Longitudinal Analysis of iOS In-App Purchase Implementation Vulnerabilities

Izabella C. Melo[1], **Vitoria Pinheiro**[1], **Divanilson R. Campelo**[1]

[1]Centro de Informática – Universidade Federal de Pernambuco (UFPE)
CEP 50740-560 – Recife – PE – Brazil

{imcm,vps3,dcameplo}@cin.ufpe.br

Abstract. *In-app purchases are a vital monetization strategy for developers, but improper implementation leads to security vulnerabilities. This paper presents the first longitudinal analysis of the evolution of these vulnerabilities in iOS apps' in-app purchases. Initially, we manually analyzed 34 native apps and found three main client-side vulnerabilities in 21 of them. Then, using an app-agnostic tool, we tested 100 trending apps and identified vulnerabilities in 30 of them. Two years later, 23 of these apps still had vulnerabilities, and five new ones emerged, showing minimal improvement. These findings underscore the ongoing issues with in-app purchase implementations and highlight the need for iOS developers to strengthen their security practices.*

1. Introduction

Mobile devices are the primary platform for digital consumption. In 2024, consumer spending in mobile app stores reached $150 billion, with the Apple App Store accounting for 67% of this revenue [Sensor Tower 2025, Business of Apps 2025]. While Google Play leads in downloads, the App Store remains significantly more profitable per user [Sensor Tower 2022]. This revenue is largely driven by in-app purchases managed via Apple's StoreKit APIs. For each transaction made using these APIs, Apple collects commissions ranging from 15% to 30% [Apple 2025a].

Despite this financial success, fraud remains a critical concern. In 2025, Apple reported blocking over US$9 billion in fraudulent transactions [Apple 2025b]. However, the report focuses on account and credit card fraud, *largely ignoring vulnerabilities in client-side in-app purchase implementations*. These flaws allow attackers to bypass payment checks by manipulating the app's execution. Exploitation typically requires access to a decrypted IPA (iOS App Store Package) file. Although Apple's FairPlay DRM encrypts these files [Foresman 2012], they are decrypted at runtime and can be extracted even on non-jailbroken devices using modern techniques [Promon 2024] or obtained via third-party repositories [iOSGods 2025]. Furthermore, recent regulations on alternative app stores further facilitate the distribution of modified IPAs [Apple 2024, Mendes 2025].

This paper demonstrates that implementation vulnerabilities in in-app purchases directly expose iOS apps to security risks. An app is considered vulnerable from an in-app purchase perspective if its premium content can be accessed without payment. Since these flaws stem from individual developer implementations rather than the operating system itself, they cannot be patched through universal iOS updates.

Our investigation is structured in the following way. We began with a manual analysis of iOS apps using static and dynamic reverse-engineering on a jailbroken iPhone. We

identified three primary implementation vulnerabilities: *i*) **Insecure Storage** of premium status; *ii*) **Missing Receipt Validation**; and *iii*) **Unprotected Code** susceptible to runtime manipulation. We successfully exploited at least one of these implementation vulnerabilities in 61% of the 34 apps analyzed.

To scale our research, we developed a black-box, app-agnostic tool that requires no knowledge of the app's internal architecture. When testing 100 top free apps with in-app purchases from App Store, our tool bypassed security in 30% of cases. Two years later, we updated the tool to run on an Apple Security Research Device (SRD) [Apple 2025c]. Despite API improvements and new iOS releases, the landscape remained largely unchanged, with 28% of the apps still exhibiting the same vulnerabilities. All findings were responsibly disclosed to the affected developers.

In a nutshell, the main contributions of this paper are:

- **First longitudinal analysis** of client-side in-app purchase vulnerabilities in modern iOS, tracking their persistence over several years and OS versions.
- **Identification and documentation** of three core in-app purchase implementation vulnerabilities, including the development of runtime exploits to demonstrate their impact.
- **Tool Release:** We provide the first open-source tool[1] for testing these vulnerabilities, enabling developers to audit their own apps.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 provides essential background on in-app purchases in iOS. Section 4 defines the attack models. Section 5 describes the methodology. Section 6 presents the experimental results. Section 7 discusses ethical considerations, and Section 8 concludes the paper.

2. Related Work

2.1. Mobile Security

While security research on Android is extensive, covering cryptographic errors [Egele et al. 2013], banking vulnerabilities [Reaves et al. 2017], and cloud API leaks [Zuo et al. 2019], research on iOS remains limited due to platform restrictions [Orikogbo et al. 2016]. Existing iOS studies focus primarily on privacy leaks [Egele et al. 2011], malware evolution [García and Rodríguez. 2016], or jailbreak evasion in banking apps [Kellner et al. 2019].

2.2. Paid Content for Free

Previous research has extensively documented in-app purchase vulnerabilities in Android. Tools like *VirtualSwindle* [Mulliner et al. 2014] and *FreeMarket* [Reynaud et al. 2012] demonstrated that between 59% and 60% of analyzed Android apps were susceptible to runtime in-app purchase bypasses. Similarly, *PaymentScope* [Zuo and Lin 2022] found that 23% of Unity-based Android games were vulnerable. In the Chinese ecosystem, studies on third-party SDKs further highlighted how improper implementation leads to payment fraud [Yang et al. 2017, Shi et al. 2021].

