regularly emerging as independent service interfaces under the reseller model. Occasional spikes in query volume and fqdn diversity, notably in late May, likely reflect short-term promotional campaigns and further underscore the overall vibrancy of the ecosystem. Moreover, the ecosystem exhibits a long-tail distribution of resolution frequency (Figure 10 in Appendix). 52.5% (1,764/3,359) were queried fewer than five times, whereas only 90 domains exceeded 100 queries. The most queried domain, *zero.xzin.top*, received 4,653 resolutions, indicating concentrated usage alongside broad participation.

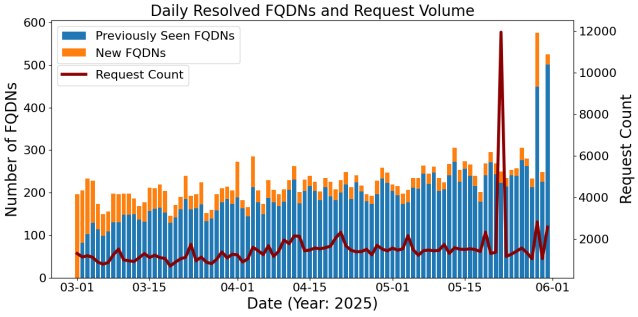


Figure 4: Temporal trend of signing sites and request volume.

Get UDID. Signing sites serve as the entry point to the self-signing process, requiring users to first extract their device UDID and then redeem a signing certificate using a prepaid code. To obtain the UDID necessary for Ad Hoc distribution, these sites instruct users to download and install a mobile configuration profile that extracts the device UDID and auto-fills it on the website (see Figure 5). Our source-code analysis reveals that 98.8% (3,319/3,359) of download links adopt a centralized format as illustrated in Figure 6, with endpoints clustered under 41 domains, indicating substantial infrastructure reuse. All these profiles collect not only UDID but also sensitive identifiers such as ICCID[1] and IMEI[2], and upload

[1] ICCID is short for Integrated Circuit Card Identifier.
[2] IMEI is short for International Mobile Equipment Identity.

them to third-party servers without explicit disclosure to the user. Notably, 97.7% of profiles are signed, displaying a green "Verified" label that boosts user trust (Figure 5). Since iOS accepts any structurally valid signature, both Apple-issued certificates and standard SSL certificates can be used for signing. Additionally, 83.7% (2,810/3,359) of signing sites download an extra *.mobileprovision* file, which is not necessary for UDID extraction but is used to trigger a system-level redirect to the Settings interface, streamlining the user interaction.

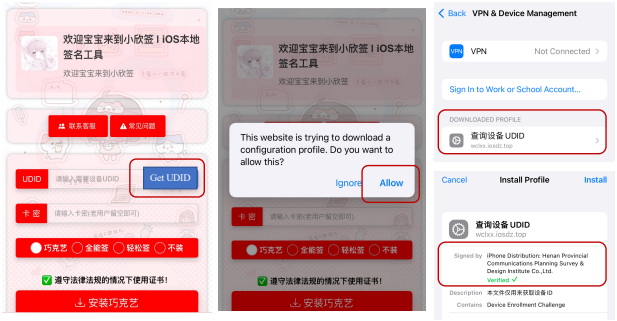


Figure 5: Procedures for users to acquire UDID.

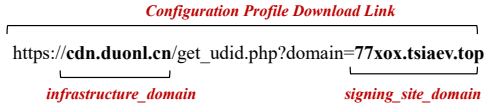


Figure 6: Common format of UDID profile download links.

Redeem for Certificate. After collecting the UDID and a prepaid code, signing sites forward them to upstream certificate providers. These backend services create a provisioning profile with the submitted UDID and return it with the developer certificate, effectively replicating Apple's provisioning process. To identify these certificate providers, we extracted AJAX request endpoints (e.g., *url:*, *$.post()*, *fetch()*) from the source code of signing sites. This allowed us to systematically recover the APIs responsible for certificate issuance. Among the 3,359 signing sites analyzed, 2,454 directly exposed certificate redemption through cross-origin requests to external domains. The remaining sites concealed this interaction, either through client-side code obfuscation or by handling certificate redemption on the server side, with frontend requests targeting only internal paths. Focusing on the observable sites, we identified 28 distinct sites responsible for certificate issuance, which we refer to as certificate sites. As illustrated in Figure 12 in the Appendix, the mapping between signing and certificate sites reveals a highly centralized structure, with a handful of providers supporting over 135 times as many downstream sites. For example, "Rainbow Site" (*ch.onxg.top*) alone supports 843 signing sites, forming the largest cluster. Notably, a single certificate site may serve signing sites across multiple SLDs, while signing sites under the same SLD may rely on different certificate sites. Such concentration not only explains the rapid expansion of the market but also points to the presence of organized groups operating behind the ecosystem.

5.2 Certificate Sites as Central Hubs

To understand the implementation of certificate sites, we manually examined these sites. All observed sites are uniformly built using the FastAdmin [22] framework and exhibit high visual and functional similarity. Acting as the central hubs, these certificate platforms serve as unified backends in the self-signing services, bridging upstream certificate provisioning with downstream service delivery.

On the upstream side, certificate sites interface with Apple's provisioning infrastructure to obtain valid signing credentials required for app signing. Our understanding of this process was enhanced by a case in which the operator of *developer.iksq.cn* publicly released its backend source code. The code employs an unofficial client library developed by MingYuanYun[1] to interact with Apple's App Store Connect API [4], automating the generation of *.p12* certificates and provisioning profiles. This process relies on *.p8* authentication keys, which are private credentials bound to specific Apple developer accounts and used to authorize programmatic access to account resources. To enable scalable issuance, certificate sites maintain pools of such keys across multiple accounts, assigning certificates in parallel or at random to balance usage and obscure identifiable binding patterns. Although source code from other sites was not available, the presence of comparable *.p8* pools suggests similar implementations.

On the downstream side, certificate sites transform issued credentials into services by distributing them to users. In addition, they use these certificates to sign the dedicated signing tool and provide its download link to users, thereby linking together the key elements of the sideloading process. The sites further support rapid expansion through the reseller-based model introduced in Section 4.1, where operators register subdomains under shared SLDs, enabling signing portals to be launched and replicated at scale. Functionally, the platforms support device management by associating submitted UDIDs with their bound certificates. They also provide prepaid code generation with configurable parameters, such as the warranty period, which entitles users to free certificate replacement if the original is revoked within that period.

5.3 Profit Model

Beyond signing sites and certificate sites, upstream developer account suppliers, downstream social media resellers, and end users collectively form a multi-layered certificate circulation process, as shown in Figure 7. Officially, issuing an individual iOS certificate requires enrollment in the Apple

[1] https://github.com/myappcloud/appstore-connect-api

Developer Program, which costs $99 USD annually (approximately RMB 688). In contrast, the multi-layered resale structure leads to a higher effective cost for end users, as markups are introduced at successive intermediary layers during certificate redistribution. In this subsection, we break down the profit flows across each layer within the resale model.

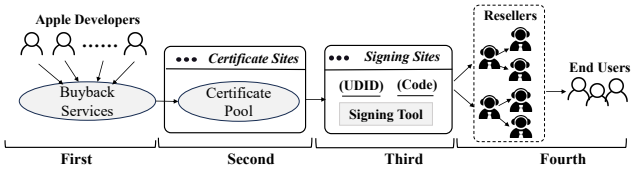


Figure 7: A multi-layered certificate circulation process.

First Layer: Upstream Suppliers. At the top of the chain, upstream suppliers typically obtain Apple Developer accounts from real users, with some platforms (e.g., *www.minclouds.com*) explicitly offering buy-back services. Newly created accounts are commonly purchased for RMB 900–1000. Older accounts (e.g., registered before 2009), exempt from Apple's current device-binding limits [30], are reclaimed at higher prices due to their larger device capacity.

