Playing Without Paying: Detecting Vulnerable Payment Verification in Native Binaries of Unity Mobile Games

Chaoshun Zuo and Zhiqiang Lin, *The Ohio State University*

https://www.usenix.org/conference/usenixsecurity22/presentation/zuo

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Playing Without Paying: Detecting Vulnerable Payment Verification in Native Binaries of Unity Mobile Games

Chaoshun Zuo
The Ohio State University
zuo.118@osu.edu

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

Abstract

Modern mobile games often contain in-app purchasing (IAP) for players to purchase digital items such as virtual currency, equipment, or extra moves. In theory, IAP should have been implemented securely; but in practice, we have found that many game developers have failed to do so, particularly by misplacing the trust of payment verification, *e.g.*, by either locally verifying the payment transactions or without using any verification at all, leading to playing without paying vulnerabilities. This paper presents PAYMENTSCOPE, a static binary analysis tool to automatically identify vulnerable IAP implementations in mobile games. Through modeling of its IAP protocols with the SDK provided APIs using a payment-aware data flow analysis, PAYMENTSCOPE directly pinpoints untrusted payment verification vulnerabilities in game native binaries. We have implemented PAYMENTSCOPE on top of binary analysis framework Ghidra, and tested with 39,121 Unity (the most popular game engine) mobile games, with which PAYMENTSCOPE has identified 8,954 (22.89%) vulnerable games. Among them, 8,233 games do not verify the validity of payment transactions and 721 games simply verify the transactions locally. We have disclosed the identified vulnerabilities to developers of vulnerable games, and many of them have acknowledged our findings.

1 Introduction

Mobile platforms have become the largest segment of the computer game industry today. In 2021, mobile games have generated 79 billion (52%) USD in global revenue, surpassed the rest of the game industry combined, including boxed and downloaded PC games, tablet games, and browser PC games [18]. There are various ways to monetize a mobile game, such as premium (players have to pay upfront), freemium (players only pay when needed), subscription (players pay monthly fees), and in-app advertisement. Increasingly, freemium, powered by the in-app purchasing (IAP) service provided by the app stores, has become the dominating monitization way for mobile games [15]. Today, both Google Play and Apple AppStore have provided IAP APIs for in-game purchasing of various digital items such as virtual currency, equipment, extra moves, or removing of the advertisements for enhanced gaming experience.

However, the APIs provided by Google Pay and Apple AppStore only handle the payment services for the IAP (*e.g.*, by communicating with the credit card companies or banks, and finishing the financial transactions). The game developers still have to be responsible for the delivery of the purchased items. To alleviate the developer's efforts, many SDK providers, *e.g.*, Unity [16], the leading game engine provider [2], have provided wrapped APIs on top of Apple's or Google's. With these wrapped APIs, mobile games can also enjoy the portable benefit of across different platforms such as Android and iOS.

Obviously, any in-game purchasing must be securely designed and implemented. While app stores have secured the payment transactions, the validation of the transactions is completely left to the developers. However, an in-game purchasing transaction is a complicated multi-party transaction. At a high level, it involves at least three parties: (*i*) mobile game app, (*ii*) game server, and (*iii*) payment provider. In theory, all these parties should have verified the validity of the purchasing transaction between each other, since both mobile games and mobile operating systems could have been compromised (*e.g.*, the games could be repackaged or executed in a virtual environment with apps such as Parallel Space [26]).

Unfortunately, in practice, we have observed that many game developers have failed to validate the transactions, even though there is a large body of work (*e.g.*, [40, 44, 48]) that have pointed out the potential security risks in multi-party transactions [49]. For instance, we found multiple extremely popular games (each with more than 100 million installs) do not verify the validity of the IAP transactions or simply rely on the client side verification to check whether the transaction is valid, allowing attackers to completely playing without paying. Even worse, the SDK providers (*e.g.*, Unity) have provided such APIs for client side verification, whose execution cannot be trusted at all. Considering the increased amount of mobile games integrating with the IAP service, it is thus important to raise developers' awareness and identify the vulnerable IAP implementations at scale.

Identifying vulnerable IAPs is not a new problem. Prior works (*e.g.*, [40, 44, 48]) have analyzed such vulnerabilities in mobile apps at Java bytecode level. However, they cannot be applied to analyze the vulnerable implementation at the native binary code level. To advance the-state-of-the-art, this paper presents a mobile game in-app payment vulnerability detector named PAYMENTSCOPE, a static native binary analysis tool to automatically identify vulnerable in-game purchases. For proof-of-concept, we focus on the games developed from the leading game engine Unity [2]. The key idea is to detect the vulnerable IAP implementation by inspecting how the validation of a payment transaction in mobile games is performed, *e.g.*, without validation, with local validation, or with server side validation, through a *payment-aware data flow analysis*. Multiple domain specific challenges have to be solved, such as how to pinpoint the IAP APIs in the game native binary, how to model the data flow of the payment transaction validations in the binary code, and how to detect the vulnerable validation.

We have addressed these challenges and implemented PAYMENTSCOPE on top of Ghidra [5], a static binary analysis framework. We have tested our tool with 39,121 Unity games and identified 8,954 (22.89%) that are vulnerable. Among them, 8,233 (91.95%) do not verify the validity of payment transactions at all, and 721 (8.05%) only verify the transactions locally. With respect to the false positives (FPs), our manual analysis with 200 randomly sampled vulnerable games confirmed they are indeed vulnerable (no FPs). We have disclosed the vulnerabilities to both Unity engine provider and also vulnerable game developers. Many of them have acknowledged our findings.

Contribution. We make the following contributions:

- **Novel discovery.** We find that even though payment security has been well studied for years, the developers still made various mistakes. More surprisingly, modern SDKs are not fully aware of this issue by even providing insecure APIs.
- **Efficient techniques.** We devise an efficient *payment-aware data flow analysis* by leveraging the metadata extracted from Unity game binaries and tracking how payment transaction is verified to identify the vulnerability.
- **Empirical evaluation.** We have implemented our techniques and conducted a large scale evaluation with 39,121 most recent Unity games. Unfortunately, we found 22.89% of them contain vulnerable in-game purchasing.

2 Background

Building a mobile game from scratch is not trivial, especially for games with sophisticated user interfaces. To alleviate this effort, developers often rely on game engines, which typically

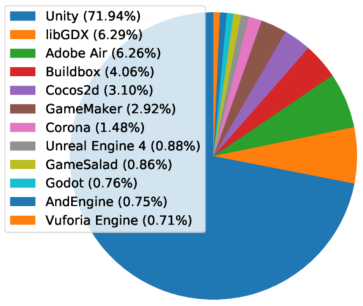


Figure 1: The Measured Distribution of Game Engines.

provide a set of tools such as SDKs, which include developer friendly IDE, graphics rendering engine, well documented APIs, and cross-platform support. With the help of those tools, developers can focus more on implementing game logic. Today, there are many game engines available. Among them, it is reported that Unity is the most popular one [43]. Meanwhile, our measured result (detailed in Appendix §A) with 293,019 Android games also confirmed this, as presented in Figure 1.

Mobile in-game purchasing. An important means for mobile app developers to monetize their games is to use IAP, a service provided by app stores. In particular, to simplify the payment setup and also ease the app development, app stores typically will ask each app user (*e.g.*, a game player) to bind his or her credit card with the platform, and then provide APIs for developers to directly handle the payment process. While the billing is completely handled by the app stores, the delivery and verification of the transaction is left to the app developers. In addition, developers could use the SDKs provided by app stores to implement IAP, but they often use the wrapped APIs provided by game engines such as Unity because the APIs provided by app stores are often in different programming languages compared with the ones in the SDKs (*e.g.*, Java in Google Play vs. C# in Unity).

While IAP protocol itself is complicated, most of the details can be abstracted and taken care by the app store (*e.g.*, Google Play) and game SDK providers (*e.g.*, Unity). At a high level, there could be up to seven steps when processing an in-game purchasing request between the three involved parties: (*i*) mobile game app, (*ii*) game server, and (*iii*) app store, as illustrated in Figure 2:

- **Step ❶:** When a game player wishes to buy a digital item (*e.g.*, an extra move), the game will first issue the payment request to the app store, which contains the item to be purchased. Assume the game is developed with the Unity SDK, the game will use API `IStoreController.InitiatePurchase(String)` to initialize this payment request where the `String` type parameter contains the ID of the product defined by the game

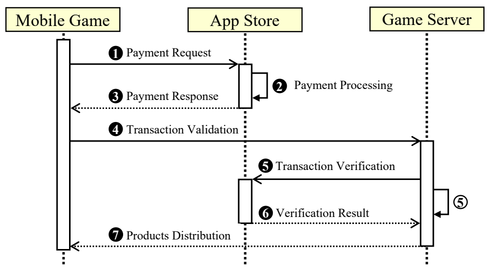


Figure 2: A Typical IAP Protocol in Mobile Games.

developers, and the app store will use this ID to look up its price.

- **Step ❷:** When receiving the purchasing request from the game app, the app store will validate the player's identity and pop up a dialog to ask the player to confirm the payment, and then the app store will communicate with either the banks or the credit card company to finish the money transfer. To ensure the integrity of this financial transaction, the app store will sign it, with the app-specific private key for Android [13] or developer-specific privatey key for iOS [11] (developers have the corresponding public key when they register their development accounts), and then return the signature to the game app.

- **Step ❸:** Once the app store finishes the payment transaction, the game will receive a payment response callback, which is either `UnityEngine. Purchasing.IStoreListener.ProcessPurchase` if the transaction succeeds, or `UnityEngine.Purchasing.IStoreListener.OnPurchaseFailed` if the transaction fails (*e.g.*, insufficient funds, or cancelled by the user) in Unity games.

- **Step ❹:** If the transaction succeeds, the game client typically will send the transaction to the game server to validate the transaction and distribute the digital items.

- **Step ❺:** To validate the integrity of the transaction, the game server can either ask the app store to verify the transaction (*e.g.*, by using REST API `purchases.subscriptions.get` [12] for Google Play), and go to Step ❻, or alternatively, the game server can verify the transaction by validating the signature of the transaction using the app-specific public key owned by the developer, and in this case the server directly goes to Step ❼.

- **Step ❻:** The app store validates the transaction and returns the transaction details (*e.g.*, transaction timestamp, payment state, item price, expiration date) to the game server.

```
Game APK
├──...
├──AndroidManifest.xml· · · · · · · ·[ 39K]
├──classes.dex· · · · · · · · · · · ·[7.8M]
├──assets
│  └──bin
│     └──Data
│        ├──...
│        └──Managed
│           ├──...
│           └──Metadata
│              └──global-metadata.dat · · ·[5.7M]
└──lib
   └──arm64-v8a
      ├──libil2cpp.so· · · · · · · · · ·[ 30M]
      ├──libmain.so· · · · · · · · · · ·[6.0K]
      └──libunity.so · · · · · · · · · ·[ 15M]
```

Figure 3: The APK Structure of Game Run Race 3D.

- **Step ❼:** Having verified the transaction by the game server, the server finally distributes the digital items if the transaction is valid.

The game binaries developed by Unity. Many mobile games today are developed using C# and with the Unity engine [3]. There are two ways to run a C# program. One is to compile the C# code to .NET Common Intermediate Language (CIL) [36], and then use a virtual machine (VM) to interpret the CIL code at runtime. The other is to translate the CIL code to C++ code via tool `IL2CPP` [14] and then compile the C++ code to native binaries. Unity has largely moved to the second approach since it is more efficient.

However, translating CIL code to C++ code is not trivial, as C# has many features that C++ does not have such as reflection and garbage collection, and `IL2CPP` needs to support those features in C++ and make the native binary work as CIL code. In particular, `IL2CPP` will translate not only the game code, but also C# system libraries and Unity libraries to C++ code. Each function in C# will be translated into a corresponding function in C++. Also, `IL2CPP` will add additional code for error checks (*e.g.*, detecting NULL pointers). While typically symbol names such function names can be stripped in native binaries, `IL2CPP` will keep them. This is because the binary still needs them to provide consistent features as C#. For instance, it needs the function name and its address when handling a reflection call, and provides a full call stack trace (including function names) when handling an exception. As such, `IL2CPP` also stores symbol information in a separate file named `global-metadata.dat` and packs it into the game, though game developers might encrypt or hide this file, to defend against the reverse engineering attempt. Ultimately, at runtime, the binary will load this file along with `libil2cpp.so` (which is generated via `IL2CPP` translation) and then use it during game execution.

