Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Oskari Teittinen

Analysis of cheat detection and prevention techniques in mobile games

Master's Thesis
Espoo, May 20, 2018

Supervisors: Professor Antti Ylä-Jääski
Advisor: Henrik Lönnroth M.Sc. (Tech.)

Aalto University
School of Science

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author:	Oskari Teittinen		
Title:	Analysis of cheat detection and prevention techniques in mobile games		
Date:	May 20, 2018	**Pages:**	56
Major:	Interactive Technologies	**Code:**	SCI3042
Supervisors:	Professor Antti Ylä-Jääski		
Advisor:	Henrik Lönnroth M.Sc. (Tech.)		

Cheating in video games has always been a problem for the players and the game developers. The problem is not just limited to the open platforms like computer games but also to closed platforms like console and mobile games. In this thesis we examine commonly used cheat protection methods that can be used to protect video games from different type of cheat tools. We focus especially on the methods that are also viable on the mobile platforms.

First, we study client-side protection methods which are directly implemented to the game's client application. These methods include protection against memory editing and tampering the game save files. We also look how code obfuscation can be used to protect against cheaters. Second, we study different type of server-side architectures that be used to prevent cheating in the game. Finally, we present our architecture that is especially designed to be used in mobile games. The architecture takes account the limitations and restrictions of the mobile devices like unreliable mobile networks and limited battery life.

Our analysis concludes that the server-side protection methods are clearly more reliable to prevent cheating in games and they are also much harder to bypass or break. Many of the client-side cheat protection methods are relying on the secure through obscurity which means that they are vulnerable for flaws in the implementation. Also, on the client-side the cheaters can easily gain an access to the game's memory, network traffic and files saved on the device storage. Using server-side protection methods where the game logic is run fully or partially on the server provides the most effective protection against cheaters.

Keywords:	cheating, mobile games, encryption, memory editing, code obfuscation, server-client, server authoritative, client authoritative
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu

Aalto-yliopisto
Perustieteiden korkeakoulu
Tieto-, tietoliikenne- ja informaatiotekniikan maisteriohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Oskari Teittinen		
Työn nimi: Analyysi huijauksen tunnistamis ja esto tekniikoista mobiilipeleissä			
Päiväys:	20. toukokuuta 2018	**Sivumäärä:**	56
Pääaine:	Interaktiiviset Tekniikat	**Koodi:**	SCI3042
Valvojat:	Professori Antti Ylä-Jääski		
Ohjaaja:	Diplomi-insinööri Henrik Lönnroth		

Huijaaminen videopeleissä on aina ollut ongelma sekä pelaajille että kehittäjille. Ongelma ei vain rajoitu avoimiin alustoihin kuten tietokonepeleihin vaan se ulottuu myös konsoli- ja mobiilipeleihin. Tässä diplomityössä tutkimme yleisesti käytettyjä huijauksenesto menetelmiä, joilla voidaan puolustautua erilaisia huijaustyökaluja vastaan. Keskitymme erityisesti menetelmiin, jotka ovat myös käytettävissä mobiilialustoilla.

Ensimmäiseksi tutkimme miten asiakaspuolen puolustusmenetelmiä, jotka voidaan suoraan toteuttaa pelin asiakassovellukseen. Nämä menetelmät sisältävät suojautumisen muistimuutoksilta ja tallennustietojen peukaloinnilta. Tutustumme myös miten koodin obfuskointia voidaan käyttää puolustautuakseen huijauksilta. Seuraavaksi tutkimme eri tyyppisiä palvelinpuolen arkkitehtuureja, joiden avulla voidaan estää huijaaminen pelissä. Lopuksi esitämme itsesuunnitellun arkkitehtuurin, joka erityisesti suunnattu mobiilipeleille. Esitetty arkkitehtuuri ottaa huomioon mobiililaitteiden useat eri rajoitukset, kuten epäluotettavat yhteydet ja akkujen lyhyen kestoiän.

Analyysimme osoittaa, että palvelinpuolen suojautumismenetelmät ovat huomattavasti luotettavampia estämään huijaaminen peleissä ja ne ovat myös paljon vaikeampi ohittaa tai rikkoa. Monet asiakaspuolen huijauksen suojautumismenetelmät luottavat epämääräisyyden tuottamaan turvallisuuteen, minkä takia ne ovat haavoittuvia, jos toteutuksessa on puutteita. Lisäksi asiakaspuolella huijarit pääsevät helposti käsiksi pelin muistiin, verkkoliikenteeseen ja laitteelle tallennettuihin tiedostoihin. Käyttämällä palvelinpuolen suojausmenetelmiä, joissa pelin logiikka ajetaan kokonaan tai osaksi palvelimella, tarjoaa tehokkaimman suojauksen huijareita vastaan.

Asiasanat:	huijaaminen, mobiilipelit, salaus, muistin muokkaus, koodin hämärtäminen, server-client, server autoratiivinen, client autoratiivinen
Kieli:	Englanti

Acknowledgements

I would like to give my gratitude and thanks to all who have supported me in this study. First and foremost, I would like thank my advisor, M.Sc. Henrik Lönnroth, for giving me the opportunity to work on this topic and also supporting and guiding me through the whole work. I would also like to thank my supervisor, Professor Antti Ylä-Jääski, for being patient with my progress on the thesis.

Espoo, May 20, 2018

Oskari Teittinen

Contents

1 Introduction 7
1.1 Research background and motivation 7
1.2 Problem statement 9
1.3 Research approach 9
1.4 Outline 10

2 Client-side cheat detection and prevention 11
2.1 Protecting game files 11
2.1.1 Introduction 11
2.1.2 Encrypting save and data files 12
2.2 Memory editing 13
2.2.1 Introduction 13
2.2.2 Using memory editing cheat tools 14
2.2.3 Protecting against memory editing 17
2.2.4 Summary 19
2.3 Time cheating 20
2.3.1 Introduction 20
2.3.2 Detecting time cheats 21
2.4 Reverse engineering 22
2.4.1 Introduction 22
2.4.2 Code obfuscation 23
2.4.3 Summary 24

3 Server-side cheat detection and prevention 25
3.1 Secure communication 25
3.1.1 Introduction 25
3.1.2 Securing connection using HTTP over TLS 26
3.1.3 Protecting against man-in-the-middle attacks 28
3.1.4 Summary 31
3.2 Client-server architectures 32
3.2.1 Introduction 32

3.2.2 Client authoritative model 33
3.2.3 Server authoritative model 35
3.2.4 Peer-to-peer model 37
3.3 Summary 39

4 Client-server architecture for mobile games 41
4.1 Introduction 41
4.2 Design and architecture 42
4.3 Implementation details 44
4.4 Summary 48

5 Conclusion and future work 49

A Code samples 56

Chapter 1

Introduction

Cheating is nothing new. There are writings from ancient Olympics where contestants would sabotage their opponent's equipment before the race in order to increase their chances to win. Sports, gambling and computer games are one of the most common activities where cheating may occur. This thesis presents different approaches how to prevent cheating in mobile games. Furthermore, this thesis evaluates each approach and their suitability in different scenarios. The purpose of analysing the scenarios is to answer what type of mobile games each approach would be optimal. Finally, this thesis introduces an combined approach on how to tackle the problem of cheating in mobile games using the lessons learned from the previously presented approaches. The resulting approach aims to provide both protection and flexibility without adding unnecessary design complexity or sacrificing game performance.

1.1 Research background and motivation

Cheating can be defined as: *to behave in a dishonest way in order to get what you want* [25]. For example, in tennis a player might use rackets of illegal string tension to gain advance. In video games a player might use a cheat where he gets more virtual currency. This can give the player unfavourable advance against other players and may break the gameplay. In online games where currency can be exchanged for other goods and items can be bought using virtual currency it could even break the economy of the game. This would affect all players and it would also hurt monetization if the gold is intended to be purchased using real money. This can be disastrous for the developer of the game if their source of income is only from in-game purchases.

There are many different ways how cheats can be applied in video games. They can be as simple as text editors which are used to modify installed game files or save game saves stored on the disk. They can also be more complex third-party applications. For example, they can modify runtime game data by injecting malicious code to game's executable code. This is achieved by editing memory used by the game's process or manipulating the game's network traffic.

Mobile games are not an exception in terms of cheating. The tools and intentions are the same. What makes it different is the free-to-play nature of modern mobile games and the massive player audience. The most popular mobile games are usually highly competitive and they can have hundreds of millions of players. These differences to traditional games make mobile games easy and desired targets for cheating. Finding and handling cheaters among the millions of players can be hard and tedious. Punishing cheaters can be difficult because the barrier to start a new game is very low due to the free price of the game.

Developers need to have tools and techniques to prevent cheating in their games. These tools and techniques can be software design patterns or algorithms that are used to protect the game data from exploits or smart ways of detecting players that are cheating. It is difficult to find research papers that would give an good overview of the tools and the techniques to prevent cheating in games. The closest one is an article written by Pritchard [26] which covers different ways that players are able to cheat at various online games and he also provides ways to prevent some of the cheats. Cano [7] has written a book about how to hack games using different techniques but he does not present any techniques to prevent them. Also, Hoglund et al. [17] presents different ways to break online game security in their book. There are also various books about software hacking and exploits that can be applied to cheating in games [8, 16, 20, 38]. Our plan is to provide a broad overview of the different tools and techniques used to prevent cheating. The majority of the existing research does not take into account the unique requirements of the mobile games and the limited resources of the mobile devices. In this thesis we will gather and study different software design patterns and algorithms from various research papers that can be used to either prevent or detect cheating in games. Finally, we present our own approach to prevent cheating in modern mobile games.

1.2 Problem statement

Cheating in mobile games can give a considerable advantage for cheaters and destroy user experience for other players. Developers might notice this as bad reviews and low revenue. Thus, in order to develop a successful modern mobile game the problem of cheating must be resolved. The problem can be divided in two parts: detection and prevention. The objective of this thesis is to study and evaluate different approaches that can be used to either prevent cheating or detect cheaters from the point of view of a mobile game developer. We pay special attention to the limited resources of mobile devices. The investigated approaches include solutions that can be software design patterns or algorithms that can be used in specific parts of the game design. They can be large architectural software designs or describe how a single operation in the game should be implemented. Suitability for mobile games is evaluated in terms of usefulness, complexity and latency. Furthermore, this thesis aims to design and present an suitable approach for modern mobile games on the basis of the studied approaches.

1.3 Research approach

This thesis studies both proactive and defensive techniques. Proactive techniques are used before cheating actually occurs thus they prevent cheating. Defensive techniques are responding to cheating after it has occurred thus they are used to detect cheating. In addition, the presented approaches can be implemented either client-side, server-side or both. These observations are used to categorize the approaches so that they can be presented in a sensible manner.

The research is divided into two parts. First, we start from the client-side approaches where we present both proactive and defensive techniques that are performed on the client device. Second, we present proactive server-side approaches which are designed to provide a reliable way to prevent cheating by handling gameplay on a remote server. Each approach is studied and evaluated using the available literature. Evaluation is based on usefulness, complexity and latency.

Finally, we put theory into practice and present a solution for modern mobile game development where we try to combine the best parts of the previously presented approaches. We aim to both design and implement a working approach that guarantees protection from cheating while focusing on being easy to implement while providing good performance and low network latency suitable for mobile devices.

1.4 Outline

- **Chapter 2 Client-side cheat detection and prevention** presents various client-side techniques that can be used to detect and prevent cheating on the client device.
- **Chapter 3 Server-side cheat detection and prevention** discusses how secure communication between the client and the server can prevent cheating and presents an overview of the client-server architectures and how they can be used to protect against cheating.
- **Chapter 4 Client-server architecture for mobile games** describes a solution for a modern mobile game which tries to combine the best parts of the previously presented approaches.
- **Chapter 5 Conclusion and future work** discusses about the results of our analysis of the cheat protection techniques and potential future research work.

Chapter 2

Client-side cheat detection and prevention

In this chapter we study how different client-side cheat protection techniques can be used to detect and prevent cheating in games. These techniques can be implemented in a way that they can be performed on the client device. This is really important especially for the single player games because otherwise they would be completely vulnerable for cheating. There a lot of different ways to cheat in the games and the client-side cheat protection techniques are the first line of the defence to protect against cheating.