Second Layer: Certificate Sites. As mentioned, certificate sites maintain certificate pools using *.p8* keys, often sourced in bulk from upstream suppliers. Besides using them internally, these platforms also resell accounts to other operators for profit. According to available price listings, standard accounts are priced between RMB 2400 and 2800, while older, unrestricted accounts can reach up to RMB 7500.

Third Layer: Signing Sites. To reduce the technical requirements of obtaining credentials via *.p8* keys, certificate sites directly offer ready-to-use *.p12* certificates and provisioning profiles for individual devices and distribute them to downstream signing sites. Pricing varies by warranty duration. For instance, services with no warranty support are typically priced at around RMB 15 per device, while services with a 330-day warranty are offered at approximately RMB 40.

Fourth Layer: Social Media Resellers. Social media resellers advertise signing services to attract users and often recruit downstream agencies to expand their reach. They sell prepaid codes that can be redeemed for signing credentials on the corresponding signing sites. While these transactions are nominally framed as certificate purchases, the quoted prices often reflect bundled services, such as access to curated app resources. Moreover, as each resale agency introduce additional markups, prices advertised on social media are often substantially inflated compared to the actual redemption cost. According to our survey, the retail prices range from RMB 30–40 for services without warranty and RMB 80–100 for those offering a 330-day warranty.

Overall, the misuse of Apple developer certificates for self-signing has fostered a highly profitable gray-market. Although each certificate is officially limited to 100 iOS devices, relaxed enforcement on macOS allows up to 200 devices to be bound in practice. Accordingly, the cost of binding a device is approximately RMB 3.4 (688/200). As the certificate passes through the illicit market, end users often pay 10 to 30 times this amount, leading to profit margins of 900% to 3,000%. While long-term warranty services may occupy more device slots and slightly increase the actual cost per device, the system remains highly lucrative, underscoring the strong economic incentives behind its expansion.

Answers to RQ1: The single-entry service is accessed through signing sites and remains highly active with substantial user traffic. Certificate sites acquire developer certificates programmatically using *.p8* keys and distribute them through signing sites. The involvement of multiple resale layers results in profit amplification across intermediaries. These practices not only violate Apple's developer agreement but also fuel the spread of unauthorized apps.

6 Demystifying iOS Signing Tools

Based on the sampling approach described in Section 4.2, we collected 12 representative samples, including two distinct versions of *All-in-One Sign*, two of *Easy Sign*, and eight custom-branded tools configured by different certificate sites. As the first unofficial apps that users must install during the self-signing process, signing tools serve as a foundation for signing other apps. In this section, we introduce their core functionalities and focus on the code signing methods (RQ2).

6.1 Feature Integration

Regardless of their origin or brand, all signing tools integrate the three essential elements of self-signing, offering usability features to support the signing and installation process. First, the tools support certificate management, allowing users to import signing credentials and view essential details such as validity (Figure 8a). In practice, these credentials are often preloaded, as the tool and certificates are typically obtained from the same signing site. In addition, the tools provide an app repository that offers direct access to popular apps for immediate download, eliminating the need to source app packages manually (Figure 8b). Third, a visual signing interface allows users to configure signing parameters such as app name, icon, bundle identifier, and dynamic library injection or removal (Figure 8c). Unlike signing service providers in the fragmented ecosystems, signing tools uniquely combine code signing with app distribution in a single interface. Unlike signing service providers in fragmented ecosystems, signing tools uniquely combine code signing and app distribution within a single interface, making them a central and user-friendly platform for self-signing.

(a) Certificate (b) App Source (c) Custom Sign

Figure 8: Core features of a typical signing tool.

6.2 Signing Method

The primary function of signing tools is to perform code signing. Through reverse engineering, we find that all tools operate without Apple's official signing utilities, such as codesign [15] or Xcode, instead relying on unofficial implementations.

Custom-Branded Tools. Custom-branded signing tools show strong similarity. 5/8 tools share the same bundle name ("HillMountPlatform"), and follow a fixed-format bundle identifier ("com.mango<4-digit>.test<4-digit>"), suggesting derivation from a common template. Moreover, we directly identified the *"zsign"* function in all custom-branded signing tools, indicating their reliance on the open-source utility zsign [39]. Instead of invoking the zsign as a command-line tool, these tools implement signing by embedding its original source code, including argument parsing, Mach-O binary processing, and CMS (Cryptographic Message Syntax) signature construction. Zsign mimics Apple's signing process to generate valid-looking signatures for iOS runtime checks. It also forges *entitlements.plist* to sign arbitrary apps with one valid provisioning profile, regardless of the App ID. While such misuse highlights the lack of enforcement in the underlying signing logic, it is part of zsign's internal behavior and not the focus of our study. To further substantiate our observation, we examined three more samples from other signing sites and found the same zsign function. This suggests that our analysis is stabilizing and the custom-branded signing tools are largely built upon a single technical foundation.

Mature Tools: All-in-One Sign and Easy Sign. Regarding All-in-One Sign and Easy Sign, both apps have two variants, a regular and a permanent one. Although the zsign function is absent in all observed versions, the regular version in each app implements code signing analogous to zsign, but appears to be a complete reimplementation in Objective-C. Its code structure includes custom classes (e.g., "YYYMachO" and "YYYSignAsset") that perform tasks such as certificate embedding and entitlements injection, replicating key elements of the original zsign workflow. In contrast, the permanent version in both apps deviates entirely from conventional methods and bypasses code signing by exploiting the IOHIDFamily kernel vulnerability (CVE-2022-46689 [16]). As this vulnerability is only effective on iOS $\leq$ 16.2, this version is compatible with a limited set of devices. Exploitation grants the app privileged access comparable to a jailbreak, enabling it to hijack system apps (e.g., Tips) via vnode replacement and use Kernel File Descriptor (KFD) techniques to persist as a disguised system app, thereby remaining operational even after the certificate expires. Once embedded, it establishes a privileged runtime environment that allows additional unsigned apps to be freely installed and executed without triggering Apple's code signing enforcement.

Answers to RQ2: Third-party tools, mostly derived from zsign, facilitate code signing in the self-signing gray-market by replicating Apple's signing process outside the official toolchain. Some variants further exploit iOS vulnerabilities to bypass verification with elevated privileges. By bundling repositories of unsigned apps and supporting customized signing parameters, these tools have become the backbone of self-signing distribution and provide a fertile ground for the proliferation of unauthorized iOS apps.

7 Distributed Apps in the Wild

Based on the method in Section 4.3, we collected 8,216 unauthorized app entries and downloaded 2,654 IPA files from the 14 app repositories obtained from signing tools. In this section, we first provide an overview of the types of these apps, then examine common modification techniques and highlight potential security risks (RQ3).

7.1 Categories of Unauthorized Apps

Our analysis of the app types considers all 8,216 collected IPA entries. Although some entries lack valid download links, they are still included in our analysis since metadata such as app names and descriptions remain informative. To classify the distributed apps, we applied two processing steps.

(1) *Name Normalization.* Unofficially distributed apps often follow arbitrary and non-standard naming practices, leading to multiple variants of the same underlying application. To support accurate grouping and large-scale analysis, we normalized app names into a canonical form, referred to as the *"baseapp"*. First, names were split using common delimiters (e.g., underscores, hyphens, brackets), and non-essential components such as version numbers or functionality tags were removed (e.g., "Spotify_8.9.68" $\rightarrow$ "Spotify"). Then, using the common modifiers collected from the previous step, we used regular expressions to strip suffixes from names without explicit delimiters (e.g., "YouTube Pro" $\rightarrow$ "YouTube"). Following the above steps, we applied manually defined mappings to unify names referring to the same app, accounting

for translations, abbreviations, and rebranded names (e.g., "X" → "Twitter"). This process extracted 1,883 unique baseapps from all 8,216 IPA entries. Among them, 1,023 baseapps are associated with multiple entries, with the largest groups corresponding to globally popular apps such as WeChat and TikTok (see Table 4 in Appendix).