[1] https://github.com/izmcm/bypass-iap-ios

For iOS, research is sparse. *PaymentGuard* [Zhou et al. 2021] was designed to detect fraudulent transactions by tracking malicious purchasing services, but it focuses on fraud detection rather than the underlying implementation flaws addressed in our work. While historical exploits like *zond80* [ZonD80 2012] targeted vulnerabilities in older iOS versions (pre-iOS 6.0), our research demonstrates that modern iOS apps remain vulnerable due to systemic developer errors in client-side checks. Additionally, previous work on Android does not transfer to iOS, as its unique security architecture requires a specialized approach. *To our knowledge, this is the first longitudinal study to analyze and exploit these vulnerabilities in recent iOS releases from an attacker's perspective.*

3. Technical Background

3.1. iOS In-App Purchase APIs

Apple currently supports two frameworks for in-app purchase implementation [Apple 2025d]:

- **Original API for in-app purchase:** Based on encrypted App Store receipts that require manual validation by the developer. Available since 2009, it remains the only option for Objective-C apps and is still widely used in legacy projects despite its recent deprecation at WWDC24.
- **In-App Purchase API:** Introduced in 2021 (iOS 15.0), it utilizes JSON Web Signatures (JWS) for transaction validation, offering a more modern security framework.

  Our research focuses on client-side vulnerabilities that persist across API versions.

3.2. Implementing In-App Purchases

Implementing in-app purchases follows a standardized sequence, as illustrated in Fig. 1:

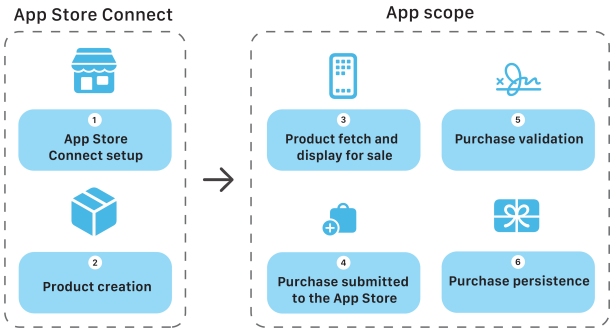

Figure 1. Steps for in-app purchase implementation.

1. **App Store Connect setup.** Developers use this platform to manage app distribution, configure banking information, and accept the legal agreements required for monetization.
2. **Product creation.** The developer creates each product by selecting its type, name, and unique Product ID. The product types can be: *i*) Consumables, which are items users can utilize only once after a purchase; *ii*) Non-Consumables, items that users can use indefinitely; *iii*) Auto-renewable subscriptions, i.e., subscriptions that will automatically renew; and *iv*) Non-renewing subscriptions, which end after the contracted time.

3. **Product fetch and display for sale.** After creating the products, developers must use the Product IDs created in the previous step to fetch the information using the chosen API: the *Original API for in-app* purchase provides `SKProductsRequest(productIdentifiers:)`, and the response is a `SKProductsResponse` object received by the `productsRequest(_:didReceive:)` delegate method; the *In-App Purchase API*, otherwise, provides a `await Product.products(for identifiers:)`. In both cases, the returned objects contain information such as the products' descriptions and prices to display for sale. Configuring and fetching product information from App Store Connect allows developers to modify product properties directly in the platform without releasing new app versions.

4. **Purchase submitted to the App Store.** To submit a purchase request in the Original API for in-app purchase, the developer must create a payment request by passing the selected item's product ID and adding it to a queue for processing. When the transaction is processed, the App Store returns the result to the app through the `paymentQueue(_:updatedTransactions:)` delegate. The delegate receives a `SKPaymentTransaction` that contains a `transactionState` indicating the transaction's state (e.g., purchased, purchasing, failed, etc.). Using the In-App Purchase API, the developer must call only the `purchase(options:)` method on the chosen `Product` object. This method returns a `Product.PurchaseResult` that can be .success, .pending, or .userCancelled.

5. **Purchase validation.** In the Original API for in-app purchase, the `transactionState` returned by the payment queue is insufficient for secure confirmation because it can be easily spoofed. To prevent fraud, developers must manually validate the encrypted App Store receipt found at the `appStoreReceiptURL`. This file contains the immutable purchase history and is essential for verifying that a transaction actually occurred. In contrast, the newer In-App Purchase API automates much of this process. A successful purchase returns a `VerificationResult<Transaction>`, which can be either `.verified` or `.unverified`. While this framework is more robust, security remains dependent on the developer's implementation; if the code fails to explicitly check for the `.verified` status, the security benefits of the API are nullified. In such instances, the app remains just as vulnerable to client-side manipulation as those using the older implementation.

6. **Purchase persistence.** Lastly, recording purchases is vital to making products available when the user needs them, even without an internet connection. For the Original API for in-app purchase, the Apple developer documentation describes ways to record the purchase according to its type, such as using the App Store receipt, saving a boolean in iCloud, or storing in UserDefaults [Apple 2025e, Apple 2025f] as shown in Fig. 2.

   On the other hand, the In-App Purchase API provides `Transaction.currentEntitlements`, which returns a list of transactions the user has made. This method returns the updated list when the device has an internet connection and has a local cache that also works offline.

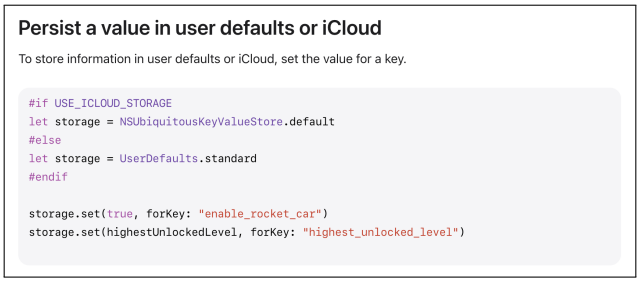

Figure 2. Persisting a purchase from the Apple developer docs (August 2025)

4. Attack Models

In our model, the attacker is an adversary with privileged device control, capable of reverse engineering and runtime instrumentation via jailbreaking or IPA extraction. Their objective is to bypass in-app purchase payments for personal gain and, optionally, redistribute modified binaries via alternative stores to enable others to gain unauthorized access. For this, we developed two attack models: i) **App-Specific Attack:** Requiring bespoke identification and exploitation of vulnerabilities unique to a target app. ii) **App-Agnostic Attack:** Leveraging common implementation flaws to launch automated, widespread exploits across multiple apps.