In the chapter 2.1 we first study how encryption can be used to protect the game files from tampering. Next, in the chapter 2.2 we explore different memory editing tools and how they are used to cheat in games. In the chapter 2.3 we investigate how to detect players that are using time cheats to progress faster in the game. Last, in the chapter 2.4 we study different techniques that can be used to prevent the cheaters from reverse engineering the game.

2.1 Protecting game files

2.1.1 Introduction

Many games are using save files that are stored locally on the device. The game stores the progress of the game to these files, so that the player can continue playing the game without losing any progress when they return back to the game. Often games are using popular data serialization formats like JSON or XML to store the game state. Both formats store the data as human-readable text which means that the cheaters can easily figure out

the save format and modify the stored values using text editors. If the game saves the amounts of the items inside the player's inventory the cheater could easily find the saved item amounts in the save data and modify them to be larger. If the save data is serialized as non-readable binary data, the cheaters can edit the save files because there are available many tools that can be used to examine the binary data. The data files of the game are also vulnerable for modifications. Usually the data files contain information about the game rules like the item properties and the shop prices. The cheater could change the shop prices to be lower or tweak to properties of the item in order to make them stronger. In order to prevent the cheaters from modifying the save and the data files they can be protected using checksum validation or encryption. In the next section we look how encryption can be used to protect the files from tampering.

2.1.2 Encrypting save and data files

The best way to protect the save and the data files is to encrypt the content of the file. Encryption is a process where the data is encoded so that only authorized parties can access it. The data is encoded using an encryption algorithm that is also known as a cipher. The encryption algorithm generates a ciphertext from the plaintext data which can be read only if it is decrypted [30]. Many cryptographic algorithms that are used to encrypt and decrypt data are using symmetric-key algorithms where the same cryptographic key is used for both encrypting and decrypting the data.

It is very important to pick a strong cryptographic algorithm; otherwise the attacker could crack the cipher using a brute force attack. The cryptographic key size can be used to determine if the cryptographic algorithm is considered weak. For example, National Institute of Standards and Technology (NIST) recommends that the Federal governments to use keys that provide at least 112 bits of security strength for the key agreement [3]. The Advanced Encryption Standard (AES) is a symmetric cipher that uses key sizes of 128, 192 and 256 bits [37]. The AES provides a very strong security and it is used by many applications. The AES cipher requires a unique binary sequence that is often called an initialization vector and a secret key. The initialization vector can be randomly generated for each encryption operation and it doesn't need to be kept secret. In the appendix A we present how the AES cipher can be used to encrypt and decrypt a string value in C# programming language. The AES cipher is a very good choice for encrypting the game files. The secret key should be generated for the save files so that the same save file can't be used by other players. Only the player that created the save file should be able to decrypt the save file. This will

prevent the players from sharing their save files. The secret key can be tied to the device that they use to play, or it can be tied to a social account like Facebook or GameCenter account. On mobile platforms both Android and iOS provide an API to get a unique identifier of the player that can be used to generate the secret key.

Encrypting the save and the data files usually prevents the cheaters from tampering the files. But in theory, it is possible to decrypt the data without using the key. The attack can use a brute force attack to crack the encryption, but it usually requires significant computational resources. Also, the attacker could try to reverse engineer the game's binaries to find the cryptographic algorithm and the key that were used to encrypt the data. In the chapter 2.4 we study how to prevent the attacker from reverse engineering the game. Encrypting and decrypting the data can also affect to the performance of the game due to extra computations of the cryptographic algorithm. There are many different cryptographic algorithms available for different platforms and some of them might be faster than other. Usually, this means that they use slightly weaker cryptographic algorithms to improve performance. The Twofish cipher is one of the fastest cryptographic algorithms and it is similar to the AES cipher [31].

2.2 Memory editing

2.2.1 Introduction

Using memory modifying cheats is one of the easiest thing to do in games. Especially on the desktop operating systems like Windows or Linux based systems there are many different tools available. Some of them are free and some of them are paid applications. Some of them are tailored for a particular game and then there are also general-purpose tools that can work with any game. Cheaters are using the memory modifying tools to tamper the memory used by the game. By tampering the memory, they can change values within the game which gives them a significant advantage in any game. Memory editing cheats are usually used to give the player unlimited ammo or health or a huge amount of the game's currency that is used to buy items in the game. They can also change a high score value to be a significantly higher value before it submitted to a leaderboard. Sometimes there is no need to modify the values but instead the cheaters might use these tools to see what their opponents are actually doing in the game. The cheater could use the memory editing tool find how much resources their opponent has or in what position their opponent is located in an online multiplayer game. This

requires that the cheater has access to the opponent's game state. Usually the game exchanges the game states with the other players frequently in an online multiplayer game in order to keep the game synchronized. In the next section we investigate what kind of memory editing tools are available for cheaters and how they can be used to cheat.

2.2.2 Using memory editing cheat tools

The *Cheat Engine* is probably the most widely used cheat tool for memory editing. It is an open source cheat tool and it comes with a collection of different tools including a debugger, disassembler, assembler, speedhack and memory scanner [15]. The memory scanner can be used to scan for values within a game and it also supports for modifying these values. Cheaters can use the memory scanner to give them infinite amount of lives or a huge amount of gold that can be then used inside the game. The memory scanner searches the memory for the memory address of the specified value. In order to find the memory address of the value the user needs first to enter an initial value that the scanner should search for. The value must be exactly the same value as in the game. For example, if the player has 100 gold in the game, then it should set the initial value to 100. Next the user must trigger the first scan where the memory scanner looks for all the matching values in the memory. Usually, after the first scan there are hundreds or even thousands of matching values. In order to narrow the results, the user needs to change the value in the game to another value and after that the memory scanner can scan again the addresses of the values in the previous results to see if any of the values matches to the new value. For example, for the second scan the player spends 20 gold so that the value changes from 100 gold to 80 gold. Now in the second scan the player searches for all values that matches to 80 using the previous results. This should narrow the number of results and it is repeated until there are only one or a few matches left. For the remaining matches the user can try to modify the values using the memory scanner and see in the game if the value has actually changed. The figure 2.1 shows the *Cheat Engine*'s user-interface for the memory scanner and the options available for the scanning process. The user can search for different value types like integer or floating-point values and also specify the memory address range to search. It also supports a fast scan where it uses an optimized algorithm to search the value, but it might not find all values.

Internally the *Cheat Engine* uses the low-level kernel APIs to attach to other processes and to read their memory. On Windows the *Cheat Engine* uses *OpenProcess*, *ReadProcessMemory* and *WriteProcessMemory* functions from the Windows API. These functions require that the user has elevated

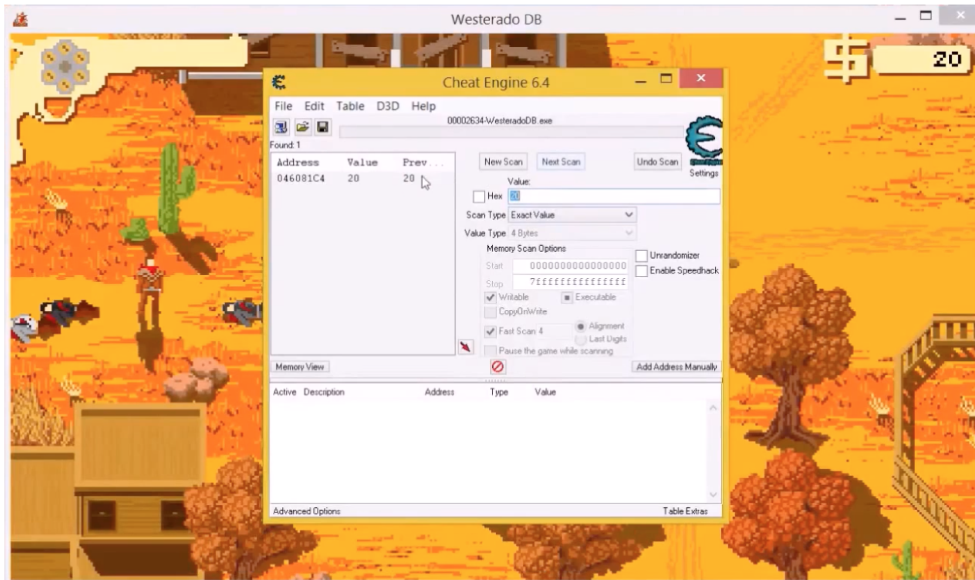

Figure 2.1: Using the *Cheat Engine* to modify the amount of the in-game currency in the *Westerado: Double Barreled*.

privileges and often the cheat tools are required to run with administrator privileges. Another ways is to use an DLL injection where the another process is forced to load a dynamic-link library (DLL) to run code within the address space of the process [35]. The Linux kernel provides access to the process memory via the proc filesystem.

The memory editing cheat tools are not only limited to desktop operating systems, but they are also available for mobile operating systems. There are many different memory editing cheat tools available for the most popular mobile operating systems. *GameGuardian* is a very popular cheat tool for the Google's Android operating system and it supports scanning and modifying memory. Similarly named *iGameGuardian* cheat tool is available for the Apple's iOS operating system and it also provides the same memory editing functionality. Both the *GameGuardian* and the *iGameGuardian* cheat tools function on the same way as the previously presented *Cheat Engine*. The user must first select the correct game process from a list of currently active processes on the device and then scan the memory for the specific value in the game. If there are too many matches to the value, then the user can change the value in the game and repeat the scan step again with the new changed value until there are only few matches left. After finding the correct memory address of the specified value the user can change the value to any

value from the tool. The figure 2.2 shows the *GameGuardian* running on a Android device.

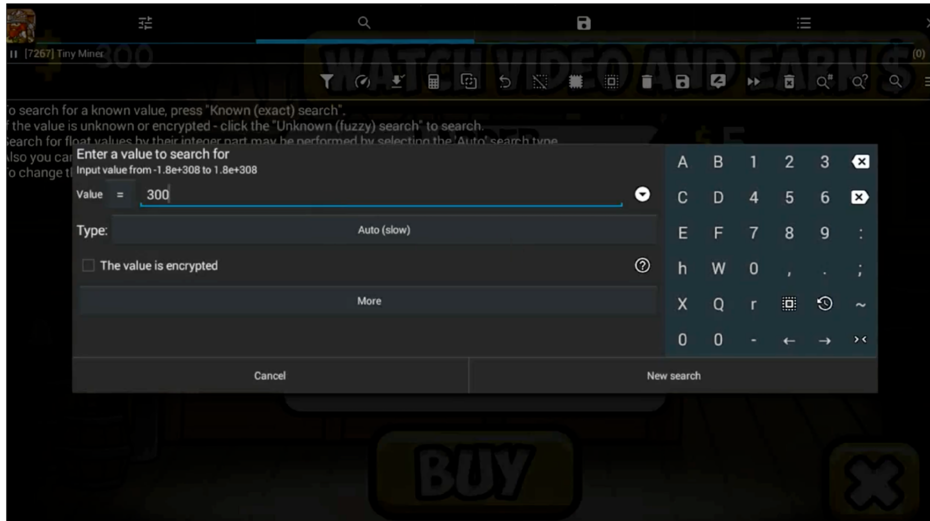

Figure 2.2: Using the *GameGuardian*'s memory scanner on the Android.

Although it seems that the memory modifying cheat tools on mobile devices are as easy to use as their desktop counterparts there is a one significant issue with them. On Android in order to use the *GameGuardian* the device must be rooted. Rooting allows the user to attain privileged control over resources that are normally protected from an application or user. Because Android uses the Linux kernel, rooting can be thought as giving access to administrative permissions as on Linux. Rooting allows the user to run specialized applications that require administrator-level permissions and perform operations that would not be possible otherwise. If the *GameGuardian* app is installed without the root access, then it would not be able to access the low-level kernel resources that are critical for the tool to work. Normally Android apps cannot access any other apps data but with the root access the permission checks are disabled for the app. This allows the app to access other apps data including memory. This is possible because the kernel API provides the system calls to read and write to other process memory. Rooting the Android device is not an easy task because it depends on the manufacturer how the device can be rooted. Some devices have official support, and some might require exploiting a known vulnerability in order to being able to root the device. In the worst case the process can brick the device. On the iOS platform the cheaters are facing a similar issue where they need to jailbreak the iOS device in order to gain the root access that is

required to run cheat tools on the device. For example, the *iGameGuardian* requires a root access to function properly. Jailbreaking the iOS device is also needed to install the cheat tools because they don't conform the App Store review guidelines thus they are not downloadable from the Apple's App Store. Android supports sideloading the apps from other places than the official Google Play Store but on iOS you can only install the apps from the App Store. Jailbreaking the iOS device removes these imposed software restrictions. The jailbreaking process is very similar to how the Android devices are rooted and it requires the same amount of work or even more. Also, there can be legitimate apps on iOS that may disable some of their features if the app is running on a device that is jailbroken due the security issues it imposes.