In Figure 3, we present the file structure of a game APK, which is an archive file in ZIP format. The sub-file `libil2cpp.so` is the eventually compiled game binary, which contains the game logic and the compiled C# libraries code. We can see that it occupies a large space of a game APK. Another sub-file worth mentioning is `global-metadata.dat`, which contains the metadata of the game including the mapping between the addresses in the binary and the functions, the classes definitions (*e.g.*, fields), and the strings that used by the game binary. Unlike debugging symbols that have source code information, `global-metadata.dat` does not include this (that is why we must perform binary analysis). In addition, while there are multiple `so` files in the APK, we only need to focus on `libil2cpp.so` since this is what `IL2CPP` eventually translated.

Ghidra. There are many frameworks that can be used to perform static binary analysis such as Angr [46], IDA Pro [27], and Ghidra [5]. Increasingly, Ghidra becomes more and more popular due to its ease of use, and multi-architecture (*e.g.*, ARM64, ARM32, x86) support with an intermediate representation called *P-Code* [10]. More specifically, *P-Code* is a register transfer language, and to generate the *P-Code* of a binary, Ghidra will translate each assembly instruction to one or multiple *P-Code* instructions. For instance, in Figure 5, instruction in the second column will be translated to one or multiple *P-Code* in the third column. For each *P-Code*, it has an *opcode* (*e.g.*, `CALL` in the example), one or more input, and an optional output. Each input or output is called `Varnode`. There are two types of `Varnode`: register, *e.g.*, `(register, 0x4000, 8)`, and memory location, *e.g.*, `(ram, 0x02212258, 8)`. Each `Varnode` is composed of the address space (*e.g.*, `register` or `ram`), the offset within the space (*e.g.*, `0x4000`), and the size of the `Varnode` (*e.g.*, `8`).

3 Overview

3.1 The Problem, Threat Model, and Scope

The problem. To complete a payment transaction, it is crucial for mobile games to verify the transactions signed by the app store to ensure its integrity. As discussed in §2, after receiving a payment request from a game app, the payment service provider (*e.g.*, Google Pay) will complete the financial transaction transparently, and when it succeeds, the transaction's metadata will be signed and returned to the game. Then, the game service provider will distribute the purchased products after verifying the transaction from the game server since a game client cannot be trusted. *Any failure or missing such verification from the game server will lead to playing without paying vulnerability*. This paper seeks to identify this vulnerability, which usually has the following two forms:

- **Lack-of-verification.** When the game app receives the payment receipt from the payment provider, it may simply check the payment receipt (in Step ❸) to see whether the payment succeeded or not without validating the integrity of the receipt at all. In other word, the Step ❹, ❺, ❻ are skipped.
- **Local-verification.** While the game app can validate the receipt locally by using the public key of the signer (by moving Step ⑤ to mobile game and skip Step ❹, ❺, ❻), this validation is fundamentally flawed. Specifically, there are multiple ways for attackers to subvert such local validation, *e.g.*, by replaying an old transaction, or directly tampering with the return value of the local validation, or removing the validation via binary patching.

Threat model. We assume a threat model in which the game client is untrusted, and the attacker is a game player who owns the client and has the incentive of playing without paying. There are multiple ways to launch the attack (detailed in §6.3), from the simplest, by using an app-level virtualization tool [26] such as Parallel Space or using repackaged victim game from 3rd parities, to the most sophisticated, by rooting the phone to tamper with anything of attacker's interests.

Our threat model is realistic for at least three reasons. First, it is consistent with many other payment security works (*e.g.*, [39, 44, 49]). In particular, when studying payment security in traditional mobile apps, Reynaud et al. [44] assume the attackers can repackage the apps; when inspecting the validation consistency between the client and the server of banking apps, WARDROID [39] assumes the client app is not trusted. Similarly, when analyzing payment security in the web domain, Wang et al. [49] assume the web clients (*i.e.*, the web browsers) can be malicious and they can cheat the web servers for shopping for free. Second, this threat model is also consistent with the reality. While a game player does not have the skills to root the phone, he or she can use the repackaged games. For instance, we have found several websites (*e.g.*, [1, 8]) host repackaged games. In addition, a game player can also use app-level virtualization [26] without any root privilege to full control the game. Third, the assumed threat is indeed threatening game developers, and many security companies actually offer commercial solutions (such as `NHN AppGuard` [9]) to defend against this.

Scope and assumptions. While there are a variety of mobile games available in the mobile platform, we focus exclusively on the native Android games developed by Unity SDK for multiple reasons. First, Unity is the most popular game engine with a 71.94% market share according to our measured result reported in Figure 1. Second, Unity is not a special case, and many other game engines (*e.g.*, Unreal Engine, Cocos2d) have the same issue—namely it allows developers to verify payment transactions locally. Third, each game engine has its own specific APIs and runtime environment. Uncovering the flawed payment verification for all of the games would require game-engine specific analysis. While we could have also focused on analyzing Cocos2d or Unreal Engine games, this would require significant amount of additional engineering effort.

Therefore, we eventually decided to focus on Unity games for proof-of-concept. Also, when analyzing game binaries developed with Unity, we assume the game developers will use

```
 1 class IAPManager : IStoreListener
 2 {
 3   public PurchaseProcessingResult ProcessPurchase(
 4                                PurchaseEventArgs args)
 5   {
 6     CrossPlatformValidator validator = new CrossPlatformValidator(
 7                    GooglePlayTangle.Data(), AppleTangle.Data(),
 8                    Application.identifier);
 9     try
10     {
11       validator.Validate(args.purchasedProduct.get_receipt());
12       ...
13     }
14     catch (IAPSecurityException)
15     {
16       Debug.Log("Invalid receipt, not unlocking content");
17     }
18     return PurchaseProcessingResult.Complete;
19   }
20 }
```

(A)

```
 1 class PurchaseManager : IStoreListener
 2 {
 3   public PurchaseProcessingResult ProcessPurchase(
 4                                PurchaseEventArgs args)
 5   {
 6     ...
 7     StoreReceipt receipt = JsonUtility.FromJson<StoreReceipt>(
 8                            args.purchasedProduct.get_receipt());
 9     GooglePayload gpayload = JsonUtility.FromJson<GooglePayload>(
10                            receipt.Payload);
11     httpRequest.AddField("signature", gpayload.signature)
12     ...
13   }
14   ...
15 }
```

(B)

```
 1 class unityInAppPurchase_LS : IStoreListener
 2 {
 3   ...
 4   private string m_LastReceipt; // 0x30
 5   public PurchaseProcessingResult ProcessPurchase(
 6                                PurchaseEventArgs args)
 7   {
 8     this.m_LastReceipt = args.purchasedProduct.get_receipt()
 9     ...
10   }
11 }
```

(C)

Figure 4: Running Examples of In-game Purchase with Decompiled or Manually Constructed Code (in C#).

the APIs to develop the game for the common logic such as logging and network communication. In addition, we assume there is no encryption or hiding of `global-metadata.dat` (which is true for 99.69% of the game we have analyzed).

3.2 Running Examples

To clearly illustrate the problem we aim to solve, we use three real-world games (the corresponding game names are removed since the vulnerable games have not been patched at this time of writing) shown in Figure 4 as running examples. Note that, the C# code in the figure is manually constructed from native binaries for clear illustration, and the excerpt of the corresponding binaries can be found in Figure 5.

At a high level, we can see that game (A) verifies the payment receipt from payment providers locally with API `validator.Validate` (line 11). Note that this API uses the public key of the payment provider (acquired by `GooglePlayTangle.Data()` at line 7) to verify the signature of the signed receipt by the app store. If the verification succeeds, the app will then complete the purchasing process (line 12); otherwise, it will generate a debug log tracking the invalid receipt (line 16). Since the entire purchasing verification can be executed in an untrusted game client, it is insecure (and can be bypassed). Game (B) extracts the receipt (line 7 and 8), adds it into an `httpRequest` object (line 11), sends the receipt to the server for the verification, which is considered to be a correct approach (though we cannot inspect its server side implementation). Game (C) stores the receipt to field `m_LastReceipt` (line 8), which tracks the last receipt,

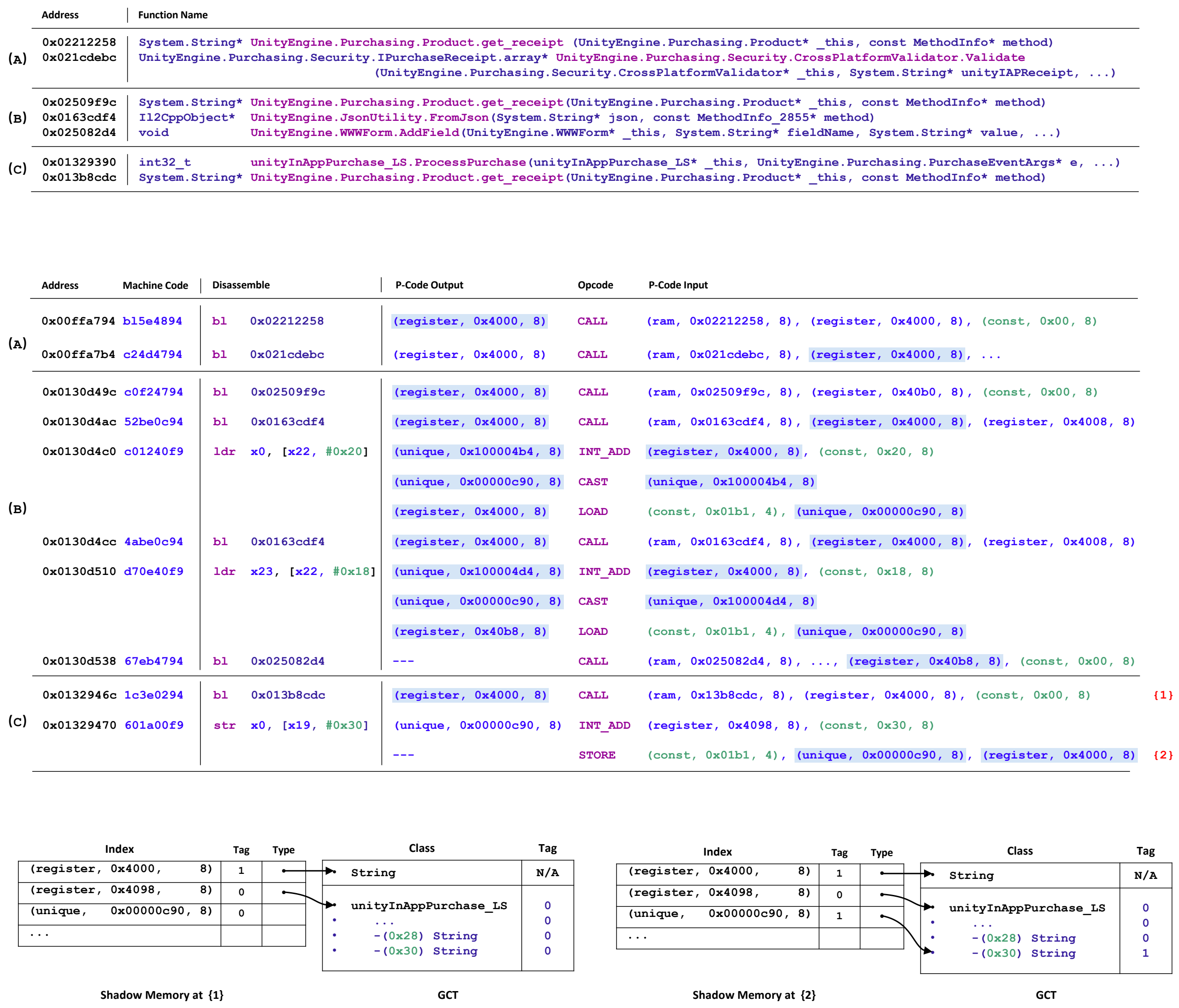
3.3 Challenges and Insights

In this study, we focus on two vulnerable in-game purchases: lack-of-verification (*i.e.*, no-verification) and local-verification. Since in both cases it refers to how data is used, obviously we need to leverage the meanings from the APIs to infer the semantics of the data use of the receipt as well as its propagation. For instance, as shown in Figure 4, we can easily detect the receipt is validated locally if the app uses `Validator.Validate`, it is logged if `Debug.Log` is invoked, it is sent to the server if it is passed to API `UnityWebRequest.Post`, and so on. Therefore, we have to solve at least two challenges: (*i*) pinpointing specific APIs in the game binary, and (*ii*) tracking the data flow [41, 45] of the payment-data.