4.1. App-Specific Attack Model

The left part of Fig. 3 illustrates the attack tree for the App-Specific Attack Model, highlighting key steps in the process. We assume the attacker already possesses the app's decrypted IPA file, as depicted in the figure's first step. A decrypted IPA can be obtained using a jailbroken iPhone, a Security Research Device, or alternative app stores that distribute decrypted versions [iOSGods 2025, und3fy.dev 2025].

Once the attacker has the decrypted IPA, they can reverse engineer the code to identify vulnerabilities. At this stage, several techniques can be employed, including dynamic method hooking to modify the app's behavior at runtime or direct binary patching.

4.2. App-Agnostic Attack Model

In the App-Agnostic Attack Model (the right part of Fig. 3), step 1 involves identifying cross-app vulnerabilities that can be exploited across multiple targets. This step aligns with the collection tactic described in the MITRE ATT&CK Matrix [MITRE ATT&CK 2018] and is typically carried out using the same techniques presented in Section 4.1. Based on the discovered vulnerabilities, the attacker can then craft a script capable of exploiting most of them.

Unlike the previous model, the App-Agnostic Attack Model requires a jailbroken device or a Security Research Device, since installing system-level scripts requires low-level access to the device. Because this model operates at the device level, it can automatically unlock premium features in *any vulnerable app installed on the iPhone*. However, while generic exploits are simpler to deploy across multiple apps, they are also less effective than targeted exploits designed for specific weaknesses in a single app.

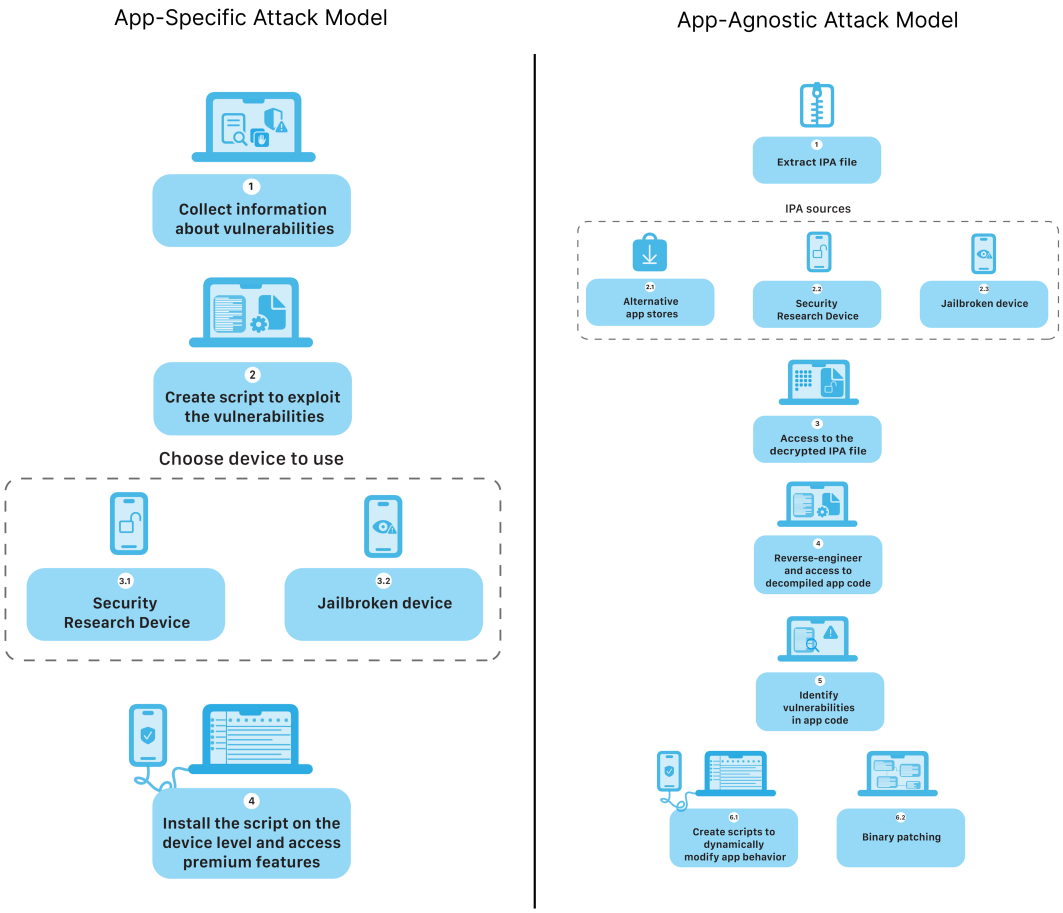

Figure 3. Attack Models.

5. Methods

Our approach focused on two main strategies for uncovering common client-side vulnerabilities in iOS in-app purchases. First, we conducted an app-specific analysis targeting the most popular free native iOS apps in the Health & Fitness, Productivity, and Education categories – areas with high user engagement and increased risk from insecure in-app purchases. We selected and tested 34 apps based on the following criteria to ensure our findings were both relevant and robust:

- The app needed to be free to download and had to offer at least one feature that users could unlock through a subscription or purchase with Apple's in-app purchase;
- The premium feature could not inherently depend on a server (e.g., cloud storage apps). Server vulnerabilities are out of the scope of this paper;
- The premium feature could not be linked only to the user account (e.g., Netflix app). Typically, these applications offer subscriptions outside the App Store to avoid paying Apple's fees.

During our app-specific analysis, we performed static analysis by reverse-engineering apps to uncover potential errors and evaluate their data storage practices. This groundwork was crucial for our subsequent app-agnostic analysis. For the broader phase, we

downloaded 100 additional apps from the App Store's "Top Free Apps" section, including Apple-featured selections like "Apps We Love" and "Editor's Choice." We then carried out dynamic analysis, modifying app behavior at runtime to identify vulnerabilities.