We could not find any cheat tools with memory editing capabilities that didn't required either rooting or jailbreaking the device. One way to get around the problem of rooting the Android device is to use an emulator. The Android Emulator comes with the official Android SDK and it can be used to simulate different Android phones and tablets. The Android Emulator also supports rooting thus the cheaters can install the cheat tool and the game to the emulator. After using the cheat tool in the emulator, the player can transfer the game's save files from the emulator to another device if needed.

2.2.3 Protecting against memory editing

Encryption can be used to protect values from the memory editing. The idea is to obfuscate all the significant values so that it becomes increasingly hard to find the values using the memory scanner. This means that we should encrypt values that have a real importance for the game like the amount of gold the player has in the inventory or the player's health and ammo values. If we would try to protect all the values in the game, it would have a significant negative impact to the game's performance because encrypting and decrypting the values adds up the CPU cycles. Also, depending on the algorithm used it would increase the memory usage. One of the easiest way to encrypt integer values is to multiply them with a specific value before they are stored to the memory. When integer value is read from the memory then the value is first divided with the same value that was used to multiply it. Another option is to use communicative function such as XOR that was suggested by Pritchard[26]. Bitwise *XOR* sets the bits in the result to 1 if either, but not both, of the corresponding bits in the two operands is 1. When the value is stored to the memory it is encrypted using the XOR function and a predefined encryption value. When the value is read from the memory it is decrypted using the same XOR function and the same

predefined encryption value. The result will be the same value that was stored to the memory before it was encrypted. The figure 2.3 below shows how the *XOR* function can be used to protect values in C# programming language. Using stronger encryption algorithms is not feasible it could have a significant impact to the performance due to more complex computations. Also, we can't guarantee that the cheater would not be able to find the value even if the value is encrypted. The problem is that the cheater can decompile the game's executable and find the algorithm used to obfuscate the values.

```
1  public class GameResources
2  {
3      // List of all game resources
4      private int[] _resourceAmounts;
5
6      // Resource encryption values
7      private int[] _encryptValues;
8
9      public void SetResource(int resourceIndex, int amount){
10         _resourceAmounts[resourceIndex] = amount ^ _encryptValues[resourceIndex];
11     }
12
13     public int GetResource(int resourceIndex){
14         return _resourceAmounts[resourceIndex] ^ _encryptValues[resourceIndex];
15     }
16 }
```

Figure 2.3: Using XOR function to protect variables from memory editing.

Another protection method is data duplication. Instead of encrypting the value we can calculate a checksum value from the value to be stored and save the checksum value to another location. When the value is next time accessed we can calculate the checksum value from the value and compare it to the checksum value that was previously saved to another location. If the checksum values are not matching, then we can assume that the value was modified using a memory editing tool. This approach can be also used not just for single values but for a structure of a data. A single checksum value can be calculated from multiple values. For example, if the game has a player information class that contains all player data, we then can calculate a single checksum from the data. The Cyclic Redundancy Check (CRC) algorithm [24] is often used calculate the checksum value because it easy to implement and it is very fast to calculate.

Sometimes the cheat tools rely on the fact that the value resides in a specific address inside the process's memory. They might have found the address by examining the game's executable or memory scanned the value in advance. For example, static variables can be stored in the data segment of

the program's address space. To prevent this exploit the game should avoid using static variables store values that are critical for the game. Instead they should use dynamic memory allocation to allocate the variables to heap memory. When the variable is allocated using the heap memory the address of the value is different every time the game executable is run.

2.2.4 Summary

The memory editing cheat tools are a serious threat for games because they are very easy to use and there many articles available in the Internet that teaches how to use them efficiently in games. Also, many of the memory editing tools are free including the most popular ones. But the games can try to protect against these tools by using different methods. Encrypting the values that are important for the game is a viable way to protect against the memory editing tools. As we learned in the previous section encryption can be used to hide the significant values from the memory scanners. Another way is to use hashes and checksums to determine if the significant values were changed by an external process or application. Both methods have an impact to the games performance because the values must be either encrypted or decrypted when they are accessed. These operations can become a bottleneck if the values are accessed frequently in the game. Another downside of these memory protection methods is that they rely on the secrecy of the implementation. If the cheaters manage to figure out the algorithm that was used to encrypt the values or to calculate the checksum value, then they can easily circumvent the protection.

The situation is similar on the mobile platforms. There are memory editing cheat tools available for both major platforms: iOS and Android. These tools have the same capabilities as their counterparts on the desktop platforms. Both iOS and Android platforms have a strict control on the applications that can be used, and they use sandboxing to control what resources the applications may use including the memory and file system resources. But these restrictions can be avoided by either rooting or jailbreaking the device which is required in order to use the memory editing tools on the mobile platforms. The good thing is that the rooting and the jailbreaking process can be very complex for the cheater. Also, they might not want to do it because it makes the device vulnerable for other threats like malware. To avoid this problem the cheater can use an emulator with rooted access instead of the cheater's own phone or tablet.

2.3 Time cheating

2.3.1 Introduction

Not all cheats need complex instructions or additional tools in order to use them. Time cheats are easy to execute because all you need to do is to change the system time from the operating system settings. There are a lot of games that are built around time mechanics where players must often wait for a specific duration until they can do the next action. Crafting games are a very popular game genre where the gameplay mechanics revolve around producing different items. Usually, in order to produce an item, the player must wait for a specific amount of time that can last from minutes to hours until the item can be collected. Players might try to cheat this by changing the system time so that the clock time goes forward. When the player returns to the game, the game needs to determine how much time has passed since the last time the player was in the game. Usually they rely on the system time that is accessible via an API provided by the operating system. The amount of time elapsed is calculated by subtracting a system time value, that was stored when the player was last time in the game, from the current system time value. The elapsed time value is then used to advance the timers in the game. If the player moved the system time one hour forward the game would think that the same amount of time has already passed in the game. In the case of the crafting game this gives the cheater an ability to produce more items in a very little time and thus giving the cheater an advantage. Speed hacks Another time-based cheat method is a speed hack. Speed hacks are used to give the cheater a significant boost to movement or to firing speeds which makes it harder for the other to defeat the cheater in the game. Some speed hacks are also used to slow down the action, so that the cheater has more time to react to the events of the game. For example, in a platformer game where the player must dodge obstacles or projectiles the cheater might use a speed hack that actually slows down the game, so that the cheater would have more time to react and give the correct inputs to avoid the obstacles and projectiles.

The *Cheat Engine* that was presented in the chapter 2.2.2 supports speed hacks that either decreases or increases the speed of the game. It uses a hook that alters the behaviour of the timing functions in the game. The *Cheat Engine* intercepts the function call when the game tries to get the current system time using a DLL injection technique. Instead of returning the correct time value it returns a modified value that makes the game think that either more time has elapsed or less time has elapsed. If the game thinks that more time has elapsed, then the game will be running faster and

vice versa. Preventing the speed hacks from working is very difficult because the game needs to be able to detect if the timing functions are hijacked by the cheat tool. Also, the cheat tools are often using different techniques to prevent the game from detecting if a malicious DLL was loaded.

2.3.2 Detecting time cheats

Detecting if the player is using a time cheat is a very challenging task because there is no single reliable way to detect it on the client-side. Especially, if the time cheat is using the DLL injection technique it is very likely that it will be not detected. But if the player tries to cheat time by manually changing the system time it is possible to detect it. In order to achieve this, we need to be able to calculate the elapsed time using timing functions that do not depend on the system time. On Unix-like systems the system time can be get using the *gettimeofday* system call which gives the number of seconds and microseconds since the Epoch [2]. The Epoch is an absolute time value that is the number of seconds elapsed since January 1, 1970 (midnight UTC/GMT). The problem is that when the user changes the system time it also affects to the *gettimeofday* function.

An alternative way to calculate the elapsed time is to use the system uptime that is typically the amount of time the system has been working and available. The system uptime is a relative time that doesn't have any fixed reference date and it is unaffected by the system time changes. Because there is no fixed reference date it might return the same value at different times, so we can't use it directly to measure the elapsed time in the game, but we can use it to validate the elapsed time that was calculated using the system time. So, when we store the system time we also store the system uptime value. When the elapsed time is calculated we calculate the time difference between the last stored system time and the current system time but also we calculate the time difference between the last stored system uptime and the current system uptime and then compare the differences between the system time and the system uptime. If the player did not change the system time, then the system time and system uptime should be the same. There might be slight differences in the values if the system time and the system uptime are using different clock resolutions. If the elapsed time calculated using the system time is notably larger than the elapsed system uptime, then we can assume that the player did change the system time. In this case we would use the elapsed system uptime to update the timers in the game instead of the elapsed system time. There is a one case where the system uptime cannot be used to detect if the player changed the system time. If the player restarts the computer or the device the system uptime values is also reset.

This means that the elapsed system uptime value can be a negative value if the current system uptime value is smaller than the stored value. In this case we can't validate the elapsed system time. The cheaters can exploit this weakness because they can first change the system time and then restart the device, thus making it impossible to detect if the system time was changed.

The system uptime value can be get using the *clock_gettime* function that is defined in the POSIX standard [1]. It is preferred to use the *CLOCK_MONOTONIC_RAW* clock instead of the *CLOCK_UPTIME_RAW* clock because the *CLOCK_UPTIME_RAW* does not increment while the system is asleep. On mobile devices this would mean that the *CLOCK_UPTIME_RAW* clock is not incremented when the device is in the sleep mode. Both *CLOCK_MONOTONIC_RAW* and *CLOCK_UPTIME_RAW* clocks are unaffected by frequency or time adjustments. The *clock_gettime* function is supported by the most operating systems including the Android and the iOS operating systems.

2.4 Reverse engineering

2.4.1 Introduction

Chikofsky et. al.[9] defines the reverse engineering as a process of analyzing a subject system to identify the system's components and their interrelationships. Also, they note that the reverse engineering can be used to create representations of the system in another or at a higher level of abstraction. Cheaters can use different reverse engineering techniques to learn how the game logic works and to access sensitive information like the cryptographic algorithms that are used to protect the game data from tampering. The cheater can decompile the game binaries using a tool to obtain the source code of the application. Using the source code, the cheater could find all the cryptographic algorithms that were used to encrypt the data in the game. For example, the cheater could use the source code to find out how to circumvent the memory protection methods that were presented in the chapter 2.2.3. Also, the cheater could use the source code to find exploits in the game logic.

Programming languages that compile the source code into intermediate languages are vulnerable for reverse engineering. The original source code can be obtained from the intermediate language code using a decompilation tool. C# is a very popular programming language that belongs to the .NET programming languages developed by Microsoft. Many game engines, including the Unity3D, are using the C# programming language. The C# source code files are compiled into intermediate language code called Common In-

termediate Language (CIL). At the runtime the CIL code is transformed to machine code by the Common Language Runtime (CLR) which is then executed by the computer. In order to translate the CIL code the interpreter needs to know the names of the classes, methods and fields. This makes the CIL code vulnerable for decompilation because it contains all the original names of the classes, methods and fields which makes the decompiled source code to easier to read and understand.

2.4.2 Code obfuscation

Code obfuscation is a process where the source code is transformed so that it is difficult to read for humans [23]. The code obfuscation can be used to make it harder to reverse engineer the game from the binaries. The obfuscated code does not change the functionality of the program and the output of the program remains unchanged. But it makes the source code more difficult to understand when it is decompiled from the binaries.

