Comparative Study of Anti-cheat Methods in Video Games

Samuli Lehtonen

Master's thesis
UNIVERSITY OF HELSINKI
Department of Computer Science

Helsinki, March 7, 2020

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Samuli Lehtonen

Työn nimi — Arbetets titel — Title

Comparative Study of Anti-cheat Methods in Video Games

Oppiaine — Läroämne — Subject		
Computer Science		
Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	March 7, 2020	71 + 48 as appendices

Tiivistelmä — Referat — Abstract

Online gaming is more popular than ever and many video game companies are reliant on the cash flow generated by online games. If a video game company wants its game to be successful, the game has to be resilient against cheating, the presence of which can ruin an otherwise successful game. Cheating in a video game can bankrupt an entire company as the non-cheating players leave the game because of unscrupulous individuals using cheats to gain an unfair advantage. Cheating can also involve criminal activity where maliciously acquired in-game items are traded against real money online. Commercial cheat programs are sold on online black markets and are available even to players who have no deep technical knowledge. The widespread availability and easy accessibility of cheats compounds the issue.

This thesis will categorize different anti-cheat techniques and give a brief history of anti-cheat starting from the early 1980s. The history section describes how the fight against online cheating began and how it has evolved over the years.

This thesis will compare different anti-cheat methods, both on the client-side and server-side, and draw conclusions about their viability. It will also look at scenarios where different anti-cheat methods are combined to create more powerful systems. All the anti-cheat methods will be evaluated based on five different criteria on a scale of 1 to 4, with one being the lowest score and four the highest. The thesis will use a custom-built client-server game as an example to illustrate many of the anti-cheat techniques. Requirements of different types of games, such as first-person shooters and strategy games, will also be considered when reviewing the anti-cheat techniques.

Lastly, the thesis will look into the future of anti-cheat and introduce video game streaming and the use of machine learning as possible new solutions to tackle cheating. The conclusion will summarize the advantages and disadvantages of different methods and show which techniques are preferable based on the analysis.

ACM Computing Classification System (CSS):

Security and privacy → Software and application security → Software security engineering

Security and privacy → Software and application security → Software reverse engineering

Security and privacy → Human and societal aspects of security and privacy → Economics of security and privacy

Avainsanat — Nyckelord — Keywords
anti-cheat, reverse engineering, comparison, game, cheating

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction

Cheating has been a problem in multiplayer online video games since they were introduced. To provide a good game experience, game developers had to develop systems to detect and prevent cheating. These systems are collectively called anti-cheat programs. Building anti-cheat systems has always been a cat-and-mouse game because new cheats are constantly being developed to avoid detection by the game's anti-cheat technology. Detecting newly made cheat programs is the fundamental problem of anti-cheat development. For example, if there is cheat-detection code in the game client and the cheaters discover a new way to circumvent the detection, then the developer has to upgrade the anti-cheat again and this cycle will repeat as long as the game is online and active. In this aspect, anti-cheat development largely resembles anti-virus development.

Video games have become more and more popular and the businesses of many companies revolve solely around their online games and the constant revenue that they generate. If a game has a large number of cheaters, it can ruin the reputation of the game or even that of the entire company, which can drive away other paying customers. The business impact of not having a proper anti-cheat system can be very high, which makes cheat detection ever more important. Nowadays many game tournaments also have prizes that range from tens of thousands to millions of dollars, so it is increasingly necessary to ensure the fairness of these competitions. Cheating can often unfairly influence the virtual economy of the game. The creators of the cheating programs also sometimes sell their cheats to make large amounts of money. Digital security company Irdeto reported in 2018 that 77% of gamers surveyed in China reported that cheating in online games happens frequently and 68% of people surveyed in South Korea described the same [58]. As a result of widespread cheating, the South Korean government took legislative action to make cheating officially illegal and punishable with fines and prison time [28]. In addition to this, there have been at least a hundred arrests of cheaters in China in 2018. These arrests were the result of cooperation between the game company Tencent and the Chinese police [2].

A successful online game needs proper anti-cheat, and it is important to provide a metric for evaluating different anti-cheat methods. This thesis will focus on comparing different anti-cheat methods, both on the server-side and the client-side. The thesis will also investigate combining server and client-side methods. Even though single-player games also have cheating programs made for them, the work will only cover online games. Cheating in single-player games is not interesting for this research as the effect of the cheating is only limited to the player themselves. The thesis will look into the history and the current trends of anti-cheat. By looking at the history of anti-cheat, the thesis will show what methods have been used over time and whether they are still preferred in today's video games.

This work will shed light on why some methods have become increasingly popular at the expense of other methods. The thesis will provide developers with valuable information about the strengths and weaknesses of different anti-cheat methods. This knowledge will help developers and other stakeholders decide what kind of anti-cheat system they should implement for their games.

2 Methodology

Different anti-cheat methods will be evaluated in terms of resistance to tampering, ease of implementation, lack of overhead, privacy, and suitability for a wide variety of games. The techniques will be rated from 1 to 4 in all categories. A higher score means that the method is better in the selected criterion.

Resistance to tampering will evaluate how difficult it is to bypass the examined anti-cheat method. This evaluation will take into account the different skill levels of cheaters and the skills required to bypass the anti-cheat. The evaluation is relevant because the developers need to be able to determine if it is worthwhile to even apply the method or whether it should be applied in conjunction with some other method.

Ease of implementation will weigh the skills needed to implement the selected anti-cheat technique. It will include considerations of the development time and other resources that may be required. Because many smaller companies are developing online games, it is useful to consider whether specialized expertise is required to implement an anti-cheat method and if it is worth it in relation to other anti-cheat techniques. Many companies might need to hire developers with special expertise in these areas, and the benefit has to be justified.

The lack of overhead will consider how much the anti-cheat method affects the functionality of the game in terms of performance, which in this case means the frame rate and latency in the game. If there is a high possibility of a method having an effect on the game performance, a lower score will be given.

The non-invasiveness angle will evaluate if an anti-cheat method is a risk to the privacy of the users. Often invasiveness means scanning personal data in the users' computers or other actions that can jeopardize the users' privacy. The evaluation will also consider whether the data gathered by the anti-cheat could be misused. People are getting more and more privacy-conscious online and an invasive anti-cheat can affect the reputation of the game; as such, the developers will need to be able to evaluate if it is still worthwhile to apply the selected technique.

Suitability for a wide variety of games will score the method based on it being general enough so that it can be applied to many different game types instead of only fitting a certain type of game. Developers need to be able to know whether a method is a good fit for their game and if the same anti-cheat system can be used for other games under development. How widely a certain method is applied today will not be used as an indicator because many games use prebuilt anti-cheat systems that rely on the same approaches. It is also very hard to accurately estimate how many games use certain anti-cheat techniques because the anti-cheat systems are often proprietary and it is in the interests of the developers to conceal the methods they use.

Most commercial anti-cheat software is proprietary, and the developers want to protect their trade secrets, so the solutions examined in this thesis will be based on public research and academic literature. Graphs and charts will be used to illustrate concepts and example code will be provided as needed. Third-party tools will be used in the reverse engineering and packet capturing section.

Ghidra is a cross-platform tool that was developed by the National Security Agency (NSA) for reverse engineering all types of binaries [19]. It will be used as the main reverse engineering tool when inspecting the game binary.

WireShark is a widely used open-source cross-platform tool for inspecting network traffic and capturing packets [62]. It will be used to examine the traffic between the game client and the server.

A small game was specifically developed for this thesis in order to test different anti-cheat systems and will be used to show how to implement the different methods and how they affect cheating. The use of an example game will provide valuable real-world data to evaluate various anti-cheat methods. In the end, both literature and data provided by the game will be used to draw conclusions about the viability of the many anti-cheat approaches. It would of course be even better to analyze real-world games but most multiplayer games are many gigabytes in size and have many layers of complexity, which means that it would take an enormous amount of time to go through and analyze them. This would derail the thesis from its main topic. The use of real games as examples instead of the purpose-built example game is also problematic in the legal sense, which gives another reason to use a custom game specifically developed for this thesis. Our small game cannot be used as an example in all the methods, however. For instance, making a kernel-based anti-cheat driver would be worth an entire thesis by itself, and the low-level implementation details would vary between the different operating systems. All code and tests for the small game were run on macOS Mojave 10.14.4, but there should be no issues with other platforms because all the tools are also available for Windows and Linux-based systems. Simple and Fast Multimedia Library (SFML) is used for the game's graphics

rendering and networking, so SFML is required to build the game server and client.

2.1 Example game client and server

The example game consists of the client and the server. The idea of the game is that the server decides on a random number between zero and one hundred and the players try to guess this number turn by turn. After a player attempts a guess, if the result is wrong then the players are told whether the guess was too high or too low, and then the turn goes to the next player. Every match has three players and the first player to guess the number wins the game. In the course of the thesis, parts of the code in the game will be referred to and some parts will be edited and compared with the original version to give a fact-based illustration of effects that different systems have.

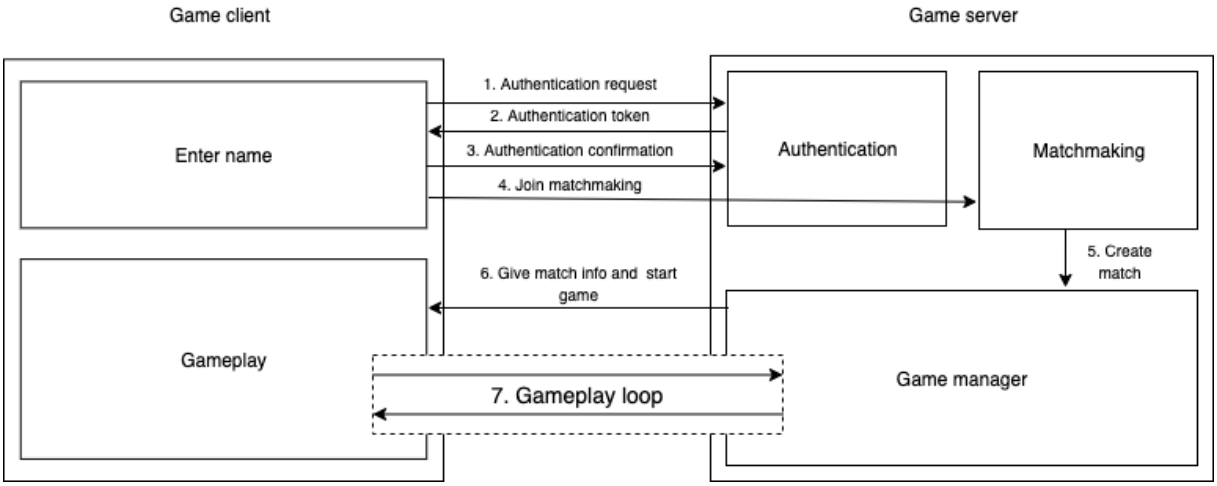


Figure 1: Architecture of the example game

Figure 1 represents the architecture and the flow of the game. In the step "1. Authentication request", the user types his username and presses enter, which causes the client to send an authentication request to the server. The server checks this request, and in step "2. Authentication token" the server assigns a unique token to the client which is used for further communications and for distinguishing the players from each other. In step "3. Authentication request", the client confirms its authentication status by sending the received token to the server and the server marks the client as authenticated, which means that it can join the matchmaking. In step "4. Join matchmaking", the client sends a message to the server telling that it wants to be placed in the matchmaking queue. The server then adds the player in matchmaking. The step "5. Create match" occurs when

there are at least three players in the matchmaking queue. The server takes these three players out of the queue and creates a match between them. The creation of the match involves generating the number that the players have to guess and informing the clients that a match is starting. In step "6. Give match info and start game", the server sends an indication to the client that the game has begun and the client changes to the game screen. The final step "7. Gameplay loop" is the main game loop that consists of messaging between the client and the server. At the start of the game loop, the first player guesses the number and the server checks whether the number was correct or not. If the number was not correct, then the server informs all players if the number was too high or too low. After this, the turn moves to the next player and the cycle repeats until someone guesses the right number, which starts a new round with a newly generated random number.

3 History of anti-cheat in video games

The history of cheating in multiplayer video games goes back at least to the 1980s, but it is hard to pinpoint the first anti-cheat system, especially in online games. Ultima Online was the first massively multiplayer online role-playing game (MMORPG) that became popular in the western part of the world, and even today it has a fair number of players [31]. The current daily player count is estimated to be around 5,000 in the official servers [52]. This is not the whole truth, however, because many players also play on private unofficial servers that have no reliable player data available. There is not much public information about the anti-cheat system in Ultima Online, but it did not have a client-side anti-cheat at all. However, most of the game logic was handled on the server-side, so cheating was not trivial. Client-side cheats such as keeping the game clock always on daytime were easy to implement due to the lack of client-side anti-cheat, but they did not affect other players. Later on, players discovered bugs that cheats could exploit to gain in-game benefits such as money and resources. For example, in 2017 online magazine Vice interviewed a man who had made his living for 20 years by cheating in games and selling the gained goods on eBay and Chinese marketplaces. In the article, the man states that his career began with Ultima Online where he tried to find weaknesses in the game protocol that allowed for unintended actions. In 1997, he managed to find an exploit that allowed him to delete other players' houses and take them over. The man ended up selling the houses for an average price of 2,000 dollars per house [16].

As online games became more and more popular, cheating also got more sophisticated and anti-cheat systems followed the trend. The first major commercial anti-cheat was PunkBuster, which was developed by Tony Ray's company Even Balance after he had bad experiences with cheaters in the game Team Fortress Classic. A few years after the initial

release of PunkBuster, it was used in about 80% of the matches in the popular shooting games such as Return to Castle Wolfenstein and Quake III [7]. PunkBuster is integrated in both the game client and the game server. The PunkBuster client performs various functions such as real-time scanning of the memory. The client tries to find cheats from the memory and compare them to signatures in a built-in database. The client can also calculate MD5 checksums of the files and send them to the server to validate whether the files are modified. If a player gets caught cheating in one server that uses PunkBuster, he is banned from all servers using PunkBuster. PunkBuster also has an option for hardware bans that can calculate hardware ID from a computer's components and block the player based on that [38].

Shortly after the introduction of PunkBuster, the game company Valve rejected Even Balance's offer to use PunkBuster in their games and started to work on their own long-term anti-cheat solution to respond to the cheating epidemic in their Counter-Strike first-person shooter game. The problem of cheating was becoming consistently worse and players were quitting the game as a result [41]. The main cheating method was the use of a "wallhack", which meant that the cheaters could see other players through walls. This gave an unfair advantage to the cheaters as it allowed them to exploit the knowledge of other players' positions and get favorable shooting positions. For example, the cheater might look where the enemy is and go behind them and shoot them in the back. Wallhack is often implemented by modifying the game engine to render walls with semi-transparent textures.

In 2002, Valve Anti-Cheat (often referred to as VAC) was finally deployed in Counter-Strike [34]. The basic idea of the Valve Anti-Cheat is very similar to Even Balance's PunkBuster system. Valve Anti-Cheat can detect cheats such as OpenGL graphics hacks that were often used for implementing wallhacks. Anti-cheat systems have also had their fair share of scandals, often related to the way they scan and send information about users' computers. For instance, in 2014 it was reported the Valve Anti-Cheat accessed the DNS cache on the users' computers and sent back data about it. Gabe Newell, the CEO of Valve, had to clarify that the DNS cache was accessed in order to catch kernel-level cheats that contacted DRM servers to verify whether the user had actually paid to use the cheat or not. He also said that this method was only used for thirteen days until the cheat producers discovered it and started to manipulate the DNS cache of their customers. Newell stated that during those thirteen days, 570 cheaters were banned by using this technique. In his explanation, Newell also noted that VAC tries to look like a scary and opaque system in order to make attacks against it more difficult [32]. This is commonly known as security through obscurity. The idea is further enforced by the fact that VAC does not immediately ban cheaters but instead marks them to be banned after a certain amount of time has passed. This makes it harder for cheaters and cheat developers to pinpoint the exact action that

caused the ban.

World of Warcraft was released by the game company Blizzard Entertainment in 2006 and it quickly became the most popular MMORPG game of all time. This meant that it also became a big target for cheaters. Blizzard had bad experiences with cheaters from their previous games, and in World of Warcraft they decided to develop a comprehensive anti-cheat system. The anti-cheat system used in World of Warcraft is called Warden and it is also used in other Blizzard games. Warden has also had its own scandals. In 2006, it was accused of scanning players' browser history and cookies. Privacy advocates argued that Blizzard had no right or need to access this kind of highly private data [25]. The Electronic Frontier Foundation (EFF) branded Blizzard's Warden as a spyware program. Software engineer Greg Hoglund managed to disassemble the code of Warden and confirmed that it was scanning running programs and checking them against known cheating programs. Hoglund also reported that Warden scans things like title bars of open programs. It also sent text strings such as emails back to Blizzard [57].

In November 2006, it was reported that many Linux users were mistakenly banned by Warden when it detected Cedega as a cheat program. Cedega is a Linux application that is meant for running Windows applications in a Linux environment, and many Linux gamers used it to run World of Warcraft. These kind of mistakes are called false positives. Blizzard acknowledged the error and restored the banned accounts 20 days later [51]. The Cedega false positive illustrates the difficulty of developing anti-cheat systems, as there are many programs that might exhibit cheat-like behavior such as rendering graphics on top of the game. For example, many voice chat programs have an option to display an in-game overlay on top of the game, and initially many of these programs were falsely classified as cheats when the overlay option was enabled. It can be hard for the developer to discern a hack that draws graphics on top of the game from a legitimate program that does so.

The three anti-cheat systems covered here all share similarities in function and all of them were developed as a response to a big cheating epidemic. All of them scan the running programs on the user's computer and perform various kinds of analysis. Since all three programs are commercial and their source code is not available, we cannot read the code itself and know exactly what these programs are doing and how they are doing it. However, the big picture about the history of anti-cheat is clear. New solutions were built as responses to problems that were driving many players away. The main method was having a client-side anti-cheat program that scanned the local computer and sent information to the server. The privacy issues about anti-cheat systems were raised with both VAC and Warden, and these issues are still relevant today.

The privacy viewpoint has been a large factor of influence in the development of modern anti-cheat methods [55].