(I) How to pinpoint target APIs in game binaries. APIs are crucial to infer the meaning of the data use. However, unlike with source code where we can see the names of variables and APIs and easily infer the semantics of data-use, for binary analysis, the input is often just hexadecimal code and data, and there are no symbol names as they can be stripped. Therefore, we have to reconstruct the high level abstraction from the binary code and recognize the specific APIs of our interest.

Interestingly, due to the nature of how Unity binaries are compiled and executed, it has surprisingly made the symbol recovery easier. Specifically, the association between the method name including APIs and the entry address has actually been recorded into the metadata file (*i.e.*, `global-metadata.dat`), which keeps all the missing symbols including even class names that are optimized by the compiler of Unity SDK. Although the metadata file is not directly readable, we can parse it with tool `Il2CppDumper` [6] to extract the information we need. The details of how we extract the function and classes metadata is presented in §4.

(II) How to identify the payment-data definition, use, and their propagation. Apparently, we need a *payment-aware data flow analysis* to identify the data-definition, propagation, and data-use of the payment receipt returned by the app store. While the use of *P-Code* has made the development of our analysis easier, we still have to locate the instructions that define the payment data, the propagations of the data (including through JSON object parsing and creation as shown in



	Address	Function Name
(A)	0x02212258	System.String* UnityEngine.Purchasing.Product.get_receipt (UnityEngine.Purchasing.Product* _this, const MethodInfo* method)
	0x021cdebc	UnityEngine.Purchasing.Security.IPurchaseReceipt.array* UnityEngine.Purchasing.Security.CrossPlatformValidator.Validate (UnityEngine.Purchasing.Security.CrossPlatformValidator* _this, System.String* unityIAPReceipt, ...)
(B)	0x02509f9c	System.String* UnityEngine.Purchasing.Product.get_receipt(UnityEngine.Purchasing.Product* _this, const MethodInfo* method)
	0x0163cdf4	Il2CppObject* UnityEngine.JsonUtility.FromJson(System.String* json, const MethodInfo_2855* method)
	0x025082d4	void UnityEngine.WWWForm.AddField(UnityEngine.WWWForm* _this, System.String* fieldName, System.String* value, ...)
(C)	0x01329390	int32_t unityInAppPurchase_LS.ProcessPurchase(unityInAppPurchase_LS* _this, UnityEngine.Purchasing.PurchaseEventArgs* e, ...)
	0x013b8cdc	System.String* UnityEngine.Purchasing.Product.get_receipt(UnityEngine.Purchasing.Product* _this, const MethodInfo* method)

	Address	Machine Code	Disassemble	P-Code Output	Opcode	P-Code Input	
(A)	0x00ffa794	b15e4894	bl 0x02212258	(register, 0x4000, 8)	CALL	(ram, 0x02212258, 8), (register, 0x4000, 8), (const, 0x00, 8)	
	0x00ffa7b4	c24d4794	bl 0x021cdebc	(register, 0x4000, 8)	CALL	(ram, 0x021cdebc, 8), (register, 0x4000, 8), ...	
(B)	0x0130d49c	c0f24794	bl 0x02509f9c	(register, 0x4000, 8)	CALL	(ram, 0x02509f9c, 8), (register, 0x40b0, 8), (const, 0x00, 8)	
	0x0130d4ac	52be0c94	bl 0x0163cdf4	(register, 0x4000, 8)	CALL	(ram, 0x0163cdf4, 8), (register, 0x4000, 8), (register, 0x4008, 8)	
	0x0130d4c0	c01240f9	ldr x0, [x22, #0x20]	(unique, 0x100004b4, 8)	INT_ADD	(register, 0x4000, 8), (const, 0x20, 8)	
				(unique, 0x00000c90, 8)	CAST	(unique, 0x100004b4, 8)	
				(register, 0x4000, 8)	LOAD	(const, 0x01b1, 4), (unique, 0x00000c90, 8)	
	0x0130d4cc	4abe0c94	bl 0x0163cdf4	(register, 0x4000, 8)	CALL	(ram, 0x0163cdf4, 8), (register, 0x4000, 8), (register, 0x4008, 8)	
	0x0130d510	d70e40f9	ldr x23, [x22, #0x18]	(unique, 0x100004d4, 8)	INT_ADD	(register, 0x4000, 8), (const, 0x18, 8)	
				(unique, 0x00000c90, 8)	CAST	(unique, 0x100004d4, 8)	
				(register, 0x40b8, 8)	LOAD	(const, 0x01b1, 4), (unique, 0x00000c90, 8)	
	0x0130d538	67eb4794	bl 0x025082d4	---	CALL	(ram, 0x025082d4, 8), ..., (register, 0x40b8, 8), (const, 0x00, 8)	
(C)	0x0132946c	1c3e0294	bl 0x013b8cdc	(register, 0x4000, 8)	CALL	(ram, 0x13b8cdc, 8), (register, 0x4000, 8), (const, 0x00, 8)	{1}
	0x01329470	601a00f9	str x0, [x19, #0x30]	(unique, 0x00000c90, 8)	INT_ADD	(register, 0x4098, 8), (const, 0x30, 8)	
				---	STORE	(const, 0x01b1, 4), (unique, 0x00000c90, 8), (register, 0x4000, 8)	{2}

Shadow Memory at {1}

Index	Tag	Type
(register, 0x4000, 8)	1	
(register, 0x4098, 8)	0	
(unique, 0x00000c90, 8)	0	
...		

GCT

Class	Tag
String	N/A
unityInAppPurchase_LS	0
• ...	0
• -(0x28) String	0
• -(0x30) String	0

Shadow Memory at {2}

Index	Tag	Type
(register, 0x4000, 8)	1	
(register, 0x4098, 8)	0	
(unique, 0x00000c90, 8)	1	
...		

GCT

Class	Tag
String	N/A
unityInAppPurchase_LS	0
• ...	0
• -(0x28) String	0
• -(0x30) String	1

Figure 5: The Excerpt of the Disassembled Code, the Corresponding P-Code, and the State of Our Shadow Memory.

Figure 4 (B)), identify the corresponding class fields that store them (*e.g.*, in the last instruction of Figure 5 (C), the payment data has been stored to a memory address, *i.e.*, `(unique, 0x00000c90, 8)`, and we need to know the definition of this memory address, which is essentially the class field `this.m_LastReceipt`), and infer the use of them based on APIs.

To identify the payment data definition, we can rely on the key APIs such as `args.purchasedProduct.get_receipt`. While directly identifying the payment data propagation inside JSON parsing functions is challenging, we can skip the detailed analysis inside these functions and instead use the API summary (an approach that has been widely used in many other applications such as symbolic execution [46]). To infer whether a class field (typically organized as an abstract base address plus an offset) stores the payment data or its propagations, we can design a taint-tracking algorithm for class field based on the taintedness of return values or arguments of well-defined APIs. Finally, to infer whether a field belongs to a particular class, we can use backward slicing [52] to identify the base address of the class, and then identify the class types using the argument types of functions extracted from the metadata. The details of how we perform our *payment-aware data flow analysis* is presented in §5.

Also, note that due to the nature of how game binaries are generated with Unity SDK, we have all the symbols of binary functions and classes, and we do not face the hard problems as in traditional binary analysis such as the aliasing. In particular, since the game binaries are translated from C# to C++ and there is no global variable in C#. The only data flow is through the access of fields of classes. For example, in Figure 4 (C), the field `m_LastReceipt` of class `unityInAppPurchase_LS` has been assigned with the payment receipt at line 8, and the data flow should continue from any other instructions that read this field. Since we have the symbols and their types, we can pinpoint exactly the class name of each variable in all the functions. As such, we can simply locate variables that are of type `unityInAppPurchase_LS` and continue analyzing from there to find out which instruction accessed field `m_LastReceipt` through a lightweight type-based data flow analysis.

4 Metadata Extraction

Since a game built with Unity contains rich information in file `global-metadata.dat` about its function signatures (types of arguments and return value) and class definitions (including fields offset and their types) of the final executables due to the use of `IL2CPP`, we can leverage this information to facilitate our *payment-aware data flow analysis*. In the following, we describe how we extract such information.

Extracting function metadata. The metadata of a function includes the starting address of the function, the type of the return value, the number of arguments, and the corresponding type for each argument. All of this information can be retrieved from `global-metadata.dat`. There is also a function name associated, and this name reveals both the original C# Class and function, from which this function was translated. As mentioned earlier, while `global-metadata.dat` cannot be directly read, we can use tool `IL2CPPDumper` [6] parse it. Since the function address is static, we store the extracted functions and their types in a global function table GFT, indexed by the address.

With the extracted function argument type, it makes our interprocedure analysis much easier. More specifically, assume we have identified a class type of our interest, then we can directly scan our GFT to locate all of the functions that have an argument referencing this class, and these functions are all possible callees. Meanwhile, we also scan all functions to identify each possible caller that has invoked these callees. The instructions of both identified callers and callees will be iterated again to identify whether there is any data-use of the payment data (and its propagations), as well as propagation of the payment data to any other class objects if there are any.

Some examples of the extracted functions and their types can be found in the top half of Figure 5. For instance, in running example (B), the function at address `0x02509f9c` is `UnityEngine.Purchasing.Product.get_receipt`. From this function name, we can easily identify that this function was translated from the C# function `get_receipt` of C# Class `UnityEngine.Purchasing.Product` and the type of its return value is `System.String`.

Extracting class metadata. Unlike function's metadata where we can extract the function's address, class' metadata is just the abstract definition of the class, and there is no concrete address associated. What we can extract only includes the class name, the field offset, and the field types. We use the class name to index them, and store the extracted class types in a global class table GCT, which will be updated to record the taintedness whenever a class field stores tainted data.

5 Payment-Aware Data Flow Analysis

In order to identify the vulnerable payment verification, we need to first identify where the payment data is defined (§5.1), how it propagates (§5.2), and whether the propagated data is sent to the server (for server-side verification), or used by local-verification APIs, or no-verification at all (§5.3). In this section, we present the details of our lightweight payment-aware data flow analysis.

5.1 Identifying Payment-Data Definition

Since we need to track the data flow of the payment receipt, our taint sources should be the APIs that receive this data. After systematically examining the documentation, we found that `UnityEngine.Purchasing.Product.get_receipt` is the only API that can be used to access the receipt. As such, we focus on this API exclusively as the taint source. More specifically, we first find the address of this function by checking the extracted metadata in the global function table. For example, in Figure 5 (A), by looking up the corresponding GFT, we find its address is `0x02212258`. Then we locate the callers of this function by traversing the call graph of the game binary. There might be several callers and each of them will be located. At each call site, the variable assigned with the return value (i.e., receipt) of this function call is the source of the payment-data.

5.2 Tracking Payment-Data Propagation

A receipt can be propagated to other objects, *e.g.*, via JSON APIs; can be assigned to local variables, *e.g.*, to variable `receipt` and `gpayload` in running example (B) of function `ProcessPurchase`; or can be assigned to a class field such as `m_LastReceipt` in class `unityInAppPurchase_LS` in running example (C). We therefore have to systematically identify these propagations. The algorithm of how we perform this analysis is presented in Algorithm 1.