Name obfuscation is the most common technique to obfuscate the source code. Usually, in the original source code the names of the classes, methods and fields have a meaningful name that describes what they are used for. When the code is compiled into the intermediate code these names are preserved. The name obfuscation is used to replace the names of the classes, methods and fields to meaningless strings. The figure 2.4 shows an example method before and after the name obfuscation process. String encryption is another code obfuscation technique and it is used to hide strings in the code that are easily readable from the binaries or from the decompiled source code. The string value is encrypted at compilation and when it is accessed on the runtime the value is decrypted. It prevents the attacker from finding critical code sections by looking for string references inside the binary. For example, the attacker could try to search code sections where a specific error code is used. It can be also used to prevent the attacker from finding the encryption keys that are defined in the source code and used in cryptographic algorithms. Also, it can be used to hide secret keys that are often required when communicating with web services. Control flow obfuscation makes the control flow of the program difficult to understand. It adds conditional instructions to the code that are always either true or false. These conditional instructions are used to increase branching in the code which makes it very difficult to follow. The obfuscator can also insert dummy code into the executable. The dummy code does not affect to the logic of the program, but the extra logic can make the decompiled code more difficult to analyze.

The code obfuscation is easy to perform because there lots of tools available for different programming languages that provide various techniques to

Before name obfuscation

```
1 public class Weapon
2 {
3     private float CalculateDamage(WeaponBoostList boosts)
4     {
5         var damage = BaseDamage;
6         while (boosts.HasMore())
7         {
8             var boost = boosts.GetNext(true);
9             boost.UpdateStatus();
10            damage += boost.ResolveDamage(Player);
11        }
12        return damage;
13    }
14 }
```

After name obfuscation

```
1 public class Z
2 {
3     private float A(B b)
4     {
5         var d = D;
6         while (b.A())
7         {
8             var e = b.B(true);
9             e.A();
10            d += e.B(B);
11        }
12        return d;
13    }
14 }
```

Figure 2.4: Using the name obfuscation to alter the names of the classes, methods and fields.

obfuscate the source code. The code obfuscation process is usually automatic, and it doesn't require any extra work from the programming perspective.

2.4.3 Summary

The code obfuscation provides an easy way to protect the game from different reverse engineering techniques. It makes the reverse engineering process more difficult and time consuming for the cheater. It also prevents the cheater from finding out how the other protection methods of the game are implemented. The downside of the code obfuscation is that it only makes the reverse engineering process more difficult but not impossible. Thus, with enough time the cheater can reverse engineer the game. Also, some of the obfuscation techniques may affect to the performance of the game due to the added extra logic.

Chapter 3

Server-side cheat detection and prevention

Online games are very popular today on all platforms. There are countless online games available on PCs, game consoles and mobile devices. It seems like online features are required to succeed in the gaming industry. For example, most of the new and well-known AAA-games on PC and console platforms implement at least the basic multiplayer features. On mobile, many of the top grossing games in the App Store and Google Play store have features like leaderboards, guilds and PvP-combat. All online games require a computer network, such as the Internet, and a server to connect with in order to player the game online.

In this chapter we will study different client-server architectures that are used in online games to communicate between the client and the server and how these architectures can used to detect and prevent cheating in the games. First, in the chapter 3.1 we examine the benefits of using a secure communication to prevent cheating in games and what kind of vulnerabilities the cheaters can use in the client-server communication to their advantage. In the chapter 3.2 we study and compare different client-server architectures in games and how they can used in protecting against cheaters.

3.1 Secure communication

3.1.1 Introduction

Security and especially network security has become increasingly important part of the software design. Nowadays users are more aware of the problems of using unsecured communication and data it could possibly expose.

Having a secure communication between the client and server is important for both applications and games. Games especially might have very specific requirements. For example, intensive FPS multiplayer games require minimal latency when sending the input actions over the network to the game's server. Popular mobile games are required to be able to handle hundreds of thousands of simultaneous players. Adding a layer of security might increase latencies and decrease server performance but it also provides a way to protect against cheaters. There are many ways how cheaters can exploit the unsecure communication. This can vary from message replication to actually modifying the message content. Creating a secure connection between the client and the server provides security through obscurity because it prevents cheaters from reading and modifying the transmitted messages.

3.1.2 Securing connection using HTTP over TLS

Transport Layer Security (TLS) and its predecessor Secure Sockets Layer (SSL) are the most popular cryptographic protocols that provide communications security over a computer network. [11] TLS is specified in RFC 5246 by Internet Engineering Task Force (IETF). SSL protocol is specified in RFC 6101, but the use of SSL has been later prohibited due to its vulnerabilities [13] [39] [4]. The TLS protocol is commonly used over HTTP to secure communication between servers and web browsers. This protocol is known as HTTPS protocol and it specified in RFC 2818 [28]. The TLS provides both privacy and data integrity. It prevents eavesdropping, tampering and message forgery because it uses symmetric cryptography to encrypt the transmitted data. The TLS also supports authentication of the client and the server using public-key cryptography. Integrity is ensured by including integrity check using a message authentication code.

The protocol is composed of two layers: TLS record and TLS handshake protocols. The TLS handshake protocol is used to negotiate a stateful connection. The handshake procedure includes the following steps: agreement of the protocol version to use, cipher suite selection, authentication of the server (and optionally the client) and session key information exchange. The client sends first a *ClientHello* message that lists cryptographic information such as the supported TLS versions, supported cipher suites and a random number. The server responds with a *ServerHello* message that contains the selected TLS version and cipher suite. These are chosen from the list provided by the client. The message also contains another random number. Optionally the server may include a session id that can be used to perform a resumed handshake. Next, the server sends a Certificate message that contains the server's digital certificate that includes the server's public key. The client ver-

ifies the certificate and extracts the public key which then used to encrypt a *PreMasterSecret* number that is then send to the server. The server decrypts the *PreMasterSecret* number using its private key. Now both the client and the server can compute the shared secret key using the *PreMasterSecret* and the previously sent random numbers. The client sends a *ChangeCipherSpec* message that tells the server that the client is now using the shared secret to encrypt the subsequent messages. The server also sends *ChangeCipherSpec* message to the client that tells the client that the server is also now using the shared secret. After this the TLS handshake is completed. The whole handshake procedure to establish a secure connection is presented below in the figure 3.1.

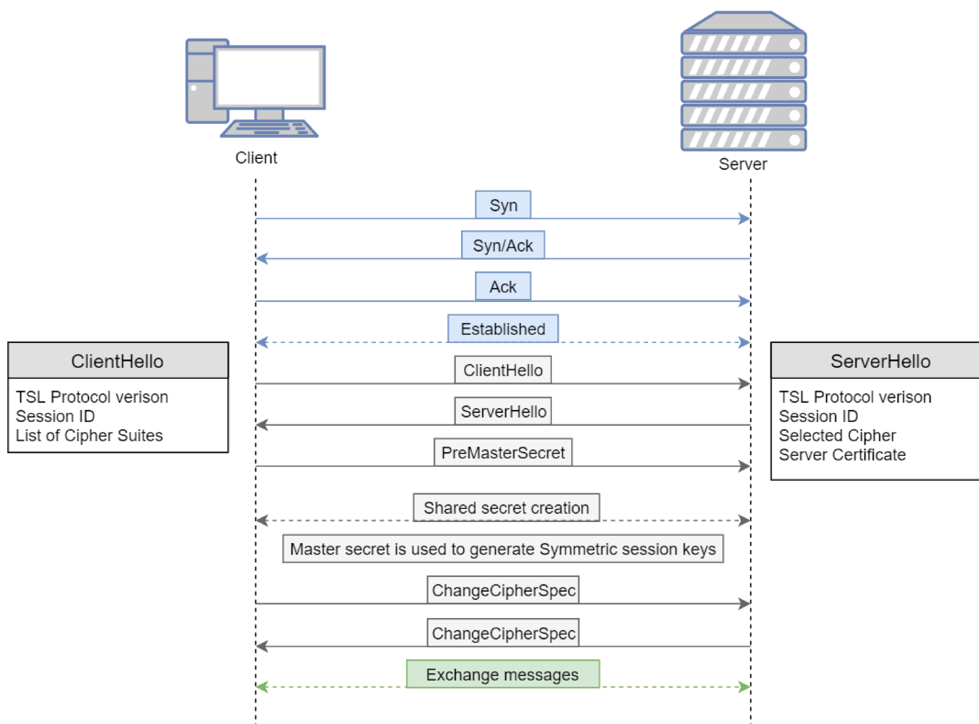

Figure 3.1: The TLS protocol handshake procedure steps.

The TLS protocol is not only restricted to be used with the HTTP protocol, but it can be used with any other reliable transport protocol such as Transmission Control Protocol (TCP). RFC 6347 [29] also defines TLS protocol for User Datagram Protocol (UDP). UDP is widely used in games to communicate between the client and server because it provides improved latency at the expense of reliably.

There have been serious attacks on TLS over the years and the protocol have been updated several times to add new features, stronger cipher suites and to remove older weak ciphers. IETF has released RFC 7457 [34] that summarizes the known attacks on TLS and DTLS. They have also released RFC 7525 [33] that provides recommendations for secure use of TLS and DTLS which includes recommended cipher suites because some cryptographic algorithms that were once considered strong have weakened over time. There have been also serious vulnerabilities in softwares implementing the TLS protocol. One of the most notable vulnerability is the Heartbleed bug that affected popular OpenSSL cryptographic software library. The bug allowed attackers to steal private keys from servers using a buffer over-read bug. This allowed attackers to eavesdrop communication and tamper with the data. In the next section we study an attack that can be used against TLS protocol.

3.1.3 Protecting against man-in-the-middle attacks

One of the potential ways to cheat in games is to intercept communication between the client and the server. A possible scenario occurs when a player finishes a level and the client reports the final score of the level to the server for updating the leaderboard. In this case the client sends a HTTP request to the server that contains the final score, but the player could intercept this request using a third-party tool and modify the final score to be a higher value. This type of attack is known as a man-in-the-middle (MITM) attack where the communication between two systems is intercepted and it is commonly used to eavesdrop or alter traffic for malicious purposes [27]. The attack relies on the fact that the attacker can impersonate each endpoint, so that they believe that they are communicating directly with each other. This can be achieved by splitting the connection into two different connections. First, when the client attempts to connect to the server, the attacker intercepts this connection. Instead of creating a connection between the server and the client it creates a connection between the client and the attacker. After the connection is established with the client the attacker connects to the server by relaying client requests to the server. Now the attacker acts as a proxy between the client and the server and it can now read and modify the messages that are sent between the endpoints. Figure 3.2 shows how the new client and server connections are created by the attacker's proxy.

The man-in-the-middle attack can be used to circumvent mutual authentication where two endpoints authenticate each other using some protocol before actually exchanging any data. For example, in the case of a public key certificate authentication when the client asks for the server's public key, the attacker intercepts this and includes its own forged public key instead of

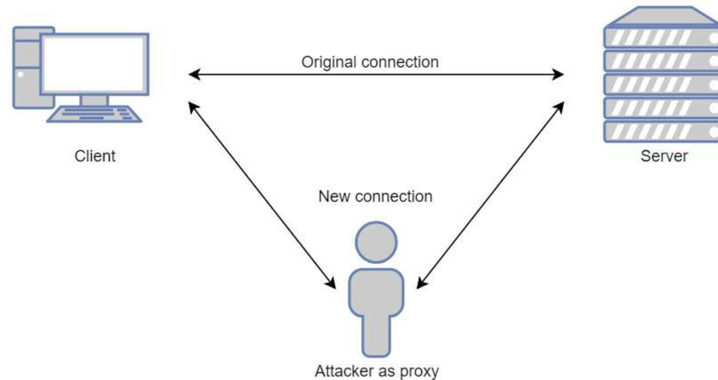

Figure 3.2: Attacker works as a proxy in the man-in-the-middle attack.

the server's public key. Now the client thinks that the public key comes from the server, but it actually encrypts messages using the forged key which the attacker can now decrypt. When the attacker communicates with the server it uses the server's public key in the same way as the client would normally do. As presented in the previous section TLS can be used to protect from the MITM attack because the client and the server exchange certificates which are issued and verified by a trusted third-party certificate authority (CA). In the TLS protocol the client would reject the attacker's forged certificate and drop the connection because it is not issued by a trusted CA. But this works only if the client itself is not vulnerable. Callegati et al. [6] have presented such an attack that replaces the original certificate authenticating the HTTPS server with a modified certificate. The weakness of the attack is that it requires that the client first trusts the forged certificate. But in the case of cheating where the client is actually the attacker it can be expected that the client trusts the forged certificate. In the next section we demonstrate how the attack can be executed on a mobile phone using called *mitmproxy* command line tool.

