Swords and Shields – A Study of Mobile Game Hacks and Existing Defenses

Yuan Tian
Carnegie Mellon University
yuan.tian@sv.cmu.edu

Eric Chen
Gridspace
eric.chen@sv.cmu.edu

Xiaojun Ma
Carnegie Mellon University
xiaojun.ma@sv.cmu.edu

Shuo Chen
Microsoft Research
shuochen@microsoft.com

Xiao Wang
Carnegie Mellon University
sean.wang@sv.cmu.edu

Patrick Tague
Carnegie Mellon University
tague@cmu.edu

ABSTRACT

The mobile game industry has been growing significantly. Mobile games are increasingly including abilities to purchase in-game objects with real currency, share achievements and updates with friends, and post high scores to global leader boards. Because of these abilities, there are new financial and social incentives for gamers to cheat. Developers and researchers have tried to apply various protection mechanisms in games, but the degrees of effectiveness vary considerably. There has not been a real-world study in this problem space. In this work, we investigate different protections in real-world applications, and we compare these approaches from different aspects such as security and deployment efforts systematically.

We first investigate 100 popular mobile games in order to understand how developers adopt these protection mechanisms, including those for protecting memory, local files, and network traffic, for obfuscating source code, and for maintaining the integrity of the game state. We have confirmed that 77 out of the 100 games can be successfully attacked, and believe that at least five more are vulnerable. Based on this first-hand experience, we propose an evaluation framework for the security of mobile game defenses. We define a five-level hierarchy to rate the protection mechanisms to help developers understand how well their games are protected relative to others in the market. Additionally, our study points out the trade-offs between security and network limitations for mobile games and suggests potential research directions. We also give a set of actionable recommendations about how developers should consider the cost and effectiveness when adopting these protection mechanisms.

1. INTRODUCTION

The mobile game industry has been booming in recent years. In 2015, mobile games accounted for 41% of the entire video game market [31]. The overall revenue of mobile games in 2015 has reached $34.8 billion (a 39.2% increase from 2014), which is 85% of mobile app market revenue [11]. Popular games can be highly profitable; for example, Clash of Clans profits $4.6 million every day [24]. It is common for mobile games to sell game points, special powers and other digital commodities for real money. Moreover, as games become more social, gamers have the motivation to compete with friends and show off their feats on leader boards. Therefore, protecting against game hacking has become an important consideration for developers. For example, Pokémon Go starts to take actions to ban users that take unfair advantage of and abusing the game. Indeed, as we show in this paper, developers have tried to apply various protection mechanisms in their games, achieving different levels of success against hacks. To the best of our knowledge, there is no systematic study about the current conditions of mobile game security. We believe that it is timely and valuable to conduct a broad study in this problem space, specifically, about mobile game hacks and protection mechanisms.

Compared to PC games, mobile games have unique limitations with respect to security protections. First, most mobile games have less frequent network communications because of limited bandwidth and the cost of data traffic. They often only communicate with servers in specific situations such as in-app purchase, leader board loading, and communication with friends. Second, the "barrier to entry" of mobile game developers is significantly lower than that of PC games. In particular, most PC games are released by large game development companies that adhere to stricter coding and security practices, whereas mobile games are often developed by small companies or even individual developers with fewer restrictions and less secure programming experience. Motivated by these differences, we approach the problem by *surveying and analyzing mobile game weaknesses leading to successful hacks, cheating, and manipulation of game activities.* We summarize our efforts as follow.

Our Work. Our study consists the following three pillars.

- We present a comprehensive view of the current landscape of mobile game hacks and protections.
- We compare the defenses deployed in the real world. The comparison focuses on multiple dimensions such as security, deployment effort, bandwidth limitation and performance.
- We identify and discuss current research challenges and point out future directions.

First, we study 100 popular mobile games, focusing on

ACSAC '16, December 05-09, 2016, Los Angeles, CA, USA

DOI: http://dx.doi.org/10.1145/2991079.2991119

their resilience against hacking techniques. Then, we discuss various hacking utility tools such as memory/file editing, traffic analysis and program analysis tools. We also investigate many protection approaches, such as local resource protection and network protection, as well as more sophisticated approaches such as code obfuscation and state-synchronization between client and server. We present a number of case studies to concretely explain how developers try to protect their games and whether the adopted protections are effective. We have confirmed that 77 of 100 popular games (such as *Angry Birds*) can be successfully hacked, and believe that at least five more are vulnerable.

Our first-hand experience suggests that game hacking requires very different levels of effort, ranging from using automatic tools to painstakingly analyzing native code libraries. This implies that certain protections and combinations of protections are very effective, while others can be trivially defeated. Accordingly, we define a five-level hierarchy to rate the effectiveness of each combination of protections that we observed. We envision that this hierarchy gives developers a clear picture about how well their games are protected relative to others in the market. We also discuss the developer's effort and the runtime overhead due to each protection approach, which are important practical considerations when making real-world engineering decisions. The feasibility of a protection mechanism is also affected by the game's genre and its development platform. Summarize all the analyses, we provide a number of actionable recommendations for developers who build different genres of games and who use popular game engines/development platforms (e.g., Android SDK/NDK, Unity3D, libGDX, Adobe Air, and AndEngine).

The remainder of our paper is organized as follows. In Section 2, we introduce common hacking tools and techniques for traffic and code analyses. We describe the threat model and the approach of our study in Section 3. We present an overview of our study results in Section 4. In Section 5, we provide a detailed account of several case studies. Based on our experience obtained through the study, we discuss and compare different protection mechanisms in terms of cost and effectiveness in Section 6. We discuss related research in Section 7 and conclude in Section 8.

2. MOBILE GAME HACKS BACKGROUND

In this section, we provide an overview about a number of mobile game hacking tools and techniques used in our study. We use these tools in our study to test the games and evaluate the effectiveness of the game protections.

2.1 Hacking Tools

There are two primary types of hacking tools that apply to mobile games: (1) tools that can be applied broadly across games and platforms and (2) tools that are game-specific. In what follows, we describe these two classes of hacking tools.

2.1.1 General Hacking Tools

General hacking tools are usually apps that are designed to hack mobile games on gamers' devices. The hacking approaches are not designed to be game-specific. We categorize them into two classes: memory editing tools and local file editing tools. Memory editing tools, such as GameKiller [16], GameCIH [14], and GameGuardian [15], search for the addresses of sensitive variables and modify them during game play. More specifically, the gamer provides the value of a sensitive variable, e.g., the number of coins, into such a tool to locate all candidate addresses in memory that contains the value. The gamer then continues the game for a while, and repeats the aforementioned steps. Usually, after several iterations, the tool is able to identify the memory address of the sensitive variable. Thus the gamer can modify it arbitrarily by editing the value in the memory address. Another class of tools is local file editing tools, such as CheatDroid [13], which allow a gamer to check the local files for sensitive values.