At a high level, starting from the caller of payment-data definition function (line 3-5 of Algorithm 1), we iterate each instruction to handle propagations (line 9-24), resolve object base address, and infer field taintedness (line 25-39). We repeat this process by iterating all of the functions in GFT (line 7-8, and line 40-55), until reaching a fixed point where the taintedness of the classes and their fields will not be updated any more (line 6). For example, when analyzing the instruction at `0x00ffa794` in Figure 5 (A), we know it is a call instruction, and by retrieving its callee with address `0x02212258`, we know it is `UnityEngine.Purchasing.Product.get_receipt`. Then our analysis will get started at line 3. Based on the API summary, we know its return value is the payment receipt data, and we therefore perform the data propagation by adding a taint tag (line 4), basically just a single bit, to the return value, *i.e.*, `(register, 0x4000, 8)`, of this function in the corresponding shadow memory (line 5), and then perform the analysis further. In the following, we describe the key procedures of our analysis.

Algorithm 1 Payment-aware Data Flow Analysis (PDFA)

```
 1: Input: $GCT$: global class table; $GFT$: global function table; $SM$: shadow memory
 2: procedure PDFA
 3:     for ($f$, $inst$) in GetCaller(Purchasing.Product.get_receipt) do
 4:         UPDATESHADOWMEMORY($SM$, TargetOperand($inst$))
 5:         TAINTPROPAGATION($f, inst$, TargetOperand($inst$))
 6:     while HASUPDATES($GCT$) do
 7:         for func in $GFT$ do
 8:             PROCESSFUNCTIONWITHTAINTEDCLASSES(func)
 9: procedure TAINTPROPAGATION($func, inst, v_0$)
10:     for ($inst_1$) ← FindDataUse($func, inst, v_0$) do
11:         if opcode($inst_1$) is Data Move then
12:             $v_1$ ← TargetOperand($inst_1$)
13:             UPDATESHADOWMEMORY($SM$, $v_1$)
14:             ($v_2$, $offset_2$) ← FIELDBASEADDRESSRESOLUTION($func$, $v_1$)
15:             $cls_1$ ← BASECLASSRESOLUTION($func$, $v_2$)
16:             TAINTGCT($GCT$, $cls_1$, $offset_2$)
17:             TAINTPROPAGATION($func, inst_1, v_1$)
18:         else if opcode($inst_1$) is CALL then
19:             $f_0$ ← GetCallee($inst_1$); $index$ ← GetParameterIndex($inst_1$, $v_0$)
20:             if isSystemAPI($f_0$) then
21:                 TAINTWITHAPISUMMARY($f_0, inst_1$, TargetOperand($inst_1$))
22:             else
23:                 for ($inst_2, v_2$) ← GetInstructUseParameter($f_0$, $index$) do
24:                     TAINTPROPAGATION($f_0, inst_2, v_2$)
25: procedure BASECLASSRESOLUTION($func, v_0$)
26:     $vdef$ ← FindDataDef($func, v_0$)
27:     if isParameter($vdef$) then
28:         $index$ ← GetParameterIndex($vdef$)
29:         return GETFUNCPARAMETERTYPE($GFT$, $func$, $index$)
30:     else if opcode($vdef$) is CALL then
31:         $f_0$ ← GetCallee($vdef$)
32:         return GETFUNCRETURNTYPE($GFT$, $f_0$)
33:     else if opcode($vdef$) is LOAD then
34:         $v_x$ ← SourceOperand($vdef$)
35:         ($v_1$, $offset_1$) ← FIELDBASEADDRESSRESOLUTION($func$, $v_x$)
36:         $cls_1$ ← BASECLASSRESOLUTION($func$, $v_1$)
37:         return GETCLASSFIELDTYPE($GCT$, $cls_1$, $offset_1$)
38:     else if opcode($vdef$) is COPY or CAST then
39:         return BASECLASSRESOLUTION($func$, SourceOperand($vdef$))
40: procedure PROCESSFUNCTIONWITHTAINTEDCLASSES($func$)
41:     for ($cls_0$, $offset$) in GETTAINTEDFIELDS($GCT$) do
42:         $indexes$ ← FindObjectIndexinFunc($func$, $cls_0$)
43:         if $indexes! = \emptyset$ then
44:             for $index_i$ in $indexes$ do
45:                 for ($f$, $inst$) in GetCaller($func$) do
46:                     $v_i$ ← GetOperand($inst$, $index_i$)
47:                     for $inst_i$ in FINDRELATEDINSTRUCTOF($f$, $v_i$) do
48:                         if ISLOADING($inst_i$, $v_i$, $offset$) then
49:                             $v_t$ ← TargetOperand($inst_i$)
50:                             TAINTPROPAGATION($fCaller, inst_i, v_t$)
51:                 for $v_i$ ← GetParameter($func$, $index_i$) do
52:                     for $inst_i$ in FINDRELATEDINSTRUCTOF($func$, $v_i$) do
53:                         if ISLOADING($inst_i$, $v_i$, $offset$) then
54:                             $v_t$ ← TargetOperand($inst_i$)
55:                             TAINTPROPAGATION($func, inst_i, v_t$)
```

Handling taint propagation. Our taint needs to be propagated for data movement instructions (*e.g.*, `LOAD`, `STORE`, `INT_ADD`), and we update the shadow memory of the register and object memory accordingly based on the taintedness of data source (line 13). However, when encountering a function call (line 18-24), we will check whether this function is a system or Unity provided API (line 20). If so, we will skip the analysis of this function, and instead directly apply the propagation rule based on the API summary (line 21). Otherwise, we will process this callee function as usual to analyze each instruction that access the tainted parameter for taint propagation (line 23-24). For instance, for the instruction at `0x0130d4ac` in Figure 5 (B), by checking the corresponding *GFT*, we find this function is `UnityEngine.JsonUtility.FromJson`. Based on the API summary, the propagation rule is to add a taint tag to the return value if the first parameter is tainted. Therefore, we add a taint tag to `(register, 0x4000, 8)`.

Tracking taintedness for classes. When encountering an instruction that propagates taint to a class field, we also need to know the specific class to which the destination field belongs. However, at the instruction level, we only observe memory addresses (due to the nature of `IL2CPP`), which is always in the form of a base address plus an offset, and we have to therefore first identify its base address, then identify the class of the base address. With the identified class information and the offset we can identify the field in *GCT*, then we taint the corresponding field and track the taintedness of the classes. As such, it eventually becomes a three-step process:

- **Step-I: Identifying the base address of a field.** When a tainted value is stored to a memory address, *e.g.*, `(unique, 0x00000c90, 8)` in Figure 5 (C), we need to identify how this address is generated (or defined). To this end, we perform backward slicing [52] starting from the current instruction to find out its definition. In this example, we find `(unique, 0x00000c90, 8)` comes from `(register, 0x4098, 8)` with an offset `0x30`, and `(register, 0x4098, 8)` comes from the first parameter of the current function. With backward slicing, we resolve the base address of field address `(unique, 0x00000c90, 8)` as `(register, 0x4098, 8)`. This procedure is referred as FIELDBASEADDRESSRESOLUTION. Note that *P-Code* has made dependence analysis such as slicing easier by simply inspecting the data-def and use chain. For instance, the Ghidra API `Varnode.getDef` can be used to find the *P-Code* instruction that defines a particular variable.

- **Step-II: Resolving the corresponding class of the base address.** Having recognized the base address of the target field, we need to identify its class types, *e.g.*, to find the class definition of `(register, 0x4098, 8)`. Fortunately, this is also an easy process given the metadata we have collected. In particular, a base address accessed in a function usually comes from three sources: *a*) the address of other class object passed from the function parameter, *b*) a return value of a function call, and *c*) a field from another class. Therefore, we resolve the class information based on its definition. This procedure is referred as BASECLASSRESOLUTION (line 25-39). In this procedure, we first locate the definition of the base address (line 26), if it is from either *a*) or *b*) (line 27, 30), we directly get its class by querying the *GFT* (line 29, 32) since *GFT* has the class type for the function parameters and return value. If the base address is from *c*) (line 33), we apply FIELDBASEADDRESSRESOLUTION to find the base address of this new field (line 35) and apply BASECLASSRESOLUTION to find the class of the new field (line 36), then we get the class of the base address by querying *GCT* (line 37). Finally, if it is copied from another variable, we will resolve it recursively (line 38-39).

- **Step-III: Tainting the class field.** With the identified class of the destination field, we then taint the destination memory address in the shadow memory, and also taint the corresponding field of the identified class in the *GCT* (line 16). Note that the purpose of having *GCT* is to track the tainted classes (and its fields), and iterate all of the functions that access any of the tainted classes. Also, we do not taint the single primitive type (*e.g.*, the first `String` type entry in the *GCT* at shadow memory `{1}` in Figure 5, and we mark it N/A), as primitive type will be used in many other fields, and instead we taint the class and the offset at that particular class (as shown in shadow memory `{2}`, our algorithm has tainted the field at offset `0x30` of class `unityInAppPurchase_LS`.

Repeating the propagation analysis. After identifying the tainted classes, we next iterate each instruction of the functions that have accessed these classes (in both the caller and the function itself), to identify whether there are any other class fields to which the tainted data can be propagated. If so, we add these classes into our tainted class as well. We call this process PROCESSFUNCTIONWITHTAINTEDCLASS (line 40-55). In particular, for each function in *GFT* (line 7), we iterate on the tainted class-field in *GCT* (line 41), to find which functions used the class (line 42-43). Then we *a*) iterate on the caller of the function call (line 45) to find out who read the class-field (line 46-48), and then we perform taint propagation from that point (line 50); *b*) iterate on the function itself (line 51) to perform the same actions (line 52-55) in our algorithm.

5.3 Vulnerability Detection

With the identified tainted classes and their corresponding tainted fields, we then iterate all of the functions again, to identify whether the known taint sinks have accessed any tainted fields. At a high level, we have two types of known taint sinks: (1) the APIs that send out data (*e.g.*, flowing into network related APIs) and these APIs include HTTP/HTTPS and Socket, and (2) payment local verification API (*e.g.*, `UnityEngine.Purchasing.Security.CrossPlatformValidator.Validate`). If any of these taint sinks have accessed the payment (and its propagation), we use the following security policies to identify the vulnerabilities.

- **Identifying local-verification vulnerability**: During the iteration of each function, if we notice the tainted data (by querying the corresponding class field) is used by Unity API `CrossPlatformValidator.Validate`, and also there is no program path that also sends the tainted data to networking APIs, we conclude a local-verification vulnerability is detected.
- **Identifying no-verification vulnerability**: We check every function to make sure there is no payment data (or its propagation) sent to the outside (*e.g.*, through network APIs) and meanwhile no local-verification APIs involved. If so, we conclude it is a no-verification vulnerability.

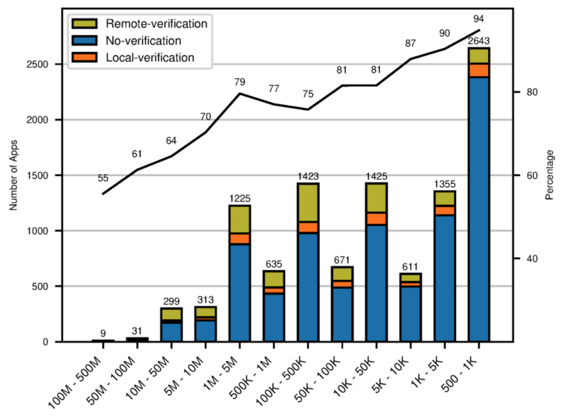
Therefore, we have used a very conservative policy to detect the vulnerability: if a game neither belongs to local verification nor no-verification, it will be classified into remote-verification, which includes the cases that we cannot identify the receipt data-use and the receipt is sent to the server. However, it is not guaranteed that those games are secure. Because we cannot confirm that the server will verify the transaction due to the lack of access to server side code. As such, we assume it is remote-verification, to avoid having too many false positives. Certainly, this will lead to false negatives. Being a vulnerability detection tool, we consider this is acceptable, as we cannot guarantee to detect all the vulnerabilities.