Mitmproxy is a free and open source interactive HTTPS proxy which can be used to intercept, inspect, modify and replay web traffic such as HTTP or any other SSL/TLS-protected protocols[10]. Mitmproxy acts as a certificate authority that dynamically generates certificates to any hostname. In order to use *mitmproxy* the cheater would need to install the tool on a computer that that is connected to the same network as the phone. Next, the *mitmproxy* CA certificate needs to be first downloaded and then installed on the phone manually from the device settings. Also, to intercept phone's HTTP/HTTPS connections the standard gateway address of the phone must

be changed to the IP address of the *mitmproxy* server. On the computer running the *mitmproxy* server the network traffic must be forwarded to it and redirected to the port that *mitmproxy* is listening. In case of HTTP/HTTPS ports 80 and 443 must be redirected. Finally, the *mitmproxy* can be started and it should start intercepting requests from the phone. Mitmproxy provides also tools to filter requests and modify them on the fly. The figure 3.3 presents the overview of the MITM attack using the *mitmproxy*.

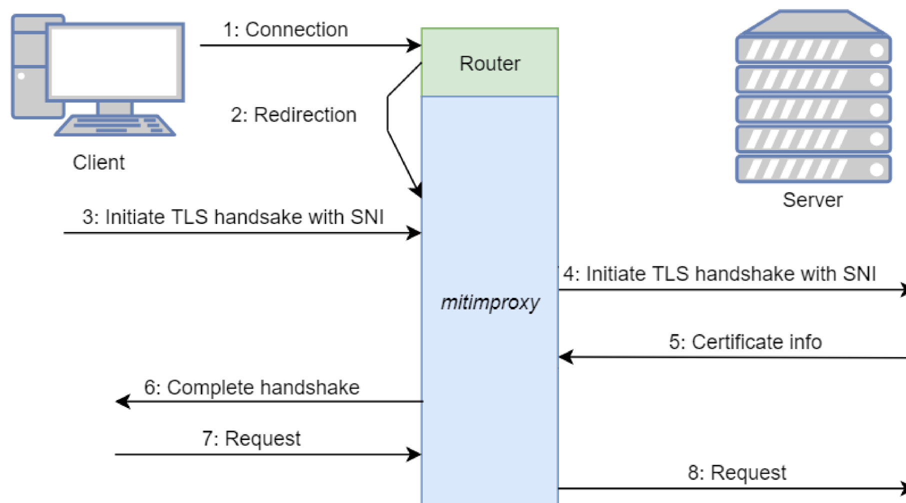

Figure 3.3: Overview of the MITM attack using *mitmproxy*

To prevent cheaters from using untrusted certificates and tools like *mitmproxy* to intercept and to modify messages the client can use technique called certificate pinning [12]. The HTTP protocol can provide a list of public key hashes via HTTP headers during the first transaction and the client must use one or more keys in the list in the following transactions. This can prevent MITM attacks if the very first transaction is not intercepted by the attacker because the attacker's forged certificate doesn't match any of the "pinned" keys. A more secure way to provide the "pinned" keys or certificates is to embed the server's public keys or certificates directly to the client. The public key hashes can be hardcoded in the code or certificates can be stored in the application binary. This prevents the attacker from intercepting the first transaction and altering the "pinned" keys. In the figure 3.4 below, we present an example how to implement certificate pinning in C# using *Unity3D 2018.1* where the "pinned" public key hashes are stored in directly into game code and each time when a request is sent to the server it validates the X509 certificate by comparing the public key hashes.

```
1 using System.Collections;
2 using System.Security.Cryptography.X509Certificates;
3 using UnityEngine;
4 using UnityEngine.Networking;
5
6 public class CertificatePinning : MonoBehaviour {
7
8     class CustomCertificateHandler : CertificateHandler
9     {
10        // Array of encoded RSAPublicKeys
11        private static string PUBLIC_KEYS[] = ...
12
13        protected override bool ValidateCertificate(byte[] certificateData {
14            X509Certificate2 certificate = new X509Certificate2(certificateData);
15            string publicKey = certificate.GetPublicKeyString();
16            if (PUBLIC_KEYS.Contains(key => string.Equals(key, publicKey))) {
17                return true; // Trusted certificate.
18            }
19            return false; // Public key mismatch.
20        }
21    }
22
23    private IEnumerator Start() {
24        UnityWebRequest www = new UnityWebRequest("https://my-game-server-address.com");
25        www.certificateHandler = new CustomCertificateHandler(); // Our certificate handler
26        yield return www.SendWebRequest();
27        if (www.isNetworkError || www.isHttpError) {
28            Debug.LogError(www.error); // Invalid certificate
29        }
30        else {
31            Debug.Log("Success!");
32        }
33    }
34 }
35
```

Figure 3.4: Certificate pinning example using *Unity3D 2018.1*.

3.1.4 Summary

From the cheating point of view having a secure connection between the client and server gives a strong protection against a wide range of attacks that would otherwise be easy to execute. TLS prevents any message replication cheats and certificate pinning prevents cheaters from modifying messages on the fly. Also, not directly related to cheating but users are nowadays more aware of the potential security issues if an unsecure connection is used to transmit user's personal information like account name, social identifiers or payment information. For example, many games are using social identifiers, like Facebook, to identify and authenticate players, thus using unsecure connection would expose this user information. Apple has also announced App Transport Security (ATS) feature that requires macOS and iOS applications to support best practice HTTPS security. This mean that applications must use HTTPS protocol when communicating with the application's backend otherwise they must statistically declare its security limitations. As of writing this thesis, this requirement is postponed but it is planned to be required in the near future.

The drawbacks of using HTTPS or other protocol with TLS are increased latency and performance cost. Naylor et al. [22] investigated the cost of using

HTTPS while browsing web pages where they measured page load times, data usage and power consumption on a mobile device. Their results show that HTTPS increased page load times more than 500ms in 90% of websites when using 3G network and the average TLS negotiation overhead was around 5% in data usage. Also, there was a notable decrease in battery life when using HTTPS in 3G network. Goldberg et al. [14] have also compared HTTP and HTTPS performance on the server side and they measured a decrease in transfer rate that was around 20% which was considered as a reasonable price to pay for security. This could lead to slightly increased server costs when using HTTPS. HTTPS supports session IDs and session tickets [sources] that can be used to resume the TLS handshake and avoid a full handshake. This can reduce latencies of the requests after the first request but it also depends on how long the session IDs and session tickets cached on the client and the server. This may vary depending on the platform.

3.2 Client-server architectures

3.2.1 Introduction

In the client-server model clients and servers communicate with each other via messages. This message exchange pattern is known as a request-response messaging pattern [18]. In this pattern the client connects to the server using a communication protocol that works on the application layer and is understood both the client and the server. Once the connection is established, the client can make a request by sending a message to the server which is then decoded and processed by the server. After processing the request, the server sends a response message back to the client. Usually this response message contains a result value indicating if the request was successfully completed. Depending on the used protocol, the server can either close the connection after sending the response message or maintain the connection to the client. If the connection is not closed, the server can listen for more messages and also send messages to the client.

In distributed computing clients are distributed across multiple different servers depending on various factors, such as location and server load [21]. However, games are often using a more traditional client-server model where a single server provides resources to all clients.

The client-server model allows executing game logic on the server instead of on the client. This allows the system to validate the player's actions before they are applied on the server's game state or sent to other clients. Using different validation techniques, the server can verify that the requests sent

by the client are actually plausible in the game. For example, the client could send a request where the player moves to a position that should be not possible from the previous position of the player. The server can validate this request by calculating the distance moved and determine if the player can move such distances. The server can also repeat the player's input on the server and determine if the server's outcome matches to the client's outcome. If the server thinks that the request is not possible the request fails validation and it is rejected. The position change is not stored in the game server's state.

There are many different designs how to implement the client-server model and in the next chapters we will take a look on the client-server models that are commonly used in games. This includes the client-authoritative and the server-authoritative models.

The client-server model is not the only architecture that is used in online games. A peer-to-peer (P2P) model is also widely used in many games because it doesn't require a centralized server. In the chapter 3.2.4 we also study how P2P model works and what needs to be considered when protecting against cheaters in P2P model.

3.2.2 Client authoritative model

The client authoritative model is probably the most commonly used client-server architecture in games. In this model the client runs the game logic and sends the result to the server that trusts that the client's result is valid without validating the data. The client authoritative model is very popular because it is easy to implement on top of an existing client-only architecture. Quite often games are first developed using a client-only architecture and then later in the development process features that requires a cooperation with a server are added in the game. Another reason to only implement a client authoritative model is that the more complex client-server architectures might not be considered as a necessity due to the nature of the game. For example, games that are single player or have only light multiplayer features, such as high score leaderboards, implementing a fully-fledged server authoritative model might considerably increase the development time and costs due to the added complexity. Many of the casual games that have only these simple multiplayer features , are client authoritative because the player competes usually against itself and not with other players. Game genres such as puzzle, arcade and trivia games, are good examples of casual games that are often client authoritative.

In the client authoritative model most of the game logic is executed on the client-side. The server assumes that the actions sent by the client are to be trusted and that there is no need to validate the actions. This can

simplify the server-side implementation of the game logic and significantly lighten the load on the server-side, but it also makes the game vulnerable for cheating. The client authoritative model relies on client-side techniques to prevent cheating. In chapter 2 we presented various techniques that can implemented on the client-side to protect from different types of cheating techniques. The client authoritative model works so that the player actions are first executed in the client's game logic after which the actions are sent to the server. The content of the message sent to the server can vary depending on how the client authoritative model is implemented on the server-side. The message can contain the action that was executed on the client which is then also executed on the server. The message can also contain only the result of the action that was executed on the client. This can be implemented by sending the delta between the previously sent state and the current state of the client. This approach allows the server to skip the execution part of the action and instead it can use directly the client's result of the same action to update the game state on the server and thus improve the server performance. This is only possible in the client authoritative model because it assumes that the messages sent by the client are always valid. For example, a player does an action that opens a chest in the game which gives a random item to the player. In this case the client sends only a message that contains the given random item to the server and the server adds the item to player's inventory without first validating the action on the server side. To reduce the number of messages sent to the server, the client can combine multiple actions into a single action that is then sent to the server.

In the terms of performance client authoritative model offers a good opportunity because the client authoritative model allows moving all the heavy work on the client-side. The obvious problem of the client authoritative model is that it relies on the client-side cheat detection and prevention. In the chapter 2 we have already learned that it is very difficult to implement effective client-side cheat prevention techniques. Without any client-side cheat prevention techniques the client authoritative model is extremely vulnerable for cheating.

A good example of the drawbacks of the client authoritative model is the Super Meat Boy video game. Super Meat Boy is a critically well received platform game that has been released on all major platforms including PC, Xbox 360 and PlayStation 4. In the game the player controls a character that must complete various levels by jumping and running on platforms to reach the end of the level. Each level completion was graded by the elapsed time and the time was also recorded to a global leaderboard. The game became a hit and it sold over million copies. Soon after the leaderboards started filling with times that were impossible to achieve and players started to complain

about cheaters on the forums. The game didn't have any validation for the leaderboard scores and it was lacking any client-side protection from cheating. To make things worse one of the players found that the address, username and password of the leaderboard database were stored in plain-text in the game's code and it was sending the level scores directly to the database [36]. This allowed anyone to modify the leaderboards without even playing the game. This caused some bad publicity and active players abandoned the game due to persistent cheaters.