2.1.2 Specific Hacking Tools

Different from the general tools, there are hacking tools designed for specific games. One example is Xmodgames [35] that provides "mods" to 32 popular games. These mods are repackaged mobile games with protections removed. Therefore, gamers can install these repackaged games on their devices and enjoy the benefits that they do not have otherwise. The mods require detailed analyses done by skillful hackers and need to be updated when the games are updated.

2.2 Analysis Techniques

Techniques such as traffic analysis and program analysis can also be used to construct game hacks. Using these techniques requires deeper knowledge about technical details and configurations.

2.2.1 Traffic Analysis

Traffic analysis is useful for hacking games that send sensitive game status updates such as scores and coins over a network. Attackers can utilize traffic analysis to identify and modify sensitive parameters for in-game benefit. The attack is often done through a network proxy. In practice, there are three situations where a hacker needs extra efforts to do traffic analysis. First, Android apps can bypass the global proxy settings so that the traffic might not be captured by the proxy. The hacker needs to force the game to use the proxy. Second, the traffic may be encoded, so the hacker needs to understand the format well enough to decode it. Third, the game traffic combines advertisement and analytics traffic and the gamer needs to filter the traffic.

2.2.2 Decompilers and Debugging Tools

Decompilers, debugging tools and hooking tools can be applied to understand the logic of the games, for the purpose of bypassing protections. Decompiling tools, such as dex2jar [6], ILSpy [22] and JD-GUI [23], enable the hacker to decompile the code and analyze the logic. Debugging tools such as GDB and hooking tools such as Android_SSL_Trustkiller help the hacker analyze the game's logic at runtime. Attackers might infer the protections deployed in a game by analyzing the game's executable and use the information to bypass the protections.

3. EVALUATION FRAMEWORK FOR MOBILE GAME DEFENSES

We developed a framework to evaluate protections in mobile games from different aspects such as security, deployment effort, bandwidth consumption, and performance.

3.1 Dimensions of Evaluation

We select the following dimensions to evaluate mobile game defenses systematically.

- Security
  When we analyze the security aspect of protections, we take the hacker's perspective to study a set of popular games. For each game, we try to understand what protection mechanisms it deploys, then explore different levels of hacking techniques to evaluate the effectiveness of these mechanisms.
- Deployment effort
  Deployment effort is also very important to evaluate a mobile game defense, because developers tend to adopt approaches that are easy to design and implement. We evaluate this aspect by analyzing whether the defense is specific to the logic of the games, whether developers need to change a lot of the code, and whether the developer need to use new technology/framework.
- Bandwidth consumption
  Mobile games have specific limitations about bandwidth consumption. Most of mobile games are designed to be light-weight for network traffic. We evaluate the network usage of each defense to see whether it is suitable for different types of mobile games.
- Performance
  We also evaluate the performance overhead introduced by each defense technique.

3.2 Threat Model

When comparing the security of the mobile game defenses, we consider the situation where the attacker is the person playing the game, namely the game hacker. The game hacker uses various hacking techniques in an attempt to defeat the protections employed by the game developers. Because the game hacker is also the owner of the mobile device, it is feasible, when necessary, for the gamer to obtain root privilege on the device. Rooting the device allows hacking tools to get access to local resources such as memory, decompiled source code, local files, and network traffic. Our threat model is standard for studying security problems pertaining to game hacks (and other digital content protection issues). We consider two levels of hackers: the first level are amateurs, who only use available hacking tools such as memory modification tools (e.g., GameKiller) and local file modification tools (e.g., cheatDroid), whereas the second level are professional hackers who also conduct program analysis and traffic analysis to hack deeper into the games.

3.3 Approach

For each game, we test it with a set of increasingly sophisticated hacking tools, described as follows.

First, we use the most general hacking tools to modify local files and memory. These tools are easily available to gamers, and they are simple to install and use. We consider these tools as amateur level.

If the general hacking tools are not effective for the game, we check the network traffic to see if the game has weaknesses that can be exploited through traffic manipulations. We use network sniffers and web proxies in this step. These tools are also easily available to gamers, but they require the effort of configurations, as well as certain basic skills for analyzing and understanding the traffic. As mentioned earlier, traffic analysis may not be trivial; a game hacker may need to use tools such as proxydroid [27] to force an Android app to use the global proxy.

If the above hacking techniques are still ineffective, we use more sophisticated tools to analyze the game to study its specific protection mechanisms. We first decompile and analyze the app to investigate its protection logic. Usually, the expertise and effort required by these techniques are beyond what amateur game hackers possess.

If the source code of the game app is heavily obfuscated or written in native code, we apply debugging tools to investigate the logic and use hooking tools to bypass protections. This level of analysis is often time-consuming even for experienced hackers.

Our study does not include game mods for two reasons: (1) the techniques to develop game mods are essentially the same as the ones we studied ; (2) game mods are often only available for a few popular games and not up-to-date.

4. OVERVIEW OF OUR STUDY

In this section, we provide an overview of our study about attacks and defenses in mobile games. We first describe the apps we investigated and then summarize the protections we observed.

4.1 Dataset Overview

Our study covers a set of 100 popular mobile games in Google Play, shown in Table 1 with an index assigned to each game for easy reference (e.g., game 43 is *Angry Birds*). We tried to be unbiased when selecting these games – they are all among the top 120 games in Google Play, from which we removed 20 games that do not present a financial or social incentive to cheat. The games are built with different development platforms such as Unity3D (C#) [33], Android SDK (Java) [18] and NDK (C++) [17], Adobe Air (ActionScript) [1], and cocos2d-x (C++) [10]. Android SDK/NDK, Unity3D, and cocos2d-x are three leading development platforms. The other platforms have fewer than 10 applications in our game set. This set of games also cover most game genres such as action, strategy, and sports [2]. The game developers range from big companies (e.g., King Digital, Zygna, and Tencent to individual developers. In addition, these games exhibit diversified network access patterns.

4.2 Summary of Results

Most mobile games expose two types of resources to hackers: local resources (e.g., memory and local files) and network traffic. Accordingly, developers try to protect these resources using various mechanisms. To protect against malicious modifications of local resources, developers adopt memory protections and file protections. To protect network traffic against modification and injection, techniques such as encoding, encryption and signing are applied to the communication between the mobile device and the game server.