We conducted the app-specific analysis from March to September 2022, followed by the app-agnostic analysis from September 2022 to July 2023. To assess changes over time, we repeated the app-agnostic analysis on the same set of apps from June to August 2025. This extended timeline allowed us to comprehensively track app trends and evolving vulnerabilities across multiple years.

5.1. Tools

Initial experiments were conducted on a jailbroken iPhone 7 (iOS 14.6), with findings later validated on an Apple Security Research Device running iOS 18 to ensure generalizability across modern OS versions. We utilized Frida [Frida 2025] for dynamic instrumentation, injecting scripts to hook and intercept function calls, thereby allowing redirection of app logic – such as changing an `isPurchased` return value from `false` to `true`.

For persistent, app-agnostic analysis, we used Theos [Theos 2025] to develop tweaks that monitor system calls at the device level. On the Security Research Device, where standard jailbreak tools are unavailable, we implemented a custom `dylib`-based hooking mechanism to achieve similar results. The final analysis pipeline involved extracting decrypted IPA files from device memory using Frida iOS Dump [AloneMonkey 2025] and performing static inspection with Hopper Disassembler [Cryptic Apps 2025].

5.2. App-Specific Analysis

The initial phase of our approach comprised an app-specific analysis of 34 selected apps. This analysis was crucial for gaining a deeper understanding of vulnerabilities introduced during the app development process. Moreover, it laid the groundwork for the subsequent app-agnostic analysis by enabling us to identify the most prevalent vulnerabilities for broader exploration.

For the app-specific analysis, we conducted both static and dynamic analyses on each of the 34 apps. Figure 4 summarizes the steps involved in the static analysis. We extracted decrypted IPA files from the jailbroken iPhone's memory and used the Hopper Disassembler to thoroughly examine the decompiled binaries. This process enabled us to assess app functionality and systematically pinpoint potential security vulnerabilities. During the analysis, we searched for keywords such as "premium", "plus", "subscription", "VIP", and "debug" to locate code segments associated with in-app purchase implementations, premium feature unlocking, or developer debug functions that could potentially be exploited.

After pinpointing potentially vulnerable code, we proceeded to dynamic analysis. This phase involved developing Frida scripts to hook into app processes during runtime, allowing us to validate and expand upon the vulnerabilities identified in the static analysis. Figure 4 also illustrates the dynamic analysis workflow.

The following subsections detail each step of our methods, supplemented with real-world examples from our research.

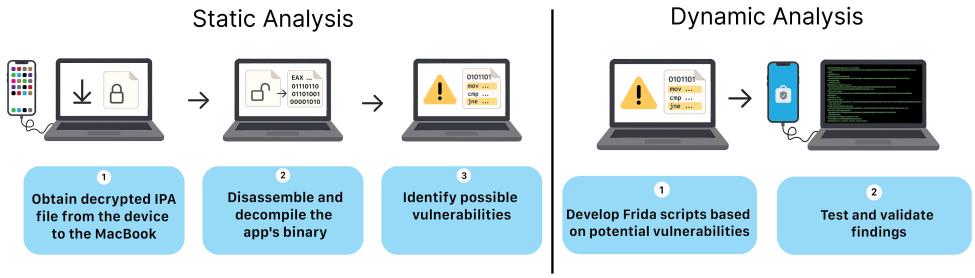

Figure 4. Static and Dynamic Analysis flow.

5.2.1. Case 1 - Unprotected Code + Insecure Storage

Fig. 5 presents simplified pseudocode for the decompiled `ISVIPUser` function, recovered from one of the apps in the set of 34 apps. As the name implies, this function determines whether a user has VIP or premium access within the app.

```
+(bool) isVIPUser {
  r20 = [SAMKeychain passwordForService:r2
                  account:@ "Purchased"];

  r19 = [NSString stringWithFormat:r20];
  r20 = @selector(isEqualToString:@"YES");

  if (objc_msgSend(r19, r20) != 0x0) {
    r20 = 0x1;
  } else {
    r0 = [NSUserDefaults standardUserDefaults];
    r20 = objc_msgSend(r0, @selector(objectForKey:@"VPNExpired"));
  }

  return r20;
}
```

Figure 5. Pseudocode of the `ISVIPUser` decompiled function.

The most straightforward exploitation technique is to alter the `ISVIPUser` function so that its return value (r0) is always *true* (`0x1`). We classify this as an Unprotected Code vulnerability: the function controlling premium status is unprotected and easily located, allowing attackers always to modify it to grant access. Figure 6 displays a Frida script used to intercept `ISVIPUser` and force it to return *true*.

While the Unprotected Code vulnerability is easily exploited, deeper inspection reveals additional security concerns. The function initially retrieves the `Purchased` key from `SAMKeychain` and checks whether its value equals *YES*. If so, the function returns *true* and exits. Otherwise, it performs an extra check: it retrieves the `VPNExpired` key from UserDefaults and returns its value. If `VPNExpired` is set to *true*, the function also returns *true*, thereby unlocking premium access.

UserDefaults is an Apple API designed to persistently store small amounts of non-sensitive data across app launches and is commonly used to save application state in iOS apps. Many tutorials – including Apple's own documentation – recommend using

```
function changeISVIPUserReturn() {
  var instance = ObjC.classes.User
  Interceptor.attach(instance.ISVIPUser.implementation, {
    onLeave: function(oldValue) {
      var newValue = ptr(0x1);
      oldValue.replace(newValue);
    }
  });
}
```

Figure 6. Frida code to exploit the unprotected code from `ISVIPUser` function.

UserDefaults to save app state after a purchase [Apple 2025e, Apple 2025f]. However, because UserDefaults stores data in an unencrypted key-value database, it should be used only for non-sensitive information, such as UI preferences or usernames.

We argue that UserDefaults should not be used to store sensitive data, such as an app's premium state, because its contents are easily accessible and modifiable by attackers. Despite these risks, many apps continue to misuse UserDefaults for sensitive information.