3.2.3 Server authoritative model

Running the game logic on the client makes the game vulnerable for cheating as we have seen in the previous chapters. Software developers often say that you can never trust the client because it can be corrupted or altered. This is also very much the case in the game development. To avoid this problem the game logic can be fully executed on the server-side. This server-side architecture is known as a server authoritative model where the server has the authority over the clients to what happens in the game. In other words, the game logic is only run on the server and the client sends only actions or key inputs to the server. The server-authoritative model is suitable for many different types of online multiplayer games, such as fast paced first-person shooters, massive multiplayer online games or turn-based games.

In the server authoritative model, the game client doesn't run any actual game logic but instead it immediately sends the actions made by the player to the server. The message that is sent to the server contains only the action that the player is going to make, for example the input key that was pressed or the name of the action to execute. This is significantly different from the client-authoritative model where the client actually sends the result of the action to the server. When the server receives the client's message it executes the action inside the server's game logic. The game logic running inside the server uses the server's own game state that does not depend on any way on the client's game state. After processing the action and updating the game state on the server, the result of the action is returned to the client. The result contains the server's updated game state. The client then uses the result to update its own local game state. The result is also broadcast to other clients by the server in order to keep all clients synchronized of the changes made by the other clients.

In most of the cases the clients can act as "dummy terminals" that only update their visual representation of the game state because the server is running all the game logic and the clients are only relaying the input actions to the server. In other words, the clients can be thought of being spectators

because they can't affect to how the actions are executed. They can only affect to what actions are executed. From the programming point of view this simplifies how the client can be developed because the client doesn't execute any game logic and it only needs to update its visual representation of the game state.

The server authoritative model provides great protection against cheaters because it renders many of the client-side cheats useless. Typical client-side cheats where the game state is modified on the client do not have any effect because the actual game state is stored on the server which cannot access directly by the client. For example, the player uses a cheat that modifies the client's game state so that the player has over 9000 energy instead of the actual value of 10. This change has no effect to the game because on the server's game state the player still has only 10 energy. The next time the client's game state is updated, the energy amount is changed to match the server's energy amount. Another benefit of running the game logic only only on the server and excluding the logic from the client, we can prevent cheaters from decompiling the client executable and accessing the game logic code. This prevents cheaters from learning any information from the game logic that could give them any advantage in the game. A good example of this would bea poker game where the cheater could use the decompiled code to analyze how the playing cards are randomized and thus gaining a significant advantage over other players.

Although the server authoritative model prevents many of the client-side cheats it is still vulnerable for some cheats. One of the potential ways to cheat in a server authoritative model is to modify the messages that are sent to the server. If the server doesn't validate the action before actually executing the action, the attacker can send actions that would not be possible for the player. One example of this is where the cheater is firing a weapon and the client sends a message that contains an action that says the player shot weapon to a specific direction. The cheater could use the man-in-the-middle attack to hijack the message between the client and the server and then change the shooting direction to point to the opponent even if the opponent was not visible in the player's view. The server will not notice this cheat unless it validates the shooting direction by comparing the previous direction to the shooting direction. Another way to cheat in server authoritative model is to use information exposure -cheats on the client-side. The cheater can use the hidden information that is not visible for the player in the client's game state to gain advantage over other players. In an online multiplayer strategy game, the cheater can expose the opponent's unit locations if the server always sends the unit movement action results to all players. The cheater can read the changed unit positions from the client's game state that is stored

in the device memory using the tools described in chapter 2.2. To protect from these kinds of cheats the client should only have partial information of the state of the game. A better approach would be that the server would determine if the result is relevant to other players before sending the results. If not, the server should postpone sending the result to the other players. In this particular case the relevancy would be determined if the unit is visible on the player's view area.

One of the drawbacks of the server authoritative model is that every action needs to be run through the server. This means that there can be delays between when the action is triggered and when the game state is actually changed. These delays can be notable especially in fast paced games and make the game feel unresponsive. To tackle this problem the client can use latency compensation methods like client-side prediction. In the client-side prediction the client assumes that the server will accept the action thus the client executes the action immediately on the client before actually receiving the result from the server. This method requires that the client can predict the result of the action either by running the actual game logic or using a simplified version of it. When the client receives the actual result from the server the, client can correct the predicted result using the server's result. The client-side prediction method is used in many FPS multiplayer games including Half-Life and Counter-Strike [5].

3.2.4 Peer-to-peer model

According to definition by Schollmeier [32], a distributed network architecture may be called a peer-to-peer network, if the participants share a part of their own hardware resources (processing power, storage capacity, network link capacity) with others. The definition emphasizes that these shared resources are necessary to provide the service and content offered by the network and they are accessible by other peers directly, without passing intermediary entities. The participants of such a network are thus resource providers as well as resource requesters [32]. When compared to the client-server model, in the peer-to-peer model clients are directly connected to each other and each act as a client and a server, thus there is no need for a central server.

The peer-to-peer model has been very popular in games. Many of the very first online FPS (first person shooter) multiplayer games were exclusively using P2P model because having a centralized client-server model would be too expensive to maintain due to high performance requirements and slow internet connections. Also, the peer-to-peer model has been used in many MMO (massive multiplayer online) games where there can be hundreds or

even thousands of players playing together at the same time. With the peer-to-peer model games have been able to move the network bandwidth load from the server to the peers because each peer acts as both a client and a server thus removing the need for a centralized server. Unfortunately this approach also includes a problem because in the peer-to-peer model peers are either non-authoritative or there is a single peer that is authoritative. If all peers are non-authoritative then each of them is responsible of their own state. If the peer's state changes it sends a message to other peers to update their state. The problem is that the other peers can't reject the change if they question the validity of the message because they don't have any authority over changes made by the other peers. This makes the peer-to-peer model vulnerable for cheating. Having a single peer with authority doesn't solve the problem completely because it still allows the authoritative peer to cheat. In the next section we will study a peer-to-peer model design that tries to solve this very problem.

In their paper, Jardine and Zappala [19] present a hybrid architecture that uses both a client-server and a peer-to-peer architectures. In this architecture there is a single central server and multiple regional servers. The central server is responsible for managing state-changing moves of the players. All players must send their state-changing moves directly to the central server which are then processed by the central server. The central server determines if each state change is valid and sends a response to the player. If the state change was a valid move, then the central server sends the move to the appropriate regional server which then distributes the move to all players in its region. Regional servers are used to divide the game in regions and they are responsible to distribute all positional moves. The central server controls which players act as regional servers. When a player makes a positional move, it sends the move to the regional server to which the player is connected. Next, the regional server distributes the move to other players connected to the region. Positional moves are never sent to the central server. This hybrid architecture provides great scalability because the peers are distributing the less important updates to other players and the central server handles only important state changing updates, thus significantly reducing the load on the central server. Also, it provides a good protection against cheating because the central server manages the important game state changes and regional server are only managing positional changes that have only low impact to the actual gameplay. Furthermore, it limits cheating only to players that act as regional server because they are distributing the central server's state changing moves. Thus, they can either refuse sending the moves forward or delay them. This can give some advantage to the player that acts as a regional server. To prevent this the players can monitor updates from the

regional server and report if the latency is high or updates are dropped to the central server. After certain amount of reports the central server can then change the player who acts as a regional server.

The peer-to-peer architecture provides a great scalability for games while keeping the operational costs low due to the reduced traffic between the game service and the players. These cost savings can be a major reason especially for smaller game companies and indie developers to choose the peer-to-peer architecture. It also removes the reliance on a single server as anyone can act as a server. From the cheating point of view the peer-to-peer provides a limited protection against cheating. Peers that are acting as servers have the authority on the game state which gives them a possibility to cheat without being detected by the other peers. This can be prevented by having multiple peers acting as authoritative servers and requiring that each of them must first accept the player's state change before it can be actually applied. Another way to prevent cheating is to limit peers so that they can't act as a server in their own region. Both solutions can increase the complexity of the game's architecture. Also, handling cases where the acting server is transferred to another peer can be a very demanding and complex operation. This can cause notable delays in the game and in the worst case the delays can be frequent if peers have poor connections.

Another downside of the peer-to-peer architecture is that that the peers acting as servers consume more network bandwidth and requires more processing power. This can be a problem especially for mobile devices as we found in chapter 3.1.4 that network traffic significantly increases power consumption. Also, mobile devices have less processing power than PCs and consoles. Latencies can be high and there can be sudden connection losses when using a mobile network. Furthermore, peers consume more bandwidth compared to the clients in the client-server architectures because they must send the messages to all other peers. In the client-server architectures the client sends the messages only to the server. Also, players on mobile devices might dislike peer-to-peer architecture because it can increase their operator data plan costs due to the increased bandwidth usage.

3.3 Summary

The use of the server-side architectures provide a significant advantage over cheating. Especially, the server authoritative models provide a great protection against the common cheats that are very effective against client-only architectures. The server authoritative model is easy to implement when the game is designed from the beginning to be server authoritative. Switching

in the middle of the development to the server authoritative model might require a significant investment and can be proven to be difficult to implement afterwards. The better approach in this case would be using the client authoritative model that provides only a limited protection but is much easier to implement afterwards. The main problem of the server authoritative model is the increased latency. Especially when the client device is using a mobile network that usually have notably higher latencies than other networks. To mitigate the latency issues and make the game appear smoother, the client can use the client-side prediction methods.

The peer-to-peer model also provides a suitable protection against cheating, but it is more complex to implement and more vulnerable than the server authoritative model. The peer-to-peer model can be an attractive option when trying to minimize the maintenance costs of the servers. The decentralized architecture of the P2P also means that there is no single point of failure.

The server-side architectures might provide a great protection against cheating, but they are also vulnerable to other problems. One such problem is scaling the server capacity depending on the number of active players. The other issue is that server-side architectures are vulnerable to denial-of-service attacks. In the worst case these issues can lead to a scenario where the game service is unavailable for the players and preventing them from playing the game.

Chapter 4

Client-server architecture for mobile games

4.1 Introduction

In this chapter we will present our architecture for a mobile game where we have taken special attention to prevent cheating in the game. The architecture is designed using the lessons learned from the previous chapters and it is also especially designed to take mobile game requirements into consideration. The architecture is designed for a mobile game that is mainly a single player game, but it should also support online multiplayer features, like leaderboards, guilds and asynchronous multiplayer. In the asynchronous multiplayer part of the game, players are matched against each other and they compete in a shared activity using a turn-based gameplay between two or more players. Since the application for the planned architecture is mobile games where the devices can have limited and unreliable mobile networks, we have focused on efficient communication between the client and the server on the architecture. While designing the architecture we have paid attention to minimize the latency when executing the actions made by the player especially in the single player mode. Also, the architecture is designed so that the game is playable even if the network connection drops for a longer duration.

From the cheaters point of view, we have assumed that they have a full access to the game on the client-side. This means that the cheaters can access and modify the device's memory, run their own code inside the executable and they can listen and alter messages that are sent from the device over the network.

As the cheaters have a full access to the game and in the earlier chap-

ters we have learned that most of the client-side cheat protection methods are unreliable, we have decided to use a server-authoritative architecture to protect against cheaters. But we still want to retain some of the features that the client-side cheat protection methods provide, such as the ability to play offline. Offline play is actually a very important for many games, but especially for mobile games where the mobile networks can be unreliable at times. The player might lose the network connection temporarily to a blind spot while traveling, for example in a subway tunnel or inside a building. Thus, it is very important for us to make the game playable even when there is no network connection but only for a limited time. Because we have chosen to use the server authoritative model, we need to design the architecture in a way that is especially efficient in network performance. We want to minimize the number of messages sent between the client and the server because constant communication between the client and the server consumes the mobile device's battery faster as we have learned in the chapter 3.1.4. Also, having fewer messages will reduce the bandwidth usage of the game. The downside of having fewer messages is that the message payloads are bigger because we need to send more data per message. This will have a negative impact to the latency of the message when the network speed is limited. In the next section we justify and present our design for the architecture of the game.