4 Existing research in anti-cheat methods

Explicit academic research of different anti-cheat methods is relatively sparse. This may be due to low academic interest in the topic; it might also be due to the fact that most anti-cheat research in the private sector is proprietary, as the companies rely on secrecy to ensure that the anti-cheat is hard to circumvent. If companies published detailed info about their anti-cheat methods, it could compromise the security of their games. While there is not much academic research in the anti-cheat space, companies such as Riot Games have published blog posts and talks about their anti-cheat systems on a general level [56]. These posts provide valuable information for research. Some research also focuses directly on the methods that are utilized in cheat detection and prevention. For instance, this thesis references many papers that cover more broad computer security related issues like man-in-the-middle attacks and memory encryption [50][36].

Many websites dedicated to cybersecurity, such as Hackmag, detail various attempts to reverse engineer proprietary anti-cheat systems [8]. The authors who post their findings usually do so under a nickname because they do not want to be identified and possibly face legal consequences from the game companies. The legal consequences can be a major factor why people are reluctant to publish information related to anti-cheat systems, especially if these findings are in any way related to any commercial product. The authors of anti-cheat related content often work full-time on the software field and are interested in computer security, but they are often not academics and, as such, do not have an interest in publishing their findings in academic research papers.

Because of the low amount of existing research directly related to game anti-cheat, this thesis draws heavily on papers and books related to computer security. It also uses news articles, anti-cheat-related content posts, and whitepapers from various authors. All these resources, combined with the custom game and benchmarks, provide valuable information to compare different anti-cheat methods.

5 Categorization of cheat methods

Video game cheats can be divided into different categories based on how they exploit the game. In this thesis, the different methods are divided into categories of soft and hard

cheats. Approaches against all of these cheats in different categories will be covered in the thesis, but the main focus will be on hard cheats that utilize external programs in cheating. For example, a cheat could manipulate how game textures are drawn and give the player the ability to see through walls. In the context of this thesis, it is important to understand the difference between soft and hard cheats as well as how different cheat techniques are classified.

The type of the game can also influence what kind of cheating is relevant. For instance, utilizing secret game data and automated tools in first-person shooting games is considered heavy cheating and are very detrimental to the game. However, utilizing secret game data and automated tools would not be that interesting in turn-based games because of the slow pace of the game; in these instances, the game was likely designed so that there would be no secret game data. In turn-based games, looking at packet modification and spoofing could prove to be more relevant.

It is important to distinguish the effects of cheating on the game company and other players. Things that might be unfair towards the game company might not always be unfair towards the other players. However, cheating that is unfair towards the other players usually affects the game company indirectly when the players start to quit the game due to the unfairness. For example, if some players use wallhacks to see through walls, the other players will quickly perceive this as unfair and quit the game which would cause losses to the company. In this thesis all forms of cheating are covered but usually the damage of cheating is evaluated from the viewpoint of the game company.

5.1 Soft cheats

Soft cheats are cheating methods that utilize game mechanics in order to gain an unfair advantage over other players. These can be things like mechanics that were not intended by the game developers, such as quick ways to make money in the game. In essence, soft cheats cover all unintended use of game mechanics in order to gain benefits. The End-user license agreements (EULA) of the games usually contain a clause that disallows the unfair use of game mechanics. For example, the EULA of the game company Blizzard disallows the use of any in-game bugs that would grant an advantage over other players [1]. This type of broad clause covers all exploits as it would be impossible to list every possible case in the EULA. Soft cheats are hard to fight against by anti-cheat methods because they do not tamper with the game client or do things that differ from ordinary gameplay. Soft cheats also do not use any external programs such as tools that press keys and move the mouse to automate functionality. The main way to counter soft cheats is by thorough testing of the game before releasing it on the market. Games will also need continuous

updates to fix exploitable loopholes.

5.1.1 Bugs and exploits

Bugs and exploits are often categorized as cheats because they provide functionality that was not intended by the game developer. For instance, in a role-playing game there could be a situation where a character could buy an item from a shop and then sell it to the same shop for a higher price. This behavior would obviously be unintended by the developer because when repeated multiple times, it would provide the player with a source of unlimited money. To exploit this bug, the players do not need to use any cheating techniques as they can just use the game client to buy and sell the item repeatedly. This thesis discusses ways to prevent these kinds of exploits from being made, but they are not the focus of anti-cheat systems as they purely work by utilizing existing game mechanics without any external programs. For example, in the game Destiny 2, players found a bug that would allow them to use a certain weapon in an unintended way to make their characters very powerful. The knowledge of this exploit spread and later the developers removed the weapon from the game to fix the bug [15].

5.1.2 Exchanging real money for in-game goods and services

Using real money to buy in-game goods and services is often considered cheating unless the game developer explicitly allows it in the game rules. Buying game services or goods with real money breaks the game economy and offers an unfair advantage in relation to other players. This kind of cheating is hard to detect because the real money transactions take place outside the game services, and the cheater gets the items in-game after the transaction is made. It is very hard to prove a link between these actions compared to just giving items away in the game for free. Even though this type of cheating is classified as soft cheating, there are ways to use statistical data analysis on the server to find suspected cheaters. For instance, all player actions can be logged and suspicious events can be searched from the logs by automated tools.

Account sharing is also increasingly common in many games. Shady companies often sell services such as character levelling or using your character to do some in-game tasks that would consume a large amount of time. These services are paid with real money, and then the password and username is given to the company to login to the service and complete the purchased tasks, which might take up to many weeks [10]. This kind of account sharing usually violates games' terms of service agreements and is considered cheating as it gives an unfair advantage to the player who buys these services. There are also concerning

reports from Asia about prisoners being forced to play World of Warcraft to earn gold and sell it for real money [48].

The main method against this type of cheating is extensive logging and retroactive inspection of the logs. Geographic restrictions can also be used.

5.2 Hard cheats

Hard cheats are cheating methods that anti-cheat systems are mainly developed to counter. Often when cheating is discussed in an everyday setting, it exclusively refers to these approaches. For example, hard cheats could be programs that edit the game client or modify the packets that the game client sends out. External cheat programs could also inject themselves into the game's memory space and create new functionality by calling the functions of the game.

5.2.1 Bots and other automated tools

Bots and other automated tools are programs that automate playing certain or all parts of the game. Bots can be roughly classified into three types: ones that read the image from the screen and simulate keyboard and click events, ones that inject code into the game client in order to control the game, and ones that control the game by sending packets to the server pretending to be the actual game client. The first method, which only simulates keyboard and mouse clicks, could also be classified as a soft cheat, but it is classified as a hard cheat because it uses external programs to simulate the input. For example, a program can automate killing enemies in a game and be left to run day and night for a week, which would grant an unfair advantage against those players that can only play the game limited hours per day. For instance, World of Warcraft players can write macros that automate small functionalities of the game, but these macros are limited by the game API. However, with cheats the player can unlock all the limitations and write very complex macros in the game that automate gameplay to a large extent [8].

5.2.2 Utilizing secret game data

Utilizing secret game data means using data from the client that is not meant to be seen by the player. For instance, all player locations could be located in the game memory even though they are not meant to be shown to the player except in certain circumstances. Now the cheat program could read the game memory and extract these locations for the

player to see. The cheater could also capture the packets and read the player location data directly instead of reading the game memory. Preventing this kind of cheating requires smart design of the application protocol and considerations about what information needs to be in the memory. Detecting memory scanning tools and encrypting the data in the memory are also possible solutions.

5.2.3 Packet modification and replay attacks

Packet modification refers to editing the data in the packets that are sent to the game server. For example, the cheater could feed the server with false movement data or exploit a badly designed application protocol where the server blindly trusts the data that arrives from the client. Malladi et al. define replay attack as an attack on a security protocol that uses replay of messages from a different context into the intended context, thereby fooling the honest participants into thinking that they have completed the protocol run successfully. They also split replay attacks into two categories: run-internal and run-external where the origin of messages defines the category of the attack [27]. In common terms, replay attack refers to an attack where a packet is copied and sent over and over again to the server. This thesis will cover run-external replay attacks where the game client tries to send replicated packets to the server. For example, a player could cast a spell on another player in a game and then replay the spell cast packet multiple times, which would make the other player lose the fight. If the server was badly designed, this type of attack could make the cheater very powerful.

5.2.4 Spoofing

Spoofing in the context of cheating in video games means pretending to be someone else and using that to exploit the game. For example the cheater could attempt to send packets and make it look like some other player was sending them instead. This could be done by editing the packet data to make it look like it was coming from someone else. In a very badly made video game this could mean that a player could give someone money in the game and then spoof the packet so that the server would think that someone is actually giving the cheater money. Spoofing is possible because the IP protocol itself does not specify any method to validate the authenticity of the source of the packet [49]. This means that the anti-spoofing measures have to be taken on the transport layer and further on in the application layer. Preventing this kind of cheating requires the game developer to implement a way to verify that the packet is really from the source that it claims to be from.

6 Server-side anti-cheat methods

Anti-cheat methods can be broken into two categories: server-side and client-side methods. Server-side methods work only on the game server itself and rely on things such as checking incoming packets and ensuring that the data and game state are correctly handled on the server. Client-side methods are things such as anti-cheat programs that operate on the client machine and send data back to the server. This chapter will take a broad look at four different ways to prevent cheating on the server-side: not trusting the client, designing a tampering resistant application protocol, obfuscating the network traffic, and statistical methods. These techniques will be analyzed from the perspective of ease of implementation, overhead, suitability for different types of games, and how easy it is for cheaters to circumvent these methods.

6.1 Do not trust the client

Not trusting the client is an essential part of designing an online game in which it is hard to cheat. In fact, verifying data from clients is an essential part of any online service design. For example, if the game client sends a packet that tells the server that the player is trying to sell an item to a shop, the server should not blindly trust the message and perform the sell operation which would give the player money. Instead, the server should perform all the necessary checks such as whether the player actually has the item and if the player is in the right location.

Not trusting the client might seem an obvious design choice, but many online services can have vulnerabilities where data is not correctly verified. For example, a mobile turn-based online strategy game could have the battles fought on the local device and then send the results of the battles to the server that blindly trusts the data. In this situation, the game developers could have calculated that it is worthwhile to save server and network resources by not having the battles simulated and fought on the server. However, this design decision paves the way for cheating by modifying the client and sending false results to the server. Developers may also believe that mobile devices are more secure than computers and thus harder to use for cheating. While this is true in some sense, it does not mean that mobile devices are immune to cheating. For instance, the iPhone's and iPad's iOS operating system is more closed than Windows or macOS, but it is still possible to open the system and modify the game client. If the network traffic is not obfuscated, this type of attack can be executed just by intercepting the packets and modifying the packet data on the fly without even touching the game client itself.

Figure 2 demonstrates a situation where a server is vulnerable to being fed false data by

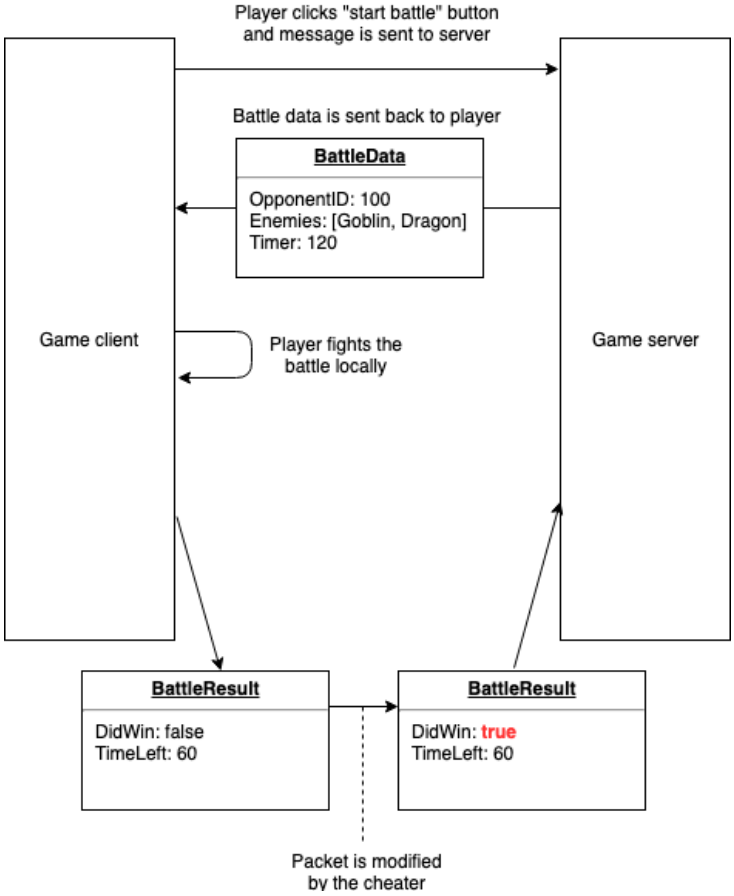


Figure 2: Protocol vulnerable to packet or client tampering

the client. For example, the cheater could edit the `BattleResult` packet and change the `DidWin` part to true instead of false. Alternatively, the client could just be modified in a way that it always sends a packet that indicates a battle was won regardless of whether it was really won or lost. In Figure 2, the game server blindly trusts the result from the client, and the server has no way of knowing the real status because the fight is only simulated in the local machine. This type of flaw is very easy to take advantage of, especially in cases where there is no anti-cheat on the client-side.

Figure 3 shows an improved protocol that does not trust anything other than input actions coming from the client; the battle is simulated solely on the server-side, and the client just displays the battle for the client. In this scenario, there is nothing that can be modified in the client in order to cheat the server; packet modification will not be useful because the only data sent to the server are actions such as move or attack. As can be seen from the

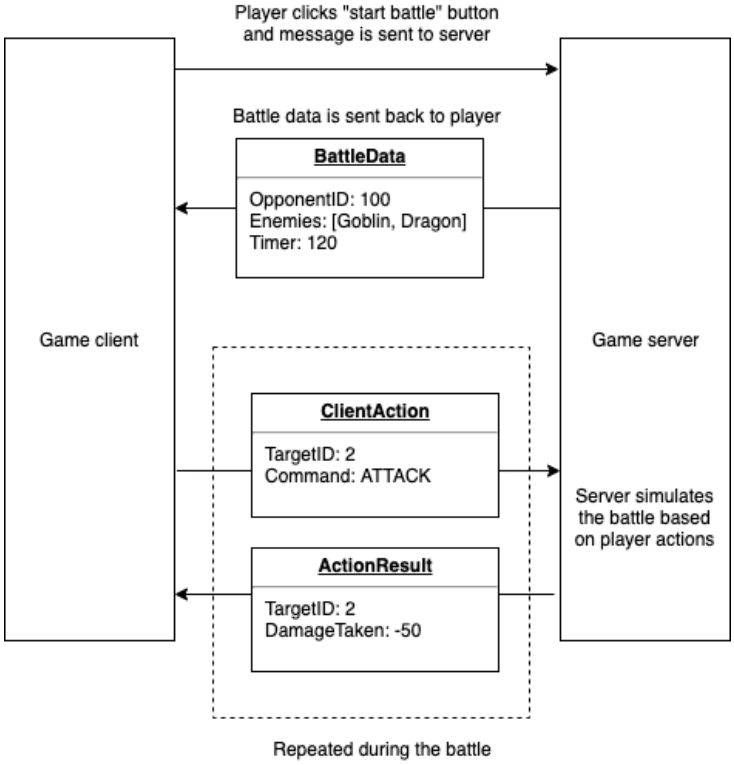


Figure 3: Protocol that is not vulnerable to packet editing

figure, the client sends an ATTACK action to the server and the server responds with the result. If the client edits the action packet, it does not matter because the actions are just inputs that tell the server how to proceed with the simulation. In essence, by editing the action packet the cheater can just change one possible action to another possible action because impossible actions are rejected by the server.

When evaluating not trusting the client as an anti-cheat method, certain trade-offs have to be taken into consideration. If the server was to run all the battle simulations, the network traffic would increase manifold because all actions would be sent to the server and the results sent back to the client. The server would also need much more resources to simulate possibly hundreds of thousands of battles at the same time. In some parts of the world, such as Southeast Asia, internet connections can be unreliable and expensive, which would make playing the game more unpleasant and possibly impossible for some users [24]. If the game is targeted at these markets, it could be justified to design the protocol as presented in Figure 2 and instead rely heavily on other anti-cheat measures.

The number guessing game, which was specifically made for this thesis, can be used to

demonstrate the real-world effects of not validating the game state and blindly trusting the user input. In the game, when a user types a number and presses enter, a packet with the structure displayed in Figure 4 is generated. The only data fields in the packet are the opcode, match id, authentication token, and the guessed number. The server uses the token to confirm the identity of the player and to check whether it is the player's turn to make a guess. If this check was not there, and the server solely relied on the client-side user interface to determine whether a client would be allowed to send a guess or not, the client could easily do a replay attack by replicating captured guess number packets and changing the guessed number or directly reverse engineer the game user interface to allow the guessing of a number on other players' turns.

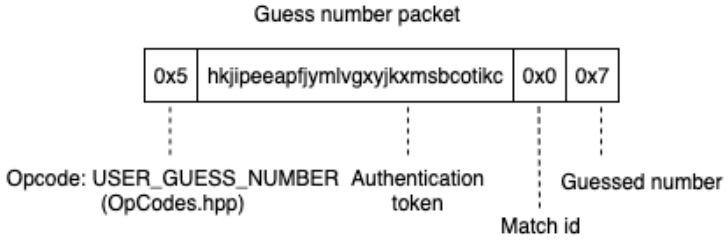
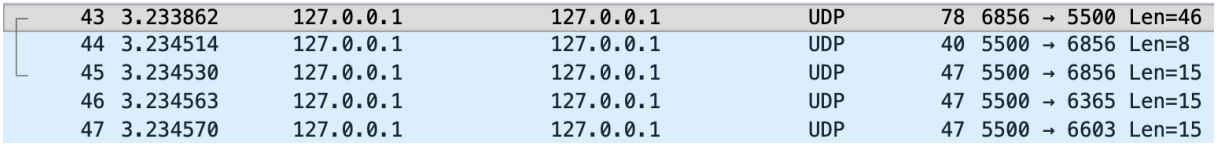


Figure 4: Packet that is sent when player guesses a number

If we want to test whether a game would be vulnerable to this type of attack, first we would find a packet that contains the number we are trying to guess. The number has to be in the packet in order for the server to validate the guess. With WireShark, we can turn on the packet capture and guess a number in the game. In this situation, the server and the client are on the same computer in the same network, so we have to examine the loopback traffic instead of Wi-Fi or Ethernet because the packets never enter those. In a real-world situation, the packets would travel over the Ethernet or Wi-Fi and we would look into those adapters instead. The packet structure would remain the same.