6 Evaluation

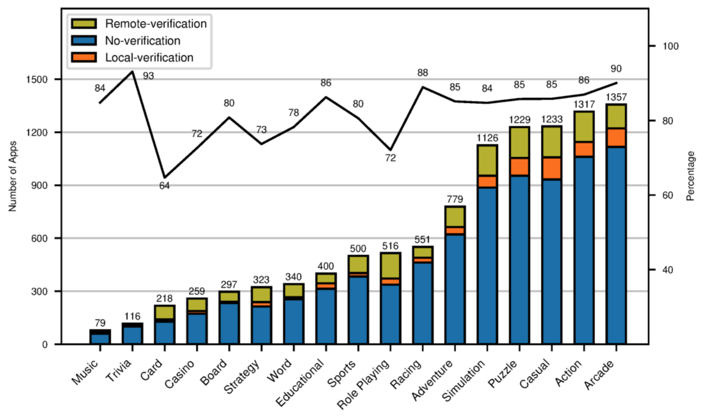
We have implemented PAYMENTSCOPE based on Ghidra, and its source code has been made available at https://github.com/OSUSecLab/PaymentScope. In this section, we present the evaluation results. We first describe how we collect the game apps and set up our testing environment in §6.1. Then, we provide our detailed results of the identified vulnerabilities and also how PAYMENTSCOPE performs in §6.2. Finally, we present the security analysis including case studies of these identified vulnerabilities in §6.3.

6.1 Experiment Setup

Dataset. We aim to understand the developer's practice at scale, and therefore we would like to test all the Android apps on Google Play as what we have done in our prior works such as LeakScope [58]. Interestingly, instead of crawling apps directly from Google Play, we notice that we can actually leverage an existing app repository, AndroZoo [21]. However, there are more than 10 million apps in AndroZoo, many of which no longer exist in Google Play. Therefore, we only focus on the apps that are still in Google Play, which eventually resulting



(a)



(b)

Figure 6: The Vulnerable Rates and Distribution of the Games Based on the App Download Numbers (a) and Categories (b).

in 2.1 million apps. By systematically iterating these apps, we found 293,019 game apps. Meanwhile, we only focus on the games built on top of Unity (by checking a unique file named `libil2cpp.so` in the APK) and used the `IL2CPP` compilation option. By applying these filtering rules, we eventually acquired 39,121 unique games (with their latest version). Note that for Google Play apps, we only need to query the metadata (by using the seeds provided in AndroZoo) and check their existence, and the concrete APKs are downloaded from AndroZoo. However, we also note that there are some games with multiple APKs [19] in which the native binaries are stored in a separate APK from the main APK. These games are not in our dataset since AndroZoo only contains the main APK.

Game Architecture Analysis. To understand the supported architectures among the analyzed 39,121 games, we have grouped them based on the download numbers and the supported architectures, and this result is presented in Table 1. We can see that most games support ARM64, then followed by ARM32, and the least supported architecture is x86. Also, we can notice that nearly none of the games only support x86, some games support both ARM32 and x86, and some support all architectures (ARM32, ARM64, and x86). A unique benefit of developing PAYMENTSCOPE based on the P-Code of Ghidra is that we can simultaneously handle the binaries of all these architectures. Also, if a game contains multiple `libil2cpp.so` files, we only need to analyze one of them, as they are all generated by tool `IL2CPP`.

Machine configuration. PAYMENTSCOPE is implemented on top of Ghidra, which is a resource (*e.g.*, CPU, memory, and storage) intensive static binary analysis framework. For better performance, we carried out our experiment in a DELL server which has two E5-2695 v2 CPUs (48 cores in total), 96GB memory, and 10TB storage, and installed with Ubuntu Server 16.04 operating system.

6.2 Evaluation Results

Effectiveness. To analyze these 39,121 Unity games, we first applied `IL2CPPDumper` to extract their metadata, which contains the APIs called by the game. By inspecting whether there is any use of in-app purchasing APIs, we filtered out those apps that never use the IAP service, and eventually we identified 10,640 games. By applying our payment-aware data flow analysis on these games, PAYMENTSCOPE identified 8,233 games with no-verification and 721 games with local-verification. For the 1,686 non-vulnerable games (*i.e.*, identified as using remote-verification), 597 of them perform both local and remote verification.

Next, to gain insights on these vulnerabilities and their corresponding vulnerable games, we report two types of distributions. One is the distribution of these vulnerable games based on popularity (in terms of number of downloads), and the other is the distribution based on game category (*e.g.*, Casino, Card, and Trivia). These two distributions are presented in Figure 6a and Figure 6b, respectively. In particular, by grouping the games based on the download numbers and vulnerable verification types (no-verification vs local-verification), we can see that the vulnerabilities widely exist among the games presented in Figure 6a, and they affect not only less popular games, but also super popular ones. For instance, there are five vulnerable games each with more than 100 million installs, and there are also in total 1,263 vulnerable games each with more than one million installs. From the vulnerable rate line shown in the figure, we can see that it goes up from 55.56% (super popular games) to 94.78% (least popular games), which indicates that less popular games have a much higher possibility of being vulnerable, and the developers of popular games are more careful to secure their IAPs.

Using the popularity of the games that could provide insights of how likely a game could contain vulnerable verification, we next seek to understand whether the vulnerability

		Disjoint (by Supported Architectures)														Joint (by Supported Architecture)					
# ↓	# Game	A32	%	A64	%	X86	%	A32, A64	%	A32, X86	%	A64, X86	%	A32, A64, X86	%	A32	%	A64	%	X86	%
1B - 5B	1	1	100.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	1	100.00	0	0.00	0	0.00
500M - 1B	2	2	100.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	2	100.00	0	0.00	0	0.00
100M - 500M	27	4	14.81	10	37.04	0	0.00	10	37.04	0	0.00	0	0.00	3	11.11	17	62.96	23	85.19	3	11.11
50M - 100M	67	3	4.48	26	38.81	0	0.00	33	49.25	2	2.99	0	0.00	3	4.48	41	61.19	62	92.54	5	7.46
10M - 50M	534	19	3.56	189	35.39	0	0.00	297	55.62	18	3.37	0	0.00	11	2.06	345	64.61	497	93.07	29	5.43
5M - 10M	579	24	4.15	216	37.31	0	0.00	305	52.68	13	2.25	0	0.00	21	3.63	363	62.69	542	93.61	34	5.87
1M - 5M	2,293	94	4.10	811	35.37	1	0.04	1,266	55.21	67	2.92	1	0.04	53	2.31	1,480	64.54	2,131	92.94	122	5.32
500K - 1M	1,372	54	3.94	413	30.10	0	0.00	790	57.58	67	4.88	1	0.07	47	3.43	958	69.83	1,251	91.18	115	8.38
100K - 500K	3,707	149	4.02	1,116	30.11	0	0.00	2,187	59.00	126	3.40	0	0.00	129	3.48	2,591	69.89	3,432	92.58	255	6.88
50K - 100K	1,820	64	3.52	584	32.09	0	0.00	1,035	56.87	52	2.86	1	0.05	84	4.62	1,235	67.86	1,704	93.63	137	7.53
10K - 50K	4,482	148	3.30	1,229	27.42	0	0.00	2,699	60.22	131	2.92	2	0.04	273	6.09	3,251	72.53	4,203	93.78	406	9.06
5K - 10K	2,112	52	2.46	544	25.76	0	0.00	1,309	61.98	72	3.41	0	0.00	135	6.39	1,568	74.24	1,988	94.13	207	9.80
1K - 5K	5,156	135	2.62	1,183	22.94	0	0.00	3,314	64.27	154	2.99	0	0.00	370	7.18	3,973	77.06	4,867	94.39	524	10.16
500 - 1K	16,969	231	1.36	2,805	16.53	0	0.00	12,195	71.87	404	2.38	12	0.07	1,322	7.79	14,152	83.40	16,334	96.26	1,738	10.24

Table 1: The Supported Architectures of the Analyzed Games (A32: arm32; A64: arm64; ↓: Download times)

has any association with the game category (based on what has been assigned by Google Play), and this result is presented in Figure 6b. Interestingly, we can see that the vulnerable rate for different categories varies a lot (from 64.68% to 93.10%), and there are two categories, namely "Arcade" (90.05%) and "Trivia" (93.10%), which have an extremely high vulnerability rate. A possible reason is that games in these two categories are more likely single-player games, and these games depend less on the servers (*i.e.*, mostly off-line games). As such, developers are very likely to implement the in-game purchasing with only local verification or without verification at all. In contrast, games in the category of "Card" (64.68%) often provide multi-player features that heavily depend on the network service, and developers are more likely to verify the in-game purchasing using server-side verification.

Efficiency. When conducting the experiments, we run 24 instances of Ghidra in parallel to analyze the games. It took 669 hours (almost 28 days) to perform the analysis. Among the processing time, it took Ghidra 665 hours to pre-process the inputs (*e.g.*, converting binary to *P-Code*), and only took 4 hours to actually perform the payment-aware data flow analysis. Also, for each game, Ghidra created a project, and those projects occupied 5.8 TB hard drive space in total.

To zoom in the internals of how PAYMENTSCOPE analyzed each game, we present a set of intermediate results in Table 2. For simplicity, we chose the top game in each category, if there are multiple games with the same download number, we randomly pick one up from them to show the result. Specifically, as shown in the table, we particularly present the binary file size of `libil2cpp.so`, the number of extracted classes and the number of extracted functions from the metadata (namely the size of GCT and GFT), the total number of tainted items, the total number of instructions involved in taint analysis, the total number of functions that are iterated by our data flow analysis, and finally the total number of identified traced paths.

We can see that all the binaries are bigger than 15MB (one game is even more than 80MB) which is the main reason why it took Ghidra so long to pre-process. For $|GCT|$ and $|GFT|$, we can see that they both contain thousands of items. But for the taintedness identification, we do not see many instructions (maximum less than two thousand instruction) are involved, partially because the payment data will not propagate to too many other variables. Also, interestingly, for game Happy Glass, the numbers are all 0. With further investigation, we found that the game never calls API `get_receipt`.

6.3 Vulnerability Analysis

(I) False positive and false negative analysis. Being a static analysis tool, PAYMENTSCOPE could have false positives (FPs) and false negatives (FNs). Specifically, the FPs in our context are the games that are secure but reported as vulnerable and the FNs are the games that are vulnerable but reported as secure (i.e., we consider it uses remote-verification as described in §5.3). To quantify FPs and FNs, we have to manually examine the games. In particular, to quantify FPs, we need to test whether the identified vulnerable games can indeed purchase for free; to quantify FNs, we need to test whether the games identified non-vulnerable indeed secure (containing remote verification). However, such a manual analysis will be very time consuming and it is not practical to test all games. For instance it took us more than 30 minutes to play the game EGGLIA until the IAP option shows up. Therefore, we have to sample the games for our manual analysis. To avoid potential biases, we eventually sampled 280 games (220 for FPs and 60 for FNs) with the following two strategies:

- **Targeted selection.** We first select the games in the targeted group based on the number of installs. Specifically, to test FPs, we select the top 10 no-verification games and the top 10 local-verification games; to test FNs, we select the top 10 remote-verification games.
- **Random selection.** We then randomly select the games. In particlar, to test FPs, we randomly selected 200 games from 8,954 vulnerable games; to test FNs, we select 50 games from 1,686 secure (remote-verification) games.