(2) *App Store Verification.* To assess the origin of these unofficial apps, we queried the iTunes Search API to check whether each baseapp exists in the official App Store by name[1]. We searched four regional stores in order, including China (CN), the United States (US), South Korea (KR), and Hong Kong (HK). Priority was given to CN, as the dataset primarily targets the Chinese distribution market. Since the API supports fuzzy matching, we compared the returned *trackName* with the queried baseapp and manually corrected ambiguous cases. Based on their presence in the App Store, we classified the distributed apps into two categories.

• ***Modified Variants.*** Among the 1,883 baseapps, 79.6% (1,498) matched official App Store entries. These baseapps are dominated by Games (548), Utilities (177), and Photo & Video (168), together accounting for 59.6% of the total. The unofficial variants of these baseapps share names with legitimate apps but are distributed through unofficial channels, suggesting they are repackaged or modified versions of official releases. Analysis of corresponding IPA metadata and App Store search results reveal widespread functional changes. The most prevalent modifications are VIP unlocking (40.9%) and ad removal (27.5%), which are relatively general-purpose changes. Other enhancements include simplified interfaces, game cheats, and location spoofing. Beyond functional alterations, distribution practices reveal deliberate repackaging to evade platform controls. Specifically, 569 baseapps that are unavailable in the Chinese App Store are redistributed to Chinese users, bypassing regional restrictions [63]. Furthermore, 222 baseapps that were originally paid are circulated for free through cracked *.ipa* files. Collectively, these unofficial variants reflect unmet user demands for broader access, lower cost, and enhanced functionality, reinforcing a parallel distribution ecosystem beyond official channels.

• ***Independent Apps.*** On the other hand, 385 baseapps had no corresponding match in the App Store. Excluding cases where ambiguous naming prevented precise matching, the corresponding IPA entries appear to be independently distributed rather than repackaged official apps. These include self-developed tools (e.g., Cymusic, SourceRead), utilities for jailbroken devices (e.g., CocoaTop, NathanLR), pirated streaming apps with adult content, and previously delisted official apps such as Renren Video and Pocket. Some entries are unofficial iOS ports of popular apps from Android or Steam, such as Undertale. These apps are blocked from the App Store due to policy or copyright issues and are only available through unofficial channels. Their presence highlights the broader scope of the unofficial iOS distribution ecosystem and raises legal and ethical concerns.

[1]Search format: "https://itunes.apple.com/search?term={Baseapp}&country={Country}&entity=software&limit=1"

7.2 Dylib-Based Modification Mechanism

As outlined in Section 7.1, apps distributed through Ad Hoc self-signing are largely modified versions of legitimate ones. Comparing file structures with their official counterparts reveals that modifications are achieved primarily through additional dynamic libraries (*.dylib* files). These injected libraries enable runtime behavioral changes, offering a flexible and minimally invasive means of modification. To understand the implementation and risks of distributed apps, we focus on dylib-based modifications, which represent the dominant and technically coherent mechanism in unauthorized app distribution. While often used to add customized features, dylibs are also embedded in some standalone apps (e.g., pirated video apps with dylibs for ad blocking), so our analysis considers each dylib as an independent unit regardless of host app type.

Dylib Extraction and Analysis. We extracted dylibs from the app bundle (*Payload/*.app/*) and excluded system libraries, such as those starting with *libswift*, focusing on additional dylibs bundled with unauthorized apps. In total, we got 8,085 third-party dylibs (averaging 3.05 per app) from 2,654 IPA files. These dylibs correspond to 1,402 unique names and are primarily used for runtime hooking to alter app behavior. To examine their hooking logic, we manually reverse engineered the five most frequent dylibs (Table 5 in Appendix) using IDA Pro and compared the binaries with radiff2 [31] to account for name collisions. The top four exhibited high internal consistency (standard deviation $\sigma < 0.1$, mean similarity $\mu > 0.9$), allowing us to directly select one representative instance from each. In contrast, *Tg@TrollstoreMios.dylib*, exhibited higher binary variance across samples ($\sigma = 0.20$, $\mu = 0.65$), and was excluded. We also examined dylibs explicitly associated with VIP unlocking and ad blocking, the two most common injected features.

7.2.1 Dylibs for Environment Setup

Our analysis reveals that the top three *dylibs* are primarily used for environment setup rather than direct feature cracking. *libsubstrate.dylib*, the core library of Cydia Substrate, is often bundled into unofficial apps to enable hooking and code injection on non-jailbroken devices. *libJailedShim.dylib* acts as a compatibility shim, offering placeholder hooks for Substrate [68], Substitute [42], and LibHooker [43] to prevent crashes when these frameworks are absent. *Tg@TrollStoreKios.dylib* achieves system check bypass by intercepting methods in system classes like "NSFileManager" and "CKContainer", injecting fake container paths and disabling CloudKit-related initialization routines to evade entitlement and configuration constraints. These dylibs act as scaffolding that prepares the environment for subsequent

modifications, and their frequent usage suggests a reusable strategy in unauthorized app tampering.

7.2.2 Dylibs for VIP Unlocking

VIP unlocking refers to bypassing in-app purchase (IAP) mechanisms to unlock premium features without actual payment. Among the 2,654 downloaded apps, 442 explicitly and solely mentioned VIP cracking in their descriptions. Given the complexity of reverse engineering, we adopt a qualitative analysis based on theoretical saturation [60]. We first randomly selected 10 apps from this subset and expanded until no new methods emerged. In total, we analyzed 15 apps until saturation, identifying three dominant cracking techniques.

• *StoreKit Hooking.* This method targets Apple's in-app purchase (IAP) flow. When the user initiates a purchase, the dylib intercepts the StoreKit transaction callback and injects a forged SKPaymentTransaction object marked as "purchased". The app accepts this fake transaction as valid, unlocking VIP content without contacting Apple's servers. This technique is commonly implemented by "Store.dylib" (the fourth most common dylib in our dataset) and is often paired with UI hooks to hide the real payment interface.

• *Local Check Hooking.* This method hooks functions that check the user's subscription status, such as "isVIP" or "hasActiveSubscription", and forces them to return true. This tricks the app into unlocking premium features without a valid subscription. In some cases, it also overrides stored values in "NSUserDefaults" to persist a fake VIP status. This approach only bypasses client-side checks and works only in apps that lack server-side validation.

• *Network Response Hooking.* This method hooks system classes such as "NSURLSession" to monitor network traffic. When it identifies subscription-related requests by matching keywords like "subscription" or "purchases", it injects forged responses in the corresponding callbacks to simulate a valid VIP state. JSON parsing classes like "NSJSONSerialization" may also be hooked to modify the response structure.

7.2.3 Dylibs for Ad Blocking

Ad blocking refers to the removal or suppression of in-app advertisements to improve user experience. Among the downloaded samples, 145 apps were modified solely for this purpose. We manually analyzed 13 representative apps until reaching saturation, identifying three ad blocking methods targeting different stages of the ad pipeline.

• *Network-Level Blocking.* This method blocks ads by hooking system methods in "NSURLSession" to intercept ad-related network traffic. Some implementations replace known ad URLs with invalid ones, effectively preventing the requests from reaching ad servers. Others allow the requests to proceed but intercept the responses and modify them to return empty data, leaving the app with no content to display.

• *SDK-Level Interference.* This approach hooks key SDK methods to disrupt ad loading logic. It targets ad initialization (e.g., "loadAdDataWithCount", "setAdManager"), configuration ("setAdSourceModel", "setAdType"), and validity checks ("isSDKAd", "isValid"), preventing the ad module from launching or rendering. By targeting common control points, these techniques are broadly compatible with major ad SDKs and effective across different apps.

• *UI-Level Suppression.* This method hides ads at the UI level by hooking system classes like "UIAlertView" or "UIViewController" to block pop-up ads. Other techniques include accelerating ad playback via the "setRate" method in "AVPlayer", or shrinking ad views to an invisible size. While straightforward and non-intrusive, these methods do not block the underlying ad content or its associated network traffic.