In WireShark's packet capture, we can notice a burst of packets right after guessing a number in the game. Figure 5 shows these five packets. We can look into the first packet and search for the number that we guessed. In this example, the number was 54 (which translates to 0x36 in hex). In order to test this method in the guessing game, we have to temporarily remove the check for the player that has the turn. For the purpose of this example, we remove the check `if (guesser == match->GetCurrentTurnPlayer())` from the game server code so the server blindly trusts the client. The game client and server code can be found in Appendix A and Appendix B.



43	3.233862	127.0.0.1	127.0.0.1	UDP	78	6856 → 5500 Len=46
44	3.234514	127.0.0.1	127.0.0.1	UDP	40	5500 → 6856 Len=8
45	3.234530	127.0.0.1	127.0.0.1	UDP	47	5500 → 6856 Len=15
46	3.234563	127.0.0.1	127.0.0.1	UDP	47	5500 → 6365 Len=15
47	3.234570	127.0.0.1	127.0.0.1	UDP	47	5500 → 6603 Len=15

Figure 5: Burst of packets related to the game

In Figure 6, we can see 36 at the end of the packet data after a long padding of zeroes. We can suspect this packet is the one that we are looking for. There are a number of ways to resend a similar packet with different guess numbers. In this example, we write a small Node.js based script that takes in the data as bytes and then edits the last two bits on every iteration to test a different number.

```
0000  02 00 00 00 45 00 00 4a  14 7b 00 00 40 11 00 00   ····E··J ·{··@···
0010  7f 00 00 01 7f 00 00 01  1a c8 15 7c 00 36 fe 49   ········ ···|·6·I
0020  00 00 00 05 00 00 00 1e  73 6d 79 63 74 75 75 67   ········ smyctuug
0030  63 74 65 67 69 6d 6d 6a  67 79 67 79 76 71 63 6c   ctegimmj gygyvqcl
0040  78 6a 74 75 65 66 00 00  00 00 00 00 00 36         xjtuef·· ·····6
```

Figure 6: Data inside the first packet

```
const dgram = require('dgram');
const socket = dgram.createSocket('udp4');

const data = [
    0x00, 0x00, 0x00, 0x05, 0x00, 0x00, 0x00, 0x1e,
    0x73, 0x6d, 0x79, 0x63, 0x74, 0x75, 0x75, 0x67,
    0x63, 0x74, 0x65, 0x67, 0x69, 0x6d, 0x6d, 0x6a,
    0x67, 0x79, 0x67, 0x79, 0x76, 0x71, 0x63, 0x6c,
    0x78, 0x6a, 0x74, 0x75, 0x65, 0x66, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x36
];

for (var num = 0; num <= 100; num++) {
    data[data.length-1] = num;
    socket.send(Buffer.from(data, 'hex'), 5500, '127.0.0.1', (err) => {});
}
```

Listing 1: Script that guesses multiple numbers in a row

The script in Listing 1 sends packets that try to guess the number for all numbers from zero to one hundred. On every iteration of the for-loop, the last byte is replaced with the number from the for-loop. In this way, the packets are sent to the server, and at some

point, the right number is hit; thus the sender wins the game. In this example, the method is easy to implement because the packets are not encrypted, nor is there any check on the order of packets or whether a certain packet has arrived or not. Even if these checks were present and the packets were encrypted, exploiting the trust of the server would still be possible but more time consuming as the encryption algorithm would have to be reverse-engineered and packets given the correct order numbers. The only way to fight against this type of cheating is to build the server so that the game state is always tracked and the content of every packet is always verified.

Based on this section, not trusting the client should be a basic building block of any anti-cheat solution unless there are very specific and important concerns why the message from the client should be trusted directly without running and evaluating the game state on the server. One special reason is that simulating lots of stuff on the server can be more expensive and will cause more overhead depending on what things need to be verified. However, utilizing cloud computing has greatly reduced costs and made it possible to quickly deploy additional resources should they be needed, so there are fewer and fewer reasons to build solutions that trust the client without verification. The example with the guessing game clearly illustrates the possibility of exploiting a weakness in the server and using it to win the game. The demonstration also displayed how simple this kind of attack can be. In addition, it showed that a single line of code `if (guesser == match->GetCurrentTurnPlayer())` could have prevented this attack.

In terms of resistance to tampering, not trusting the client ranks very high because as shown here, the data verification is done on the server side and thus it is very hard to exploit.

When looking at the ease of implementing the game in a server authoritative way, it is not very straightforward as it requires verifying all parts of the game so that no client info is directly trusted. Especially if the developers are modifying an existing game, it can take a lot of work to upgrade the game so that all information is verified. Even then the programmers might miss something and leave in critical exploits. If the game was designed from ground up to not have the simulation done on the server side, it might be too time-consuming and difficult to fully change the architecture of the game.

When it comes to overhead, the solution cannot be given the highest score because verifying all types of data that comes from the client increases the processing time on the server. However, the overhead is dependent on the amount of data and also on the processing power of the server, so in the best case scenario it may be a non-issue.

In terms of privacy, not trusting the client is a good anti-cheat method as it does not interfere with anything on the client machine.

When looking at the suitability for different types of games, the method is widely applicable to various types of games, although it is more suitable for slower games, such as strategy games, that do not need to send large amounts of data continuously.

Overall, building a server, that accurately keeps track of the game state, makes the fight against cheating much easier and also reduces the need for other types of anti-cheat methods. Server-client design, that does not trust the client without checks, provides a solid foundation for any type of anti-cheat solution.

6.2 Designing a tampering resistant application protocol

User Datagram Protocol (UDP) [54], Transmission Control Protocol (TCP) [39], and the protocols below them are responsible for the delivery of the messages between the client and the server. Application protocol encompasses the packet structure, what kind of data is sent in different types of situations, and how the data is handled on both the client and the server. The tampering resistant application protocol is viewed as a server-side anti-cheat method because all the packet verification is done on the server-side even though the client-side also uses the protocol. Despite UDP and TCP being on the lower layers, the choice between them will affect how the application protocol is built. While this chapter focuses on the application protocol that the developer can affect, in theory, the cheating could happen on any layer of the protocol stack, such as the IP protocol layer.

Most games tend to use UDP as a network protocol and implement some of the TCP functionality in their own application protocols. However, there are famous games that use TCP, such as World of Warcraft [46]. Peer-to-peer games have their own unique issues related to their distributed nature but they will not be covered in this thesis due to its focus on the more popular client-server paradigm. A vulnerable application protocol can cause heavy damage to the game in the form of cheating but also in the form of data theft. For example, Valve's popular Source Engine contained a memory corruption bug that could be exploited with fragmented packets. This vulnerability was caused by too small of a heap buffer that was assigned to hold the entire packet. The checks done on the packet's POS and LEN fields were also faulty. This vulnerability allowed the cheater to use fragmented packets to overflow the buffer and execute malicious code [3].

Firstly the developers need to decide whether to choose UDP or TCP. The TCP was designed for bulk transfers and as such is not very widely used in online games. It was reported that TCP can often cause an unneeded delay in video games due to the way games work by continuously sending data that did not exist before. For example, in TCP if an earlier packet is delivered unsuccessfully then the following packets will get

blocked. This phenomenon can cause delays that would not happen with UDP. When it comes to anti-cheat, TCP provides useful out-of-the-box functionality like packet sequence numbering. This means that if the cheater tries a replay attack by simply resending packets, the cheat will not work because TCP will notice that a packet with the same sequence number has already been received. In the case of UDP, this kind of protection would have to be implemented by the developer. In comparison to UDP, it would also be much harder for the cheater to inject fake packets into the stream because they would need to have the correct sequence numbers on TCP [39].

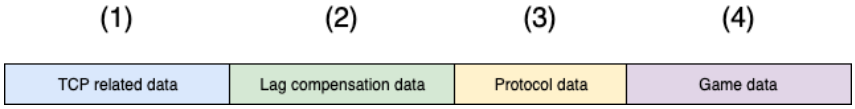


Figure 7: Possible UDP packet structure in an online game

Figure 7 represents an example UDP packet structure in an online game. In this case the developer has chosen to use UDP but build a TCP-like functionality on top of the UDP protocol. This is commonly known as TCP over UDP. The structure of the packet in Figure 7 is as follows:

1. TCP-related data contains information that TCP packets normally contain such as sequence numbers.
2. Lag compensation data contains time-related information such as the time when the packet was sent by the server. This data can be utilized to make calculations about the game latency. It can also be taken into account when evaluating things such as character movement.
3. The protocol data can contain miscellaneous data that the developers have chosen to use for the protocol. For example, this can be things like data reliability hashes.
4. Game data is actual data of the packet. This can be anything related to the game such as character movement data, enemy health data or account information.

Auriemma and Ferrante state in their paper that many games often fail to properly verify the headers of the fragmented packets which can allow the cheater to send in malformed packets. They also highlight that when searching for game vulnerabilities, the packets are often a good place to look at [3]. For example, cheaters could edit the data in the (3) lag compensation portion of the packet and fool the server into thinking that there is more latency than there is.

Whether the developer chooses to use TCP or UDP, the most important part of a smart

application protocol is sending only information that is needed and nothing else. For example, many games have a minimap that shows the players that are close to you, which means that the locations of these players have to be stored inside the game memory. In a badly designed application protocol, the memory would not only contain the locations of the nearby players that are displayed on the minimap, but all the players, even those that are not displayed. This implies that a smart cheater could either get these non-displayed player locations from the packets themselves or dig them up from the game memory.

Designing an application protocol that does not send any extra information is not always possible due to other constraints like the speed of the game and latency. For instance, in many shooting games it is necessary to react quickly to a player that comes around the corner and is not visible on the map. If the location of this player was not in the memory, it would have to be sent from the server to all players in the situation. If the packet arrived to one player before the others, it would cause an unfair situation where that player would see the other players even though they would not see him or her for a few milliseconds. In fast-paced shooting games, a few milliseconds might be all that is needed to win a fight. This situation is best demonstrated in Figure 8. This is why in many shooting games the player locations are held in the memory, which makes it possible for cheats to utilize them.

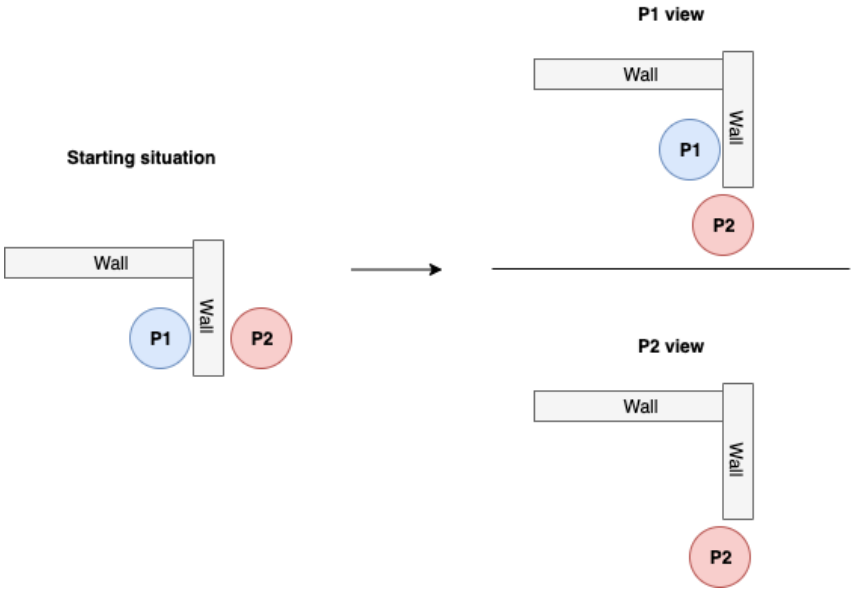


Figure 8: Player one (P1) receives the location packet before player two (P2)

In Figure 8 we can see that in the starting situation the players are on the different sides

of the wall and, therefore, cannot see each other. When the game progresses, the player two (P2) starts moving to the other side of the wall. Player one notices player two (P2) because he received the packet before player two, who has not yet received it. If both players already had the locations of other players in the memory, this situation could have been avoided. A smart application protocol takes into account the specific circumstances of the game that the protocol is built on. In this example case, it would be smarter to just store the locations of the nearby players in the memory and rely more on other anti-cheat methods. On the other hand, in a slow turn-based game, it would be perfectly suitable to send all the necessary data at the exact moment when it is needed. Obviously, games can implement much more complicated systems that, for example, take into account the range of players and only send the location data of players that are sufficiently close by. This will not work for games that operate on long distances but can be a suitable solution for a small subset of games. A very popular commercial game engine, Unreal Engine 3, tries to minimize the data that is sent to the client. In other words, only the exact data that is needed for the simulation is sent to the client so the cheater cannot take advantage of the extra data [6]. A well-implemented application protocol can be very hard to utilize for cheating and thus is very resistant to tampering.

When designing the application protocol for a game, it is necessary to find out what kind of information is needed in different types of situations. It is also reasonable to think in each of these situations whether the data could be transmitted at the exact moment when the situation occurs or if it is necessary to have it ready before.

Tamper-resistant application protocols are not hard to implement on the technical level, but the difficulty comes from determining what type of information is needed and where. As shown before, TCP offers a lot features out-of-the-box, so it can be a good choice for teams that do not have a lot of experience implementing networking solutions.

A good application protocol will not generate much overhead, but often more packets can be needed compared to a protocol that does not have any checks in place. If the game is operating in areas with very slow internet speeds, extra attention should be paid to the overhead.

The application protocol itself is not invasive in terms of the user data and is suitable for all types of games, although special considerations might be needed in certain fast-paced games. Building a tamper-resistant application protocol should be at the core of a game's anti-cheat approach.

6.3 Obfuscating the network traffic

When data travels over the network, by default it is not encrypted; this means that anyone can get the data by using tools such as Wireshark [62] or Ettercap [14] and fully understand what is transmitted over the network. For example, in Figure 9, the game server sends other players' positions unencrypted to the client and a third party is capturing the packets by using a packet sniffer tool. The packet sniffer then feeds the packet data to a custom cheating program that constructs a world model from the data. Cheaters can then use this world model to their advantage. For instance, the cheating program can be a "maphack" that shows the players' positions on a map even though normally they would not be visible.

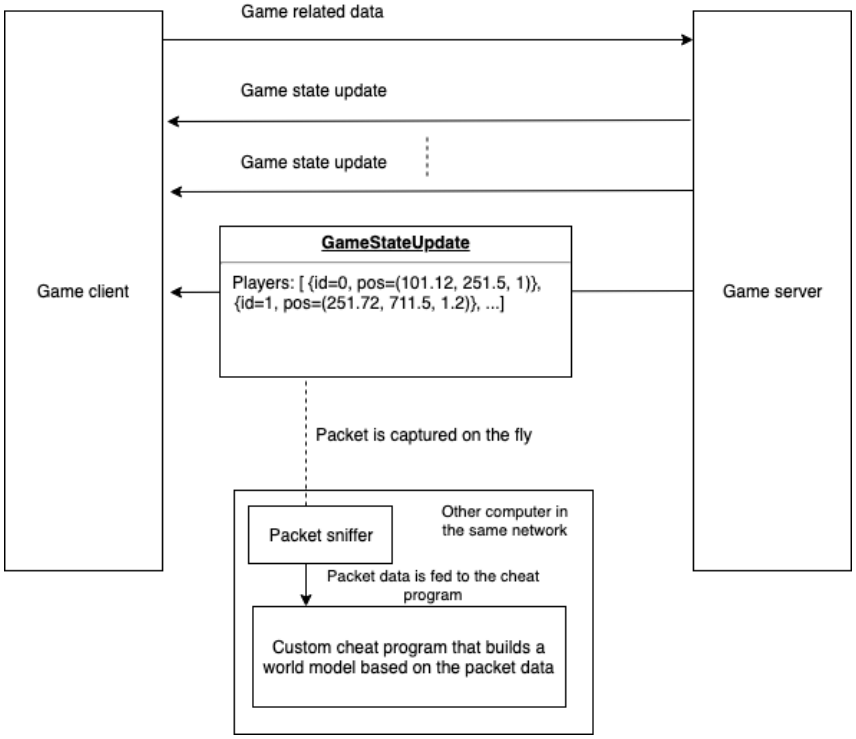


Figure 9: Packet sniffer uses data from packets to build a model of the game world

The situation in Figure 9 could be prevented by encrypting the packets so that the data contained in them would not be understandable. However, even if the packets were encrypted, the game client would need to decrypt the packets at some point to utilize the data inside. If the cheaters managed to discover the decryption algorithm and reverse it,

they could then decrypt the sniffed packets and cheat as before. Packet obfuscation is not a foolproof solution and will not ultimately stop packet sniffing, but it will make it harder and more time-consuming for the cheaters. The game developer could also choose to change the encryption algorithm every update, forcing the cheaters to find the new algorithm and update their decryptors. After some time, the decryption process would become a routine step for the cheaters, but it would possibly get rid of many inexperienced cheaters.

The encryption can be demonstrated with a real-world example by using the number guessing game. We compare how the encrypted packets of the guessing game look compared to the normal non-encrypted versions of these packets. We also show how much overhead the encryption and decryption would generate both in the server and the client. The code that encrypts the packet is shown in Listing 2. As we can see, the encryption is unique for each player because their random token is used to encrypt the packet, so even if the cheaters managed to discover how the encrypted algorithm works, they would still need to have the unique key for each player which would make the cheating process longer and harder, although not impossible.