Techniques to launch the attacks. To confirm the vulnerability of these games, there are multiple ways such as

Game Name	# ↓	|Size| (MB)	# Classes in GCT	# Functions in GFT	# Tainted Items	# Instruction in Tainting	# Iterated Functions	# Traced Paths
aquapark.io	100,000K	31.6	8,149	76,743	295	385	11	125
Critical Action	50,000K	30.0	6,810	57,446	20	26	5	10
Crowd City	100,000K	27.4	5,630	51,839	527	671	11	207
Onnect	10,000K	19.8	4,252	36,528	860	1,073	9	370
RummyCircle	10,000K	21.1	4,717	44,182	45	64	6	27
GSN Casino	10,000K	84.6	17,510	132,353	58	68	8	18
Hello Kitty Nail Salon	100,000K	34.8	7,071	67,915	1,589	1,739	21	782
Little Panda's Restaurant	10,000K	17.5	5,184	42,727	11	18	5	9
Dot n Beat	10,000K	26.5	5,830	49,911	130	176	9	69
Happy Glass	100,000K	26.2	5,802	52,955	0	0	0	1
Moto Rider GO	100,000K	22.3	5,021	44,175	48	66	6	18
Gun Strike	50,000K	26.3	6,408	52,562	19	26	5	10
Real Cake Maker 3D	50,000K	30.7	6,878	63,386	553	720	13	224
Run Race 3D	100,000K	29.6	5,677	51,171	38	51	4	20
Game of Warriors	10,000K	26.8	5,996	56,311	7	11	3	5
MEGA QUIZ GAMING 2020	1,000K	23.4	4,681	43,173	265	331	12	97
Word Connect	10,000K	15.6	3,820	27,806	56	75	6	22

Table 2: The Statistics of How PaymentScope Performed for Each Analyzed Game.

Vul.	Game MD5	Version	# ↓	# Reviews	"Purchased" Item	Price
No-verification	AE74936431D1268E6F1814F41393E916	4.3.18	100,000K	4,432,126	Android Robot	$0.99
	765A97DB5AC1D8C11DDEBFCAC50805FE	1.0.55	100,000K	909,418	900 Coins	$0.99
	C6D1E7DB5D3DAD11F3DD097260F52A9C	1.4.5	100,000K	1,142,902	1000 Coins	$1.99
	E6D852636E3A5B4EDB87AF70758B3405	1.3.5	100,000K	1,744,386	Remove Ads	$4.49
	5AAD7FF4794457E90F68BD34894FDCE7	1.1.4	50,000K	416,650	Remove Ads	$1.99
	1D591C2FA5687DF35D7B0F39B94D94E7	2.10	50,000K	1,502,696	5000 Coins	$1.99
	4D3502CEFAB5699073D64F9343A405A8	4.1.0	50,000K	230,424	6000 Coins	$0.99
	4FDA1515973B164603928AC80E3C57CB	2.6.9	50,000K	115,886	Remove Ads	$2.99
	9204AEDD6665A1BA5374A064AD2E49D6	1.185	50,000K	444,558	15 Power	$1.99
	22A40571718A137EA646F0073CDAD361	1.2.5	50,000K	778,685	Remove Ads	$2.99
Local-verification	4057B81EFE3BAFEA151AF910E92AF015	1.27.1	100,000K	648,749	Special Case	$1.99
	9F63D671E0812355CE39CF7D1EE15BF0	4.3.39	50,000K	665,712	200 Diamonds	$1.99
	86D4E0E5C9DB42253A26D48A3ADCB4E1	1.21.1	50,000K	1,822,549	50 Gold	$5.99
	A4BE318C5CCF94FFFA74E6041A3F4632	1.4.44	10,000K	3,457,150	Bunch of Gems	$0.99
	A2CE43BF4E99D429ADDBB169E24928BA	1.36.05.0	10,000K	110,187	Pile of Gems	$0.99
	2B6D5AA12FB0D4AE3FC2303C9410E218	1.4.6	10,000K	142,123	Remove Ads	$2.99
	B7AB0E7969CD1E8D9D5C231769E0CF35	1.3.9	10,000K	300,158	Remove Ads	$3.99
	7943CD8FB582625871BC1645680D8FAB	9	10,000K	552,885	Double Coins	$1.99
	205F76E38A5A91E7E5FB08D4D3CE2F47	1.2.4	10,000K	464,386	125 Coins	$1.99
	6562953B4FAEFA34AEC3779EE2DE828E	4.4.35	10,000K	83,784	200 Diamonds	$1.99

Table 3: The Detailed Case Study of the Top-10 Vulnerable Games in Each Vulnerability Category.

virtualization and repackaging. In the following, we describe how to test them to confirm the vulnerabilities:

- **Injecting fake transaction using virtualization.** A fake transaction needs to be injected to the game when a purchase request has been initialized. In particular, we need to hijack the return value of the purchasing API (Step ❸ in Figure 2), and replace it with a fake success transaction so that the game believes the transaction has succeeded. To this end, we first launch the Payment Request (Step ❶ in Figure 2), but we then cancel the payment when the purchasing confirmation dialog pops up. Through the use of virtualization [26], we dynamically hook the Android system APIs by using our prior work AutoForge [59], to hijack the `Intent` which contains the return values (*e.g.*, state, receipt) of the payment transaction, this attack succeeds.

- **Disabling local-verification using code patching.** Some tested games would verify the transaction locally. Therefore, when this fake transaction receipt is passed to `CrossPlatformValidator.Validate` API, it will certainly fail the validation check if we use the original code, but we can first patch this API to make the validation always return true regardless of the receipt. Note that the static binary code patching will change the signatures of the entire game APK, and the games that check their integrity [29] may refuse to run. As such, we use `Frida` [4] to instrument the code at run time. `Frida` will be enabled right before we trigger the IAP.

With the above attack techniques, we then run these 280 games to launch an in-app purchasing request. To minimize the damages to the developers, we immediately cancel the transaction while interacting with app store (basically at Step ❷ shown in Figure 2). Note that this cancelling process is through a pop-up dialog, which is transparent to the game (and handled by the Android framework). If we can "purchase" these virtual products for free, then we confirm the game is indeed vulnerable.

Results. When analyzing the 220 games to quantify FPs, we found that 30 games could not be tested. Some of them crashed when running in our test phone; some needed an

Anonymized Developer ID	# Vul. Games	No Verification	Local Verification	# Total Games	Vul. Rate	∑(#↓)
1	91	91	0	91	100.00	169K
2	69	69	0	69	100.00	41,130K
3	35	35	0	35	100.00	182,321K
4	34	34	0	34	100.00	120,550K
5	33	33	0	33	100.00	37,850K
6	28	28	0	28	100.00	554K
7	25	25	0	25	100.00	1,126K
8	24	24	0	24	100.00	65,910K
9	23	23	0	23	100.00	998K
10	22	17	5	23	95.65	32,410K

Table 4: Top 10 Vulnerable Game Developers Ranked Based on Total Number of Vulnerable Games.

Anonymized Developer ID	∑(#↓)	No Verification	Local Verification	# Total Games	Vul. Rate
A	321,100K	7	1	11	72.73
B	198,150K	11	1	16	75.00
C	182,321K	35	0	35	100.00
D	125,215K	13	1	17	82.35
E	120,550K	34	0	34	100.00
F	112,500K	4	2	14	42.86
G	100,000K	1	0	1	100.00
H	95,000K	3	3	8	75.00
I	90,000K	5	0	28	17.86
J	87,710K	17	0	31	54.84

Table 5: Top 10 Vulnerable Game Developers Ranked by Total Number of Downloads of Vulnerable Games.

update that was not available in our Google Play (*i.e.*, United States) due to region restriction. For the other 190 games, all of them were vulnerable and the attacks succeeded. We show the detailed information of the top games in Table 3. The "purchased" item and its price are reported in last two column of Table 3. Although the experiment shows that PAYMENTSCOPE has no FPs, theoretically it may still have FPs. For instance, the game may load additional code at runtime which will perform remote-verification, or the game uses reflection call to access receipt which is hard to detect, though we did not witness such cases in our manual analysis.

When analyzing the 60 games to quantify FNs, we found that 9 games cannot be tested due to the same reasons. Among the rest of the 51 games, 37 of them are secure. The attacks failed and some games show dialog, such as "invalid purchase". Therefore, we confirmed that the games performed receipt verification at the server side. By intercepting the traffic of the tested games, we found that the response included both the verification result and the game state such as the number of coins. For the other 14 games, we were still able to purchase for free. Consequently, this eventually leads to the FNs of 29%. By performing manual reverse engineering on the games, we found that, in some games the receipts have indeed been sent to the servers but the responses do not indicate that the receipts are invalid. The servers may have just used the receipts for the logging purpose without any receipt verification, but PAYMENTSCOPE cannot confirm this automatically since the code of the server side is unavailable. Therefore, it has a high FN. However, on the other hand, this just indicates that the true rate of this vulnerability could be even higher, which further shows how prevalent this problem is.

(II) Top vulnerable game developers. After confirming the FPs and FNs of PAYMENTSCOPE, we next seek to understand why there are so many vulnerable games, who developed them, and how many total installs these vulnerable games have accumulated (to estimate potential losses for developers). To this end, we retrieve the `developerName` (as well as the email address used for responsible disclosure) from the metadata returned by Google Play when querying each vulnerable game. We then cluster the vulnerable games by `developerName`, and also accumulate the total number of downloads. Then we rank the developers based on the total number of vulnerable games they have developed, and also the total number downloads their vulnerable game have accumulated.

The top 10 developers who produced the most number of vulnerable games are listed in Table 4. We can see that the top developer has 91 vulnerable games. Also, except the 10th developer who has one game that is not vulnerable, all of mobile games developed by these developers are vulnerable to no-verification attack. The top 10 developers whose vulnerable games have gained the most downloads are presented in Table 5. We can see that the vulnerable games developed by the top 7 developers have more than 100 million downloads. Most of the vulnerabilities in these games are no-verification. Also, interestingly, by looking at all of the games from the corresponding developers, we found that not all of their games are vulnerable. It could be the reasons that these games are developed by different individuals or teams.

(III) Potential Financial Impact. To understand the financial impact, we would like to estimate, if the vulnerabilities were exploited, and if so, what the potential financial losses for developers could have been. To this end, we created a heat map in Figure 7 to show the distribution of product prices of the vulnerable games. Specifically, we group the games based on the number of downloads in the X-axis. Each item in the Y-axis represents a price range. The color of each cell indicates the percentage of the vulnerable games out of all of the vulnerable games that provide products whose prices fall into the corresponding price range in that specific downloading category. Also, for simplicity, for each game, we directly retrieve the minimum price and maximum price of the sold products for each game from Google Play, and use this price information to fill the corresponding cell. According to the result presented in Figure 7, we can see that less popular games usually sell cheaper products, and high price products are usually sold in extremely popular games. This result makes sense since it is more likely to have profit when having large

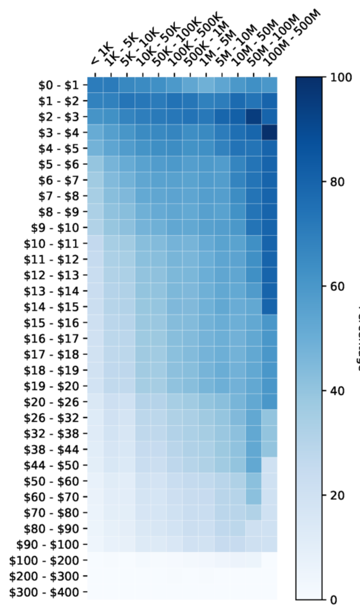


Figure 7: Distribution of the Prices of Sold Products in the Vulnerable Games

number of users. Also, only few games sell products beyond $100, and most common prices are less than $5.

7 Discussion

7.1 Root Causes

Fundamentally, the root cause of the identified vulnerable in-game purchases comes from the misplacement of trust. In particular, a game without any payment verification or with local-verification places the trust on the mobile operating system (by assuming uncompromised operating systems without being rooted) and the game app itself (*i.e.*, by assuming the integrity of the app). Unfortunately, neither the mobile operating systems nor the game app itself are always trustworthy given the large number of hacking tools (*e.g.*, virtualization, repackaging, and rooting) in Android.