7.3 Security and Privacy Risks

The distribution of unauthorized apps undermines developer revenues and reduces Apple's App Store income by diverting users from official distribution channels. More importantly, such apps may also introduce security and privacy risks to end users. To characterize these risks, we first scanned all 8,085 extracted dylibs using VirusTotal [37] and found that only three were flagged as malicious. This suggests that most injected components are added to support additional functionality rather than to distribute overt malware. However, their implementation of these functionalities still poses substantial security risks. The injected dylibs hook into the app process to tamper with runtime memory, manipulate traffic, and bypass system controls. These interventions compromise the app logic and weaken iOS's security isolation, creating a reusable attack surface for further abuse. To illustrate how these general risks manifest in specific apps, we conducted a case study of WeChat-specific *dylibs*, as WeChat contains the largest number of modified variants (as mentioned in Section 7.1) and manages highly sensitive user data.

Injected Dylibs in WeChat. In total, we identified 140 downloadable WeChat variants, from which we extracted 175 dylibs after deduplication, averaging 18.7 dylibs per variant. Each injected *dylib* functions as a plugin that extends app capabilities. Based on app descriptions and supplemental online sources, we classified them into five categories, including display enhancements, social feature extensions, comprehensive toolkits, plugin management utilities, and stability support components. The distribution is shown in Table 3, with comprehensive toolkits and display enhancements being the most prevalent. Unlike the single-purpose *dylibs* analyzed in Section 7.2, these *dylibs* support more complex behaviors and require broader hook coverage. For instance, *PKCWeChatTools.dylib*, a common comprehensive toolkit, hooks 132 functions spanning message handling, UI rendering, settings management, image processing, and payment logic, illustrating the breadth of such modifications. These large-scale modi-

Table 3: Categories of WeChat dylibs and their distribution.

Category	Subcategory	Description	Dylibs	Typical Example	Usage
Display Enhancements	UI Theming	Customize UI themes, like fonts, icons.	14	ThemeBox.dylib	532
	Elements Removal	Modif UI components to change layout.	6	WCMyPageInfoCenter.dylib	146
	Ad Removal	Remove advertisement within WeChat.	7	WeChatNoAd.dylib	19
Social Features	Chat Functionality	Add chat-related features, like message filtering.	12	DouTu.dylib	274
	Friend Management	Hide or fake friend relationships in contact.	4	FakeFollow.dylib	128
	Moments	Customize Moments features, like post pinning.	3	WCEnableFriendsStar.dylib	10
Comprehensive Toolkits	/	Bundle multi advanced features.	15	PKCWeChatTools.dylib	540
Plugin Management	/	Provide user-facing plugin management.	7	wcplugins.dylib	164
Stability Support	Runtime Support	Provide basic runtime support.	7	libsubstrate.dylib	161
	Integrity Bypass	Bypass system integrity mechanisms.	7	WXGetVersion52.dylib	72
	Push Notifications	Restore system push notifications.	4	WechatPushMsgPage.dylib	49

fications inevitably create security risks, which we further summarized in the following three threats.

• ***Perform Unauthorized Behaviors.*** Injected dylibs effectively bypass WeChat's built-in restrictions and gain control over core functionalities. They manipulate app logic to introduce customized features and gain unfair advantages by modifying wallet balances, spoofing fitness data, or faking location. These modifications enable cheating behaviors and undermine app trust assumptions. In addition, some dylibs directly invoke internal methods to simulate interactions without genuine user involvement and authorization. For example, they can send messages on behalf of users upon detecting specific events, automatically join group chats through QR scanning, or call payment APIs to initiate low-value transactions that silently probe contact relationships. These behaviors bypass user consent, leading to unauthorized social actions.

• ***Sensitive Data Access.*** To support advanced social features such as message filtering and anti-recall, injected dylibs often access sensitive user data, including real-time messaging, WeChat IDs, and contact lists. This access breaks platform data boundaries and enables the manipulation of private information, posing significant risks to user privacy and data integrity. Alarmingly, certain dylibs go further by exfiltrating data to third-party servers. For instance, *xnsp.dylib* stealthily uploads user messages to an external endpoint when triggered by a special keyword "TTS+". This mechanism is intended to provide a text-to-speech (TTS) feature, but ultimately results in silent and unauthorized transmission of private conversations. In addition, other dylibs bypass iOS sandbox protections by recursively modifying file system permissions. This grants access to restricted directories, undermining the isolation guarantees of the iOS sandbox.

• ***System Capability Exploitation.*** Some dylibs extend beyond WeChat's internal logic to exploit iOS system capabilities for unauthorized access. For example, by modifying "NSUserDefaults" and toggling undocumented flags such as camenable, they force WeChat to use the native iOS camera interface instead of its built-in camera module. Similarly, dylibs activate CallKit [25] to present WeChat calls as the native phone calls, enabling call answering from the lock screen. When combined with the multitasking camera access entitlement, WeChat can silently activate the camera and maintain call sessions in the background, without user awareness. These actions bypass WeChat's permission mechanisms and iOS runtime protections, granting silent access to sensitive system features and raising risks of covert surveillance.

In summary, injected *dylibs* hook into WeChat's internal and system methods to provide customized features. While these may enhance user experience, they break app and system security boundaries. This not only introduces direct security and privacy risks but also creates a foothold for further exploitation. Although WeChat is used as a case study, similar threats exist in other unofficial apps that rely on similar hooking mechanisms. As with many gray-market apps [57, 61], users are often unaware of the risks or willingly trade security for enhanced functionality, contributing to the continued growth of this underground ecosystem.

Answers to RQ3: Most apps distributed through the self-signing market are modified versions of legitimate apps that rely on dynamic library injection to add customized features. Their distribution disrupts the revenue models of iOS and app developers, while also raising copyright concerns. Moreover, such modifications bypass system safeguards, introducing security threats such as unauthorized user actions, data collection, and system capability exploitation.

8 Discussion

Our study reveals the gray-market of Ad Hoc sideloading. It is sustained by the resale of developer certificates, advanced unofficial signing techniques, and large-scale app distribution, forming a mature and well-organized industry. In this section, we present actionable strategies to curb the continued growth

of the market, and discuss the limitations of our study.

8.1 Mitigation Recommendations

Ad Hoc self-signing distribution undermines the integrity of the iOS ecosystem, inflicts economic losses on developers, and exposes users to serious security and privacy risks. While some official app developers (e.g., Tencent) have adopted reactive measures such as suspending user accounts [13] to curb the use of unauthorized apps, current efforts target end users and neglect issues in the distribution pipeline. Furthermore, reliance on legitimate Apple-issued certificates, rather than overtly malicious exploits, blurs the line between normal use and abuse. Collectively, these factors underscore the urgent need for stronger, multi-level regulatory intervention. Based on our findings, we propose the following recommendations.

iOS-Level: Restricting Certificate Abuse. Apple plays a key role in curbing certificate abuse. Reducing the validity of Ad Hoc provisioning profiles from one year to a shorter period (e.g., three months) would raise the cost of misuse while minimally affecting legitimate testing. In parallel, Apple should enhance oversight of individual developer accounts, as it already does with enterprise accounts. For example, provisioning profiles bound to known blacklisted App IDs could be flagged and revoked. We have already reported our findings and practical recommendations to Apple, hoping to support effective mitigation of certificate misuse.

User-Level: Containing the Spread of Illicit Service. User demand fuels the growth of this ecosystem. To contain such illicit services, it is essential to raise user awareness of unofficial distribution risks and block their exposure channels, particularly on social media. Based on our observation, these services are currently advertised openly without textual obfuscation (e.g., keyword alteration), revealing weak platform restrictions. To further evade accountability, some vendors issue disclaimers that the distributed apps are just "for educational use only" and require users to delete them shortly after download, thus shifting responsibility for misuse to end users. These practices underscore the need for stronger content moderation. Simple keyword-based detection would be effective, and the domain patterns of signing sites identified in our study can also support detection.