5.2.2. Case 2 - Insecure Storage + Unprotected Code.

During app development, developers commonly introduce debug code – such as shortcuts and bypass mechanisms – to facilitate testing and troubleshooting. If this code is not thoroughly removed or secured prior to release, it may persist in the production build, resulting in an Unprotected Code vulnerability that exposes the app to security risks.

While analyzing a decrypted IPA file with Hopper Disassembler, we identified multiple debug-related classes in one of the 34 apps examined. Among them was the `DebugManager`, which handle debugging functions to streamline development. Our initial attempt to access the `DebugManager` class targeted the `isAvailableDebug` method. In the production version, this method always returns *false*, indicating that debug functionality is disabled by default. Although the return is always *false*, the debug code remains present in the app, allowing us to intercept and modify the method's return value to enable debug functionality, as demonstrated in Fig. 7.

```
function changeIsAvailableDebugReturn() {
  var instance = ObjC.classes.DebugManager.instance()
  Interceptor.attach(instance.isAvailableDebug.implementation, {
    onLeave: function(oldValue) {
      var newValue = ptr(0x1);
      oldValue.replace(newValue);
    }
  });
}
```

Figure 7. Frida code to exploit `isAvailableDebug` function.

Enabling debug mode alone did not immediately grant access. Further investigation re-

vealed another method, `isFastAccessAvailable` (Fig. 8). This method facilitates direct access to debug mode through the app's UI by retrieving a Boolean value from the UserDefaults key `ud_de-bug_content_screen_state` and returning it. To further manipulate the app's behavior, we used an alternative UserDefaults-hooking strategy. Rather than altering the app function's return value or directly modifying UserDefaults, as shown in Fig. 9, we target the system API. This approach prevents inadvertent overwriting of stored values and offers greater scalability, as it can be applied across multiple apps rather than being limited to a specific method.

```
-(bool) isFastAccessAvailable {
  r0 = [NSUserDefaults standardUserDefaults];
  return [r0 boolForKey: @"ud_debug_content_screen_state"];
}
```

Figure 8. Pseudocode of the `isFastAccessAvailable` decompiled function.

```
function hookUserDefaultsForSpecificKey() {
  var shouldForceTrue = false
  var instance = ObjC.classes.NSUserDefaults.standardUserDefaults;
  Interceptor.attach(instance.boolForKey.implementation, {
    onEnter: function (args) {
      var key = new ObjC.Object(args[2]).toString();
      if (key === "ud_debug_content_screen_state") {
        shouldForceTrue = true;
      }
    },
    onLeave: function (retval) {
      if (shouldForceTrue) {
        retval.replace(ptr("0x1"));
      }
    }
  });
}
```

Figure 9. Frida code to exploit `NSUserDefaults` system API.

Finally, the methods `isAvailableDebug` and `isFastAccessAvailable` work together to activate the app's debug mode, making it accessible via a debug menu. The app's debug menu included an option to enable premium features, granting access to its full capabilities.

5.2.3. Case 3 - Missing Receipt Validation.

Another prevalent issue with improper in-app purchase implementation, previously discussed in Section 3.2, involves handling the `transactionState` variable during the final stages of the Original API for in-app purchase workflow. This variable, an *enum*, indicates the current state of each transaction and can assume the following values: i) `purchasing (= 0)`: The App Store has started processing the transaction; ii) `purchased (= 1)`: The transaction was successfully processed; iii) `failed (= 2)`: The

transaction failed; iv) `restored (= 3)`: The content was previously purchased; and v) `deferred (= 4)`: The transaction requires external action.

Upon receiving a `purchased (= 1)` transaction, Apple advises developers to verify the App Store receipt—an encrypted file, signed with an Apple certificate, that records all in-app purchases. This verification can be performed locally or previously via an App Store server. However, with Apple's receipt verification server now deprecated, developers are responsible for implementing the receipt validation process in full.

The design of the Original API for in-app purchase makes clear that omitting receipt verification introduces a significant security risk. This vulnerability can be readily exploited and demonstrated with Frida by modifying the `transactionState` variable of the `SKPaymentTransaction` class.

```
1 function hookTransactionState() {
2   var instance = ObjC.classes.SKPaymentTransaction;
3   Interceptor.attach(instance["-
     transactionState"].implementation, {
4     onLeave: function(oldValue) {
5       oldValue.replace(ptr('0x1'));
6     }
7   });
8 }
```

Figure 10. Frida code to exploit `SKPaymentTransaction` system API.

Fig. 10 shows code that modifies the value returned by the `transactionState` system API. This script is scalable to unlock premium app content in any app that uses the Original API for in-app purchases and does not perform receipt validation, since it operates at the system level.

5.3. App-Agnostic Analysis

After completing static and dynamic analyses on 34 apps, we shifted to an app-agnostic analysis. By rewriting our Frida scripts in Theos syntax, we enhanced their versatility and enabled us to test an additional 100 apps with a single tweak on our jailbroken device, eliminating the need for app-specific modifications.

Our generic script targeted vulnerabilities such as Insecure Storage and Missing Receipt Validation. While Unprotected Code was common in the initial analysis, exploiting it typically demands manual intervention and relies heavily on individual app implementations, making app-specific exploits more practical. The following subsections explain the logic and operation of the Theos scripts used in our app-agnostic analysis.

5.3.1. Generalizing Insecure Storage exploits.

Our analysis identified two `NSUserDefaults` methods frequently used to gate premium content: `boolForKey(_:)` and `doubleForKey(_:)`. By intercepting these methods at runtime, an attacker can manipulate an app's perception of the user's subscription status without modifying the underlying database.