We also observed developers adopting different approaches to protect the logic of their games, such as applying code obfuscation and compiling critical components into native code. These mechanisms try to discourage hackers from understanding how the games behave internally. In addition, some developers implement mechanisms for synchronizing client-side state with servers. Such client-server synchronization ensures that malicious modification of the client-side state cannot persist, instead reverting back to the server-side value during subsequent synchronization.

In the following sections, we explain the aforementioned mechanisms: *local resource protection*, *network protection*, *code obfuscation*, *native code implementation*, and *client-server sync*.

Index	Game	Index	Game	Index	Game	Index	Game	Index	Game
1	Subway Surf	2	AA	3	Temple Run 2	4	Crossy Road	5	Trivia Crack
6	Agent Alice	7	ZigZag	8	Clash of Clans	9	Coin Dozer	10	Mary Knots: Garden
11	Kill Shot	12	Witchy World	13	Minion Rush	14	AdVenture Capitalist	15	94%
16	Video Poker	17	Coin Trip	18	Journey of Magic HD+	19	Castle Clash	20	Prize Claw 2
21	Sniper 3D	22	Tank League	23	Quiz Battle	24	Jelly Jump	25	Empire : Rome Rising
26	Poker Deluxe	27	Clash of Lords 2	28	Shipwrecked: Volcano	29	3 Pyramid Tripeaks	30	Elemental Kingdoms
31	Flow Free	32	Fruit Ninja Free	33	Word Search	34	PAC-MAN	35	MARVEL War of Heroes
36	blood & glory: immortals	37	Fruit Land	38	Hill Climb Racing	39	Solitaire vegas free card game	40	Dumb Ways to Die 2
41	My Talking Angela	42	Legend of EmpireKingdom War	43	Angry Birds	44	Bingo crushfree bingo game	45	Solitaire
46	Farm Heroes Saga	47	My Talking Tom	48	Don't Tap The White Tile	49	Five nights at Freddy's 3 demo	50	TETRIS
51	Marvel Contest of Champions	52	Panda Pop	53	Racing Rivals	54	SimCity BuildIt	55	Bounce
56	Geometry Dash Lite	57	Spring Ninja	58	Hungry Shark Evolution	59	11+	60	Westbound: Gold Rush
61	DEER HUNTER 2014	62	Winter Craft 3: Mine Build	63	Pou	64	Fashion Story: Daring Red	65	Rock Hero
66	Boom Beach	67	Maternity Doctor	68	Cookie Jam	69	Sonic Dash	70	Madden NFL Mobile
71	Bad Piggies	72	Bubble witches saga 2	73	MiniCraft 2	74	Looney Tunes Dash!	75	Candy Crush Soda Saga
76	Candy Crush Saga	77	War of Nations	78	Need A Hero	79	Pharaoh's War	80	Brave Trials
81	Angry Birds Stella POP!	82	Plants vs. Zombies	83	King of Thieves	84	Hay Day	85	Hardest Game Ever 2
86	Dawn of the Dragons	87	Bike Race Free	88	Call of Duty: Heroes	89	Transformers: Battle Tactics	90	Gods Rush
91	Surgery Simulator	92	8 Ball Pool	93	Cinderella Free Fall	94	Army of Toys	95	Pet Rescue Saga
96	Ninja go go go	97	Galaxy Online 3	98	Racing Fever	99	YAHTZEE With Buddies	100	Dragon city

Table 1: We tabulate and index the representative set of games included in our study.

Game Engines / Protections	Unity3D	Android SDK/NDK	Adobe AIR	Cocos2d-x	libGDX	AndEngine
Local resources protection	1 7 14 17 20 24 30 53 58 98	11 13 18 25 34 42 43 46 63 73 76 78 79 81 85	10 37	19 27 31 91	28 60	2
Network protection	1 4 6 7 17 24 30 26 39 51 53 89 93 94 96 98	5 8 11 13 15 25 26 32 33 34 42 43 54 55 64 66 70 74 75 77 78 79 80 81 82 83 92 95 97	12 37 40 86	27 38 44 56 57 59 90 100	28 60 65	2
Code obfuscation	53 88	5 8 11 13 18 23 25 33 35 43 63 66 74 77 83 84 87	12	22 67 90	65	
Compilation to native code		8 11 13 18 23 25 32 34 42 43 46 50 54 55 63 64 66 70 72 73 74 75 76 77 78 79 80 81 82 83 84 85 95 97		19 22 27 31 38 44 48 56 57 59 67 90 91 100		
Client-server sync	30 36 51 89 94	5 8 23 25 35 42 64 66 70 77 80 83 84 97 99		19 27 100	28 60	

Table 2: Summary of protections in mobile games

4.2.1 *Local Resource Protection*

Local resource protection prevents hackers from editing local resources. To protect against memory editing tools (e.g., GameKiller), developers use different approaches such as encrypting the sensitive variables and detecting memory editing operations. To protect against file editing tools (e.g., CheatDroid), developers often encrypt sensitive values before writing them into files. Overall, we observe that 34% of the studied games have local resource protection in place. For games with proper local resource protections, it is hard

to use the general hacking tools to modify memory or local files because these tools can effectively get correct values from the memory. Attackers need to use more advanced analyses such as traffic analysis to modify traffic or decompilation to figure out the protections.

4.2.2 Network Protections

Although mobile games only generate a small amount of network traffic, developers do want to protect the traffic from hackers, because it often carries sensitive data, which, if controlled by the hacker, would void the effort of local resource protection. The most basic protection is using standard HTTPS for network communication, which we observed in 26% of the studied games. While HTTPS provides some protection, it is not effective against an attacker using a web proxy with a fake certificate to decrypt the HTTPS traffic. A more effective approach is to make the traffic obscure for the hacker. For example, some studied games use non-public encoding or encryption, do certificate pinning or maintain a list of approved certificates. There are also games whose traffic cannot be captured by the proxy and games that sign their packets.

4.2.3 Code Obfuscation and Hiding

The goal of code obfuscation and hiding is to increase the barrier for the hacker to reverse-engineer the game's internal logic. Our study shows that 24% of games use code obfuscation or hiding at different sophistication levels, ranging from obfuscating class names and variable games to dynamic library downloading.

4.2.4 Compilation into Native Code

We also observed 48% of studied games contain components that are compiled into native code as opposed to Java byte-code. This approach creates barriers for hackers because decompiling native code is much harder than Java byte-code. Although there are existing decompilers for native code, the decompilation quality is usually far from satisfactory, thus reverse-engineering the logic is still very time-consuming.

4.2.5 Client-Server Sync

The goal of client-server sync is to keep the client-side state in sync with that of the server. We only found 25 games that implement this protection . Client-server sync, if effectively implemented, is considerably secure against most hacks that we are aware of. However, a correct implementation requires an insightful design consideration about how to partition the game logic between the client and the server, so that important computations are not solely performed on the client side. We found several examples of games that attempt client-server sync but are still vulnerable, including *Trivia Crack* and *Dragon City*.