As such, a secure verification approach is to place the transaction verification in a trusted place such as either the remote game servers or the local trusted execution environment (TEE) such as ARM TrustZone, which is an isolated space guaranteed by the hardware and has already been integrated in mainstream mobile SoCs. Recently, there are quite a number of efforts studying how to use TEE to protect games (e.g., [22, 30, 42]). However, , it is a non-trival task to protect IAP with TEE because only moving the transaction verification to TEE is not secure since the TEE calls can be bypassed or manipulated. As such, the TEE should also maintain the game states. In other words, the game need to track its states (*e.g.*, handling in-game currency) in the aforementioned trusted places (in fact many server-based games have already implemented this mechanism for the purpose of cross-device-game-state-syncing); otherwise, the attacker can just remove the code of transaction verification. For game state tracking in server side, there are many inexpensive and easy to use 3rd-party services that can be leveraged, such as PlayFab provided by Microsoft [7] which offers a free plan for games that have up to 100,000 users. Moreover, we also believe that SDK providers such as Unity should remove the local verification API and instead provide server side APIs to help the app developments.

7.2 Limitations

While PaymentScope has successfully identified a significant number of mobile games containing vulnerable in-game purchases, it still has at least two limitations. First, the key technique in PaymentScope is the static data flow analysis, which can fail in various cases, such as when encountering indirect propagations (*e.g.*, data propagation via conditions, or via writing and reading through files). If a game uses such payment-based propagations, PaymentScope will have false positives (*e.g.*, concluding a game is vulnerable but there is actually indirectly propagated payment that flows to network APIs). While currently we have not witnessed such a case in FP analysis, it could exist in theory.

Second, as shown in our experiment, our tool could also have FNs, and 14 out of 51 manually verified games were indeed vulnerable but not detected by PaymentScope. One reason for this FN is that our detection policy blindly considers the game is not vulnerable if the receipt is sent to the server. However, the server may still not perform payment verification, but we do not have the server side code to confirm this. Also, theoretically, another source of FN could come from over-tainting because of our type-based taint analysis, which could cause the non-payment data be considered as payment.In addition, if the tainted data flows into a container (e.g., an array), it is challenging to identify its further usage and PaymentScope will just conclude it is secure due to its very conservative detection policy, leading to FNs as well.

7.3 Future Work

In addition to addressing the aforementioned limitations, there are mutiple avenues in the future work, such as extending PaymentScope to handle (1) other game engines and (2) other platforms such as iOS.

Handling other game engines. Currently PaymentScope only detects vulnerabilities in games built with Unity Engine, but ignores games implemented using other game engines. In fact, similar to Unity engine, we did find many other engines such as Unreal Engine (UE4) also provide wrapped payment API [17] for the game developers to implement IAP. In particular, games built with UE4 can call APIs to initialize a transaction, and the APIs will perform call backs (*e.g.*, `On Success`, `On Failure`) when the payment provider succeeded or not to process the transaction. Then the game can verify the transaction at the server side, or it can perform local verification or no verification at all, which will lead to payment bypass. To confirm this, we have downloaded the top 5 games that are developed based on UE4 and performed a manual study to check whether they are vulnerable to our payment bypass attacks. Through dynamic analysis of the games, we found two of them (i.e., games with MD5 `6cd6314b084514647c8f067fe34dad32` and `6e260f7d5ba30cf2c78d61b2e007a6e0`) are also vulnerable.

Our future work will aim to automate PAYMENTSCOPE for these games. Although PAYMENTSCOPE is targeting Unity based games, the key observation of *abstracting the payment-bypassing vulnerability detection problem to a data flow analysis problem and solving it using taint analysis* can be applied to other game engines. However, we anticipate there will be at least two game-engine specific challenges:

- **Selecting a proper static analysis framework**. Note that game engines are often language specific. For instance, Unity based games can be compiled to native binary or `IL` bytecode (older version), libGDX based game is in `Java` bytecode format, and Cocos2d based game may in `Lua` script format. For a particular file format, a proper static analysis framework is needed to parse the game and implement the analysis algorithm. There are some existing mature frameworks such as Ghidra for binary and Soot for `Java` bytecode. Therefore, we have to properly select the corresponding static analysis framework for each specific engine.

- **Recognizing the specific APIs in the framework**. Different game engines provide different APIs for IAP and other functionalities. We have to identify the corresponding APIs for the taint source (*i.e.*, the payment receipt definition APIs), taint sinks (*e.g.*, networking APIs, receipt validation APIs), and taint propagations (*e.g.*, string or JSON data manipulation APIs). However, not all of the symbols are available as in Unity, and how to recognize them will be non-trivial.

Handling other platforms. Certainly, the identified vulnerabilities not only exist in Android games but also iOS games. To confirm that, we have downloaded two iOS games: Game A with MD5 `86D4E0E5C9DB42253A26D48A3ADCB4E1` and Game B with MD5 `A2CE43BF4E99D429ADDBB169E24928BA` from Table 3, through AppleJam [50]. By analyzing these two games we found that they are consistent with the corresponding Android version and vulnerable to payment-bypassing attack.

7.4 Ethics and Responsible Disclosure

We did take ethics seriously when conducting this study. First, when confirming FPs and FNs of the vulnerabilities (§6.3), we did launch playing without paying attack and we chose to "purchase" the cheapest product in the tested games for proof-of-concept. However, in order not to cause any damages to the developers, we then did a real purchase in a separate transaction for the tested games. Therefore, from developer's perspective, there is no financial loss and they even gained income because of our test. Second, we shared our findings with Unity. We also learned from Unity that they have acquired a startup company recently to particularly focus on alleviating the developers' efforts of server side verification. Finally, we disclosed our identified vulnerabilities, with detailed explanation (including the root cause, game detail) to all the vulnerable games through the email they left on the Google Play.

In total, we contacted 5,494 game developers who developed these 8,954 vulnerable games (note that some developers developed multiple vulnerable games). Many of the developers have acknowledged our findings. More specifically, some developers promised to patch the vulnerability as soon as possible; some replies indicate that the team will investigate the reported issue; some developers asked for articles where they can find more detailed information about this attack. Also, one developer mentioned that they have already switched to remote-verification in a later version of the tested version; one developer stated that he/she cannot update the game anymore due to that the source code having been lost; one developer shared with us that there is a game hacking tool named Lucky Patcher [20] that can actually be used to attack no-verification games.

8 Related Work

Payment security. Wang et al. [49] studied the shopping websites that allow users to use 3rd-party payment services, and discovered that several websites (*e.g.*, Buy.com and JR.com) allow a malicious customer to purchase products with low price or even for free due to the logic flaw in the payment integration, which also exists in many e-commerce applications [47]. When moving to mobile payment, many of the old problems such as authentication and malware threats still exist [51], but the directly integration of payment via app store also introduces new threats. For instance, Reynaud et al. [44] and Lai et al. [33] found a vulnerability which can lead to payment bypassing in mobile apps with in-app purchasing. Also, Mulliner et al. [40] designed a framework to protect the apps from automatic in-app billing attacks. In addition, Yang et al. [54, 55] found several flaws in 3rd-party payment SDKs in mobile apps, rather than the Unity SDKs focused by our work with novel native binary analysis. There are also efforts to study the security of the mobile payment protocols [31] [56] including the video-on-demand subscription services [35].

Game security. There has been an arms race between game cheating and anti-cheating [38]. Particularly, in the desktop online games, numerous efforts have been made to fight for game bots by exploiting inconsistencies [23, 37], similarities [34], and human observational proofs [28]. There are also efforts to defeat secret revealing in game states such as exploring private set intersection protocols as in OpenConflict to protect game maps [24], and also Intel SGX to defeat wallhacks as in BlackMirror [42]. In the mobile games, Tian et al. [48] studied the existing attacks such as modification of memory and network traffic in the mobile game cheating, and also provided a reference framework for the game defense. This study has particularly mentioned the untrusted client attack, and our work provides concrete evidences for such attacks and their potential impact.

Binary analysis. Binary analysis is a powerful technique for vulnerability identification. Over the past decades, a large body of research has been carried out of either improving the binary analysis itself or applying binary analysis for vulnerability discovery, as summarized by Shoshitaishvili et al. [46]. Built with foundation techniques including program slicing [52], data flow analysis [32, 53] (or taint analysis [25,41,57]), PAYMENTSCOPE complements existing work by exploring the direction of mobile game binary analysis and focusing on detecting the in-game purchasing bypassing vulnerabilities.

9 Conclusion

We have presented PAYMENTSCOPE, a static binary analysis tool built on top of Ghidra to automatically identify vulnerable in-app purchasing implementations in mobile games binaries developed by the Unity SDK. The key idea is to model the vulnerability detection problem using a payment-aware data flow analysis, and leverage the metadata inside Unity game for the binary analysis. We have implemented PAYMENTSCOPE and tested with 39,121 games. Surprisingly, our tool has identified 8,233 games that do not verify the validity of payment transactions and 721 games that simply verify the transactions locally. Such a high rate of vulnerability shows how prevalent the insecure programming practice (by misplacing the trust) is for in-app purchasing. Finally, to really make in-game purchasing secure, we believe SDK providers should provide APIs to ease the server side payment verification.

Acknowledgment

We thanks the anonymous reviewers for their invaluable feedbacks. This research was supported in part by NSF award 1834215. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] "Apkmody - download mod apk games & premium apps," https://apkmody.io/, (Accessed on 2/12/2022).

[2] "The best 10 mobile game engines and development platforms & tools in 2019 | computools," https://computools.com/the-best-10-mobile-game-engines-and-development-platforms-tools-in-2019/.

[3] "Creating and Using Scripts," https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html.

[4] "Frida - Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers." https://github.com/frida/frida.

[5] "Github - nationalsecurityagency/ghidra: Ghidra is a software reverse engineering (sre) framework," https://github.com/NationalSecurityAgency/ghidra, (Accessed on 2/13/2022).

[6] "Il2CppDumper," https://github.com/Perfare/Il2CppDumper.

[7] "Microsoft Azure PlayFab | Full Stack LiveOps, Real-time Control," https://playfab.com/pricing/.

[8] "News, free games and program for android," https://an1.com, (Accessed on 2/12/2022).

[9] "NHN AppGuard," https://www.toast.com/kr/service/security/nhn-appguard.

[10] "P-code reference manual," https://github.com/NationalSecurityAgency/ghidra/blob/master/GhidraDocs/languages/html/pcoderef.html, (Accessed on 2/13/2022).

[11] "Receipt validation programming guide," https://developer.apple.com/library/archive/releasenotes/General/ValidateAppStoreReceipt/Chapters/ValidateLocally.html#//apple_ref/doc/uid/TP40010573-CH1-SW9, (Accessed on 2/13/2022).

[12] "Rest resource: purchases.subscriptions," https://developers.google.com/android-publisher/api-ref/rest/v3/purchases.subscriptions, (Accessed on 2/12/2022).

[13] "Setting up a publisher account," https://developer.android.com/google/play/licensing/setting-up#account, (Accessed on 2/13/2022).

[14] "The future of scripting in Unity," https://blogs.unity3d.com/2014/05/20/the-future-of-scripting-in-unity/.

[15] "The ultimate mobile in-app purchases guide," https://instabug.com/blog/mobile-in-app-purchases/, (Accessed on 2/12/2022).

[16] "Unity real-time development platform | 3d, 2d vr & ar engine," https://unity.com/.

[17] "Using in-app purchases," https://docs.unrealengine.com/en-US/SharingAndReleasing/Mobile/InAppPurchases/index.html, (Accessed on 2/12/2022).

[18] "Video game market revenue worldwide in 2021," https://www.statista.com/statistics/292751/mobile-gaming-revenue-worldwide-device/, (Accessed on 2/12/2022).

[19] "Multiple APK support | Android Developers," Dec 2019, [Online; accessed 18. Feb. 2022]. [Online]. Available: https://developer.android.com/google/play/publishing/multiple-apks

[20] "Lucky Patcher Official Website By ChelpuS - Lucky Patcher," Jan 2021, [Online; accessed 18. Feb. 2022]. [Online]. Available: https://www.luckypatchers.com

[21] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 468–471. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903508

[22] E. Bauman and Z. Lin, "A case for protecting computer games with sgx," in *Proceedings of the 1st Workshop on System Software for Trusted Execution (SysTEX'16)*, Trento, Italy, December 2016.