The code shown in Listing 2 takes in an unencrypted packet, removes the opcode, and encrypts the remaining data in the packet. The opcode cannot be encrypted because the server has to identify packets before taking any action. The first few packets are not encrypted so the server decides based on the opcode whether to decrypt the data or not. The encryption itself is done using bitwise operation XOR on all the bytes with the encryption key. The algorithm takes the i:th byte of the data and XORs it with the i:th byte of the key. If the key is shorter than the data, then the key is treated as a circular and the index is the division remainder of the i divided with the key length. The overhead of the encryption scheme (Listing 2) was measured to be around 0,004 milliseconds on the test computer that was introduced in Chapter 2. On a thousand packets, the combined overhead would be four milliseconds. Obviously, on a more complicated encryption method, the overhead would be higher but most likely not high enough to have a substantial effect on the performance.

Figure 10 shows the unencrypted join matchmaking packet and Figure 11 shows the encrypted version of the same packet. We can clearly notice that the player name “Player” is in plain text at the end of the unencrypted packet while the content of the encrypted one looks unclear and it is hard to tell what the packet is transmitting. Being able to read the player name from the packet is not very dangerous, but in many games, the packets contain things such as player locations that are not supposed to be displayed to the client. An encryption with unique keys for each player is also a strong tool against MitM (Man in the middle) attacks [50].

```
sf::Packet PacketHandler::EncryptData(sf::Packet packet, std::string key) {
    // If encryption is not in use, just return the packet as normal
    if (!this->mUseEncryption) {
        return packet;
    }
    // Extract opcode from the packet
    sf::Packet encryptedPacket;

    int opCode;
    packet >> opCode;
    encryptedPacket << opCode;

    // Get data of the packet into buffer
    const char *charBuffer = (char*)packet.getData()+sizeof(char)*4;
    int n = packet.getDataSize()-sizeof(char)*4;

    // Convert the char* data buffer to std::vector data buffer
    std::vector<char> data(charBuffer, charBuffer + n);
    std::size_t dataSize = n;
    std::vector<char> encryptedData;

    // Encrypt by XOR:ing with encryption key
    for (std::size_t i = 0; i < dataSize; i++) {
        encryptedData.push_back(data[i] ^ key[i % key.length()]);
    }
    // Create the encrypted packet and append the data
    encryptedPacket.append(encryptedData.data(), dataSize);
    return encryptedPacket;
}
```

Listing 2: Packet encryption in the guessing game

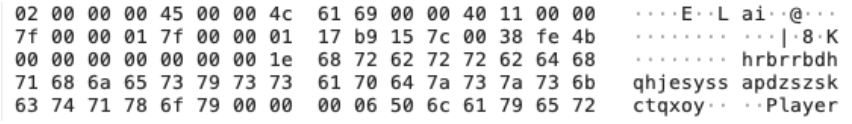


Figure 10: Unencrypted version of enter matchmaking packet

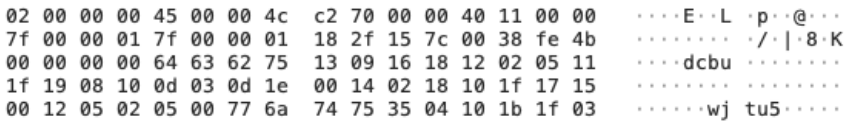


Figure 11: Encrypted version of enter matchmaking packet

A well-designed packet encryption scheme is an important part of designing an anti-cheat system. It is a good method because it has very few disadvantages when compared to the benefits that it offers. While it is not a bulletproof system, it still adds a strong barrier to cheating and makes spoofing the game data harder for inexperienced cheat developers. It can be said that packet encryption tries to hide the game protocol and thus brings security through obscurity.

The simple XOR-based encryption scheme demonstrated here should never be used in real games due to its various weaknesses. In real games, developers should consider using already established networking libraries that provide well-tested networking functionalities such as secure connections. They also provide things like different lanes for packets that have to arrive and for packets that are not very critical. For example, popular networking library RakNet [40] provides data security on par with 256-bit TLS. It offers an efficient 256-bit Elliptic Curve key agreement with forward secrecy that protects each connection with the server. RakNet encrypts each message and stamps them with a message authentication code and a unique identifier that protects sensitive data and prevents replay attacks. Setting up the encryption in RakNet only takes a few steps, which involve generating a public and private key and setting the private key to the server and putting the public key inside the game client [40].

Using a popular and well-tested library such as RakNet is much preferable to making a custom solution; making a well-made custom encryption scheme takes considerable time that could otherwise be spent on other anti-cheat measures. Furthermore, testing a custom solution with a large number of players before launching the game might prove difficult.

Based on the results in this chapter, encrypting the game network traffic should be part of

any game's anti-cheat toolkit as the overhead is very minimal and there are well-tested networking libraries available that take care of the heavy lifting. Network traffic encryption is one of the prime methods of anti-cheat, but it is still limited by the fact that the client has full control of the computer and can work through how the encryption works on the client-side. As noted in this chapter, this can be alleviated by changing the encryption method every time the game receives an update.

Finally, we evaluate this method in the context of the five key attributes: resistance to tampering, ease of implementation, lack of overhead, non-invasiveness, and suitability for a wide variety of games.

In terms of resistance to tampering, the packet encryption makes it harder for cheaters to determine the contents of the game packets. It can also effectively make more inexperienced cheaters abandon their attempts to cheat because figuring out the encryption scheme is much more difficult than just capturing the packets and viewing the data. Note that both client and server need keys for both decryption and encryption. However, the player should not have access, or at least not an easy way to access, these keys. Public key cryptography would not really help in this because only one of the keys (encryption key) can be made public, the other one (decryption key) still needs to be kept secret. Even if the encryption was completely secure, the data could still be retrieved before the encryption.

When it comes to ease of implementation, the encryption scheme is usually quite easy to implement because of the many premade libraries available that take care of the heavy lifting. Even if the game developers decide to implement the whole system from scratch, it still needs to be deployed only on one central place in the code that handles all the packet encryption/decryption, and thus the other game code can remain unchanged. Overall, network traffic encryption is one of the easiest methods to implement.

In terms of overhead, decryption and encryption should not be a major issue because the algorithm is fast and processing power is very high on modern computers. However, utilizing encrypted network traffic will still be slightly slower than using non-encrypted traffic because of the extra overhead.

The packet encryption is by definition non-invasive in terms of user privacy because it does not access private user data. Passing the data through the encryption algorithm does not make it any more vulnerable in terms of privacy.

This method is suitable for all types of games and does not have fundamental limitations that would make it unfit for certain games.

Overall, packet obfuscation is a powerful anti-cheat technique given its small overhead, ease of implementation, and the fact that it can make it much more difficult to understand

game data from the captured packets. It is also the only anti-cheat method that works directly on the packet-level and thus it is hard to replace with other techniques.

6.4 Utilizing statistical methods to discover cheaters

It is possible for the server to collect various types of player data such as number of wins, kills, deaths, or whatever interesting data the game has. For example, in a first-person shooter, the game server might track the number of kills the player gets every match and compare it to the game averages and the player's past record. If the kill count diverges enough from the global average or the player's own past record, the system could flag the player for further investigation by a human reviewer. The problem with these kinds of systems is that it is often very hard to make a direct judgement based on statistical data. For instance, there are players that practise many hours every day and are much better than most players, so their score diverges a lot from the average. If the system automatically banned people who are much better, it could generate many false positives and hurt the reputation of the game. This is why many statistics-based systems are often linked with a human-operated review system. Statistics-based systems are also well suited to catch exploiting of bugs such as selling an item to a shop for a higher price than it was bought for as mentioned in the Chapter 5 Section 1.

FairFight is an example of an anti-cheat system that purely uses these types of server-side statistical methods without any client-side anti-cheat. It is very effective and is used in many high-profile games such as Battlefield V, Tom Clancy's The Division, and Titanfall 2. The company behind the development of Fairfight, GameBlocks, explains that the players' actions are tested against multiple statistical markers to determine if cheating occurs. The anti-cheat also allows the game developers to customize the rule sets and data that the anti-cheat uses to determine whether a player is cheating or not [17]. What makes Fairfight so powerful is that the developers do not need to spend time implementing the anti-cheat itself; rather, they can just integrate the premade system in the game and customize the rules and determine what data will be used.

The popular online competitive shooting game Counter-Strike: Global Offensive utilizes a system called Overwatch [35]. The system is based around other players reviewing footage from players that are suspected of cheating. Many reviewers are looking at the same cases and the majority decides whether the suspect was cheating or not. The reviewers are selected based on their game participation such as competitive wins, skill group, account age, and hours played. If they have served as an investigator before, their previous accuracy rate also affects whether they can keep reviewing cases. For example, if they have continuously made wrong judgements, then they might not be able to review cases in the

future. Wrong judgements are decisions that go against the majority of the other players who are reviewing the same case. The investigator can view the game from the viewpoint of the suspected cheater and also fly around the map in free mode and look at the match from different perspectives. Overwatch also allows the reviewers to see through walls to determine if the cheater actually saw the other players through walls and if the actions give it away. The game company obviously does not share the algorithm that selects the suspected cheaters, but they state that it is based on the behavioral patterns and whether a certain player stands out from other players in terms of being reported for cheating or otherwise being flagged by the server. Figure 12 shows how the person who reviewed the evidence can choose to vote if the inspected player was cheating or not.

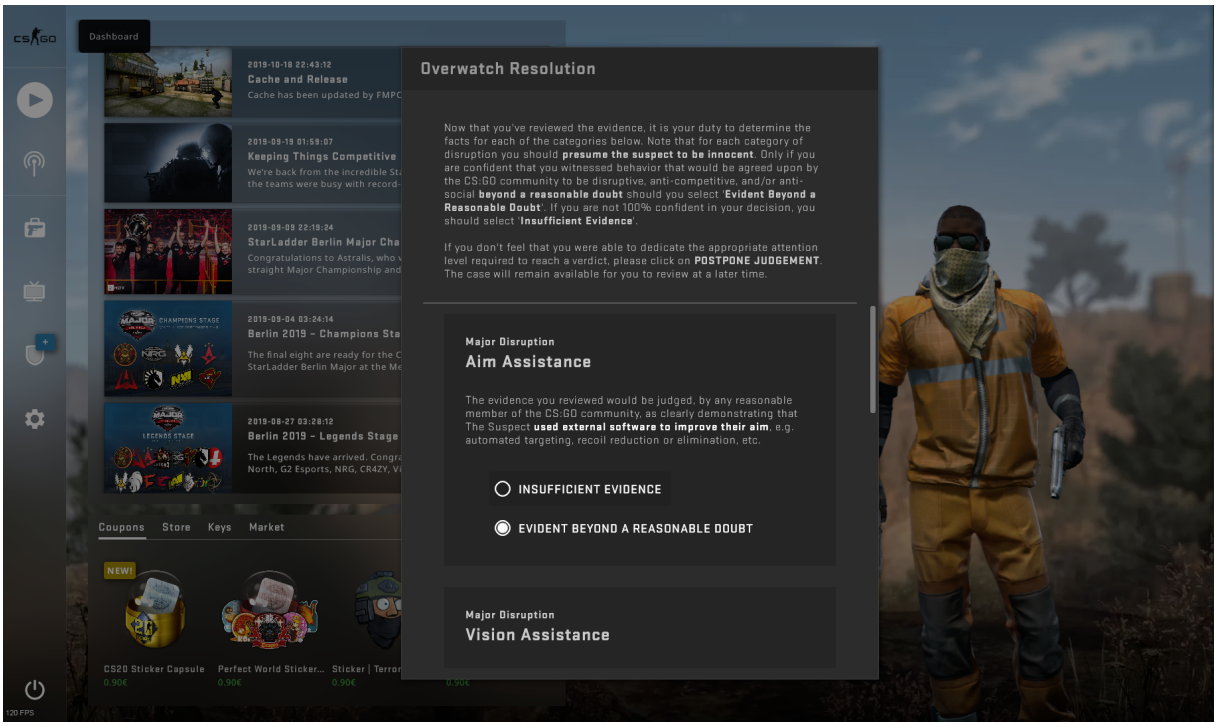


Figure 12: Passing judgement after reviewing evidence in Counter Strike: Global Offensive's Overwatch system

The reviewers have access to a replay of randomly selected eight-round segment from the suspected cheater's match. Things like chat and voice are disabled and other players' names are blurred so the determination has to be solely based on the actions of the player. After reviewing all the footage, if the investigators collectively determine by a large margin that the player cheated, then the cheater is banned by the system. The length of the

ban is determined by the severity of the offense that the cheaters committed and also whether they have any prior cheating history. It should be noted that a judgement of a higher-ranked investigator will have more weight in the system. For instance, if one investigator has a success rate of ninety percent and the other has a success rate of fifty percent, the judgement of the former investigator will weigh more in the conclusion. The system rewards players who participate and review cases.

A review system like Overwatch is a very powerful tool for curbing cheating because it is purely server-side and it combines the statistical analysis approach with experienced human reviewers. However, this type of system requires a large and active playerbase, which all games do not have. It also requires the game to be competitive (i.e., where players compete against each other), but many games focus primarily on cooperation where players fight together against AI opponents. These types of games do not give players a big incentive to report other players because everyone is on the same team against computer opponents. Even though all players are on the same team, cheating can negatively affect the game systems such as the in-game economy.

The problem in using statistical methods is primarily the need for human resources to analyze the flagged players. These human reviewers can be players as in the Overwatch system, or they can be people that are directly hired by the game company to evaluate these cases. Many smaller companies might find it difficult to utilize these types of systems because of financial constraints or low number of players. This leaves them to either very slowly go through the marked players with limited manpower or take risks and immediately ban players who significantly diverge from the average. Neither of these situations are ideal, so every company has to plan a model that works for their specific situation. There are also external companies that offer these types of data-based anti-cheat systems as a service to game companies. One of these anti-cheat systems is Easy Anti-Cheat[12], which was recently acquired by Epic Games [20].

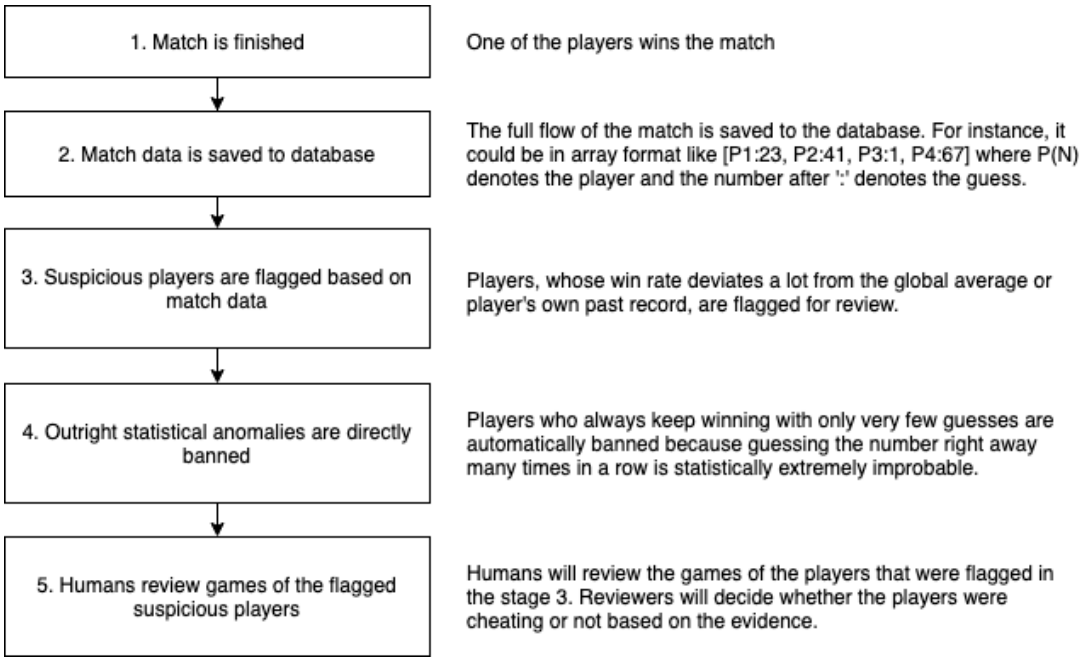


Figure 13: Hypothetical flow of the guessing game's statistics-based anti-cheat

The number guessing game that was implemented for this thesis does not have any statistics-based anti-cheat system. Figure 13 introduces a flow that represents a hypothetical system that would be suitable for this type of game. Since the rounds in the number guessing game are very simple, each round can be recorded in a database by writing down the sequence of moves that happened in the round. The system would flag players who statistically win more games than the average of other players. The recordings would be stored in the database for few months so a human reviewer could review the games if needed. The system would also outright ban players who keep winning many games in a row on only very few guesses because that would be statistically extremely unlikely unless they were cheating.

As more and more games collect huge amounts of data, the use of statistical methods has become very common. It has even spawned companies whose main product is an anti-cheat system that heavily utilizes game data. Examples of these companies are the aforementioned Easy Anti-Cheat [12] and FairFight [17]. Unlike most other methods, it is harder to bypass an anti-cheat that is based on statistical methods because modifying the client code will not help to avoid the anti-cheat as long as the data is correctly handled on the server and the data from the client is not trusted directly. The weaknesses of this type of system are usually the need for large amounts of data and the need for some form of

human oversight, which can be challenging for smaller companies.

When evaluating statistical anti-cheat methods in terms of resistance to tampering, we can conclude that they are very resistant because they are employed on the server side and purely rely on the data that comes from different clients in large numbers. Of course, there is always a possibility that the data is not verified adequately and tampered data slips through. Other techniques such as a tamper-resistant application protocol can help mitigate bad data. A weakness of these types of methods is that the cheater can manage to cheat just enough to not to cross the detection threshold of the anti-cheat. For instance, the cheater can purposefully not use the cheats to the full extent and make it look like normal improvement in player skill level.

Statistical anti-cheat methods are usually difficult to implement because of the need for large amounts of data to be used in the analysis. The developers also need to consider how this data can be used to evaluate the possibility of cheating. The company might need to hire data scientists to tackle the development of this type of anti-cheat because a game developer might not have the needed skills for the data-related work.

The overhead of statistical methods is non-existent because they are usually run in parallel and most likely on different machines. The data analysis does not interfere with the game pipeline at all. However, more data will usually require more computation resources, which means that the game company has to spend more money.