Developer-Level: Strengthening Integrity Protections. Distributed apps are predominantly modified through injected *dylib* files, a simple and reusable approach that lowers the barrier for cracking. To prevent tampering and unauthorized modification, official app developers should adopt effective defensive strategies. Although certain apps implement jailbreak detection, such measures can be easily bypassed through dynamic library hooking, limiting their effectiveness. As a result, relying solely on client-side mechanisms is inadequate. Critical logic, such as VIP status and environment verification, should be handled through server-side validation. In addition, runtime integrity checks, such as monitoring unexpected memory changes, are effective against dylib-based tampering.

8.2 Limitations

Drawing on our collected data and analysis results, we gained a comprehensive understanding of the self-signing gray-market. Nevertheless, several limitations should be noted.

Geographic Focus. First, our data collection focuses on integrated self-signing services in China. While similar services exist in other regions, they are typically fragmented. Some sites focus solely on certificate reselling, such as Applep12 [11], while others provide only IPA files, such as CyPwn [18]. These services represent elements of the broader ecosystem but do not provide a unified platform that enables full-process analysis. We therefore selected the Chinese market as it offers a structured and observable workflow for analysis. Nevertheless, our methodology can be applied to fragmented services elsewhere, and the insights gained apply to Ad Hoc sideloading more broadly.

Coverage of Signing Sites. The discovery of signing sites serves as the foundation of our dataset, as it directly determines access to signing tools and distributed apps. While full coverage is not feasible, the collected data is sufficient to capture the key structural patterns of the ecosystem. While our method focuses on reseller-based sites identified through domain patterns, some self-managed sites with custom domains and direct API integration may be missed. However, as they share the same backend for certificate redemption, their effect is confined to scale estimation and does not affect our analysis results of certificate circulation.

Sampling of Signing Tools. Regarding signing tools, our sampling strategy balances practical constraints with the need for diversity. The sample set captures a representative range of tools and is sufficient to reach analytical saturation, revealing common patterns in code signing implementations.

Scope of Distributed Apps. Our dataset consists of distributed apps collected through signing tools, providing sufficient scale for ecosystem-level analysis. While community-curated repositories (e.g., *iappsrepo.github.io*) also exist, our focus is specifically on apps distributed through this self-signing channel rather than on all unauthorized sideloaded apps. Moreover, our analysis focuses on dylib-based modifications. For independent grayware apps, although we examined the embedded *dylibs*, their core functionality is typically implemented in heavily obfuscated binaries with heterogeneous designs, making systematic analysis impractical and thus left for future work. Nevertheless, our dylib-based analysis reveals the key characteristics of the majority of distributed apps and highlights their significant security risks.

9 Related work

Sideloading has long been a primary security concern for Apple [40]. Zheng et al. [81] revealed the abuse of enterprise

certificates for sideloading and the misuse of private APIs in distributed apps. Likewise, Bashan [21] demonstrated the widespread use of enterprise certificates to distribute unofficial or even malicious apps. More recently, Guo et al. [50] measured the underground app distribution on Telegram, documenting frequently used sideloading methods in the wild, including TestFlight and WebClips. Hu et al. [52] further analyzed the abuse of WebClips and showed that they are widely exploited to disseminate gambling, fraud, and pornographic content. However, none of these studies address Ad Hoc self-signing sideloading. This model misuses personal developer certificates and supports peer-to-peer distribution, making it difficult for Apple to monitor.

Jailbreak App Ecosystem. Beyond sideloading, jailbreaking enables the installation of unauthorized iOS applications by granting elevated system privileges. Amaral et al. [44] noted that jailbreaking is primarily motivated by users' desire to extend device capabilities and achieve greater personalization. For some highly committed Apple users, it reflects a pursuit of self-expression and resistance to platform control [56]. As a result, jailbreak app stores such as Cydia [17] distribute pirated and modified apps that provide additional capabilities for users [44]. Despite the lack of formal review, Egele et al. [48] suggested apps are not necessarily more malicious than those on the App Store in terms of data usage. However, other characteristics of modified iOS apps, particularly their modification methods and intended purposes, remain largely unexplored. Although jailbreaking and sideloading rely on different technical mechanisms, they share the same goal of bypassing iOS restrictions to access unauthorized apps with enhanced capabilities. Accordingly, the existing studies on jailbreak ecosystems offer relevant background for understanding unauthorized app distribution on iOS.

Third-Party App Distribution on Android. Developers on Android can freely generate and manage their own signing keys, resulting in an ecosystem where app development and distribution are less restricted than on iOS. While Google Play serves as the official app store, third-party markets remain widespread and officially permitted. Due to the absence of rigorous security vetting, these markets show a significantly higher prevalence of malware, fake, and cloned apps than Google Play [55, 66, 76]. A major driver of this phenomenon is the widespread practice of app repackaging [58]. Zhou et al. found that approximately 13% of apps in third-party markets are repackaged [83]. Repackaging typically involves modifying an existing APK and re-signing it. Attackers can embed malicious code into the repackaged app [64] or alter its original logic to introduce additional functionality such as game cheating, premium-feature unlocking, or advertisement suppression [67]. These repackaged variants also tend to request more permissions than their legitimate counterparts [78], amplifying security and privacy risks. Beyond repackaging, Shi et al. [71] identified a growing trend in which Android malware samples abuse app-virtualization technologies as an alternative distribution channel. Virtualization frameworks embed additional code as plugins, allowing malicious logic to be loaded at runtime and enabling it to evade repackaging-based detection mechanisms (e.g., [65, 70, 79, 82]). Such behavior closely resembles the dylib injection observed on iOS, providing important context for our investigation into modification practices on iOS.

To our knowledge, no prior work has systematically examined unauthorized iOS distribution via Ad Hoc self-signing and its security implications, particularly those arising from modded apps. Beyond work on app distribution, other research has addressed broader iOS security issues [46, 69, 77], such as sandbox policy flaws [47] and exposed network services [73], which have informed defenses that progressively reinforced the iOS security design. In this context, our study complements existing work by highlighting the overlooked yet practically significant weakness, with the same goal of strengthening the resilience of the iOS ecosystem.

10 Conclusion

Abuse of Apple's Ad Hoc provisioning with individual developer certificates has fostered a covert, integrated gray-market for unauthorized iOS app distribution. Leveraging its single-entry structure, our study provides the first systematic analysis of this ecosystem, revealing the multi-layered certificate circulation, common signing techniques with evasion tactics, and widespread distribution of modified apps. We further detail the technical methods used for app modification and the associated security risks, including unauthorized actions, sensitive data exfiltration, and system capability exploitation. These findings highlight the urgent need for intervention to curb such unauthorized distribution, and we propose actionable countermeasures at the platform, developer, and user levels.

Acknowledgments

We would like to thank our shepherd and the anonymous reviewers for their insightful comments. This work is supported by the National Key Research and Development Program of China (No. 2023YFC3321303), the National Natural Science Foundation of China (62102218, 62302258). Baojun Liu is the corresponding author.

Ethical Considerations

Our work examined the gray-market iOS sideloading ecosystem using a user-centric data collection approach. Although our institution does not have an Institutional Review Board (IRB), we carefully considered ethical guidelines before starting the research to ensure compliance with the Menlo Report's principles [54]. Below, we discuss the ethical implications for each stakeholder and the dual-use considerations.

Stakeholder Analysis The key stakeholders involved in our research include legitimate app developers, Apple, end users, and gray-market resellers. Additionally, researchers constitute an important stakeholder, as the research process itself may expose them to personal safety risks.

• Apple and legitimate app developers are the most direct victims of this abusive ecosystem, as it disrupts their legitimate business models and results in financial losses. To avoid further harm, we did not distribute, reuse, or share any materials obtained during our measurements. Reverse engineering was limited to non-legitimate apps and focused solely on identifying implementation details and security risks, without reproducing or exploiting their functionality. After completing the experiments, we reported the identified gray-market activities to Apple, including our purchasing activities and the misused developer IDs we obtained, along with suggested mitigation measures to help address the ongoing abuse.