4.3 Real-world deployments of the protections

Table 2 shows the 100 games in a grid, in which each row is one of the five protection mechanisms discussed above, and each column is one of the popular game engines. Note that it is fairly common for a game to deploy multiple protection mechanisms, so an index may appear in multiple cells in a column. From the table, we can see that games developed using different game engines have different tendencies of applying the protection mechanisms. For example, games developed using Android SDK/NDK tend to apply obfuscation more often due to native support in the development environments. Conversely, games developed using Unity3D do not deploy code obfuscation very often, because developers have to implement their own techniques. We also observed that client-server sync is most often applied on multi-player games that have frequent network communications.

The deployments of the protections are also related to the vendors of the games. For example, large vendors tend to adopt better protections. Out of the 100 apps we studied, ten vendors such as EA games, Supercell develop more than one app (32 apps in total). These games from larger vendors are more likely to deploy full client-server sync (34.4%) than games from smaller vendors (10.3%). These larger vendors also tend to use one developing platform over their games. Six companies utilize the same platform for their 17 games. Using the same platform makes it easier for the vendor to share the same protection techniques across games. However, the protections adopted are still different because of other factors. Since the majority of games (68%) are from small vendors that do not have good resources for security development, we envision the analysis of protection techniques and suggestions for best practices would benefit them tremendously.

It is worth noting that even for games adopting the same set of protections, resilience against hacking still varies considerably across games. Understanding how effectively the games implement their protections requires a much deeper investigation about individual games. In the next section, we present several representative cases, including games that are trivial to hack and ones that are much more secure.

5. CASE STUDIES OF REAL-WORLD PROTECTIONS IN MOBILE GAMES

In order to obtain a deeper understanding about how effectively developers implement protection mechanisms, we conducted a number of case studies, in which we took the game hacker's perspective and used hacking tools with different levels of sophistication. Our analysis uses the categorization introduced in Section 4.

5.1 Local Resources Protection

We observed different approaches for local resource protections. Most of them aim at protecting the memory, but there are a few approaches for local file protection. Examples of local file protection techniques include encrypting or encoding local files. Note that it only makes sense to protect both local files and memory, because protecting one without the other would be too trivial to break. Unfortunately, this mistake was made by a number of games. For example, *Jelly Jump* and *94%* only protect memory, while *AdVenture Capitalist* and *ZigZag* only protect local files.

5.1.1 Basic Memory Protection

We observed several games that encrypt sensitive values before storing them . An example is *Subway Surfers*, a popular game with over 100,000,000 downloads. In the game, the main character runs on railways to collect coins while avoiding incoming trains. Gamers can buy equipments with coins, which are, therefore, an important resource that the game developers want to protect. We used general memory editing tools attempting to modify the number of coins but

```
public int amountOfCoins
{
  get
  {
    return Utils.XORValue(this._xoredAmount);
  }
  set
  {
    int num = Utils.XORValue(this._xoredAmount);
    if (num != value)
    {
      this._xoredAmount= Utils.XORValue(value);
      Action action = this.onCoinsChanged;
    }
  }
}
public static int XORValue(int value)
{
  return value ^ Utils.GetXorRandomValue();
}
```

Figure 1: *Subway surfers* computes the XOR of the coin number with a random number when updating the coin number (code is simplified for display purpose).

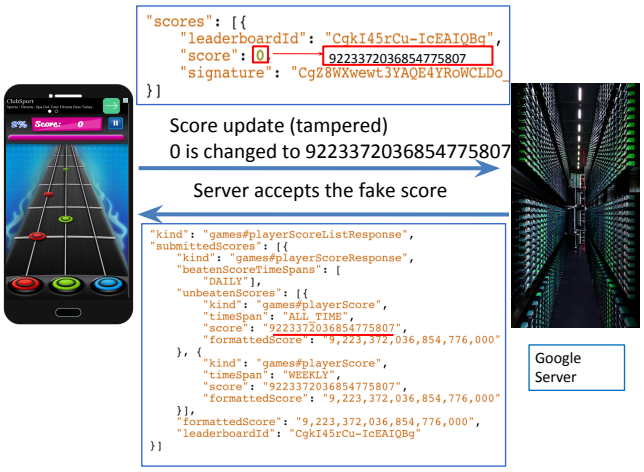

Figure 2: Modifying traffic in *Rock Hero* to rank high in the leader board.

did not succeed. We then decompiled the game with ILSpy to analyze the coin counting logic, and we eventually realized that the developers XOR the coin number with a random number so that a hacker cannot simply search for the coin value in memory. The details are shown in Figure 1.

5.1.2 Local File Protection

We also used decompliers such as dex2jar, ILSpy, and JD-GUI to identify games that protect sensitive values in local files . *ZigZag* is a game in which the gamer tries to move balls forward and still keeps balls on the pathway by changing the directions of balls. It enables in-app purchases for more balls. *ZigZag* stores level number in SharedReferences (a common local file for storing values), but we cannot modify it directly. By studying its decompiled code, we found that *ZigZag* used `CryptoPlayerPrefs` (XOR or Rijndael) before saving values to SharedPreferences. We decompile the app and track the function for storing values locally to identify the encryption key. Then we realized that the key is hashed from a variable name. Even without the encryption key, a game hacker can still modify scores because *ZigZag* does not implement memory protection for updating two sensitive variables `GameController.score` and `GameController.bestScore`. Attackers can just search the score number in the memory and modify the score.

5.2 Network Protections

A majority of mobile games allow the gamer to play mostly offline and only communicate with servers in specific situations such as updating high scores, purchasing equipment, sending awards, and socializing with friends. These operations are essential to the games, so the traffic should be well protected. We observed different approaches of protecting traffic with different levels of effectiveness. Following are our case studies to explain what developers implement in the real world.

5.2.1 Basic HTTPS

The first level of network protection is using HTTPS alone. It is not effective against game hacks, because the hacker can use a standard HTTPS proxy to decrypt the traffic. Note that the attacker would intentionally ignore and bypass any certificate errors.

The game *Rock Heroes* uses Google's leader board – it sends the high score to Google's server to update the leader board using HTTPS. However, the gamer can use a proxy to modify the traffic and set an arbitrary high score on the leader board, as shown in Figure 2. Although Google claims that they have protections for fake high score [20], we did not observe the effects of such protections. Instead, we were able to confirm at least three gamers who are hackers in the leader board, as their scores are impossibly high. According to our analysis, the variable for the in-game score is of type `Int32`. However, Google's `submitScore` API for their leader board takes a score of type `Int64`. We have confirmed that the traffic was modified so that arbitrary scores under $2^{63}-1$ can be successfully set. Most notably, and somewhat to our surprise, the leader board already showed three other instances of high scores equal to $2^{63}-1$.