Statistical methods are generally non-invasive if they only collect game-related data and not other types of data from the users' computers. However, the developers need to be careful how to manage possibly sensitive data and make it clear to the user how their data is being handled.

Statistical methods are not well-suited for all types of games, as they require large amounts of data. For instance, a game might have a low playerbase and thus does not have the needed amount of data for statistical analysis. The data also has to be in a format that lends itself well to use in statistical methods. This can require big changes to the game, especially if the anti-cheat is being implemented later.

Overall, statistical anti-cheat methods are among the best tools to fight cheating, especially considering their high tampering resistance, low overhead, and non-invasiveness. All these are reasons why commercial data-based anti-cheat systems like FairFight and Easy Anti-Cheat are increasingly popular and preferred over the old systems that mostly rely on client-side techniques.

7 Client-side anti-cheat methods

Client-side anti-cheat methods are solutions that are run on the player's local machine. Examples of these are the Valve Anti-Cheat (VAC) and PunkBuster, both of which were introduced in Chapter 3. The weakness of client-side anti-cheat approaches is the fact that they are run on the local machine that the cheater has full access to. This chapter will analyze five different client-side anti-cheat methods: code encryption, verifying files by hashing, detecting known cheat programs, obfuscating memory, and kernel-based anti-cheat drivers. Some types of cheats such as wallhacks are very hard to detect using server-side techniques, which is why client-side methods are still relevant today.

In the context of client-side anti-cheat, it is important to remember that mobile devices and gaming consoles are also computers. Many console and mobile game developers have not given much thought to anti-cheat and instead have relied on the assumed hard-to-hack nature of the devices. For instance, Elder Scrolls Blades mobile game lacked an anti-cheat and cheaters released many cheat programs for it [22]. In the context of PlayStation consoles, the console needs to be "Jailbroken", which means that the user gets to run uncertified software and do things that are not permitted on a standard console. This enables the user, among other things, to run cheat programs [33]. Jailbreaking a console is not easy, as it requires taking multiple non-trivial steps. For instance, the first PS4 Jailbreaks exploited a Webkit vulnerability where the payload software was uploaded to a website and the user changed the DNS settings of the PS4 to point to a custom DNS server. After the user went to a certain menu in the PS4 settings, the custom DNS server returned the address of the site that served the exploit to the PS4.

Another important thing to note is that writing robust code is an essential part of preventing vulnerabilities in the game. If the game contains faulty code that enables the user to inject their own code into the game, it makes the game very vulnerable to cheating. For example, if a string is read from user input and put into a buffer of predefined size and the user supplies a string that is larger than the buffer, the remaining part of the string overflows to some other area in the memory. In computer security, this type of attack is known as buffer overflow attack [11]. The cheater can then determine a string of a certain size and make it overflow to the code region of the game and overwrite game code with cheat code and have it executed. While auditing code for vulnerabilities is not an anti-cheat method itself, it is a notable part of making the game client secure.

7.1 Code encryption

Code encryption means the process of encrypting the code section of the game (Figure 14). This can mean full encryption that is removed when the application is launched or partial decryption where parts of the game executable are decrypted once they are needed and then re-encrypted. This chapter will look into different code encryption techniques and analyze their various strengths and weaknesses. It should be noted that the game client has to be able to both encrypt and decrypt the code so the keys need to be stored somewhere in the game memory. This is in contrast to public-key cryptography, where public keys can be distributed to everyone but only the owner has the private key that can be used to decrypt information.

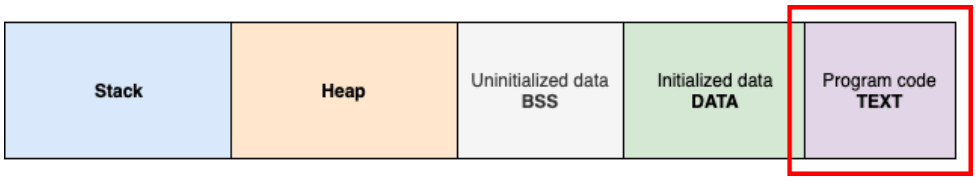


Figure 14: Typical memory layout of an application with the code (TEXT) section highlighted

Code packing is the simplest form of code encryption. Most of the code in the executable becomes unreadable for the static analysis and the code is restored at the runtime by an unpacking routine. The use of code packing is very popular in malware development because malware developers want to make their malicious software harder to detect and analyze. One of the most popular packing programs is called *UPX* [53]. Figure 15 illustrates how the entry point code of the non-packed game executable looks in the reverse engineering tool Ghidra [19]. Both in Figure 15a and Figure 15b, the disassembled and decompiled code is clear. Figure 16 respectively illustrates how the entry point code looks when it is packed. We can note that the packed code in Figure 16 is much harder to make sense of when compared to the unpacked code in Figure 15. For instance, in the non-packed executable in Figure 15, Ghidra can automatically detect and make much sense of the main function while in the packed executable the code does not give much away and Ghidra cannot provide much info about it either. In Figure 17, Ghidra cannot even view the code because it is packed and thus not recognizable to the analyzer. Code packing can be a cost-effective way to give some protection to the game, but if the cheater discovers which packer has been used, the same packer can usually be used to unpack the executable. For example, with UPX the cheater could just run `upx -d <executable-name>` and the unpacked game would be vulnerable to analysis.

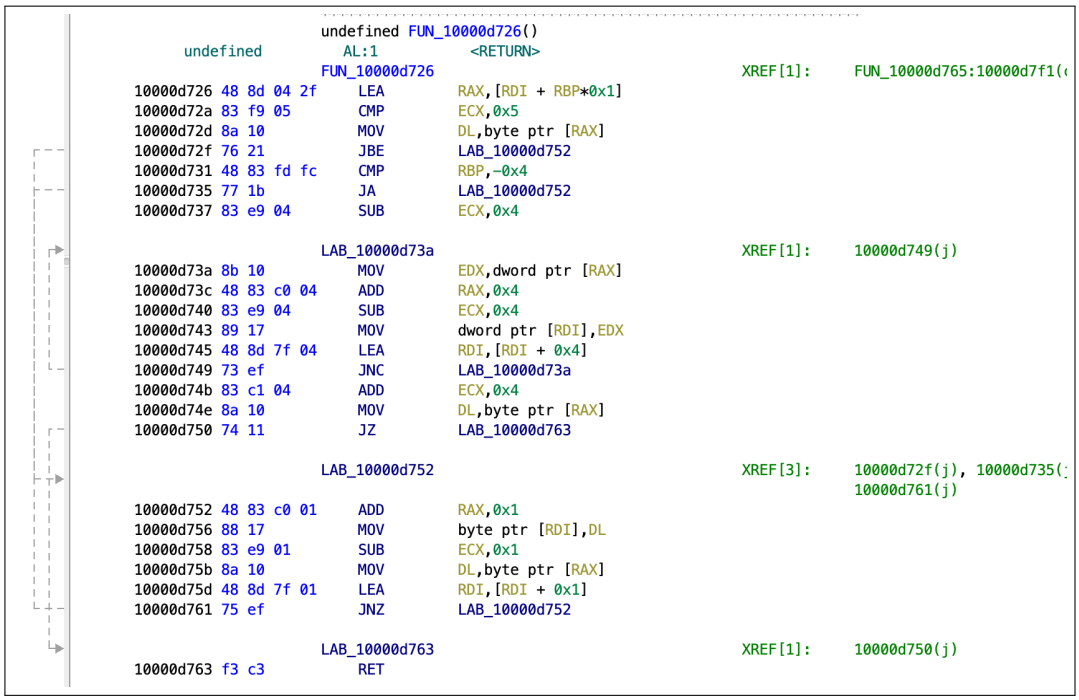
```
                        _main                                           XREF[2]:     Entry Point(*),
                                                                                     entry:10000128f(c)
1000085b8 55              PUSH        RBP
1000085b9 48 89 e5        MOV         RBP,RSP
1000085bc 41 57           PUSH        R15
1000085be 41 56           PUSH        R14
1000085c0 53              PUSH        RBX
1000085c1 48 81 ec        SUB         RSP,0x358
          58 03 00 00
1000085c8 48 8d 9d        LEA         RBX=>local_378,[RBP + -0x370]
          90 fc ff ff
1000085cf 48 89 df        MOV         RDI,RBX
1000085d2 e8 25 f4        CALL        Game::Game                                undefined Game(Game * this)
          ff ff
1000085d7 48 89 df        MOV         RDI,RBX
1000085da e8 0d f7        CALL        Game::Run                                 undefined Run(Game * this)
          ff ff
1000085df 48 8d 7d 80     LEA         RDI=>local_88,[RBP + -0x80]
1000085e3 e8 84 e2        CALL        SceneHandler::~SceneHandler               undefined ~SceneHandler(SceneHan...
          ff ff
1000085e8 48 8d 9d        LEA         RBX=>local_b8,[RBP + -0xb0]
          50 ff ff ff
1000085ef 48 8b 05        MOV         RAX,qword ptr [->sf::UdpSocket::vtable]   = 100012a40
          02 9b 00 00
1000085f6 48 83 c0 10     ADD         RAX,0x10
1000085fa 48 89 03        MOV         qword ptr [RBX]=>local_b8,RAX=>PTR_~UdpSocket_... = 1000079c4
1000085fd 48 8b 7b 18     MOV         RDI,qword ptr [RBX + local_a0]
10000860l 48 85 ff        TEST        RDI,RDI
10000860d 74 0c           JZ          LAB_100008612
10000860f 48 89 bd        MOV         qword ptr [RBP + local_98],RDI
          70 ff ff ff
10000861d e8 02 43        CALL        __stubs::__ZdlPv                          undefined __ZdlPv()
          00 00

                        LAB_100008612                                   XREF[1]:     100008604(j)
100008612 48 89 df        MOV         RDI,RBX
100008615 e8 80 41        CALL        __stubs::__ZN2sf6SocketD2Ev               undefined __ZN2sf6SocketD2Ev()
          00 00
10000861a 48 8d bd        LEA         RDI=>local_e0,[RBP + -0xd8]
          28 ff ff ff
```

(a) ASM code of the non-packed game executable's entry point

```
undefined8 _main(void)

{
  Game local_378 [544];
  byte local_158;
  undefined8 local_148;

  __hash_table<std--__1--__hash_value_type<int,std--__1--function<void(sf--Packet,sf--IpAddress,int)
  >>,std--__1--__unordered_map_hasher<int,std--__1--__hash_value_type<int,std--__1--function<void(sf
  --Packet,sf--IpAddress,int)>>,std--__1--hash<int>,true>,std--__1--__unordered_map_equal<int,std--__
  1--__hash_value_type<int,std--__1--function<void(sf--Packet,sf--IpAddress,int)>>,std--__1--equal_
  to<int>,true>,std--__1--allocator<std--__1--__hash_value_type<int,std--__1--function<void(sf--Pack
  et,sf--IpAddress,int)>>>>
  local_130 [40];

  __hash_table<std--__1--__hash_value_type<int,bool>,std--__1--__unordered_map_hasher<int,std--__1--
  __hash_value_type<int,bool>,std--__1--hash<int>,true>,std--__1--__unordered_map_equal<int,std--__1
  --__hash_value_type<int,bool>,std--__1--equal_to<int>,true>,std--__1--allocator<std--__1--__hash_v
  alue_type<int,bool>>>
  local_108 [40];

  __hash_table<std--__1--__hash_value_type<std--__1--basic_string<char,std--__1--char_traits<char>,s
  td--__1--allocator<char>>,std--__1--basic_string<char,std--__1--char_traits<char>,std--__1--alloca
  tor<char>>>,std--__1--__unordered_map_hasher<std--__1--basic_string<char,std--__1--char_traits<cha
  r>,std--__1--allocator<char>>,std--__1--__hash_value_type<std--__1--basic_string<char,std--__1--ch
  ar_traits<char>,std--__1--allocator<char>>,std--__1--basic_string<char,std--__1--char_traits<char>
  ,std--__1--allocator<char>>,std--__1--hash<std--__1--basic_string<char,std--__1--char_traits<char
  >,std--__1--allocator<char>>>,true>,std--__1--__unordered_map_equal<std--__1--basic_string<char,st
  d--__1--char_traits<char>,std--__1--allocator<char>>,std--__1--__hash_value_type<std--__1--basic_s
  tring<char,std--__1--char_traits<char>,std--__1--allocator<char>>,std--__1--basic_string<char,std-
  -__1--char_traits<char>,std--__1--allocator<char>>>,std--__1--equal_to<std--__1--basic_string<char
  ,std--__1--char_traits<char>,std--__1--allocator<char>>>,true>,std--__1--allocator<std--__1--__has
  h_value_type<std--__1--basic_string<char,std--__1--char_traits<char>,std--__1--allocator<char>>,st
  d--__1--basic_string<char,std--__1--char_traits<char>,std--__1--allocator<char>>>>>
  local_e0 [40];
  undefined *local_b8 [3];
  long local_a0;
  long local_98;
  SceneHandler local_88 [104];

  Game(local_378);
  Run(local_378);
  ~SceneHandler(local_88);
  local_b8[0] = PTR_vtable_1000120f8 + 0x10;
  if (local_a0 != 0) {
    local_98 = local_a0;
    __ZdlPv();
  }
  __ZN2sf6SocketD2Ev(local_b8);
  ~__hash_table(local_e0);
  ~__hash_table(local_108);
  ~__hash_table(local_130);
  if ((local_158 & 1) != 0) {
```

(b) Decompiled C code of the non-packed game executable's entry point

Figure 15: Codes of the non-packed and UPX packed game executable's entry point viewed in Ghidra



```
                        undefined FUN_10000d726()
      undefined           AL:1           <RETURN>
                        FUN_10000d726                           XREF[1]:     FUN_10000d765:10000d7f1(
10000d726 48 8d 04 2f     LEA         RAX,[RDI + RBP*0x1]
10000d72a 83 f9 05        CMP         ECX,0x5
10000d72d 8a 10           MOV         DL,byte ptr [RAX]
10000d72f 76 21           JBE         LAB_10000d752
10000d731 48 83 fd fc     CMP         RBP,-0x4
10000d735 77 1b           JA          LAB_10000d752
10000d737 83 e9 04        SUB         ECX,0x4

                        LAB_10000d73a                           XREF[1]:     10000d749(j)
10000d73a 8b 10           MOV         EDX,dword ptr [RAX]
10000d73c 48 83 c0 04     ADD         RAX,0x4
10000d740 83 e9 04        SUB         ECX,0x4
10000d743 89 17           MOV         dword ptr [RDI],EDX
10000d745 48 8d 7f 04     LEA         RDI,[RDI + 0x4]
10000d749 73 ef           JNC         LAB_10000d73a
10000d74b 83 c1 04        ADD         ECX,0x4
10000d74e 8a 10           MOV         DL,byte ptr [RAX]
10000d750 74 11           JZ          LAB_10000d763

                        LAB_10000d752                           XREF[3]:     10000d72f(j), 10000d735(j
                                                                             10000d761(j)
10000d752 48 83 c0 01     ADD         RAX,0x1
10000d756 88 17           MOV         byte ptr [RDI],DL
10000d758 83 e9 01        SUB         ECX,0x1
10000d75b 8a 10           MOV         DL,byte ptr [RAX]
10000d75d 48 8d 7f 01     LEA         RDI,[RDI + 0x1]
10000d761 75 ef           JNZ         LAB_10000d752

                        LAB_10000d763                           XREF[1]:     10000d750(j)
10000d763 f3 c3           RET
```

(a) ASM code of the non-packed game executable's entry point

```
1
2  /* WARNING: Removing unreachable block (ram,0x00010000d718) */
3
4  void entry(long param_1,long param_2,undefined8 param_3,undefined8 param_4,undefined8 param_5,
5             undefined8 param_6)
6
7  {
8    undefined extraout_DL;
9    undefined7 extraout_var;
10
11   FUN_10000d974();
12   FUN_10000d765(CONCAT71(extraout_var,extraout_DL),param_1,extraout_DL,0,param_5,param_6,
13                 param_2 + param_1,CONCAT71(extraout_var,extraout_DL),param_4);
14   return;
15 }
16
```

(b) Decompiled C code of the non-packed game executable's entry point

Figure 16: ASM code (above) and decompiled C code (below) of the UPX packed game executable's entry point viewed in Ghidra

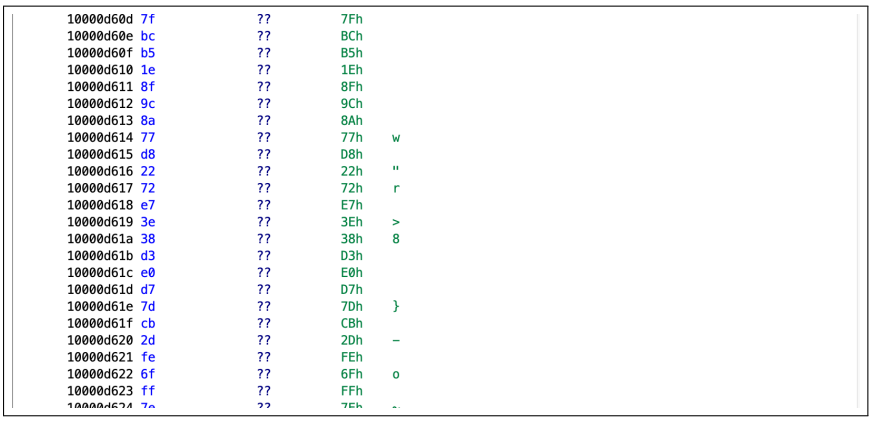


Figure 17: UPX packed code viewed in Ghidra

Cappaert et al. introduce an on-demand decryption framework that will decrypt and encrypt an entire function at once [4]. This is in contrast to typical packing tools that pack an executable to defend against static analysis but use a small unencrypted bootstrap section to decrypt the entire program at the start of the application, thereby making it vulnerable to dynamic code analysis. The problem with this kind of full decryption is that once the application is fully decrypted, one can dump the entire process image to disk and analyze it later. According to [4], the main advantage of encryption is that it hides the internals of the program and protects against static analysis. It is also hard for the attacker to statically change bits because the changed bits will result in bit flips in the decrypted version of the code, which in turn results in modified instructions that may lead to a crash or unintended behavior [4].