4.2 Design and architecture

The simplest solution would be to use a server authoritative model where all the game logic is run on the server, but this would mean that the game would require a constant connection between the client and the server. A fully server authoritative model would prevent players from using any client-side cheats, but it would also mean that offline play would not be possible. A more suitable solution would be to use a mixed authority model where one part of the game logic is client authoritative and the other part would be server authoritative. In our example game, the single player mode would be client authoritative and the multiplayer mode would be server authoritative. This would satisfy the requirement that the game must be playable in a single player mode when there is no network connection because the single player game logic is only run on the client. Also, it would reduce the amount of traffic between the client and the server because when using the client authoritative model, the client can synchronize the client's game state less frequently with the server.The client could synchronize the client authoritative game state only before entering to the multiplayer mode. But it would also mean that the client authoritative game state would vulnera-

ble for client-side cheats as we have learned in the previous chapters. This could have also a significant effect on the multiplayer if the player can achieve things in the single player mode that are usable in the multiplayer mode and thus gain an advantage against other players. The player could use cheats to achieve more powerful weapons in the single player mode and then use these weapons in the multiplayer mode against other players. This would have a significant effect to the game balance and eventually it would lead to a bad user experience for the other players. Also, splitting the game logic in two parts would mean that the same game code must be implemented for both the client authoritative model and the server authoritative model.

Because splitting the game logic to the client authoritative and the server authoritative parts is not suitable for our game we need to find an alternative solution that would work with the server authoritative model. Instead of using a fully server authoritative model we decided to use a client optimistic server authoritative model. It resembles the client-side prediction method that was introduced in the chapter 3.2.3. The client-side prediction method is used to reduce the latency between triggering and executing the action when the it needs to be validated on the server. In the client optimistic model, the client processes the action first on the client and then stores the action so that it can then later be sent to the server. So, instead of sending the action immediately to the server it is delayed until the client decides otherwise. Finally, when the client sends the actions to the server in a single, batched message the server processes the actions and updates the server's game state. After processing the actions, the server sends the new game state to the client which then determines if the client's game state is in sync with the server's game state.

Using the client optimistic server authoritative model, the game can be played in an offline mode because the actions can be stored on the client and it can send them to the server after the connection with the server is restored. As the single player mode has no effect to other players, this approach allows reducing the number of messages send to the server, because the actions can be delayed and send to the server in a batch thus reducing the number of messages sent to the server.

In order to make the client server optimistic server authoritative model to work the client and the server must stay synchronized. This means that the client's game state must match to the server's game state of the client. To keep the game states synchronized between the client and the server, the game logic needs to be deterministic. This means that the game logic on the client and the server must always produce the same output when processing the game actions. Otherwise they get out of sync and subsequent actions may fail when processed. The client can calculate a checksum value of the client's

game state and send it to the server along the actions to prevent the game states going out of sync between the client and the server. The server can calculate the checksum value of the client's game state stored on the server and compare it to the checksum value that the client sent to the server. If the checksums don't match, then the client's game state is overwritten with server's game state.

In the multiplayer mode, the client optimistic model does not work because the server must first process the actions. Otherwise the client might try to execute an action that is no longer valid due to another opponent's actions. To avoid this problem the game must fall back to the server only authoritative model in the multiplayer mode.

4.3 Implementation details

The game logic is designed around the model-view-controller (MVC) concept [Bernier], where the game logic is divided into interconnected parts. The *model component* contains the player's game state data and the *view component* contains the user-interface logic. The *controller component* is responsible to process the actions triggered by the *view component* and update the *model component*. The MVC concept and its interactions are presented in the figure 4.1 below. The MVC concept allows us to share the model and the controller components with the server. The client game logic which makes it easier to create a deterministic game logic implementation, that is required for the client optimistic server authoritative model. Both the client and the server game logic must output the exact same results with the same input values. The *view component* is only implemented on the client side.

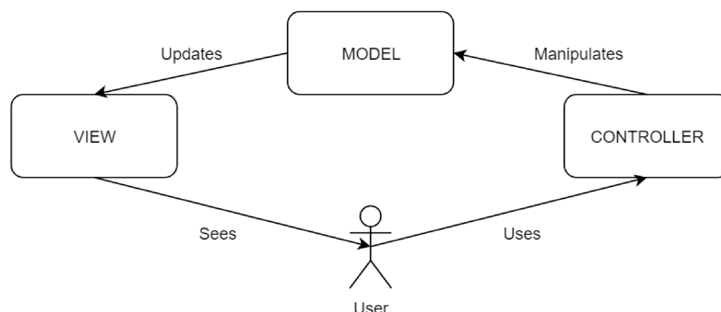

Figure 4.1: The MVC concept and the interactions between the components.

The *controller component* listens for actions that are triggered by the *view component*. A common case would be where the player presses a button in

the user-interface that triggers a specific action. Once the controller receives an action, it creates a command object that stores the triggered action and its parameters. It also stores the timestamp value when the action was triggered on the client using the internal clock of the game. The game clock is always synchronized with the server at the beginning of each play session. The timestamp value is later used on the server to determine if the player tried to cheat by changing the device clock. There are two different types of commands: *ServerClient* and *ServerFirst*. The type of the command depends on the action that was triggered. Actions that only affect the player's game state and has no effect to other player's game states are always marked as *ServerClient* commands. This means that they don't depend on the server's result and they can be immediately processed by the client's game logic. Actions that also have an effect to the other players are marked as *ServerFirst* commands. The *ServerFirst* commands are always first processed on the server and only after that on the client.

If the action is marked as a *ServerClient* command, then the action immediately processed by the client's game logic. After processing the action on the client, it calculates a checksum value from the client's game state. This checksum value is stored to the command object. Later when the command object is sent to the server the checksum value is used to determine if the client- and the server game states are still matching after the action on the client and the server are processed. In other words, we can use the checksum value to detect if the player tried to cheat on the client by changing the game logic or modifying the game state. The checksum values should match after processing the actions due to the deterministic game logic model. The checksum value can be a simple numeric value or a small byte array that is calculated from the game state. In our project, that adapts the client optimistic model, uses a checksum value instead of comparing the full game state to reduce the amount of data sent between the client and the server. Next, the client stores the *ServerClient* command object to an array inside the controller. The *ServerClient* command object is then later sent to the server for validation and processing.

If the action is marked as a *ServerFirst* command, the action needs to be executed first on the server and only after the server has validated and processed the action the client can process the action. Before the *ServerFirst* command can be executed on the server we need to execute the pending *ServerClient* commands on the server. Otherwise the server's game state would not be in a sync with the client's game state as the commands would be processed in a wrong order. To prevent this, the client creates a command batch message that is sent to the server. The command batch message contains an array of *ServerClient* commands and a single *ServerFirst* command.

The idea is that the server first executes all of the *ServerClient* commands and only after that the *ServerFirst* command. So, when the client is processing an action that is marked as a *ServerFirst* command, it adds all the stored *ServerClient* commands that are pending for the server execution to the command batch message along with the currently triggered action that was marked as *ServerFirst* command. After creating the command batch message, the client sends the message to the server which validates and processes all the commands inside the command batch message. First, the server executes all the *ServerClient* commands and if they were executed successfully without any errors it then executes the *ServerFirst* command. For each command the server validates that the command action and its parameters are valid. Also, the server checks that the timestamp value that is stored to the command is valid. The timestamp value must be greater or equal to the last command that was processed by the server and the timestamp value must be less or equal to the current timestamp of the server's internal clock. Otherwise we suspect that the player tried to cheat the game clock and the validation fails. After processing the command's action, the server calculates a checksum value from the player's game state that is stored on the server. If the action was marked as a *ServerClient* command, the server compares the checksum value to the checksum value that was stored to the command after the action was processed on the client. If the validation step succeeds and the checksum values are matching, the server acknowledges the action and saves the changes to the player's game state. In this case the server sends a result indicating that the commands were executed on the server successfully. When the client receives the result that was successful it still needs to process the *ServerFirst* command on the client and compare the checksum values to see if the client's game state is matching to the server's game state. If the checksum value is not matching on the client, the client triggers a rollback procedure where it downloads the last valid game state from the server. If the server result was successful,the pending *ServerClient* commands, that were sent with the command batch message, are removed from the controller. If the validation step failed or if the checksum values are not matching on the server, we assume that the player tried to cheat and reject the changes. If the changes are rejected, the server responds with a result indicating that a command was rejected and that the client needs to rollback to the last valid game state that is stored on the server. In the figure 4.2 below, we have presented a flowchart of the implemented client optimistic server authoritative model.

There are also cases where we want to send the pending *ServerClient* commands to the server even if there is no *ServerFirst* command to execute. In this case we send the command batch message to the server without the

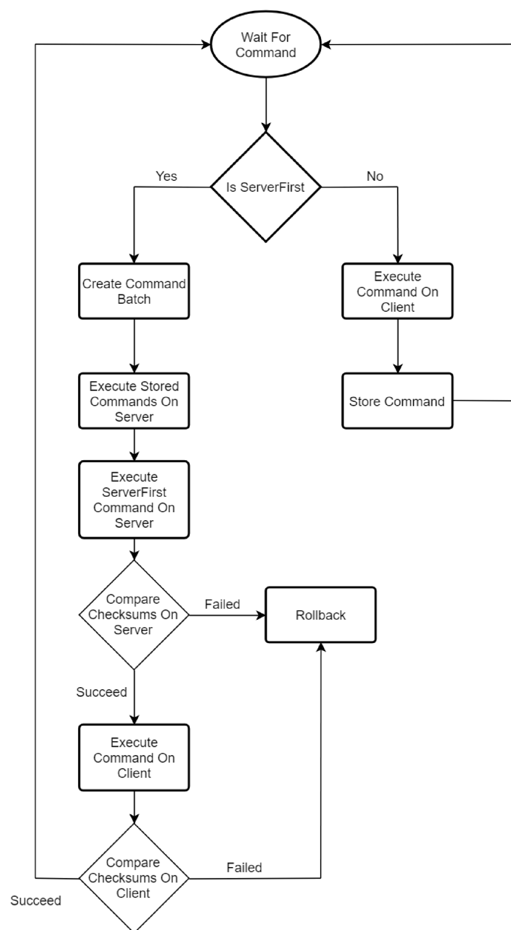

Figure 4.2: The flowchart of the client optimistic server authoritative model.

ServerFirst command. An example case where this is needed could be that the game should to support playing on multiple devices. This requires that the client send the pending commands to the server at certain interval to keep the player's game state on the server in sync with the client's game state. When the player starts playing the game with the other device then we can check if the last validated checksum on the client matches to the server's game state. If not, then we can assume that player had played with other device and download the latest state from the server.

4.4 Summary

We aimed to create an architecture for a game that supports both single player and multiplayer modes without compromising the protection against cheaters and the presented client optimistic server authoritative model performs these requirements excellently. The client optimistic server authoritative model makes it possible to play the game in a single player mode even when there is no network connection for a longer duration. Because the actions are stored and later sent to the server for validation and processing, it prevents the cheaters from using most of the client-side cheat methods in the single player mode. The server can force the player's game state to be overwritten or rolled back if the server detects mismatch in the client and the server checksum values. Also, the client optimistic model allows the client to combine multiple actions into a single message that is sent to the server which reduces the number of messages exchanged between the client and the server. This can significantly reduce the load on the server and increase the amount of supported concurrent players.

There are also drawbacks in the client optimistic server authoritative model when it is compared to the server only authoritative model where all the game logic is run only on the server. Because the client optimistic model requires that the client must be able to process the actions and run the game logic on the client it exposes the game's logic for the cheaters. Because the client and the server game logic are deterministic the cheaters can decompile the client's executable and learn how the game logic functions. The cheaters can use this information to predict how the game behaves in different situations and thus gain an advantage in the game. To prevent these kind of cheats the game client should use code obfuscation to make it harder for cheaters to learn the client's game logic. Another problem is that the player's game state must be fully available for the client in order to being able run the game logic. In other words, the game state must be also stored on the client instead of having it stored only on the server and having only a partial game state stored on the client. The cheaters can expose the information that is stored in the client's local game state and use the information to cheat in the game. For example, in a single player mode they could determine if they should attack the AI opponent by first checking the AI opponent's power from the client's local game state. Unfortunately, there is no definite way to protect the client's local game state against information exposure cheats. The client could use methods presented in the chapter 2 to protect the critical game state data, but they don't guarantee that the cheaters wouldn't be able expose the protected game state data.