5.2.2 HTTPS with Additional Protections

A few game developers attempt to place additional barriers so that gamers cannot use a proxy to hack them easily. These techniques include certificate pinning to block unauthorized certificates and encrypting inside HTTPS payload to block hackers from decrypting their HTTPS traffic. Certain games, such as *Game of War*, have their predefined valid certificate list hardcoded in the apk. If the signer of a server certificate is not in this list, the game will refuse to communicate. This approach can be somewhat effective, because it forces the hacker to investigate how to bypass the certificate verification. However, a knowledgeable and more persistent hacker can still succeed. For example, we were able to hook and override certain Java and Apache SSL libraries to bypass these certificate verifications.

5.2.3 Message Signing

Besides using HTTPS (or SSL), some games sign their messages in order to protect the integrity and authenticity of the communication. The effectiveness depends on how well the signing key is protected. Typically for a multi-

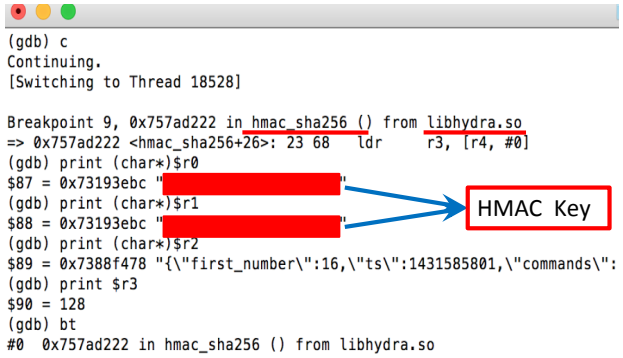

Figure 3: Debugging *Dragon City* to identify the HMAC key.

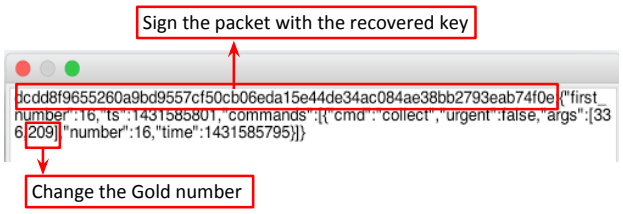

Figure 4: We change the gold quantity and sign the modified packet with the recovered key in *Dragon City*.

player battle game such as *Dragon City*, an update from the client to the server contains only the start state of the battle state and the end result. The update message is protected by an HMAC, such as SHA-256 HMAC. In other words, the gold collecting result or battle result is computed exclusively on the client side, so the server has no way to detect a hack if the signature is faked. We decompiled the code by dex2jar, but found that the important logic was inside the native libraries. We tried to debug the native libraries as follows. We first used the command `'ndk-which nm' -D -demangle libgame.so` to list the memory addresses of functions, then we identified the function for computing packet signatures. Among the four libraries (`libcocs2d.so`, `libdc.so`, `libgnustl_shared.so` and `libhydra.so`), we found the `hmac_sha256` function inside `libhydra.so` to be a strong candidate. We verified our estimation by setting a stop point at `hmac_sha256` and recovered the parameters, as shown in Figure 3. R0 and R1 are the static HMAC keys, R2 is the request content, and R3 is the length of the packet. After recovering the key, we tried to modify the coin number from 209 to 500 and signed the modified packet with the HMAC key. As the signature was valid, the server sent a success response and we updated a fake coin number. Similarly, we can modify the game result with the recovered key, as is shown in Figure 4.

5.2.4 *Communicating using Customized Protocol*

Besides HTTP/HTTPS, some apps also use customized protocol to send data. Such traffic is harder to analyze than that HTTP/HTTPS traffic because it is more flexible; for example, message format and encoding can be defined by the game developer. Moreover, games that send customized encoded traffic usually use client-server sync as well, which we will discuss in Section 5.5.

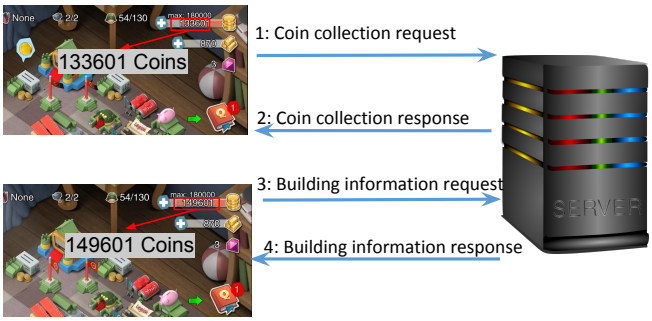

Figure 5: *Army of Toys* exchanges messages between server and client to update the coin collection event. Developers compute how many coins the gamer can collect in the server and sends to the client to update the coins. Client also updates building information from the server.

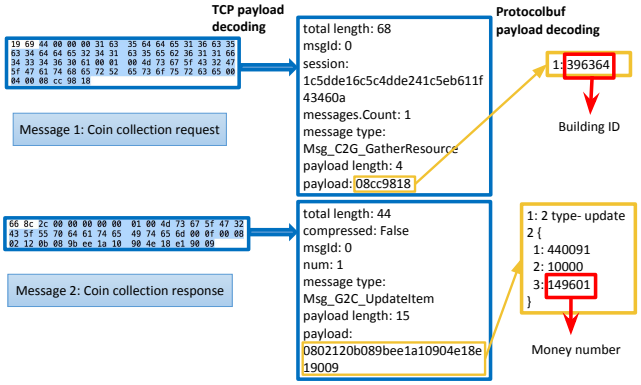

Figure 6: Decoding traffic of *Army of Toys*: message 1 and 2. We analyzed the decompiled code to figure out the encoding protocols of two layers. Here we zoom in the message 1 and 2 in Figure 5 to explain how the server maintains the coins the gamer can collect and how the server informs the client about the coins update.

As a concrete example, we describe our observations about *Army of Toys*, which is a multi-player strategy game. The game uses a special message format, and we managed to reverse engineer its encoding protocols by decompiling the code. It took us a considerable amount of effort to understand the message format of their communication. As shown in Figure 6, we observed from the decompiled source code that *Army of Toys* uses two layers of encoding. We first decode the TCP payload (the left textbox). According to the decoding protocol we found from the decompiled code, we obtain the content in the middle textbox. We see that even this decoded content still constains an encoded payload. After further investigation, we found that the second encoding is in `protobuf` format [21], and we were able to further decode the payload to obtain the content in the right textbox. This multi-step analysis and decoding process allows us to understand the traffic, but with significant effort.