Figure 18 shows the process of bulk decryption where the entire program is decrypted at once by the small portion that is not encrypted. Figure 19 shows the partial decryption technique. As we can see in Figure 18, the code is instantly vulnerable after the decryption. Extracting the game executable will require the cheater to observe the code in the unencrypted bootstrap section and implement the algorithm there to fully decrypt the executable. If the game uses partial encryption and decryption, the cheater has to observe the different functions when they are in their decrypted state and build a clear picture of what is happening in the game code. The encryption code could also be changed on each game update to make life harder for the cheaters.

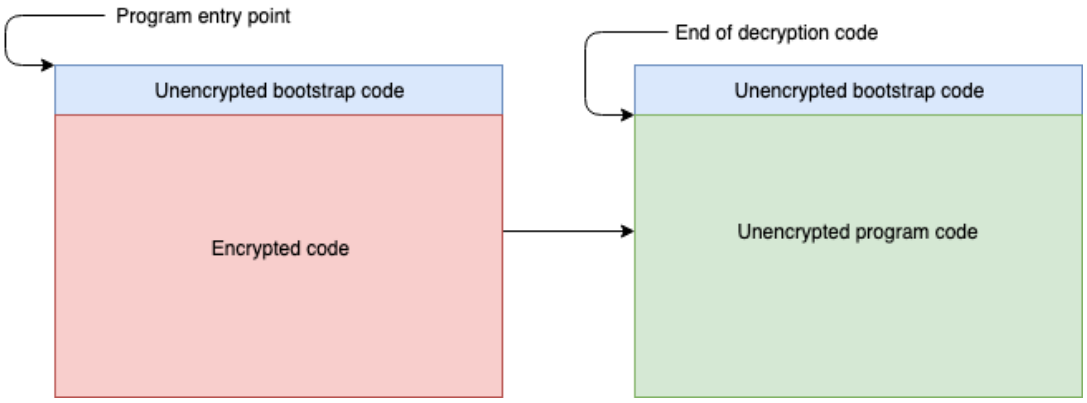


Figure 18: Bulk decryption of the game executable

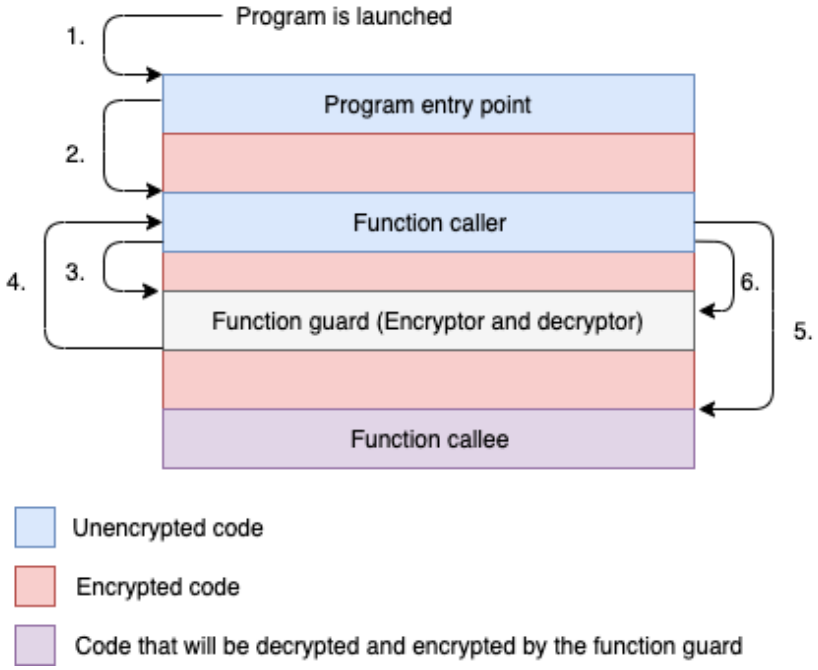


Figure 19: On-demand decryption of the game executable

We will go through the process presented in Figure 19. Step (1) represents the entry point of the program, which is unencrypted because otherwise the operating system would not be able to run the program. In step (2), some unencrypted part of the code wants to call another function that is encrypted. First it needs to call the function guard (3) to decrypt

the function. In this model, the decryption works so that the guard computes a hash of the immediate dominator of the callee and then the callee is decrypted with the hash acting as a key. In step (4), the guard returns and the function caller can call the decrypted function (5). Finally, in step (6) the function caller calls the guard to re-encrypt the function with the same method of calculating a checksum of the immediate dominator of the callee, and the callee is encrypted with the checksum being the key [4]. In short, there are three steps when running a method inside the partially decrypted program: calling function guard to decrypt the method, running the method, and calling function guard again to re-encrypt the method.

Compared to decrypting the whole executable in bulk, the overhead is obviously greater because the decryption and encryption have to be run constantly and not just once at the beginning of the program. Cappert et al. looked at the overhead generated by the full decryption method and the on-demand method. The tests were conducted using Diablo [9] link-time binary rewriter to patch the binary code of different programs, including inserting the extra encryption functionality inside the executables. The results for the bulk decryption method were clear: the overhead was less than 1%. However, Table 1 shows that the execution time of the on-demand decryption method was relatively high on some programs such as *milc* and *sphinx_livepretend* [4]. Note that the Table 1 shows the execution time relative to the unedited version of the same program. For example, *sphinx_livepretend* took 6.65 times longer when compared to version that does not use the on-demand encryption.

Program name	Total functions	Encrypted functions	Execution time
mcf	22	20	1.09
milc	159	146	8.17
hmmer	234	184	3.20
lbm	19	12	1.00
sphinx_livepretend	210	192	6.65

Table 1: Execution time of programs using the standard on-demand decryption model

Table 1 reveals that *milc* encrypted with the on-demand scheme suffers a high performance cost with execution taking more than eight times the time of the non-encrypted base program. It can be observed that the size of the overhead is dependent on the nature of the program, such as how many function calls there are, how large the functions are, and how rapidly they are called over a period of time. Table 1, *milc* details a program used by the MIMD Lattice Computation (MILC) to run large scale numerical simulation

and study quantum chromodynamics, which is the theory of the strong interactions of subatomic physics. The program is very heavy numerically and has many methods that deal with matrix calculations. These methods are run multiple times, which requires constant decrypting and encrypting [30]. If we extrapolate the results in Table 1 to games, they do not bode well because many games use 3D rendering, for example. The graphics engine executes the same rendering functions constantly, and if they are continuously decrypted and encrypted, it can unnecessarily impact performance and make for a poor game experience.

When building an anti-cheat system, we are not generally interested in protecting functions related to rendering, audio, or other functions unrelated to important game mechanics that can be exploited. Cappert et al. propose an improved system that utilizes the dynamic profile information that is generated by Diablo [4]. Based on this information, the so-called hot functions are excluded from the on-demand encryption and are instead protected with bulk encryption. A given function is considered hot if the amount of times it is called exceeds or equals the threshold value that is calculated with the formula below.

$$threshold = min\{i | \sum_{j=1}^{i} calls(f_i) > K \sum_{j=1}^{n} calls(f_i)\}$$

In the threshold calculation formula, the functions are indexed from 1 onwards and are arranged in a decreasing order by the number of times that they are called. Additionally, the formula implies that the threshold represents the smallest number of function calls needed to be classified as a hot function. K represents the ratio of function calls and is a number between zero and one. K is always chosen first and only then the threshold can be calculated. If K was 0, it would mean that all functions were hot, and adversely, if K was 1, it would mean that none of the functions would be hot. N represents the number of functions.

Program name	Total functions	Encrypted functions	Execution time
mcf	22	20	1.04
milc	159	146	1.95
hmmer	234	184	1.15
lbm	19	12	1.00
sphinx_livepretend	210	192	1.72

Table 2: Execution time of programs using the improved on-demand decryption model

When using the improved scheme on the same programs as before, Table 2 shows that the execution times were vastly improved. For instance, the *milc* now has almost four times faster execution time when compared to the scheme without the hot function classification (Figure 1). However, the problem with the hot function classification in anti-cheat systems is that it is purely based on the number of function calls and is done dynamically without any input from the game developer. There could be situations in certain games where we want to on-demand encrypt and decrypt functions that exceed the threshold value but are too important to be left for bulk encryption. With the scheme proposed by Cappert et al., this is not possible to implement.

Based on the principle of on-demand encryption, we now introduce an improved scheme. This model takes into account the unique needs of anti-cheat systems where we want to select functions that we want to protect. In this scheme, the threshold value would be used the same way as the model presented by Cappert et al., but the developer could manually mark certain methods to be exempt from the threshold value check and encrypted on-demand whether or not they exceed the threshold value [4]. This scheme would also relieve the programmer from the extra work of going through all the methods and leave the focus on the methods that the programmer knows will be run many times and need to be protected regardless. Games also often use external libraries and components such as graphics engines and audio tools that game developers do not even touch on the code level. The improved scheme encrypts methods based on the threshold value just like in the [4] on-demand encryption method but it also allows the developer to exempt methods from encryption.

In practice, the improved scheme would work by informing the encrypting software of the function addresses that are exempt. Cappert et al. utilized Diablo link-time binary rewriter to patch the binary code of the program and insert the encryption and decryption functionality. In our case, Diablo or a similar program can be provided with the exempt function addresses and everything else can be done exactly as in the improved on-demand scheme with the hot functions threshold. Figure 20 represents one way to implement the final anti-cheat optimized scheme.

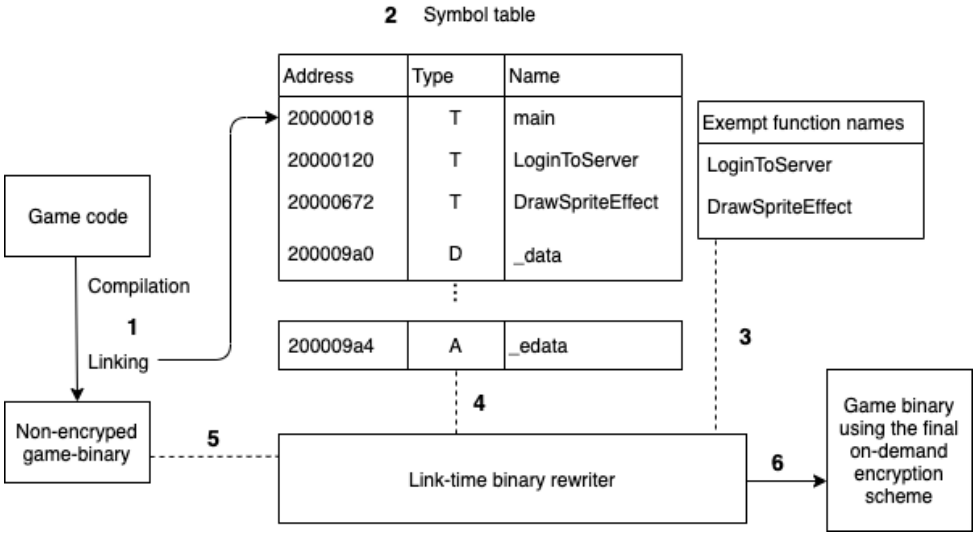


Figure 20: Anti-cheat optimized on-demand encryption scheme

The process in the Figure 20 can be summed as follows:

1. The game code is compiled and then linked to make the unencrypted binary file.
2. During the compilation process the symbol table is formed with the names of all functions and their addresses along with other data such as section addresses.
3. Link-time binary rewriter reads the names of the exempt functions from a file that the developer has created.
4. Link-time binary rewriter reads the symbol table and matches the names of the exempt functions from the step 3 with their addresses.
5. Link-time binary rewriter takes in the original game binary.
6. Link-time binary rewriter modifies the game binary according to the on-demand encryption scheme and writes in the encryption code while taking into account the methods that were exempt by the developer and will always be encrypted.

The baseline speed of the final scheme with zero exemptions is the same as in the threshold version of the on-demand encryption system presented in [4]. The more frequent and complex the exempt functions, the more the performance will suffer. It is impossible to calculate a standard performance overhead that exempted functions generate because they can vary widely. The threshold formula also does not take into account the size of the functions, so it is left to the developer to decide which functions must be protected in

order to thwart cheaters. Evaluating the gained benefit against the implied overhead costs becomes a part of the anti-cheat development process.

Within the group of client-side anti-cheat methods, code encryption stands out as the single core method because it also protects the other anti-cheat approaches such as file verification, cheat program detection, and everything else that has to do with the game memory. For example, if the game's anti-cheat system contains a module that scans the computer for known cheat programs, the code of this module would also be encrypted, which would make it many times harder to crack. In short, the code encryption protects all the other client-side anti-cheat mechanisms from tampering and makes the whole system much more robust.

A well-designed encryption system does not require much work from the developers after building the initial system, so encrypting the game code becomes part of the standard build process. Once the game executable is built, it is run through the encryptor and that is all that needs to be done. With an on-demand encryption scheme that allows developers to exempt methods from encryption, there is almost no reason not to include it in a game's anti-cheat solution.

When looking at code encryption from the viewpoint of tampering resistance, it is clear that code can be decrypted by a skilled cheater. The difficulty of decrypting depends on the code encryption scheme. A partial encryption scheme will be harder to crack than an encryption scheme that decrypts the whole program at the start. When deployed alone, the code encryption does not fare well in terms of tampering resistance. This is similar to what is the case with the other client-side anti-cheat methods.

When it comes to the ease of implementing the code encryption, it is much trickier than many other anti-cheat methods considered in this chapter. Developing a good partial encryption scheme requires major time investments from the developers at the outset. It also requires changes and maintenance when the game is released because the code encryption might be breached by the cheaters. The developers also need to determine which parts of the game to encrypt and which not to. However, the developers could choose to use an existing packer like UPX which would be trivial, but at the same time, it would be much easier for cheaters to circumvent as shown in the start of this section.

Code encryption always introduces some overhead, and as demonstrated earlier, the size of the overhead depends on the amount of encrypted code and the frequency that this code is encrypted and decrypted. Thus the overhead is very dependent on the specific game.

In terms of invasiveness, the code encryption only works within the game's own memory space and has no effect on other programs. The method is suitable for all types of games as it works on a very low-level and is unaffected by what is built on top of it. Obviously,

certain games can be faster-paced and thus generate more overhead.

Overall, the code encryption is a non-invasive anti-cheat method that introduces some amount of overhead depending on how often the code encryption is utilized and how large portions of code are encrypted. Well-implemented partial code encryption provides good tampering resistance, but as with all other client-side methods, it is still vulnerable to skilled cheaters. The main strength of the technique is its ability to protect other anti-cheat methods and make the whole solution tougher to crack.

7.2 Verifying files by hashing

Verifying files by hashing is a method to ensure that the game files have not been modified. In many games, modifying files such as textures on walls can give the player an unfair advantage. For example, the cheater could modify the wall textures to be transparent so all enemies could be seen through walls. The cheater could also adjust the lightning to make it easier to see enemies. Obviously these types of modifications could not be easily caught by any server-side anti-cheat method because they are purely done on the client-side.

In order to combat the changing of important game files, developers can utilize a technique known as hashing where the content of the file is read and a (in practice unique) hash code is generated based on the content. The hash code matches the content of the file exactly, so even if one byte is changed, the hash code will be different. The hash code can also be generated from only the file metadata, such as the date modified and the file size, but this kind of hash generation is much easier to circumvent. Obviously, the cheater can try to modify the hash generator method and generate fake hashes that make it look like the file is unchanged. Preventing method modification requires other anti-cheat methods that are presented in this thesis. This section introduces an example file verification system built on top of the guessing game that was shown in Chapter 2.

Because the example game is very simple and does not contain resource files that many larger games have, we will demonstrate the hashing system by creating a small example resource file and coding the game to take its hash and send it to the server. The server tries to verify the integrity of this file, and if it succeeds, the game is allowed to proceed. In the event that the verification fails, the user is immediately disconnected from the game. Figure 21 represents the file verification system used in our example game. In a more complex real-world game, the server would have a database of hashes of the important game files such as maps and textures.

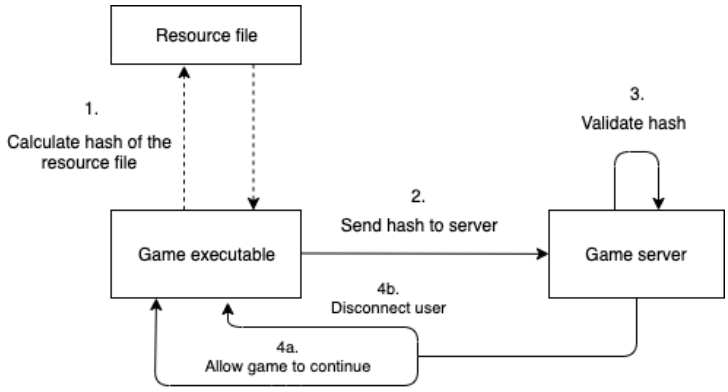


Figure 21: Simple file verification scheme in the example game

Our system presented in Figure 21 works as follows:

1. The game calculates a hash of its resource file.
2. The game sends the hash to the game server.
3. The game server validates the hash by comparing it to the one in its database.
4. Action is taken depending on the result of the validation.
    a If the hash does not match, the user is disconnected from the server.
    b If the hash matches, the game is allowed to continue.

On Listing 3, we can see the code that does the integration check on the client-side. The hash is calculated from the file and sent to the server in a packet using the opcode USER_SEND_FILE_HASH. In the server-side, the server receives the packet and inspects the hash code by comparing it to the hash that is stored on the server-side. Note that the implementation of *std::hash* is platform-dependant so the result might be different on different operating systems [44]. A real game should use its own hashing mechanism that is consistent across platforms.