• End users are both beneficiaries and potential victims within the ecosystem. While they benefit from the functionality of unauthorized apps, they also face associated security risks. Our measurement does not involve direct interaction with end users. The passive DNS dataset used in this study contained no personally identifiable information (PII). Specifically, fields such as client IP addresses were excluded, thereby eliminating the risk of disclosing users' privacy.

• Gray-market resellers are the primary operators in this ecosystem. While we collected data on various service providers and their websites, our study focuses on analyzing their operational models rather than identifying individual actors. We did not collect, nor attempt to infer, any personally identifiable information about resellers. To mitigate the risk of unintentionally promoting illegal activities, sensitive data related to these services (i.e., the signing sites) are only available upon request, following the practices of prior work [51].

• Researchers, as third-party observers, constitute an important stakeholder. To mitigate potential risks and safeguard researcher privacy, we created several new accounts with fictitious identity information for interactions with service providers and employed WeChat Pay to purchase the necessary data, which better protects user identity compared to credit card payments. Additionally, all analysis of acquired app samples was conducted in controlled sandbox environments to prevent exposure to potential malicious software.

Dual-Use Considerations Our research involves an inherent dual-use dilemma. Although we dissected a mature abusive practice, the analysis may inadvertently facilitate other malicious actors entering the market as resellers or modified app developers for profit. Additionally, monetary transactions and interactions with these services necessarily involve limited financial support to the gray-market.

To reduce these risks, we followed established best practices from prior work [45, 59]. All purchases were made strictly for research purposes, and transactions were limited to the minimum necessary to obtain a representative sample. We selected services across different price tiers to capture ecosystem diversity. After analyzing several platforms, we found that additional services exhibited nearly identical operational patterns, suggesting that further diversity was not needed. As a result, we limited our study to 12 services, with a total expenditure of RMB 500 ($\approx$ USD 70), thereby minimizing financial support to gray-market operators. This limited engagement, guided by the public interest, avoided stimulating market activity while still enabling meaningful analysis. Additionally, to avoid unintentionally enabling abuse, we promptly reported our findings to Apple to alert them to the potential risks and encourage remedial action. We also restricted access to sensitive materials in our artifact release, including complete lists of signing sites, collected IPA files, and extracted dylibs, as such data could otherwise be misused.

Overall, our study minimizes potential negative impacts on key stakeholders and provides actionable insights to help them better understand and respond to the underlying risks, supporting more effective efforts to counter this illicit market.

Open Science

The initial datasets we collected consist of 3,359 URLs of signing sites, 12 sampled signing tools, and metadata for 8,216 IPA entries, which formed the foundation for our subsequent analysis. While open artifact dissemination is essential for transparency and reproducibility, unrestricted release of these data could inadvertently facilitate such unauthorized distribution channel and enable malicious actors to exploit the gray-market as resellers or modified app developers. To balance transparency with the need to prevent unintended promotion of the ecosystem, we adopt a tiered artifact-release model. We provide a public repository [1] containing representative samples from each data category, as well as some essential analysis results, to help readers better understand the ecosystem we focus on. We also release the analysis scripts used in our study, allowing other researchers to reproduce our workflow and apply the methodology in different contexts. The complete datasets are hosted in a restricted-access repository [2]. The data is available upon request for academic research purposes only. To limit the potential for misuse, requesters must provide a brief description of their intended research use. By

[1] Public repository: https://doi.org/10.5281/zenodo.17850998

[2] Restricted-access repository: https://doi.org/10.5281/zenodo.17846379

combining public and restricted access with appropriate vetting of requesters, we responsibly balance the need to prevent misuse of sensitive data with the goal of supporting open and reproducible science.

References

[1] Alist. https://alistgo.com/. Accessed: 2025-07.

[2] All-in-one sign. https://sign.drnrt8.cn/sign/. Accessed: 2025-07.

[3] Altstore pal. https://altstore.io/. Accessed:2025-07.

[4] App store connect api. https://developer.apple.com/documentation/appstoreconnectapi. Accessed: 2025-07-08.

[5] Apple announces changes to ios, safari, and the app store in the european union. https://www.apple.com/newsroom/2024/01/apple-announces-changes-to-ios-safari-and-the-app-store-in-the-european-union/. 2024-01.

[6] Apple closes access to apple developer enterprise program (adep) for russian developers. https://appleworld.today/2025/02/apple-closes-access-to-apple-developer-enterprise-program-adep-for-russian-developers. 2025-02.

[7] Apple developer agreement. https://developer.apple.com/support/downloads/terms/apple-developer-agreement/Apple-Developer-Agreement-20250318-English.pdf. Accessed: 2025-07.

[8] Apple developer program: From code to customer. https://developer.apple.com/programs/. Accessed:2025-07.

[9] Apple has blocked us from renewing our enterprise developer license. what are our alternatives? https://www.reddit.com/r/iOSProgramming/comments/vhi1vh/apple_has_blocked_us_from_renewing_our_enterprise/. Accessed:2025-07.

[10] The apple wiki of trollstore. https://theapplewiki.com/wiki/TrollStore. Accessed: 2025-07.

[11] Applep12. https://applep12.com/. Accessed: 2025-07.

[12] Apptesters ipa signer. https://signer.apptesters.org/. Accessed: 2025-07.

[13] Can visitors be viewed in wechat moments? tencent responded: We are cracking down on it. http://news.china.com.cn/2025-05/27/content_117898336.shtml. Accessed: 2025-07.

[14] Can visitors be viewed in wechat moments? tencent response. https://mp.weixin.qq.com/s/N7hzjGHUdOvNo9kJ37COqg. 2025-05.

[15] Code signing tasks. https://developer.apple.com/library/archive/documentation/Security/Conceptual/CodeSigningGuide/Procedures/Procedures.html. Accessed: 2025-07.

[16] Cve-2022-46689. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-46689. Accessed: 2025-07.

[17] Cydia: An alternative application distribution platform for jailbroken ios devices. https://www.cydiafree.com/. Accessed: 2025-12.

[18] Cypwn ipa library. https://ipa.cypwn.xyz/. Accessed: 2025-12.

[19] Discord-group chat that's all fun & games. https://discord.com/. Accessed: 2025-07.

[20] Easy sign. https://esign.yyyue.xyz/. Accessed: 2025-07.

[21] Enterprise apps: Bypassing the ios gatekeeper. https://infocondb.org/con/black-hat/black-hat-asia-2016/enterprise-apps-bypassing-the-ios-gatekeeper. 2016-03.

[22] Fastadmin. https://www.fastadmin.net/. Accessed: 2025-07.

[23] How fairplay works: Apple's itunes drm dilemma. http://www.roughlydrafted.com/RD/RDM.Tech.Q1.07/2A351C60-A4E5-4764-A083-FF8610E66A46.html. 2007-02.

[24] Ida pro-a powerful disassembler, decompiler and a versatile debugger. https://hex-rays.com/ida-pro. Accessed: 2025-07.

[25] Ios developer documentation-callkit. https://developer.apple.com/documentation/callkit. Accessed: 2025-07.

[26] ios security-ios 8.3 or later. https://www.apple.com/vn/privacy/docs/iOS_Security_Guide.pdf. 2015-04.

[27] Ipasign. https://sign.ipasign.cc/. Accessed: 2025-07.

[28] ipastore. https://ipa.store/. Accessed: 2025-07.

[29] Kravasign. https://kravasign.com/. Accessed: 2025-07.

[30] Managing your registered devices list. https://developer.apple.com/news/?id=11022009. 2009-12.

[31] radiff2-r2wiki. https://r2wiki.readthedocs.io/en/latest/tools/radiff2/. Accessed: 2025-07.

[32] r/sideloaded in reddit. https://www.reddit.com/r/sideloaded/. Accessed:2025-07.

[33] Scammers have been using apple's testflight to distribute malicious ios apps. https://9to5mac.com/2022/03/16/scammers-have-been-using-apples-testflight-to-distribute-malicious-ios-apps. 2022-05.