5.3 Code Obfuscation and Hiding

Code obfuscation and hiding are commonly used by games

developed with the Google SDK, which provides obfuscation functionality. As we discussed earlier, our hacking effort succeeded in many of the studied cases because we were able to figure out the program logic. If done effectively, code obfuscation and hiding can be a sufficient (though not perfect) deterrent to game hackers who try to reverse engineer the game.

5.3.1 Code Obfuscation

Kill Shot is a shooting game developed using Android SDK and NDK. We did not succeed in hacking the game by local resource editing or traffic analysis, so we attempted to decompile the code. We found that class names and variable names of the program are obfuscated, which makes it very difficult to figure out how to take control of sensitive data that attackers want to manipulate.

5.3.2 Code Hiding

A number of games try to hide the existence of important code components, rather than obfuscate them. For example, *Army of Toys* mentioned earlier does not load certain important libraries when the game starts. Instead, it downloads these compressed libraries at certain points during the game.

5.4 Compilation into Native Code

Implementing important game logic in native code is another approach to increase the difficulty of reverse engineering. Once compiled into native code, a program has lost most compile-time information, such as variable names and type information. Moreover, native code instructions are much harder to be reverted to their source code statements. If compiler optimizations are applied, then the code is even harder to understand. For this reason, developers can take advantage of this feature of native code compilation to defend against game hacking attempts. This approach can be further combined with code obfuscation techniques.

For example, the game *Angry Birds* hides the local storage protection inside a native code library `libAngryBirdsClassic.so`. We did not get sufficient information from decompilation to recover the logic. However, we found an encrypted file named `highscores.lua` and used a debugging tool to analyze it. We recovered the symbol table for function names and then tried to use the GDB debugger to follow the interesting functions for the game logic. *Angry Birds* is time-consuming to analyze because the game has many functions related to security. We had to debug many encryption functions to get their parameters and test if each parameter is the right key for decrypting `highscore.lua`. After much effort, we were able to get the right key from the function `AES::StartEncryption`. This step is shown in Figure 7. Using the key, we were able to decrypt, modify, and re-encrypt the `highscore.lua` file, which resulted in a modified in-game score.

5.5 Client-Server Sync

A small number of games implement client-server sync. The effectiveness of protection comes from the fact that the computation is performed on the server-side, and the client is forced to synchronize state with the server. We explained in Section 5.2.4 that the client-server sync can be effective if the client side code is basically a "renderer", whereas all the game states are computed and maintained by the server.

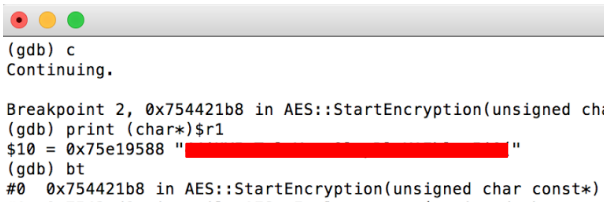

Figure 7: Debugging *Angry Birds* for recovering the keys. We try many functions for encryptions to recover the key, and finally get it in `AES:StartEncryption` from `libAngryBirdsClassic.so`. The figure is a screenshot when the game hits the breakpoint in `AES:StartEncryption` and R1 stores the key as is shown in the red box.

Figure 8: Developers of *Trivia Crack* try to maintain player status on the server side and sync with client side, but they fail because they still trust the client to compute important logic such as whether gamers have enough coins or when the gamers use power-ups. Therefore, we can modify message 2 to increase coins and use the coins locally, also tamper message 3 not to report using powder-ups.

However, some games still depend on the client side to do sensitive computations, such as those about coins and battle results. These games are vulnerable to hacks. Below are the details about some of these games.

5.5.1 Partial Client-Server Sync

Trivia Crack is a game in which a gamer competes with friends in answering multiple-choice questions. The gamer can purchase coins using in-app purchases and use "power-ups" to eliminate some wrong options. We use traffic analysis to get a rough understanding of the program's logic. When one gamer answers a question, *Trivia Crack* sends a message to its server, including question id, the chosen option, and the power-ups that the gamer uses for this question; the process is shown in Figure 8. *Trivia crack* computes the score on the server side according to the message. A direct memory modification would fail to change the score because the authoritative copy of the score is not in the client's memory. However, *Trivia Crack* still depends on the client to do important computations such as deciding whether the gamer has enough coins to use power-ups and whether power-ups have been used to remove incorrect options. Therefore, the server to client message can be mod-

ified to trick the client to believe that the gamer has more coins (message 2 in Figure 8). Similarly, the client-to-server message can be modified to persuade the server that the gamer did not use power-ups (message 3 in Figure 8). There are several other weaknesses of this nature in *Trivia Crack*. For example, it judges whether the gamer is right on the client side, so a hacker can just switch the device to the offline mode after the question is loaded and answer it many times until he gets the correct answer. Then, he can switch the device back to the online mode to update the result.

5.5.2 *Full Client-Server Sync*

Unlike the partial client-server sync, if the game does not depend on the client side to do any important computation, the program will be very robust against hacks. We only see a few examples of such games, all in the multiplayer game category. *Army of Toys* uses encoded traffic as a protection, as we discussed previously. It also utilizes encoded communication for synchronizing client states with the server. All the game logic is maintained on the server side, which only exposes the display interfaces to the client-side code. Every action the gamer performs triggers a number of messages exchanged between the server and the client. For example, as shown in Figure 5, when a gamer collects coins from a factory, the client sends a packet "Msg_C2G_GatherResource" to inform the server that the gamer wants to collect resources from the factory building. Server replies with "Msg_G2C_UpdateItem" to update the coins that the gamer should be able to collect. The server also sends building information to control the display of buildings, such as what time will the gamer be able to collect coins again, in packet "Msg_G2C_AskBuildingInfo". Because the server performs all the sensitive logic computations, gamers cannot hack the client or traffic to fool the server. Although we spent efforts to decode two layers of payload and understand every fields in its traffic (see Figure 6), our efforts to hack this game were unsuccessful.

6. SUGGESTIONS TO MOBILE GAME DEVELOPERS

The previous sections focused on understanding various protection mechanisms. Following the understanding, a practical question to ask is how game developers should consider adopting these protections. The consideration is a judgement about the cost of each mechanism and the protection strength.

6.1 Cost of Protection

The cost of each protection mechanism includes the developer's effort and the runtime cost, which are the two dimensions shown in Figure 9. Based on our studied cases, we place the protection mechanisms in this space.

Local resource protection is easy to implement because it does not affect the game's logic. A developer just decides which sensitive variables need protection, and applies encryption or encoding on them. The runtime cost is also low.