There are examples of real games that did not have any kind of hash verification of their resources, such as map files. For instance, in early versions of World of Warcraft it was actually possible to edit the map files to be able to go to areas in the game that were normally off-limits. Lack of any file verification allowed the cheater to build bridges in the sky and walk on them. To other players with the correct map files, it would appear as if the cheater was walking in the air because the server was just translating the packets from the client without any verification that the map files were untouched. Obviously nowadays this not possible in most games, at least not easily.

```
void InputNameScene::VerifyGameIntegrity() {
    try {
        // Support Windows and OS X
        std::ifstream in(getPathForResource(), std::ios::in | std::ios::
    binary);

        std::ifstream::pos_type pos = in.tellg();
        std::vector<char> buffer(pos);
        in.seekg(0, std::ios::beg);
        in.read(&buffer[0], pos);

        std::hash<char*> hash;
        size_t resultHash = hash(buffer.data());
        std::cout << resultHash;

        sf::Packet gameIntegrityPacket = this->CreatePacketWithOpCode(
    OpCodes::USER_SEND_FILE_HASH);
        std::string hashAsString = std::to_string(resultHash);
        gameIntegrityPacket << hashAsString;
        this->mPacketHandler->SendPacket(gameIntegrityPacket, this->
    mPacketHandler->GetSendAddress());

    } catch (std::exception &e) {
        std::cout << "Failed to verify game integrity, aborting";
    }
}
```

Listing 3: Code demonstrating the hash calculation in the guessing game client-side

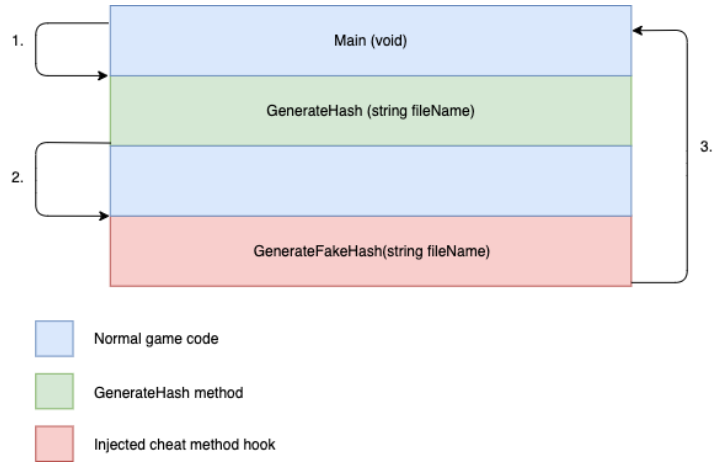


Figure 22: `GenerateFakeHash` cheat method hooked to the `GenerateHash` method

In terms of tampering resistance, a hash-based file verification system is very easy to exploit when deployed alone. As explained, the cheater could edit the client-side hash calculation methods so that they always return the correct hash even if the file was modified (Figure 3 Step 1). This is futher demonstrated in Figure 22. In Figure 22, (1) the game normally calls the `GenerateHash` method, but at the end of the method, the cheater has patched the game so that the `GenerateHash` method actually calls the fake `GenerateFakeHash` method (2). The `GenerateFakeHash` method outputs a fake hash that makes it look like nothing has changed. Instead of the legitimate hash, the fake hash is returned to the caller function (3). This type of cheating is called hooking and it is more closely explored in Chapter 5, Section 5. The hook can be either placed at the start or at the end of the function, like in this example. If the hook is placed at the beginning, the hook usually modifies the data that is passed to the function. If the hook is placed at end, the function result is usually modified and returned to the caller. On it own, the hash calculation method is very vulnerable to being hijacked, as demonstrated in Figure 22.

The cheater could also directly edit the packet data to send the correct hash regardless of what is really calculated (Figure 3, Step 2). The method itself is easy to implement as it only requires a function that calculates the hashes and sends them to the server for verification. Any developer should be able to quickly implement this system.

The file hashing method barely generates any overhead as the hash calculation is usually very quick to perform and it only needs to be performed when the file (or part of the file) is loaded into the memory. If the file is loaded again, the hash verification will obviously need to be performed again.

The hashing method is generally not invasive as it only calculates hashes of the game files which should not contain anything related to the users' privacy. The method also has no limitations when it comes to different types of games.

Overall, while hash-based file verification is easy to implement and has almost non-existent overhead, all these benefits mostly go to waste because it is so easy to circumvent. It is hard to argue that hash-based file verification should be used as a standalone anti-cheat method, but when combined with code encryption, it is harder to tamper with and works well in deterring cheaters from editing the game files. Out of all client-side anti-cheat methods, hash verification is the weakest when it is deployed alone.

7.3 Detecting known cheat programs

A proper anti-cheat program should have a way to scan the user's computer for known cheating programs based on various signatures. The simplest method can simply entail comparing hashes or process names, but these methods are easily circumvented and thus not recommended. In this chapter, we look at different ways of detecting known cheat software on the client computer. The importance of maintaining some form of cheat database is highlighted by the fact that many cheats get sold or otherwise distributed by their developers and quickly become popular among cheaters in the game community. The spread of commercial cheating tools benefits criminals and destroys the game experience of normal players.

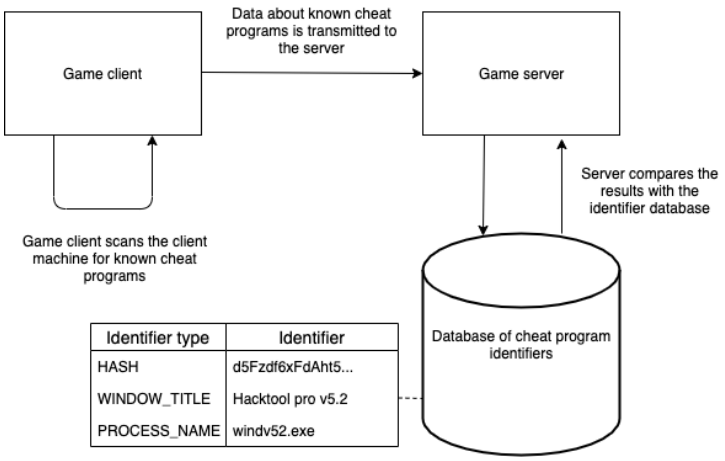


Figure 23: Example of identifier-based anti-cheat system

Figure 23 represents an example of an identifier-based anti-cheat system that has a database of different types of identifiers that can be used to detect known cheating programs. Identifier databases make it possible to use various kinds of markers such as window titles, hashes of executable files, and process names. This makes it harder for the cheater to pinpoint what exact information is being utilized on the server side.

At the implementation level, this anti-cheat method would more or less rely on the API functions of the operating systems to retrieve the list of process names and window titles. For instance, on Windows the method `EnumProcesses` would be used [13]. Without any other anti-cheat methods to support it, the identifier-based anti-cheat is very easy to hijack and make the program send fabricated data to the server. The same problem was highlighted in the hash-based file verification anti-cheat in Chapter 6, Section 2. Figure 24 demonstrates the main issue with using this anti-cheat method as a standalone solution.

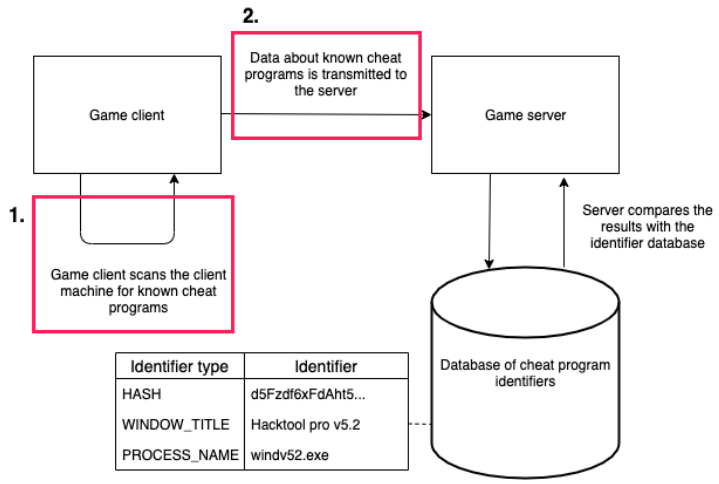


Figure 24: Main problem points with identifier-based anti-cheat

The first weakness in the system, shown in Figure 24, is the scanning that the client performs on the local machine. The cheater can easily manipulate the scanning methods and feed them false data which then gets sent to the server. The cheater can also use other means to evade detection. For example, cheat developers can randomly generate window titles and also randomly change the executable itself, which would break the hashing because the anti-cheat developers would not be able to update the database fast enough. The second weakness is at the point where the client sends the data to the server. Without the help of other anti-cheat methods, the packet can easily be modified on the fly and false data passed to the server. The other way is that the cheater directly hijacks the

methods that send packets to the server and modifies these methods to send false data. As we can see, there are many ways to exploit identifier-based anti-cheat.

Even with a more complex scheme of getting program identifiers, the method would be vulnerable to all of the same exploits. Without the help of other anti-cheat methods, the detection of known cheat programs by the use of identifiers is not an effective way of blocking cheating. When combined with other methods such as code encryption and memory obfuscation, it can be an important addition to the game's anti-cheat toolkit. It provides a way to combat the mass adoption of different anti-cheat programs, allowing anti-cheat developers to catch other people who are utilizing the exact same tools that were discovered earlier by the anti-cheat.

Some cheat tools update themselves automatically and thus they connect to certain addresses. Anti-cheat programs can use this to their advantage and inspect the DNS history for update addresses of known cheats. If the cheating tool is smartly done, it can clear the DNS history, but some cheat developers have overlooked this which has led to detecting many cheaters. In Chapter 3 we discussed Valve Anti-Cheat, which used to access the DNS cache in order to catch kernel-level cheats that contacted their DRM server in order to verify that the user had actually bought the cheat. This allowed the game company Valve to catch and ban many cheaters, which demonstrates that looking though the DNS history can be a powerful addition to other identifiers.

One important factor in the usefulness of this anti-cheat method is that the developer acquires a large amount of cheat signatures. These signatures can be added to the database once a cheat is identified. For instance, the developers of the popular online game PlayerUnknown's Battleground (PUBG) reported that they have around 100 people dedicated to monitoring sites where online cheats are being sold [2]. In this way, the company can quickly acquire the identifiers of the popular cheats and block them. The identifiers can be also acquired automatically by other anti-cheat methods. For example, if the kernel-based anti-cheat driver detects a cheating attempt, it can save the identifiers of the cheat executable and send them to the server.

In terms of the evaluation criteria introduced in Chapter 2, the method fares badly when it comes to resistance to tampering. As pointed out in this section, there are many ways for the cheater to hijack and modify the data that is sent to the server. For example, the cheater can reverse engineer the game and edit the methods to take in false data.

When it comes to the ease of implementation, it really depends on how complex the identification system will be. A simple system is cheap and fast to implement, but a system that uses more complex identifiers (such as parts of the program code) can become very complicated.

Overhead generated by the system will be minimal because it simultaneously collects and sends data to the server and does not interfere with the game logic itself.

An identifier-based anti-cheat method is very intrusive when it comes to the users' privacy, however. Things such as window titles and program names can reveal sensitive data like credit card numbers and other private information.

The method works the same regardless of what type of game it is used on, as it works outside the game logic itself and performs its function simultaneously.

Finally, we can see that the method suffers from the same issues as other client-side methods but on a larger scale because its code is not encrypted and its memory is not obfuscated. If deployed alone, the identifier-based anti-cheat will be easy to circumvent for a skilled cheater.

7.4 Obfuscating memory

All programs have a part called the heap (Figure 25) that contains dynamically allocated variables. Even small programs can have hundreds of thousands of variables in the heap.

Figure 25: Typical application memory layout with heap highlighted

Among the variables in the heap, there may be important data such as player coordinates that are not meant to be seen by the player but only to be processed by the game itself. This vulnerability can be easily exploited by cheaters. For instance, a cheater might be able to narrow down the location of the player coordinates in the memory by scanning all the variables in the memory and then moving the character and removing those variables from consideration that did not change when the character moved (Figure 26). In this way, the cheater can narrow down the actual variables that represent the player coordinates. Furthermore, the cheater can then utilize the location of these values to map out even larger data structures in the memory, such as the player object itself. After having the location of the player object in the memory, the cheater can easily access other values

related to the player. By having the location of important hidden variables, the cheater can develop tools that read these variables automatically.

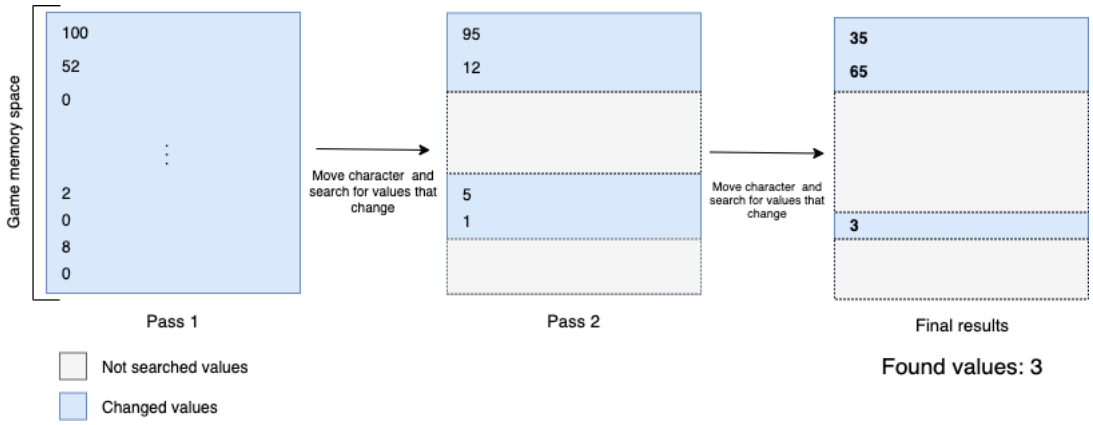
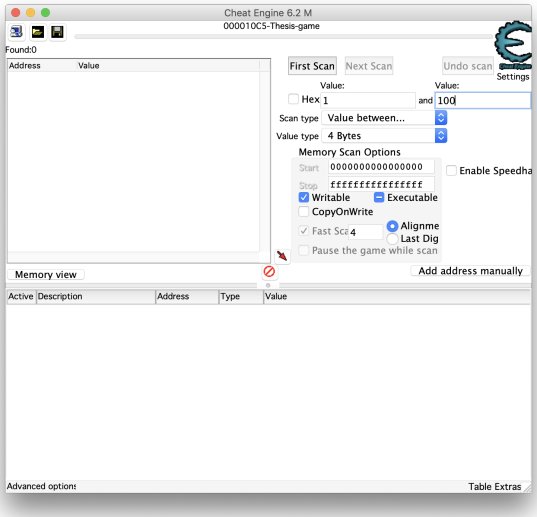
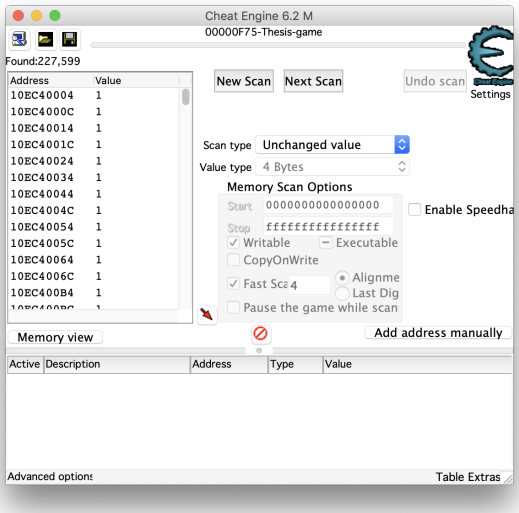


Figure 26: Using memory scanning to narrow down the values that represent player location

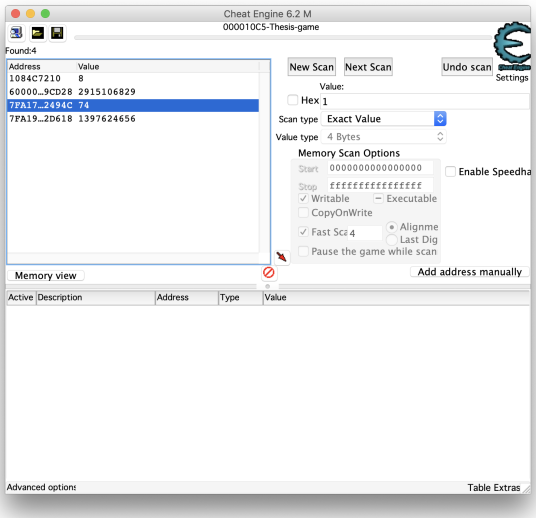
We can demonstrate the ease of this method with the number guessing game developed for this thesis. In this example, we can assume that the game is badly designed and the correct number is stored in the local memory. We can narrow this number down by playing a few rounds of the game and looking for variables that change when a new round begins with a new number. Figure 27 shows the searching of the correct number by using the popular memory scanning and editing tool Cheat Engine [5].



(a) Before scanning



(b) In the middle of the search



(c) The end of the search

Figure 27: Using Cheat Engine to find the correct number from the game memory

The process of reducing the search to only four values took over ten passes and guessing the correct numbers two times (Figure 27). The idea was to first search for all values in the

memory that are between 1-100; the next step was to systematically search for unchanged values every time a player guessed the wrong number, and then search for changed values every time someone guessed the correct number and a new number was generated. The approach was fairly straightforward and reduced the hits from around 200,000 (Figure 27b) to four, as shown in Figure 27c. From the last three hits, we can conclude that the correct number is either 74 or 8 because the other two memory addresses show garbage data.

This example illustrates how strong a simple memory scanning tool, such as Cheat Engine, can really be. In a real game, this method could be used to find all sorts of important data such as player coordinates. Cheat Engine also has a built-in debugger that can be attached to the game. The debugger is useful when the cheater wants to see what code writes or reads from the selected memory address. For instance, in the example game we could see which part of the program writes and reads the correct number variable and then we could try to trace back the code to find out important game methods and see how they are called. Finding important variables is a good entry point into finding other data to utilize in cheating. Often simple variables such as player health can lead the cheater to larger data structures, such as the player object which can contain variables like player position and methods related to the player control.

In the development blog of the popular multiplayer online game League of Legends, Michael VanKuipers outlines a method that tries to make life hard for memory-searching tools. The idea of the method is to move the variables once they are changed so the most basic cheating tools will get confused as the address of the variable changes. The method also encrypts these values when they are moved, and each value uses a slightly different encryption method. Figure 28 represents the improved approach that was outlined by the Riot Games developer Michael VanKuipers [56].