The exploitation strategy relies on semantic key matching, using a generalized hook to monitor key-name parameters for descriptive keywords. For the `boolForKey(_:)` method, the logic bifurcates based on intent: for status-related keys containing terms like *"plus"*, *"premium"*, or *"vip"*, the hook overrides the original result to return *true*, while for restriction-related keys suggesting expiration or cancellation, such as *"expire"* or *"subscriptionCancelled"*, it forces a *false* result to ensure continued access.

A similar approach is applied to the `doubleForKey(_:)` method, which apps often use to store Unix timestamps for subscription expiration. When the hook detects keys associated with time limits (e.g., "*vipDate*" or "*expireTimestamp*"), it bypasses the stored value and returns an arbitrarily large constant. This ensures that the "expiration date" always exists far in the future, effectively perpetuating premium access indefinitely.

By continuously monitoring and updating these targeted keywords during app execution, this automated approach remains highly effective across a wide range of applications without requiring prior knowledge of their specific internal architectures.

5.3.2. Generalizing Missing Receipt Validation exploits.

As discussed in Section 3.2, receipt validation is a critical security step that many developers omit or defer to prioritize a seamless user experience. This omission creates a significant vulnerability that can be exploited through runtime method hooking.

Our attack targets the `SKPaymentTransaction` class, specifically intercepting the `transactionState` property. By hijacking this call, we override the transaction's actual state and force it to return the integer value `1`, which corresponds to the `SKPaymentTransactionStatePurchased` constant in Apple's documentation. Consequently, even if a purchase is canceled or fails, the application is misled into believing the transaction was successful. This bypass is effective because the app relies solely on this locally reported state rather than independently verifying the encrypted receipt with Apple's servers.

6. Results

In this section, we detail the exploitation results of the vulnerabilities identified in our study. We first report the outcomes of the App-Specific Attack Model, followed by those of the App-Agnostic Attack Model. Table 1 shows the consolidated results for all apps identified as vulnerable in at least one scenario, including both app-specific and app-agnostic analyses. The complete tables with all analyzed apps are available in our public repository at `https://github.com/izmcm/bypass-iap-ios`.

Results with the App-Specific Attack Model.
For the app-specific analysis, we selected 34 top free apps from the Health & Fitness, Productivity, and Education categories of the App Store. Each app was decompiled, systematically analyzed, and tested for the vulnerabilities described earlier. The app-specific analysis uncovered widespread weaknesses in the implementation of digital goods validation. Specifically, 21 of the 34 analyzed apps were vulnerable to at least one exploit, allowing us to unlock premium features without paying. The most common vulnerabilities were: *i*) Unprotected Code (11 apps), *ii*) Insecure Storage (6 apps) – highlighting a

Table 1. Consolidated Vulnerable Apps by Analysis Type

App-Specific Analysis			
Name	**Bundle ID**	**Vulnerable**	**Version**
MyFitnessPal: Calorie Counter	com.myfitnesspal.mfp	✓	22.5.5
Flo Period & Ovulation Tracker	org.iggymedia.periodtracker	✓	8.7.1
Calm: Sleep & Meditation	com.calm.calmapp	✓	5.35
Yuka - Food & Cosmetic scanner	yuca.scanner	✓	4.17
Fitness Coach: Home Workout	fitnessworkout.app	✓	2.9.7
Fitness Coach & Diet: FitCoach	com.fitcoach	✓	5.0.2
I am - Daily Affirmations	com.MonkeyTaps.IAM	✓	4.17
AllTrails: Hike, Bike & Run	com.alltrails.AllTrails	✓	15.4.0
Motivation - Daily quotes	com.MonkeyTaps.Motivation	✓	4.21.1
Nutrition Coach: Food tracker	nutrition.coach	✓	1.5.1
VPN - Super Unlimited Proxy	mobi.mobilejump.freevpn	✓	1.8.1
VPN Tunnel-solo VPN for iPhone	com.vpntunnel.solo.vpn	✓	1.2.0
ViVi Keyboard: Theme Maker	com.metaversetech.vikeyboard	✓	1.5
Ringtones Maker - the ring app	com.csj.ling	✓	1.9.01
Fing - Network Scanner	overlook.fing	✓	6.161.0
PictureThis - Plant Identifier	cn.danatech.xingseus	✓	3.22
Photomath	com.microblink.PhotoMath	✓	8.4.0
Impulse - Brain Training	gen.tech.impulse	✓	1.26.6
PlantIn: Plant Identifier	xyz.plantin.app	✓	3.8.2
Babbel - Language Learning	com.babbel.babbelMulti	✓	21.2.0
Headway: Fun & Easy Growth	com.headway.books	✓	177.4