[34] "super signature" exposes the underground cybercrime ecosystem. https://www.spp.gov.cn/spp/zdgz/202203/t20220328_552074.shtml. 2022-03.

[35] Testflight. https://developer.apple.com/testflight/. Accessed:2025-07.

[36] Tutorial posts from the r/sideloaded subreddit on reddit. https://www.reddit.com/r/sideloaded/?f=flair_name%3A%22Tutorial%22. Accessed:2025-07.

[37] Virustotal. https://www.virustotal.com/gui/home/upload. Accessed: 2025-12.

[38] Xiaohongshu (rednote). https://www.xiaohongshu.com/download. Accessed: 2025-07.

[39] Zsign. https://github.com/zhlynn/zsign. Accessed: 2025-07.

[40] Apple Inc. Building a trusted ecosystem for millions of apps. https://www.apple.com/privacy/docs/Building_a_Trusted_Ecosystem_for_Millions_of_Apps.pdf. Accessed: 2025-07.

[41] Apple Inc. Device registration updates. https://developer.apple.com/help/account/reference/device-registration-updates/. Accessed: 2025-07.

[42] comex. Substitute - an open-source code substitution library for jailbroken ios devices. https://github.com/comex/substitute. Accessed: 2025-07.

[43] CoolStar. Libhooker - modern hooking library for ios and ipados. https://libhooker.com/docs/. Accessed: 2025-07.

[44] Adriana da Rosa Amaral and Rosana Vieira Souza. User resistance and repurposing: a look at the ios jailbreaking scene in brazil. *AoIR Selected Papers of Internet Research*, 2013.

[45] Louis F DeKoven, Trevor Pottinger, Stefan Savage, Geoffrey M Voelker, and Nektarios Leontiadis. Following their footsteps: Characterizing account automation abuse and defenses. In *Proceedings of the Internet Measurement Conference 2018*, pages 43–55, 2018.

[46] Zhui Deng, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. iris: Vetting private api abuse in ios applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 44–56, 2015.

[47] Luke Deshotels, Razvan Deaconescu, Mihai Chiroiu, Lucas Davi, William Enck, and Ahmad-Reza Sadeghi. Sandscout: Automatic detection of flaws in ios sandbox profiles. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 704–0716, 2016.

[48] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Pios: Detecting privacy leaks in ios applications. In *NDSS*, volume 2011, page 18th, 2011.

[49] Craig N Goodwin and Sandra Woolley. Sideloading: An exploration of drivers and motivations. In *35th International BCS Human-Computer Interaction Conference*, pages 1–6. BCS Learning & Development, 2022.

[50] Yanhui Guo, Dong Wang, Liu Wang, Yongsheng Fang, Chao Wang, Minghui Yang, Tianming Liu, and Haoyu Wang. Beyond app markets: Demystifying underground mobile app distribution via telegram. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 8(3):1–25, 2024.

[51] Catherine Han, Anne Li, Deepak Kumar, and Zakir Durumeric. Characterizing the {MrDeepFakes} sexual deepfake marketplace. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 5169–5188, 2025.

[52] Qinyu Hu, Songyang Wu, Wenqi Sun, Zhushou Tang, Chaofan Chen, Zhiguo Ding, and Xiaomei Zhang. Measurement of the usage of web clips in underground economy. *arXiv preprint arXiv:2209.03319*, 2022.

[53] iOSGods Community. Sideloadly - ios ipa sideloading tool. https://sideloadly.io, 2025. Accessed:2025-07.

[54] Erin Kenneally and David Dittrich. The menlo report: Ethical principles guiding information and communication technology research. *Available at SSRN 2445102*, 2012.

[55] Yosuke Kikuchi, Hiroshi Mori, Hiroki Nakano, Katsunari Yoshioka, Tsutomu Matsumoto, and Michel Van Eeten. Evaluating malware mitigation by android market operators. In *9th Workshop on Cyber Security Experimentation and Test (CSET 16)*, 2016.

[56] Michael SW Lee and Ian Soon. Taking a bite out of apple: Jailbreaking and the confluence of brand loyalty, consumer resistance and the co-creation of value. *Journal of Product & Brand Management*, 26(4):351–364, 2017.

[57] Yijing Liu, Yiming Zhang, Baojun Liu, Haixin Duan, Qiang Li, Mingxuan Liu, Ruixuan Li, and Jia Yao. Tickets or privacy? understand the ecosystem of chinese ticket grabbing apps. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5107–5124, 2024.

[58] Lannan Luo, Yu Fu, Dinghao Wu, Sencun Zhu, and Peng Liu. Repackage-proofing android apps. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 550–561. IEEE, 2016.

[59] Ariana Mirian, Joe DeBlasio, Stefan Savage, Geoffrey M Voelker, and Kurt Thomas. Hack for hire: Exploring the emerging market for account hijacking. In *The World Wide Web Conference*, pages 1279–1289, 2019.

[60] Janice M Morse. The significance of saturation, 1995.

[61] Collins W Munyendo, Yasemin Acar, and Adam J Aviv. "desperate times call for desperate measures": User concerns with mobile loan apps in kenya. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2304–2319. IEEE, 2022.

[62] Mahin Naderifar, Hamideh Goli, Fereshteh Ghaljaie, et al. Snowball sampling: A purposeful method of sampling in qualitative research. *Strides in development of medical education*, 14(3):1–6, 2017.

[63] Jay Peters. Apple is reportedly removing apps in china that don't have a government license. 2023-10.

[64] Husnain Rafiq, Nauman Aslam, Muhammad Aleem, Biju Issac, and Rizwan Hamid Randhawa. Andromalpack: enhancing the ml-based malware classification by detection and removal of repacked apps for android systems. *Scientific Reports*, 12(1):19534, 2022.

[65] Sajal Rastogi, Kriti Bhushan, and BB Gupta. Android applications repackaging detection techniques for smartphone devices. *Procedia Computer Science*, 78:26–32, 2016.

[66] Chuangang Ren, Kai Chen, and Peng Liu. Droidmarking: resilient software watermarking for impeding android application repackaging. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 635–646, 2014.

[67] Luis A Saavedra, Hridoy S Dutta, Alastair R Beresford, and Alice Hutchings. Modzoo: A large-scale study of modded android apps and their markets. In *2024 APWG Symposium on Electronic Crime Research (eCrime)*, pages 162–174. IEEE, 2024.

[68] Jay Freeman (saurik). Cydia substrate. https://www.cydiasubstrate.com/. Accessed: 2025-07.

[69] David Schmidt, Alexander Ponticello, Magdalena Steinböck, Katharina Krombholz, and Martina Lindorfer. Analyzing the ios local network permission from a technical and user perspective. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 4229–4247. IEEE, 2025.

[70] Yuru Shao, Xiapu Luo, Chenxiong Qian, Pengfei Zhu, and Lei Zhang. Towards a scalable resource-driven approach for detecting repackaged android applications. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 56–65, 2014.

[71] Luman Shi, Jiang Ming, Jianming Fu, Guojun Peng, Dongpeng Xu, Kun Gao, and Xuanchen Pan. Vahunt: Warding off new repackaged android malware in app-virtualization's clothing. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pages 535–549, 2020.

[72] Charlie Sorrel. Crackulous strips copy protection from iphone apps. https://www.wired.com/2009/02/crackulous-stri/. 2009-02.

[73] Zhushou Tang, Ke Tang, Minhui Xue, Yuan Tian, Sen Chen, Muhammad Ikram, Tielei Wang, and Haojin Zhu. {iOS}, your {OS}, everybody's {OS}: Vetting and analyzing network services of {iOS} applications. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2415–2432, 2020.

[74] ThinkSkySoft. i4tools - free ios device manager. https://www.i4.cn/. Accessed:2025-07.

[75] Verimatrix. ios sideloading explained: Risks and security tips. https://www.verimatrix.com/cybersecurity/knowledge-base/ios-sideloading-explained-risks-and-security-tips. 2024-12.