Chapter 5

Conclusion and future work

This thesis has studied the different approaches how to prevent cheating in video games and evaluated their usefulness, complexity and performance. We have also focused on how these approaches can be used in the mobile games where there are special restrictions and limitations. Our analysis of the client-side protection methods shows that they provide only a limited protection against different cheat methods. The memory protection methods provide an effective way to protect against memory editing tools and they are also easy to implement. But they have also a significant impact on the performance. The constant encryption and decryption of the protected variables consumes a lot of processor power. This means that only the most important variables can be protected. Also, the encryption algorithms are usually fairly trivial in order to reduce the impact on the game's performance. This means that the cheaters can crack the encryption algorithm after some time. Time cheating causes serious problems for the games because it is very easy to carry out. As our investigation shows that the player can cheat the time by simply changing the system time without using any external tools. There is no reliable way to detect the changes in the system time on the client-side. It is even harder to detect time cheats that are using code injection to change the behaviour of the timing functions in the game. Finally, the code obfuscation provides an efficient way to strengthen the protection and to make it more difficult for cheaters to use different reverse engineering techniques. The code obfuscation can have a slight impact on the performance of the game, but it doesn't require any extra work to implement because the obfuscation process is automatic. The downside of the code obfuscation is that it only makes the reverse engineering process more difficult, but it doesn't prevent it.

On the other hand, our studies show that the server-side protection methods provide a considerably better protection against the cheaters. The client-server model provides a better protection against cheating by moving the

game logic on the server. In the server authoritative model all the game logic can be run only on the server thus making the client only to display the current state of the game. This makes most of the client-side cheats ineffective because they can't affect to the game logic that is run on the server. The server can also validate the actions sent by the client in order to prevent cheats that tamper the messages between the client and the server. From the performance point of view the server-side protection methods have no effect to the game's performance. But they have a significant impact to the latency and battery usage. Especially on the mobile devices the added latency from the client-server communication is noticeable. Also, constant communication between the client and the server can quickly drain the batter of the mobile device. In the list below, we have identified several key points and limitations of the different protection methods.

- The client-side protection methods provide only a limited protection against the cheaters. With enough time the cheaters can either crack the protection methods or circumvent them.

- The client-side protection methods can have a significant impact to the game's performance, usually only part of the game can be protected without having notable performance penalties.

- Detecting time cheating using client-side methods is unreliable. The only way is to prevent time cheating reliable is to use a server synchronized time.

- The code obfuscation makes it difficult to reverse engineer the cheat protection methods, but the code obfuscation alone doesn't provide any protection against cheating. The code obfuscation is used to make the reverse engineering process too time consuming for the cheaters.

- Only the server-authoritative model can provide almost impenetrable protection against cheating. But it is still vulnerable for information exposure and reflex augmentation cheats.

- The client-server communication increases the latency of the actions thus the client needs to be able to compensate the added latency.

- Cheating in the mobile games is not different from the other games. The same tools exist for both the mobile and the desktop platforms.

In the chapter 4 we presented our architecture for a mobile game that was designed using the using the lessons learned from the other chapters. The

architecture takes into account the limitations of the mobile devices while still being almost cheat-proof. It even allows playing the game when there is no network available while still being a fully server authoritative. Our initial results have been promising and we haven't found any significant flaws in the design of the architecture that could not be resolved.

Overall, this thesis has been able to provide the analysis of the cheat methods that are commonly used in the games. We have also identified the problems of the client-side and the server-side cheat protection methods. We have been also able to evaluate the performance and latency impact of the different cheat protection methods. Also, we have pointed out how the server-side cheat protection methods provide significantly better protection than the client-side protection methods.

We have mostly concentrated on the commonly used cheat techniques in this thesis. It may be further studied to analyze more different cheat techniques. For example, bots are a major problem in many games and especially in MMO games. The bots can have significant impact to the game's balance and even break the economy of the game. Furthermore, it would be interesting to study if the statistical techniques of the machine learning could be used in the cheat detection.

Bibliography

[1] clock_gettime. http://pubs.opengroup.org/onlinepubs/009695399/functions/clock_gettime.html, 2017. Accessed: 2018-05-13.

[2] gettimeofday(2) - Linux manual page. http://man7.org/linux/man-pages/man2/gettimeofday.2.html, 2017. Accessed: 2018-05-13.

[3] Barker, E., and Roginsky, A. Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. *NIST Special Publication 800* (2011), 131A.

[4] Barnes, R., Thomson, M., Pironti, A., and Langley, A. Deprecating secure sockets layer version 3.0. Tech. rep., 2015.

[5] Bernier, Y. W. Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization.

[6] Callegati, F., Cerroni, W., and Ramilli, M. Man-in-the-middle attack to the HTTPS protocol. *IEEE Security and Privacy 7*, 1 (2009), 78–81.

[7] Cano, N. *Game hacking : developing autonomous bots for online games*. 2016.

[8] Chell, D., Erasmus, T., Lindsay, J., Colley, S., and Whitehouse, O. *The Mobile Application Hacker's Handbook*. Wiley, 2015.

[9] Chikofsky, E. J., and Cross, J. H. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software 7*, 1 (jan 1990), 13–17.

[10] Cortesi, A., Hils, M., Kriechbaumer, T., and contributors. mitmproxy: A free and open source interactive HTTPS proxy. https://mitmproxy.org/, 2010–. [Version 4.0].

[11] Dierks, T., and Rescorla, E. RFC 5246 - The transport layer security (TLS) protocol - Version 1.2. In *Network Working Group, IETF* (2008), pp. 1–105.

[12] Evans, C., Palmer, C., and Sleevi, R. RFC 7469 -Public Key Pinning Extension for HTTP (Certificate Pinning).

[13] Freier, A., Karlton, P., and Kocher, P. RFC 6101 - The Secure Sockets Layer SSL Protocol Version 3.0. In *Internet Engineering Task Force (IETF)* (2011), pp. 1–67.

[14] Goldberg, A., Buff, R., and Schmitt, A. A Comparison Of Http And Https Performance. *Computer Measurement Group, CMG98* (1998).

[15] Heijnen, E. Cheat Engine. https://www.cheatengine.org/aboutce.php, 2016. Accessed: 2018-05-10.

[16] Hoglund, G., and McGraw, G. *Exploiting software : how to break code.* Addison-Wesley, 2004.

[17] Hoglund, G., and McGraw, G. *Exploiting Online Games: Cheating Massively Distributed Systems.* Addison-Wesley, 2007.

[18] Hohpe, G., and Woolf, B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley, 2003. p. 184.

[19] Jardine, J., and Zappala, D. A hybrid architecture for massively multiplayer online games. *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games - NetGames '08* (2008), 60.

[20] Kotipalli, S. R., and Imran, M. A. *Hacking Android : explore every nook and cranny of the Android OS to modify your device and guard it against security threats.* 2016.

[21] Kshemkalyani, A. D., and Singhal, M. *Distributed computing: Principles, algorithms, and systems*, vol. 9780521876. Cambridge University Press, 2008.

[22] Naylor, D., Finamore, A., Leontiadis, I., Grunenberger, Y., Mellia, M., Munafò, M., Papagiannaki, K., and Steenkiste, P. The Cost of the "S" in HTTPS. *Proceedings of the 10th ACM*

International on Conference on emerging Networking Experiments and Technologies - CoNEXT '14, April 2018 (2014), 133–140.

[23] Nolan, G., and Zukowski, J. *Decompiling Java.* Apress, 2004.

[24] Peterson, W., and Brown, D. Cyclic Codes for Error Detection. *Proceedings of the IRE 49*, 1 (1961), 228–235.

[25] Press, C. U. cheat - meaning in the cambridge english dictionary. https://dictionary.cambridge.org/dictionary/english/cheat, 2016. Accessed: 2018-03-24.

[26] Pritchard, M. How to Hurt the Hackers : The Scoop on the Internet Cheating and How You Can Combat It. http://www.gamasutra.com/view/feature/131557/how_to_hurt_the_hackers_the_scoop_.php, 2000. Accessed: 2018-04-25.

[27] Prowell, S., Kraus, R., Borkin, M., Prowell, S., Kraus, R., and Borkin, M. Man-in-the-Middle. In *Seven Deadliest Network Attacks*. Elsevier, 2010, pp. 101–120.

[28] Rescorla, E. RFC 2818 - HTTP Over TLS. Tech. rep., 2000.

[29] Rescorla, E., and Modadugu, N. Datagram transport layer security version 1.2. Tech. rep., 2012.

[30] Schneier, B. *Applied cryptography : protocols, algorithms, and source code in C.* Wiley, 1996.

[31] Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C., and Ferguson, N. Twofish: A 128-bit block cipher. *NIST AES Proposal 15* (1998), 23.

[32] Schollmeier, R. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. *Proceedings - 1st International Conference on Peer-to-Peer Computing, P2P 2001* (2001), 101–102.

[33] Sheffer, Y., and Holz, R. P. saint-andre," recommendations for secure use of transport layer security (tls) and datagram transport layer security (dtls). Tech. rep., BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015,¡ http://www. rfc-editor. org/info/rfc7525, 2015.

[34] Sheffer, Y., Holz, R., and Saint-Andre, P. Summarizing known attacks on transport layer security (tls) and datagram tls (dtls). Tech. rep., 2015.

[35] Shewmaker, J. Analyzing dll injection. *GSM Presentation* (2006).

[36] Something Awful Forums. Coding Horrors: Turn screen off to reveal nemesis. `https://forums.somethingawful.com/showthread.php?noseen=0&threadid=2803713&pagenumber=258#post398884189`, 2011. Accessed: 2018-04-13.

[37] Standard, N.-F. Announcing the advanced encryption standard (aes). *Federal Information Processing Standards Publication 197* (2001), 1–51.

[38] Thiel, D. *IOS application security : the definitive guide for hackers and developers*. 2016.

[39] Turner, S., and Polk, T. Prohibiting secure sockets layer (ssl) version 2.0. Tech. rep., 2011.

Appendix A

Code samples

```
1  using System.Security.Cryptography;
2  using System.IO;
3  using System;
4  using System.Text;
5
6  public class DataSecurity
7  {
8      static byte[] EncryptData(string data, string key) {
9          using (Aes aes = Aes.Create()) {
10             aes.Mode = CipherMode.CBC;
11             aes.Key = Encoding.ASCII.GetBytes(key);
12             aes.GenerateIV(); // Random initialization vector
13
14             var encryptor = aes.CreateEncryptor(aes.Key, aes.IV);
15             byte[] encrypted;
16
17             using (var memoryStream = new MemoryStream())
18             using (var cryptoStream = new CryptoStream(memoryStream,encryptor,CryptoStreamMode.Write))
19             using (var steamWriter = new StreamWriter(cryptoStream)) {
20                 steamWriter.Write(data); // Write actual data
21                 encrypted = memoryStream.ToArray();
22             }
23             // Combine initialization vector and encrypted data
24             var combined = new byte[aes.IV.Length + encrypted.Length];
25             Array.Copy(aes.IV, 0, combined, 0, aes.IV.Length);
26             Array.Copy(encrypted, 0, combined, aes.IV.Length, encrypted.Length);
27             return combined;
28         }
29     }
30
31     static string DecryptData(byte[] combined, byte[] key) {
32         using (Aes aes = Aes.Create())
33         {
34             byte[] iv = new byte[aes.BlockSize / 8];
35             byte[] encrypted = new byte[combined.Length - iv.Length];
36
37             // Read initialization vector and encrypted data
38             Array.Copy(combined, iv, iv.Length);
39             Array.Copy(combined, iv.Length, encrypted, 0, encrypted.Length);
40
41             aes.Mode = CipherMode.CBC;
42             aes.Key = key;
43             aes.IV = iv;
44
45             ICryptoTransform decryptor = aes.CreateDecryptor(aes.Key, aes.IV);
46
47             using (var memoryStream = new MemoryStream(encrypted))
48             using (var cryptoStream = new CryptoStream(memoryStream,decryptor,CryptoStreamMode.Read))
49             using (var streamReader = new StreamReader(cryptoStream)) {
50                 return streamReader.ReadToEnd(); // Reads actual data from the decrypting stream
51             }
52         }
53
54     }
55 }
```

Figure A.1: Encrypting and decrypting data using AES cipher.