Code obfuscation is easy on Android SDK/NDK – the developers only need to configure the project to enable ProGuard [19]. However, if the platform does not support obfuscation, developers' effort will be significantly higher. Similarly, the developer's effort for native code compilation also

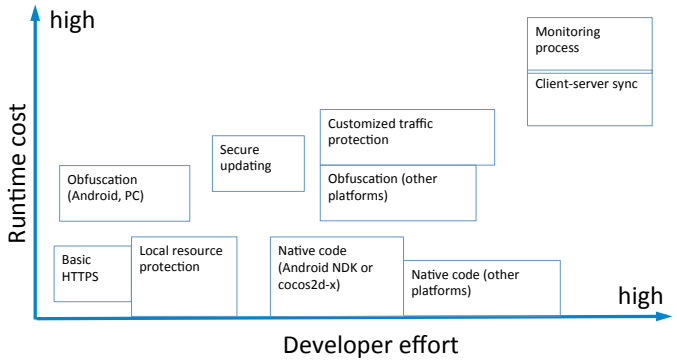

Figure 9: Protection cost comparison.

depends on the platform support. It is easier for games developed using Android SDK/NDK or cocos2d-x(a game development engine in C++) to include native code libraries than those using other platforms. Usually, compilation to native code does not incur much runtime cost. In fact, the code might run even slightly faster as a result.

Traffic protection can be an easy job if it uses standard protocols, such as basic HTTPS. However, developing a customized traffic protection requires much more effort.

Client-server sync is most difficult because it is game specific. Developers need to consider carefully about the game logic so that they can make sure all important logic are computed correctly in the server side. The runtime cost is also high, because it incurs a large amount of traffic for synchronizing the game states. For light-traffic games, this protection is not a reasonable choice. Only multi-player games that already have frequent network communications are suitable for client-server sync.

6.2 Strength of Protection

Software development involves many aspects of consideration, such as the go-to-market schedule, the available manpower, and the budget constraints. We believe that mobile game development is no exception. Every game is developed under a specific set of time pressures and financial constraints. It would be biased for us, as security researchers, to suggest that protecting against game-hacks is of paramount importance over all other considerations. Nevertheless, the knowledge obtained through our study can help developers make more informed decisions about which mechanisms to adopt.

Specifically, Table 3 shows all combinations of protections that we observe in the studied games. We rank their protection strengths using a 5-level rating. Level 1 is weakest. The hacker can use general tools to hack the game. Level 2 can resists the general tools, but is vulnerable to traffic analysis. Level 3 forces the hacker to do decompilation to understand the game's logic. Level 4 requires even manual debugging effort. Level 5 is assigned to those that we have not found vulnerabilities in the principles of their protection approaches. Levels 1 - 4 consist of sub-levels indicated by letters in an ascending order of strength. The right-most two columns show the number of games in each sub-level and how many of them we have successfully hacked. In total, we have succeeded in 77 cases, including every game below the 4B rating and 7 games with 4B - 4D ratings. There are 18 games that we do not think vulnerable (i.e., with rating 5)in

Protection / Level	Local resource protection	Code obfuscation	Compilation into native code	Network protection	Client-server sync	No. of games	No. of games hacked
1A	NA	*	*	*	NA	48	48
1B	Partial	*	*	*	NA	4	4
2A	Full	*	*	NA	NA	9	9
2B	Full	*	*	HTTPS	NA	3	3
2C	NA	*	*	NA	Partial	1	1
3A	Full	NA	NA	Customized	NA	3	3
3B	NA	NA	NA	Customized	Partial	1	1
4A	Full	Yes	NA	Customized	NA	1	1
4B	Full	NA	Yes	Customized	NA	5	1
4C	Full	Yes	Yes	Customized	NA	2	1
4D	NA	+	+	Customized	Partial	5	5
5	*	*	*	*	Full	18	0

Table 3: Levels of protection strength. "Partial" in the local resources protection column means that developers protect memory or local resources, but not both. "*" means that the protection does not matter at this level. "+" means that the game either has code obfuscation or compilation into native code. "Partial" in the client-server sync column means that developers partially rely on the client to compute sensitive logic.

our current threat model. The practical value of this table is that it gives developers an estimate about how well their games are protected relative to others in the market. For example, if a game implements local resource protection and customized traffic protection, and its important libraries are compiled into native code, then the game has a 4B rating, which means that its developer probably has done a better job than 70% of others.

A game's genre and its development platform should also be taken into consideration. For a multi-player game, we suggest implementing client-server sync, because it gives the strongest protection, and the traffic overhead is reasonable given that the game has already generating heavy traffic. For a game of another genre, its development platform becomes the main consideration. Generally speaking, we suggest a combination of local resource protection, network protection, and code obfuscation/native code implementation, if the platform provides support of them. For example, if a game is developed on Android SDK/NDK, all these mechanisms can be implemented without significant effort. Among other development platforms, we suggest local resource protection, network protection, and native code for cocos2d-x because it is a C++ game engine. For the other development platforms covered in our study (Unity3D, libGDX, Adobe Air, and AndEngine), it is reasonable to use a combination of local resource protection, network protection, and code obfuscation, because it would require significant developer's effort to build native code libraries, and frequently invocations of native code functions from high-level languages of these frameworks, such as Java, or C#, or ActionScript, would incur high runtime overhead. We also suggest developers to consider strengthening their protections by deploying per-device keys and periodically updating keys, which would make it difficult for professional hackers to share their results with other gamers.

7. RELATED WORK

Attacks and Defenses in Games. About online multiplayer PC games, researchers have investigated specific threats and categorized different attacks. Specific threats have been discussed in [26, 7] about state exposure and general map hacking techniques by analyzing the memory respectively. Yan et al. provide an overview of different cheating methods and the security problems behind these methods [37]. Another review paper [34] covers known attacks and gives a few real world examples in multi-player games. The authors focused on comparing the attacks and defenses in Client/Server architectures and P2P architectures of online mobile games, while our study has a much wider coverage about different types of games and also we have first-hand experiences about the efforts of hackers to break these games.

Researchers have proposed defenses against these attacks. Most of these proposals design protocols to sync client-server states and verify game status on the server side. For example, Baughman et al. propose a protocol for cheat proof for online games [3]. Researchers also implement approaches to verify client behavior observed in the server side to figure out abnormal behaviors [4, 9, 25, 32]. Minimizing information disclosed to the client is discussed in [28, 36].

A few client-side protection approaches are proposed in the research literature: Monch et al. try to protect games from attacks by creating a trustworthy client [29]. Their approach is to check the integrity of the client, mask sensitive functions, and update these protections overtime. Researchers propose the use of trust computing platforms as hosts for game consoles [2]. Comparing with these papers, we study the defenses in real mobile games systematically and provide practical suggestions.