[76] Haoyu Wang, Zhe Liu, Jingyue Liang, Narseo Vallina-Rodriguez, Yao Guo, Li Li, Juan Tapiador, Jingcun Cao,

and Guoai Xu. Beyond google play: A large-scale comparative study of chinese android app markets. In *Proceedings of the Internet Measurement Conference 2018*, pages 293–307, 2018.

[77] Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee. Jekyll on {iOS}: When benign apps become evil. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 559–572, 2013.

[78] Shishuai Yang, Guangdong Bai, Ruoyan Lin, Jialong Guo, and Wenrui Diao. Beyond the horizon: Exploring cross-market security discrepancies in parallel android apps. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*, pages 558–569. IEEE, 2024.

[79] Qiang Zeng, Lannan Luo, Zhiyun Qian, Xiaojiang Du, and Zhoujun Li. Resilient decentralized android application repackaging detection using logic bombs. In *Proceedings of the 2018 international symposium on code generation and optimization*, pages 50–61, 2018.

[80] Mingming Zhang, Xiang Li, Baojun Liu, Jianyu Lu, Yiming Zhang, Jianjun Chen, Haixin Duan, Shuang Hao, and Xiaofeng Zheng. Detecting and measuring security risks of hosting-based dangling domains. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 7(1):1–28, 2023.

[81] Min Zheng, Hui Xue, Yulong Zhang, Tao Wei, and John CS Lui. Enpublic apps: Security threats using ios enterprise and developer certificates. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 463–474, 2015.

[82] Wu Zhou, Xinwen Zhang, and Xuxian Jiang. Appink: watermarking android apps for repackaging deterrence. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 1–12, 2013.

[83] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pages 317–326, 2012.

[84] Zimperium. Beyond the app store: The hidden risks of sideloading apps. https://zimperium.com/blog/the-hidden-risks-of-sideloading-apps. 2024-07.

A Supplementary Charts

Following are some supplementary figures and tables provided for a better understanding of the paper.


```
{
  "name": "qweather",
  "type": 0,
  "version": "3.5.16",
  "versionDate": "2025-04-03",
  "Description":
        "Unlock premium features without login, supports
         certificate installation and desktop widgets.",
  "lock": "0",
  "downloadURL":
        "https://pan2.iosr.cn/d/%E7%A7%BB%E5%8A%A8%E4%BA%
        91%E7%9B%983/ipa/20250330/%E5%92%8C%E9%A3%8E%E
        5%A4%A9%E6%B0%94_3.5.16.ipa",
  "isLanZouCloud": "index",
  "iconURL":
        "https://www.iosr.cn/uploads/logo/2025/03/30/c2fbd38d6b3150f9c
        5cd92c7111b03a0.jpg",
  "tintColor": "",
  "size": "72687288.32"
}
```


Figure 9: The signing tool integrates the app repository through URLs, which provide JSON manifests describing app metadata (illustrated with "qweather" as an example).

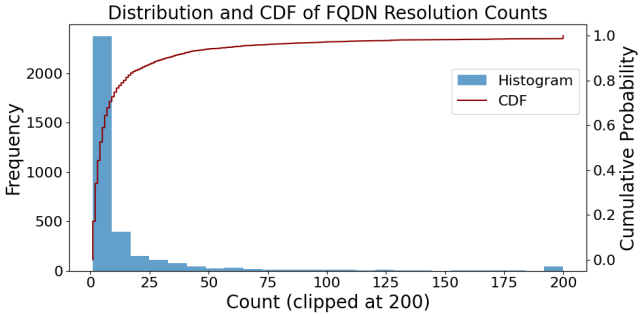
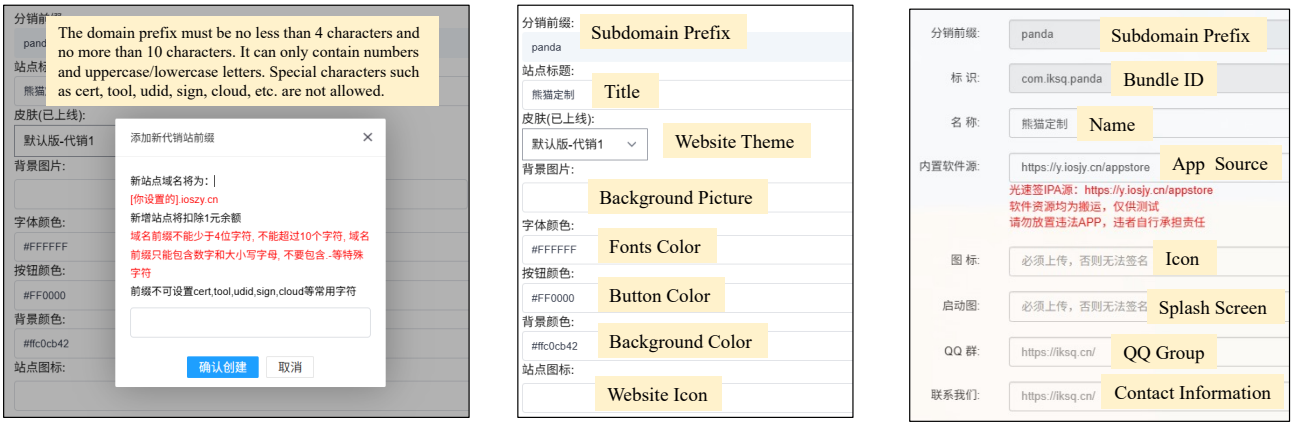


Figure 10: Distribution and CDF of the resolution counts for signing sites ("fqdn" in passive DNS dataset), with the majority observed fewer than five times.

Table 4: Top 10 baseapps by number of unofficial variants.

Baseapp	Bundle ID	Numbers
WeChat	com.tencent.xin	492
Xiaohongshu	com.xingin.discover	141
Douyin	com.ss.iphone.ugc.Aweme	100
X	com.atebits.Tweetie2	97
TikTok	com.zhiliaoapp.musically	91
Soul	com.soulapp.cn	74
Kwai	com.jiangjia.gif	68
Bilibili	tv.danmaku.bilianime	65
Spotify	com.spotify.client	65
Youtube Music	com.google.ios.youtubemusic	61



(1) Create a reseller-based signing site. (2) Configure site information. (3) Set up the associated signing tool.

Figure 11: The key steps of reseller-based deployment model provided by certificate sites. Resellers can quickly set up their own signing sites under specific subdomain rules and create custom-branded signing tools.

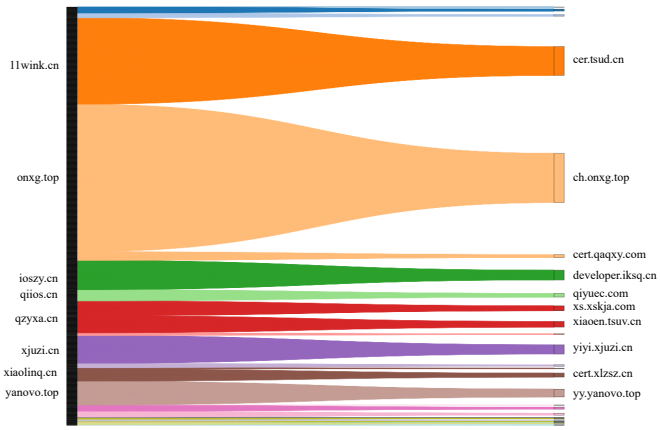


Figure 12: Relationship between signing sites (left, aggregated at the SLD level) and their corresponding certificate sites (right). Only major signing sites are labeled.

Table 5: Top 5 common used third-party dylibs and their sample dissimilarity distribution.

Dylib	**Numbers**	σ[1]	μ[2]
libsubstrate.dylib	1217	0.02	0.97
libJailedShim.dylib	272	0.09	0.94
Tg@TrollStoreKios.dylib	196	0.0	1.0
Store.dylib	178	0.0	1.0
Tg@TrollstoreMios.dylib	113	0.20	0.65

[1] σ means standard deviation.
[2] μ means mean similarity.