[23] D. Bethea, R. A. Cochran, and M. K. Reiter, "Server-side verification of client behavior in online games," *ACM Transactions on Information and System Security (TISSEC)*, vol. 14, no. 4, pp. 1–27, 2008.

[24] E. Bursztein, M. Hamburg, J. Lagarenne, and D. Boneh, "Openconflict: Preventing real time map hacks in online games," in *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 506–520.

[25] S. Chen, Z. Lin, and Y. Zhang, "{SelectiveTaint}: Efficient data flow tracking with static binary rewriting," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1665–1682.

[26] D. Dai, R. Li, J. Tang, A. Davanian, and H. Yin, "Parallel space traveling: A security analysis of app-level virtualization in android," in *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies*, 2020, pp. 25–32.

[27] C. Eagle, *The IDA pro book*. no starch press, 2011.

[28] S. Gianvecchio, Z. Wu, M. Xie, and H. Wang, "Battle of botcraft: fighting bots in online games with human observational proofs," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 256–268.

[29] M. Ibrahim, A. Imran, and A. Bianchi, "Safetynot: on the usage of the safetynet attestation api in android," in *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, 2021, pp. 150–162.

[30] S. Jeon and H. K. Kim, "Tzmon: Improving mobile game security with arm trustzone," *Computers & Security*, vol. 109, p. 102391, 2021.

[31] S. Kadhiwal and A. U. S. Zulfiquar, "Analysis of mobile payment security measures and different standards," *Computer Fraud & Security*, vol. 2007, no. 6, pp. 12–16, 2007.

[32] U. Khedker, A. Sanyal, and B. Sathe, *Data flow analysis: theory and practice*. CRC Press, 2017.

[33] Y.-c. Lai and M. Husain, "A holistic approach for securing in-app purchase (iap) vulnerability in mobile applications."

[34] E. Lee, J. Woo, H. Kim, A. Mohaisen, and H. K. Kim, "You are a game bot!: Uncovering game bots in mmorpgs via self-similarity in the wild." in *Ndss*, 2016.

[35] S. Lee, J. Kim, S. Ko, and H. Kim, "A security analysis of paid subscription video-on-demand services for online learning," in *2016 International Conference on Software Security and Assurance (ICSSA)*. IEEE, 2016, pp. 43–48.

[36] S. Lidin, *Inside Microsoft. net il assembler*. Microsoft Press, 2002.

[37] D. Liu, X. Gao, M. Zhang, H. Wang, and A. Stavrou, "Detecting passive cheats in online games via performance-skillfulness inconsistency," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 615–626.

[38] G. McGraw, *Exploiting online games: cheating massively distributed systems*. Addison-Wesley, 2008.

[39] A. Mendoza and G. Gu, "Mobile application web api reconnaissance: Web-to-mobile inconsistencies & vulnerabilities," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 756–769.

[40] C. Mulliner, W. Robertson, and E. Kirda, "Virtualswindle: An automated attack against in-app billing on android," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014, pp. 459–470.

[41] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software." in *NDSS*, vol. 5. Citeseer, 2005, pp. 3–4.

[42] S. Park, A. Ahmad, and B. Lee, "Blackmirror: Preventing wallhacks in 3d online fps games," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 987–1000.

[43] E. Peckham, "Unity ipo aims to fuel growth across gaming and beyond," https://techcrunch.com/2020/09/10/how-unity-built-a-gaming-engine-for-the-future/, (Accessed on 2/12/2022).

[44] D. Reynaud, D. X. Song, T. R. Magrino, E. X. Wu, and E. C. R. Shin, "Freemarket: Shopping for free in android applications." in *NDSS*, 2012.

[45] M. Sharir, A. Pnueli *et al.*, *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences . . . , 1978.

[46] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 138–157.

[47] F. Sun, L. Xu, and Z. Su, "Detecting logic vulnerabilities in e-commerce applications." in *NDSS*, 2014.

[48] Y. Tian, E. Chen, X. Ma, S. Chen, X. Wang, and P. Tague, "Swords and shields: a study of mobile game hacks and existing defenses," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 386–397.

[49] R. Wang, S. Chen, X. Wang, and S. Qadeer, "How to shop for free online–security analysis of cashier-as-a-service based web stores," in *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 465–480.

[50] Y. Wang, Y. Xie, and J. Li, "Applejam: An open dataset for ios app binary code analysis," https://github.com/xros-wyz/AppleJam, 2021, accessed: 2021-06-01.

[51] Y. Wang, C. Hahn, and K. Sutrave, "Mobile payment security, threats, and challenges," in *2016 second international conference on mobile and secure services (MobiSecServ)*. IEEE, 2016, pp. 1–5.

[52] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[53] H. Wen, Z. Lin, and Y. Zhang, "Firmxray: Detecting bluetooth link layer vulnerabilities from bare-metal firmware," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 167–180.

[54] W. Yang, J. Li, Y. Zhang, and D. Gu, "Security analysis of third-party in-app payment in mobile applications," *Journal of Information Security and Applications*, vol. 48, p. 102358, 2019.

[55] W. Yang, Y. Zhang, J. Li, H. Liu, Q. Wang, Y. Zhang, and D. Gu, "Show me the money! finding flawed implementations of third-party in-app payment in android apps." in *NDSS*, 2017.

[56] Q. Ye, G. Bai, N. Dong, and J. S. Dong, "Inferring implicit assumptions and correct usage of mobile payment protocols," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2017, pp. 469–488.

[57] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 116–127.

[58] C. Zuo, Z. Lin, and Y. Zhang, "Why does your data leak? uncovering the data leakage in cloud from mobile apps," in *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2019.

[59] C. Zuo, W. Wang, R. Wang, and Z. Lin, "Automatic forgery of cryptographically consistent messages to identify security vulnerabilities in mobile services," in *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS'16)*, San Diego, CA, February 2016.

Shared Object File Name	# Appearances	Type	Game Engine?
libmain.so	108,408	Unity	✓
libunity.so	107,694	Unity	✓
libmono.so	65,105	Unity	✓
libil2cpp.so	39,121	Unity	✓
libgpg.so	15,851	Library	
libgdx.so	9,480	libGDX	✓
libCore.so	9,428	Adobe Air	✓
libysshared.so	9,427	Adobe Air	✓
libc++_shared.so	8,691	Library	
libswappy.so	6,973	Library	
libswappyVk.so	6,967	Library	
libopenal.so	6,811	Library	
libplayer.so	6,118	Buildbox	✓
libFirebaseCppAnalytics.so	5,304	Library	
libstlport_shared.so	4,680	Library	
libMyGame.so	4,668	Cocos2d	✓
libyoyo.so	4,406	GameMaker	✓
libjs.so	4,373	Library	
libcocos2dcpp.so	4,145	Library	
libadcolony.so	4,132	Library	
libgdx-box2d.so	3,858	libGDX	✓
libmonobdwgc-2.0.so	3,655	Unity	✓
libstagefright_froyo.so	3,632	Adobe Air	✓
libstagefright_honeycomb.so	3,632	Adobe Air	✓
libgdx-freetype.so	3,574	libGDX	✓
libgnustl_shared.so	2,409	Library	
liblua.so	2,367	Library	
libmpg123.so	2,355	Library	
libFirebaseCppMessaging.so	2,338	Library	
libads.so	2,230	Corona	✓
libjnlua5.1.so	2,196	Library	
libxwalkcore.so	2,179	Library	
libxwalkdummy.so	2,091	Library	
libalmixer.so	2,064	Corona	✓
libcorona.so	2,064	Corona	✓
libanalytlibgameNetworkics.so	2,052	Library	
libgameNetwork.so	2,047	Library	
liblicensing.so	2,047	Library	
libMonoPosixHelper.so	1,695	Library	
libAnalytics.so	1,651	Library	
libcocos2djs.so	1,578	Cocos2d	✓
libApp.so	1,547	Unity	✓
libsqlite3.so	1,526	Library	
libplugins.so	1,487	Corona	✓
libeasymobile.so	1,467	Library	
libimagepipeline.so	1,407	Library	
libUE4.so	1,319	Unreal Engine 4	✓
libgsengine.so	1,295	GameSalad	✓
libFirebaseCppRemoteConfig.so	1,279	Library	
libgame.so	1,237	Cocos2d	✓
libmonosgen-2.0.so	1,223	Unity	✓
libmonodroid.so	1,216	Unity	✓
libBugly.so	1,177	Library	
libgodot_android.so	1,147	Godot	✓
libNativeABI.so	1,138	Adobe Air	✓
libandengine.so	1,136	AndEngine	✓
libfb.so	1,126	Library	
libfolly_json.so	1,123	Library	
libglog.so	1,123	Library	
libreactnativejni.so	1,123	Library	
libglog_init.so	1,121	Library	
libFirebaseCppCrashlytics.so	1,113	Library	
libcrashlytics.so	1,110	Library	
libyoga.so	1,100	Library	
libFirebaseCppAuth.so	1,084	Library	
libjsc.so	1,070	Library	
libVuforia.so	1,067	Vuforia Engine	✓

Shared Object File Name	# Appearances	Engine Name	Market Share
libmain.so	108,408	Unity	71.94%
libgdx.so	9,480	libGDX	6.29%
libCore.so	9,428	Adobe Air	6.26%
libplayer.so	6,118	Buildbox	4.06%
libMyGame.so	4,668	Cocos2d	3.10%
libyoyo.so	4,406	GameMaker	2.92%
libads.so	2,230	Corona	1.48%
libUE4.so	1,319	Unreal Engine 4	0.88%
libgsengine.so	1,295	GameSalad	0.86%
libgodot_android.so	1,147	Godot	0.76%
libandengine.so	1,136	AndEngine	0.75%
libVuforia.so	1,067	Vuforia Engine	0.71%

Table 6: Detailed Information about Shared Objects in Mobile Games with the Estimated Game Engine Market Share.

A Measuring the Popularity of Game Engines

Today, there are many mobile game engines, such as Unity, Unreal Engine and Cocos2D. However, it is unclear how popular mobile game engines really are among the mobile games. To answer this question, we have thus performed a measurement study with 293,019 mobile games crawled from Google Play. Our key insight is that the shared objects in the games that developed with the same engine should have the same names. For instance, the games developed with Unity should have shared objects `libmain.so`; the games developed with Unreal Engine should have shared objects `libUE4.so`.

As such, we first unpacked each game and collected its shared objects. For each shared object name, we count the number of games that contains it. Then we focus on the shared objects that appeared in more than 1,000 games as presented in the first column of Table 6. However, for many shared objects, it is hard to tell which game engines they belong to based on their names. So we performed a manual investigation to find out such information with two strategies.

- **Reverse engineering of popular game engine based on the available games (bottom-up).** For each popular game engine, we downloaded 5 games that have developed with it. Through reverse engineering, we find out the names of shared objects they should have in the released games. In addition, by looking at the strings or the exported method names of the shared objects, we find clues about the game engine such as the engine name.

- **Search engine, such as `Google` (top-down).** A game engine typically has a forum, while developers discussing bugs, they may post logs (may contain shared object names). And the search engine will collect such information. So searching the shared object name may lead us to the forum. And eventually help us identify the game engine.

With our manual investigation, we find out that a shared object can fall into two category: (1) a game engine, (2)a general library (*e.g.*, `libc++`, advertisement library). The detailed results are presented in Table 6. From this information, we have identified 12 popular game engines. For each game engine, we use the appearance number of its highest shared object as its market share. Then we created a pie chart for the market share as shown in Figure 1. It is clear that Unity Engine is holding the dominant market position.

Note that, not all the games are developed with games engines. We found that 116,084 (39.62%) games that do not contain shared objects. Those games are normally simple games (*e.g.*, puzzles such as `com.puzzle.mathpuzzle`) that do not need fancy UI, and they are developed in pure `Java`. In addition, the binaries of some games are missing in AndroZoo due to Multiple APK feature as mentioned in §6.1.