App-Agnostic Analysis					
Name	**Bundle ID**	**Vulnerable (2022/2023)**	**Version (2022/2023)**	**Vulnerable (2025)**	**Version (2025)**
Top Sticker Maker - WaSticker	com.aplicativoslegais.Memes-For-WhatsApp	✓	4.29	✓	4.47
InShot - Video Editor	com.camerasideas.InstaShot	✓	1.70.2	✓	1.88.0
Moovit: Bus & Transit Tracker	com.tranzmate.tranzmate1	✓	5.122.0	✓	5.171.0
Secret Santa 22: Gift exchange	com.bocadil.amigoinvisible	-	3.1.2	✓	4.4.2
BoomArt: FaceChanger&Cartoon	com.photo.boomart	✓	1.3.3	✓	1.4.1
InStories Reels & Story Maker	ylee.studio.instories	✓	5.35.0	✓	5.83.0
Object Removal AI Retouch Fix	com.skysoft.removalfree	✓	4.0.0	✓	4.9.1
Universal remote tv smart	com.teix.allinOne	✓	2.6.3	-	3.3
ToonMe	com.informelab.toonme	✓	0.10.47	✓	0.11.22
Period Tracker Period Calendar	com.abishkking.periodcalendar	✓	2.54.0	✓	2.93.0
Blemish Editor - Face Tuner	com.shestakova.pimpleremover	✓	1.0.1	✓	1.0.1
Magazine Maker - Photo Editor	com.haavard.mk.magazinemaker	✓	1.3	✓	1.3
VPN Lumos: Open & Connect	bernikovich.vpn	✓	2.2	-	2.20
Impulse - Brain Training Games	gen.tech.impulse	✓	1.29.13	-	1.33.5
Eris Protector	com.wagroupsia.erisprotector	✓	1.0.0	No longer available [2]	
Egis Pro	com.sunforestsia.egispro	✓	1.0.0	No longer available [2]	
LED Banner Pro: Marquee maker	com.clidoapps.ledbannerpro	-	2.13.1	✓	3.11
AI Chatbot: AI Chat Smith 4	co.vulcanlabs.moodtracker	✓	5.8.1	✓	8.1.0
BabyGenerator Guess baby face	com.dbai.BabyGenerator	✓	1.9	✓	1.14
Photo Enhancer - EnhanceFox AI	com.risingcabbage.enhancefox	✓	6.6	✓	7.5.1
Falou - Fast language learning	com.moymer.falou	✓	0.0.282	✓	0.0.369
one sec — screen time + focus	wtf.riedel.one-sec	✓	3.5.1	-	4.1.18
Make A Baby AI Future Face	com.ninja.baby3	✓	6.1.3	✓	6.1.15
Tempi Art-Cartoon&AnimatePhoto	com.face.video.facev	✓	2.4.1	✓	2.4.8
PackPoint Travel Packing List	net.drinkpoint.Packpoint	✓	1.38	✓	1.40
Citymapper: All Live Transit	azdev.citymapper	✓	11.3	✓	11.32.1
SONNO FLOWS: Sleep & Relax	com.movenext.sonno	✓	1.1.2	✓	1.1.2
Afterlight: Film Photo Editor	us.afterlight.Afterlight	✓	4.5.4	✓	4.9.19
Bazaart AI Photo Editor Design	co.bazaart.app	✓	9.9.3	✓	10.8.0
Polarr: Photo Filters & Editor	co.polarr.ppe	✓	6.8.4	✓	6.11.6
Paper: Sketch, Draw & Create	com.fiftythree.paper	✓	5.4.2	-	5.5.9
Insight Timer-Medidate & Sleep	com.spotlightsix.zentimer	-	17.4.6	✓	19.30.0
Sweat: Fitness App For Women	com.PixelForceSystems.SweatWithKayla	-	6.46.1	✓	7.6.2
Explain Everything Whiteboard	com.explaineverything.explaineverything	-	7.5	✓	8.2
Plantyx-Flower Tree Identifier	com.angelscope.ios.plantyx	✓	1.35.0	✓	1.75.0

[2] No longer available on the country App Store utilized to download the app.

need to revise Apple's developer guidance, and *iii*) Missing Receipt Validation (6 apps).

Results with the App-Agnostic Attack Model

Our app-agnostic analysis of 100 trending App Store apps – from Top Free, "Apps We Love," and "Editor's Choice" – between September 2022 and July 2023 exposed widespread in-app purchase vulnerabilities across both popular and lesser-known titles. During testing, we used the Console app to analyze app logs, including keys accessed by the app in UserDefaults, which helped us improve our black-box tool. In multiple cases, these refinements allowed us to bypass protections in previously resistant apps. We assessed 100 iOS apps using our tool and found 30 vulnerable to unauthorized access to premium features. We notified the developers with detailed reports. Two years later, in 2025, we reassessed the same apps using a Security Research Device running iOS 18: five previously secure apps became vulnerable; only five fixed the issues; seven were removed from the App Store; and 23 remained at risk. We also notified developers of any newly vulnerable apps. The App-Agnostic Attack Model provides broader coverage but is less efficient than the App-Specific model, mainly because black-box testing makes it difficult

to detect Unprotected Code vulnerabilities and identify precise issues within each app.

7. Ethical Considerations

We conducted this research following established ethical principles in information security:

- **Use of real-world apps:** Testing actual iOS apps was essential to capture the complexity of real production environments, which simulated apps cannot replicate.
- **Responsible disclosure:** We identified 56 vulnerable apps: 21 from the app-specific analysis, 30 from the app-agnostic analysis, and 5 more from a follow-up analysis two years later. All developers were contacted. This process revealed a significant gap in security maturity: only 4% had structured reporting channels. Of the developers contacted, the majority did not respond; only five acknowledged the vulnerabilities and committed to fixes. This highlights a widespread lack of dedicated resources for vulnerability management.
- **Premium feature access:** Features were unlocked strictly for research purposes. We neither benefited from nor exploited these features, and the apps were uninstalled from test devices immediately after verification.
- **Focus on client-side attacks:** Our exploits targeted flaws in client-side logic and did not cause server-side damage or affect any systems beyond the local installation on our test device.

Throughout this study, we adhered to these practices to contribute to a more secure iOS ecosystem while ensuring no harm to developers or users.

8. Conclusion

This paper presents the first longitudinal analysis of the evolution of in-app purchase implementation vulnerabilities in iOS. Our analysis of 34 specific apps revealed that 21 were vulnerable to at least one of three identified flaws. By extending our study to a broader, app-agnostic approach, our automated tool successfully exploited 30 of 100 popular apps on the App Store.

Two years later, using a Security Research Device running iOS 18, we found that 23 apps remained affected and that 5 new apps had been added to the list. This confirms that these vulnerabilities persist across modern operating system versions and do not require a jailbroken environment. Our findings demonstrate that the issue stems from insecure client-side logic rather than in platform-specific limitations. Our findings underscore that security is a shared responsibility. While developers must adopt more rigorous coding practices, industry leaders such as Apple should revise documentation and APIs to provide clearer, more secure implementation paths. Addressing these systemic implementation flaws is essential to safeguarding the financial integrity of the iOS app ecosystem.