Security Issues Due to Untrusted Client. The problem that we discuss in this paper about mobile games is a special and critical case of a more general security problem in many client-server systems. For example, web developers often make mistakes by placing some sensitive data in browser cookies or doing some critical computations in client-side javascript. Bisht et al. described several such logic bugs in real-world websites [5] and proposed a client-side detection approach to find these bugs. Felmetsger et al. also studied this type of vulnerabilities, but their proposed approach is based on program analysis of web application (i.e., server-side) [12]. Related work can also be extended

to studies about protocol implementation bugs. For example, Chen et al. described many incorrect implementations of the OAuth protocol, which allow an untrusted client to victim users' accounts without knowing the passwords [8]. Researchers also describe an attack against Google's In-App purchase service [30].

8. CONCLUSIONS

Our study tries to understand the effectiveness of existing defense techniques against mobile-game hacking. These techniques are designed to protect various elements in mobile games, including memory, local files, network traffic, source code, and game states. The result of our study suggests that many developers (over 50% in our dataset) have attempted to implement some of these protections, which is encouraging. However, the effectiveness varies. Some protections can be trivially defeated by automatic tools, some others need significant manual effort to bypass, and some are perhaps not vulnerable in the principles of their approaches. We provide a reference framework to help developers understand how effective their implementations are relative to others. Besides the effectiveness, an important consideration is cost, including developer's effort and runtime overhead. Many protection mechanisms need developers to make a trade-off between effectiveness and cost. For example, client-server sync is a very effective protection, but the cost is so high that it is not suitable for most game genres; basic HTTPS traffic protection is very inexpensive, but it is trivially breakable. We did observe that platform support is an important factor. It makes developers more willing to adopt these protection mechanisms, and presents a better value proposition in terms of cost-effectiveness.

9. ACKNOWLEDGEMENTS

We thank Harshit Agarwal, Sohil Habib, Kenny Sung, Xiaofeng Wang, and anonymous reviewers for valuable comments.

10. REFERENCES

[1] Adobe. Adobe air. http://www.adobe.com/products/air.html.

[2] S. Balfe and A. Mohammed. Final fantasy–securing on-line gaming with trusted computing. In *Autonomic and Trusted Computing*, pages 123–134. Springer, 2007.

[3] N. E. Baughman and B. N. Levine. Cheat-proof playout for centralized and distributed online games. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 104–113. IEEE, 2001.

[4] 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)*, 14(4):32, 2011.

[5] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. Venkatakrishnan. Notamper: Automatically detecting parameter tampering vulnerabilities in web applications. In *ACM Conf. on Computer and Communications Security*, 2010.

[6] Bob Pan. Dex2jar. https://github.com/pxb1988/dex2jar.

[7] E. Bursztein, M. Hamburg, J. Lagarenne, and D. Boneh. Openconflict: Preventing real time map hacks in online games. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 506–520. IEEE, 2011.

[8] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague. Oauth demystified for mobile application developers. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 892–903. ACM, 2014.

[9] R. A. Cochran and M. K. Reiter. Toward online verification of client behavior in distributed applications. In *NDSS*, 2013.

[10] Cocos2d-x. Cocos2d-x. http://www.cocos2d-x.org/.

[11] Dean Takahashi. Mobile games hit 34.8B in 2015. http://venturebeat.com/2016/02/10/mobile-games-hit-34-8b-in-2015-taking-85-of-all-app-revenues/.

[12] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security Symposium*, pages 143–160, 2010.

[13] FLX App. Cheatdroid. https://play.google.com/store/apps/details?id=com.felixheller.sharedprefseditor.

[14] GameCIH. Gamecih. www.cih.com.tw/gamecih.html.

[15] GameGuardian. Gameguardian. http://gameguardian.net/forum/.

[16] GameKiller. Game killer. http://game-killer.com/.

[17] Google Inc. Android ndk. https://developer.android.com/tools/sdk/ndk/index.html.

[18] Google Inc. Android sdk. https://developer.android.com/sdk/index.html.

[19] Google Inc. Hiding leaderboard scores. https://developers.google.com/games/services/common/concepts/leaderboards#hiding_leaderboard_scores.

[20] Google Inc. Proguard. http://developer.android.com/tools/help/proguard.html.

[21] Google Inc. Protobuf. https://github.com/google/protobuf.

[22] ICSharpCode. Dex2jar. http://ilspy.net/.

[23] Java Decompiler. Jd-gui. http://jd.benow.ca/.

[24] Joshua Brustein. Finland's new tech power: Game maker supercell. http://goo.gl/9woZTj.

[25] P. Laurens, R. F. Paige, P. J. Brooke, and H. Chivers. A novel approach to the detection of cheating in multiplayer online games. In *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, pages 97–106. IEEE, 2007.

[26] K. Li, S. Ding, D. McCreary, and S. Webb. Analysis of state exposure control to prevent cheating in online games. In *Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*, pages 140–145. ACM, 2004.

[27] Max Lv. Proxydroid. https://github.com/madeye/proxydroid.

[28] S. Moffatt, A. Dua, and W.-c. Feng. Spotcheck: an efficient defense against information exposure cheats. In *Proceedings of the 10th Annual Workshop on Network and Systems Support for Games*, page 8. IEEE Press, 2011.

[29] C. Mönch, G. Grimen, and R. Midtstraum. Protecting online games against cheating. In *Proceedings of 5th*

ACM SIGCOMM workshop on Network and system support for games, page 20. ACM, 2006.

[30] 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*, pages 459–470. ACM, 2014.

[31] SuperData Research. Worldwide digital games market. https://www.superdataresearch.com/blog/us-digital-games-market/.

[32] H. Tian, P. J. Brooke, and A.-G. Bosser. Behaviour-based cheat detection in multiplayer games with event-b. In *Integrated Formal Methods*, pages 206–220. Springer, 2012.

[33] Unity3D. Unity3d. https://unity3d.com/.

[34] S. D. Webb and S. Soh. Cheating in networked computer games: a review. In *Proceedings of the 2nd international conference on Digital interactive media in entertainment and arts*, pages 105–112. ACM, 2007.

[35] Xmodgames. Xmodgames. http://www.xmodgames.com/.

[36] A. Yahyavi, K. Huguenin, J. Gascon-Samson, J. Kienzle, and B. Kemme. Watchmen: Scalable cheat-resistant support for distributed multi-player online games. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 134–144. IEEE, 2013.

[37] J. Yan and B. Randell. A systematic classification of cheating in online games. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–9. ACM, 2005.