Future work includes the following tasks: 1) *Expanded Analysis Across Categories*: Identify which sectors are more or less susceptible to specific flaws; 2) *Automated Detection Enhancement*: Although our app-agnostic tool is effective, there is room to improve its efficiency to match the depth of app-specific analysis; and 3) *Comparative API Security Analysis*: Isolate API implementations to assess whether the security improvements introduced by Apple in iOS 15 have resulted in a measurable reduction in real-world vulnerability exposure.

References

AloneMonkey (2025). frida-ios-dump. https://github.com/AloneMonkey/frida-ios-dump Accessed in Aug, 2025.

Apple (2024). About alternative app marketplaces in the european union. https://support.apple.com/en-us/118110 Accessed in Aug, 2025.

Apple (2025a). Agreements and guidelines for Apple developers. https://developer.apple.com/support/terms/ Accessed in Aug, 2025.

Apple (2025b). App Store stopped over $9 billion in potentially fraudulent transactions in the last five years. https://www.apple.com/newsroom/2025/05/the-app-store-prevented-more-than-9-billion-usd-in-fraudulent-transactions/ Accessed in Aug, 2025.

Apple (2025c). Apple Security Research Device Program. https://security.apple.com/research-device/ Accessed in Aug, 2025.

Apple (2025d). Choosing a StoreKit API for in-app purchases. https://developer.apple.com/documentation/storekit/in-app_purchase/choosing_a_storekit_api_for_in-app_purchases Accessed in Aug, 2025.

Apple (2025e). Persisting a purchase. https://developer.apple.com/documentation/storekit/persisting-a-purchase Accessed in Aug, 2025.

Apple (2025f). Unlocking purchased content. https://developer.apple.com/documentation/storekit/in-app_purchase/original_api_for_in-app_purchase/unlocking_purchased_content Accessed in Aug, 2025.

Business of Apps (2025). App Revenue Data (2024). https://www.businessofapps.com/data/app-revenues/, Accessed in Aug, 2025.

Cryptic Apps (2025). Hopper Disassembler. https://www.hopperapp.com/ Accessed in Aug, 2025.

Egele, M. et al. (2011). PiOS: Detecting privacy leaks in iOS applications. In *Proc. NDSS*.

Egele, M. et al. (2013). An empirical study of cryptographic misuse in Android applications. In *Proc. CCS '13*, pages 73–84.

Foresman, C. (2012). Recent iOS, Mac app crashes linked to botched FairPlay DRM. https://arstechnica.com/gadgets/2012/07/recent-ios-mac-app-crashes-linked-to-botched-fairplay-drm/ Accessed in Aug, 2025.

Frida (2025). Frida. https://frida.re/docs/ios/ Accessed in Aug, 2025.

García, L. and Rodríguez., R. J. (2016). A peek under the hood of iOS malware. In *2016 11th International Conference on Availability, Reliability and Security (ARES)*, pages 590–598.

iOSGods (2025). Decrypted iOS IPA App Store. `https://armconverter.com/decryptedappstore/us` Accessed in Aug, 2025.

Kellner, A. et al. (2019). False sense of security: Jailbreak detection in banking apps. In *Proc. IEEE EuroS&P*, pages 1–14.

Mendes, M. (2025). iOS 26.5 RC sets up support for app sideloading in Brazil. `https://9to5mac.com/2026/05/04/ios-26-5-rc-sets-up-support-for-app-sideloading-in-brazil/` Accessed in May, 2026.

MITRE ATT&CK (2018). Collection. `https://attack.mitre.org/versions/v15/tactics/TA0035/`, Accessed in Aug, 2025.

Mulliner, C. et al. (2014). Virtualswindle: Automated attack against in-app billing on android. In *Proc. ASIA CCS '14*, pages 459–470.

Orikogbo, D. et al. (2016). CRiOS: Toward large-scale iOS application analysis. In *Proc. SPSM '16*, pages 33–42.

Promon (2024). App threat report: The state of iOS app security. `https://promon.co/app-threat-report-ios-app-security/` Accessed in Aug, 2025.

Reaves, B. et al. (2017). Mo(bile) money, mo(bile) problems: Branchless banking applications. *ACM Trans. Priv. Secur.*, 20(3).

Reynaud, D. et al. (2012). FreeMarket: Shopping for free in Android applications. In *19th Annual Network And Distributed System Security Symposium*.

Sensor Tower (2022). Mobile market forecast 2022-2026. `https://go.sensortower.com/rs/351-RWH-315/images/Sensor-Tower-2022-2026-Market-Forecast.pdf` Accessed in Aug, 2025.

Sensor Tower (2025). State of mobile 2025. `https://sensortower.com/state-of-mobile-2025` Accessed in Aug, 2025.

Shi, S. et al. (2021). Breaking and fixing third-party payment service for mobile apps. In *Proc. ACNS 2021*.

Theos (2025). Theos. `https://theos.dev/` Accessed in Aug, 2025.

und3fy.dev (2025). Decrypt.day. `https://decrypt.day/` Accessed in Aug, 2025.

Yang, W. et al. (2017). Show me the money! Finding flawed implementations of third-party in-app payment in Android apps. In *Proc. NDSS*.

Zhou, Y. et al. (2021). Payment-Guard: Detecting fraudulent in-app purchases in iOS system. *Neurocomputing*, 422:263–276.

ZonD80 (2012). Zond80/in-app-proxy: Apple in-app purchases store emulator for iOS $<$ 6. `https://github.com/ZonD80/in-app-proxy` Accessed in Aug, 2025.

Zuo, C. et al. (2019). Why does your data leak? uncovering the data leakage in cloud from mobile apps. In *Proc. IEEE S&P*, pages 1296–1310.

Zuo, C. and Lin, Z. (2022). Playing without paying: Detecting vulnerable payment verification in native binaries of unity mobile games. In *31th USENIX Security Symposium (USENIX Security 22)*.