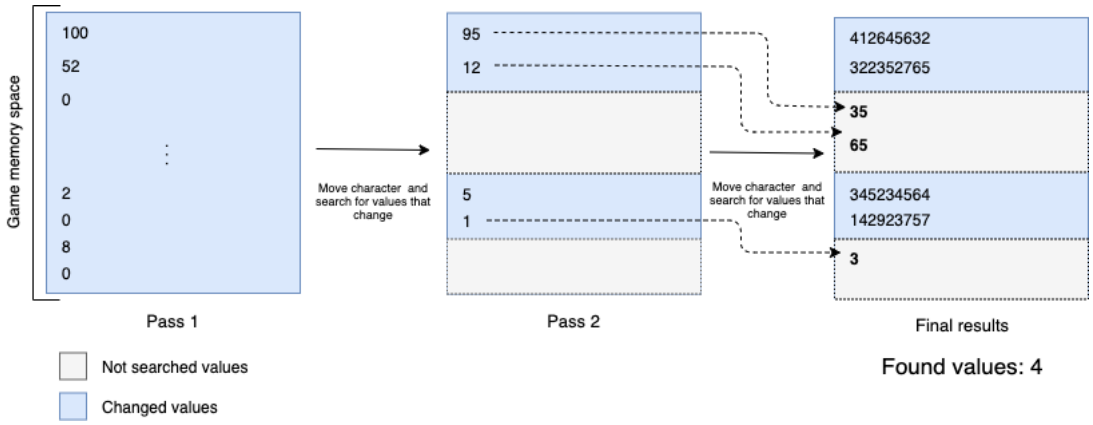


Figure 28: Anti-cheat method that is resistant to memory scanning

In Figure 28, we can clearly see that after the second pass, the values change their locations and thus become harder to find because they are no longer in the memory address that the scanning tool is tracking. Meanwhile, the tracked addresses are now pointing to memory that is undefined, so it could contain any value. This approach will fool the memory scanner to find fake variables that end up not being the correct ones even though it looks like four variables were found like in Figure 28. All the values are encrypted in the heap so they need to be decrypted before use, which will obviously add some overhead to these operations. Additional overhead comes from relocating the value. In short, when a variable is read from the heap, it is read, decrypted, and then the original value is passed on. When a variable is changed, first the new value is encrypted and then placed in the new location while the old location is cleared or filled with fake information.

In order to test how much overhead the relocation and encryption scheme generates, we build a small C++ program that reads and writes a value a million times. This program has three versions: one with the value relocation and encryption, one with only value relocation, and one without any protections. Appendix C shows that the standard method `TestWithoutRelocation` simply reassigns a normal integer value a million times without anything extra. In the method `TestWithRelocation`, the old value is always deleted and the new value is allocated somewhere else in the heap and a pointer to the new place is returned. This means that the place where the old pointer was pointing to becomes undefined and the memory reading tool will just show random data. `TestWithRelocationAndEncryption` is a method that first uses a XOR cipher on the value and then relocates it in the same way as in the previous method. When benchmarking the functions `TestWithRelocation`, `TestWithRelocationAndEncryption` and `TestWithoutRelocation` against each other, we get the results shown in Table 3.

The benchmark was done with 3.6 GHz Intel Core i9, 40 GB 2667 MHz DDR4 running macOS Mojave 10.14.5.

Action	Execution time (ms)
Write value normally	1.544
Relocate value	59.697
Relocate and encrypt value (XOR)	63.851

Table 3: Execution time comparison of writing integer value a million times with three different methods

In Table 3, we can clearly notice that the normal assignment operation is the fastest by a large margin, while the value relocation takes almost sixty times more time. The difference between the method using the value relocation and encryption and the method only using value relocation is not that big. Although in this comparison we used a simple XOR cipher for the encryption, in a real situation we would not want to use encryption that is too heavy, so a simple value transformation serves us for testing purposes. The more complex the encryption, the more time the encryption and decryption would take on read-and-write operations. In order to discover the encryption method, the cheater would have to do some reverse engineering and see how the program encrypts the values. As such, a more complex encrypting algorithm would be discovered almost as easily as the simple one, which would render the extra effort useless. The real question should be whether it is worthwhile to apply the relocation scheme to important variables when considering the performance of the game. It is not reasonable to universally apply it to every single value in the game, but only to values that are important, such as the members of a player class. For instance, it does not matter if the cheater discovers the rendering coordinates of a building in the memory because everyone knows the static location of the building anyway.

Papadoupolos et al. introduce an alternative method to selectively encrypt memory [36]. This can also be applied to encrypting variables in the heap. They introduce a new method `s_malloc` which taints the memory region that it allocates, allowing the program to know whether a memory address has to be encrypted or decrypted before being accessed [36]. The custom `s_malloc` places the size and the starting address of the region at the beginning of each allocation so the decryptor knows what to decrypt. Papadoupolos et al. report overheads ranging from ten times to twenty-six times in terms of the execution time depending on the test. Based on these results, the method using the custom allocator is faster than the previous method presented in this section. The code in Listing 4 shows an example of how to use the custom `s_malloc` method. As we can see, its usage is fully

similar to `malloc` so it will not cause any extra effort when allocating memory. It would even be possible to build a cleaner wrapper on top of the `s_malloc` to make the code cleaner.

```
int main(int argc, const char * argv[]) {
    int *a = (int*)s_malloc(sizeof(int));
    *a = 5;
    return 0;
}
```

Listing 4: Allocating memory with the `s_malloc` method

In the wider context of all anti-cheat methods, memory obfuscation plays an important role because the code encryption itself does not affect variables in the heap, as it only encrypts the code (TEXT) section of the program. If the important values in the heap are not relocated and encrypted, then the cheater can easily find the variables and use them to go deeper into the code by tracking what addresses read and write them. The encryption of heap variables is not as essential as code encryption. Even if the heap was not encrypted, the code found by tracking the variables' reads and writes would still be encrypted. This would make it very hard for the cheater to understand the structure of the program. If the cheater only discovered single variables without understanding the entire context, the damage would not be as high as it would be if all of the code or the network traffic was unencrypted.

A proper anti-cheat should contain relocation and encryption of heap variables because it takes very little development time to implement and the benefit would be very substantial. Even though the overhead was very high in our tests (Table 3) and somewhat less in the tests conducted by [36], we can conclude that some form of heap encryption should at least be used on important variables. The developers need to judge what variables are important and how many variables can be encrypted while maintaining good game performance.

Obfuscating memory is a client-side anti-cheat method and thus is fundamentally susceptible to tampering if the cheater discovers the obfuscation logic. This problem can be alleviated by other anti-cheat methods, but on its own, memory obfuscation is very susceptible to meddling.

Implementing the obfuscation logic is not very straightforward, especially if the developer wants to make it harder to tamper with. Creating the obfuscation scheme with relocation requires deep knowledge of the working of memory, and as such, it is not trivial to build.

When it comes to overhead, memory obfuscation and relocation will cause overhead

regardless of how they are implemented because the game has to take extra steps to resolve the needed value from the memory, as demonstrated in this section. However, depending on the frequency of memory access and the obfuscation scheme, the overhead can be very low and not noticeable at all.

Memory obfuscation only works within the memory space of the game itself and thus does not affect other programs or the users' privacy. Memory obfuscation is a very low-level anti-cheat technique and is totally independent of the type of game it is used on.

Despite all its flaws, memory obfuscation is one of most powerful client-side anti-cheat techniques and should be part of any serious anti-cheat system. At the very least, memory obfuscation can be used to defend the other anti-cheat methods from tampering by obfuscating the memory that they use.

7.5 Kernel-based anti-cheat driver

Many anti-cheat systems are built as standard programs that operate in the user space. This means that they are limited in their ability to protect the game. They are also easier for the cheaters to bypass. Another increasingly popular option is building the anti-cheat system as a kernel driver that operates in the kernel space. The previously mentioned, BattleEye and Easy Anti-Cheat are examples of kernel-based anti-cheat drivers that spy on requests for interfacing with the game process memory. For instance, if a cheat program tries to open a handle to the game process, the kernel driver will detect this and block it.

	User space	Kernel space
Memory access	Limited access	Full access
Hardware access	No direct access	Full access
Access to CPU instructions	Only unprivileged instructions	All instructions
Access to critical OS data structures	No access	Full access

Table 4: Comparison of user space and kernel space privileges

Figure 29 demonstrates how the kernel-level anti-cheat works by communicating with the game process in the user space. The user space program and the kernel can exchange messages. We can see from Table 4 that kernel drivers also have full access to the memory. In essence, kernel drivers can do anything on the computer while user space programs are limited in their access to memory and CPU instructions. These facts make kernel-based

anti-cheat drivers powerful tools for scanning the memory for signs of cheating programs and also for protecting the user space executable.

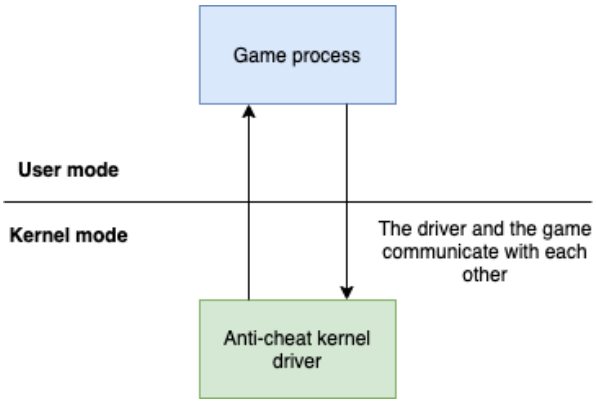


Figure 29: Kernel mode anti-cheat communicates with the game in user mode

The kernel anti-cheat driver can intercept harmful actions that try to affect the memory of the game. For example, Figure 30 shows the kernel anti-cheat driver intercepting an attempt to write to the game memory. In this case, the cheat program could be trying to edit game textures to make it easier to see other players, for instance. Once the kernel driver intercepts and blocks the cheating attempt, it can close down the game and report the cheating to the server so action can be taken against the cheater. The driver can also collect identifiers from the cheat program and send them to the server for further analysis.

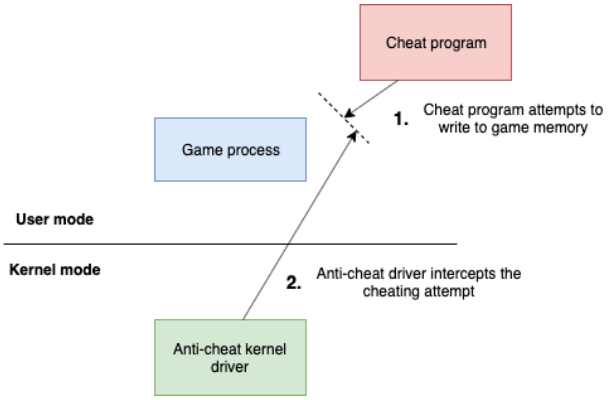


Figure 30: Kernel mode anti-cheat protects the game process

On a practical level, the kernel driver can hook into important system methods that cheats utilize. On Windows, some of the methods commonly used for cheating are: `WriteProcessMemory` [61], `CreateRemoteThread` [59], and `SetWindowsHookExA` [60]. When the cheat program tries to call these methods, the method calls go through the kernel driver first, where their arguments can be analyzed. The kernel driver can terminate the game process if it suspects that the called system method is being used for cheating, otherwise the method will be allowed to run as normal. Figure 31 illustrates an example of inline hooking [21] on a Windows system method `WriteProcessMemory`. We can see that in the hooked method the first line is replaced with a `jmp` instruction to a custom anti-cheat memory section. In this section, the anti-cheat can inspect the arguments of the call to determine whether it is being used for cheating. The anti-cheat can also send data back to the server. At the end of the anti-cheat section, there is a `jmp` instruction to a section with the original code that was replaced by the `jmp` instruction in the original `WriteProcessMemory` function. In the end, the control is returned to the next instruction in the hooked method. Hooking of system methods makes the kernel-level anti-cheat a very powerful tool because it works on a very low-level and is not attached to the game executable.

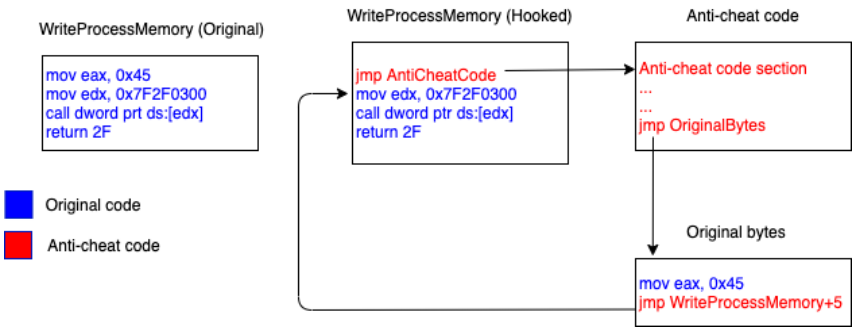


Figure 31: Inline hook for Windows system function `WriteProcessMemory`

The main weakness of kernel-based anti-cheat drivers is that the cheat developers can also develop their own cheats as kernel drivers which can effectively bypass the detection. According to the developers of the popular competitive shooting game PlayerUnknown's Battlegrounds, there are indications that many cheaters are moving their cheats to work in the kernel space to avoid detection [2]. This trend is predictable and is a result of kernel-level anti-cheats becoming more common. The developers noted that defending against kernel-level cheats has been very difficult compared to standard user space cheats. They also report that cheaters are abusing illegally traded certificates to sign the kernel drives that are used for cheating [2].

When it comes to tampering resistance, kernel-level anti-cheat drivers are the best among all different client-side anti-cheat methods. Bypassing them requires the cheat itself to be on the kernel level, which in turn requires the cheat developer to have the skills to write kernel-level drivers on the operating system that the cheat is being made for. The cheaters could also analyze the anti-cheat driver and find vulnerabilities to exploit, but it would also require good skills in reverse engineering, especially at the kernel-level. In addition, at least on Windows and macOS operating systems, kernel drivers need to be signed in order to work. The operating system vendor, in this case Apple and Microsoft, needs to grant the developer the certificate to be able to sign the driver. This requires an application from the developer [47][23]. There are ways to bypass this, such as running the driver in debug mode or using a vulnerable driver that allows programs from the user mode to execute code in the kernel mode. One example of this type of vulnerable driver is the Capcom driver that is installed with the PC version of Street Fighter V. The driver opens a backdoor which allows users to execute code in the kernel space [45][42]. In short, without a kernel-level cheat program, it is difficult to avoid kernel-based anti-cheat.

Kernel-level anti-cheat can be hard to implement for developers with little to no experience in kernel driver programming. Kernel driver development requires a deep understating of how the specific operating system communicates between programs on the user space and drivers on the kernel space. In the case of kernel-level anti-cheats, developer skill is more important compared to other methods because there is a risk of creating new exploits with vulnerable kernel drivers. For instance, the aforementioned Capcom driver opened an exploit for every program to run code in the kernel space [42]. This included malware as well. An unskilled developer should not attempt to create a kernel-level anti-cheat driver because in the worst case it can be vulnerable to exploitation and can put users' computers at risk.

In terms of overhead, kernel-level anti-cheat systems should not generate any more overhead than systems running on the user mode. The overhead depends strictly on what kind of functionality the system performs. Often it protects the game executable by preventing other programs from accessing its memory. In the usual use cases, the overhead is minimal and should not factor into the decision to use a kernel-level anti-cheat.

In the worst-case scenarios, kernel space anti-cheat systems can be very invasive because they have full access to the memory in the user space, which means that they can theoretically snoop around the memory space and gain access to private information. This concern has been raised regarding the use of kernel drivers, and they have even been compared to rootkits [37]. It is up to the developer to maintain ethics and use the kernel-level driver responsibly. Kernel-mode anti-cheats are game agnostic, so they are suitable for all types of games.

It can be argued that out of all the client-side anti-cheat methods, the kernel-level anti-cheat driver is the most effective, especially if it is implemented in the correct way. It can also be combined with some of the other client-side anti-cheat methods in order to make them more effective. For example, file verification by hashing and detection of known cheat programs can be implemented by the kernel driver to enhance their tampering resistance. All this makes kernel drivers a very powerful tool in the fight against cheating.

8 Comparison of anti-cheat methods

This chapter will draw a comparison between the anti-cheat methods covered in this thesis. All methods will be rated from one to four (1-4) with four being the best score and one being the worst. The methods will be rated with five different criteria: resistance to tampering, ease of implementation, lack of overhead, non-invasiveness, and suitability for a wide variety of games.

	Resistance to tampering	Ease of implementation	Lack of overhead	Non-invasiveness	Suitability for wide variety of games
Server-side methods					
Not trusting the client	4	2	2	4	3
Tampering resistant application protocol	4	2	2	4	4
Obfuscating the network traffic	2	4	3	4	4
Statistical methods	3	1	4	4	2
Client-side methods					
Code encryption	2	1	2	4	4
Verifying files by hashing	1	4	4	3	4
Detecting known cheat programs	1	2	4	1	4
Obfuscating memory	2	2	2	4	4
Kernel-based anti-cheat driver	3	2	4	2	4

Table 5: Comparison of different anti-cheat methods

When we look at the client-side and server-side anti-cheat methods as a whole, we can conclude that the client-side methods are much more susceptible to tampering than the server-side methods. As discussed in Chapter 6, the fundamental limits of client-side anti-cheat methods make them bad in terms of tampering resistance. If the cheater has access to the client machine, it is impossible to fully protect the game in a foolproof manner. A lot can be done to improve tamper resistance, but tampering cannot be totally ruled out. Despite being strong anti-cheat methods, even code encryption and obfuscated memory can be bypassed by a skilful cheater who has enough time. Out of all the client-side anti-cheat methods, the kernel-based anti-cheat driver was found out to be the most tampering resistant. This is because the kernel-level anti-cheat module operates in kernel mode, which puts it out of reach of cheats that only operate in user mode. While the server-side methods are superior to the client-side methods in terms of tampering resistance, it does not mean that they are superior as a whole because there are certain types of cheats that the server-side methods simply cannot easily catch. For instance, in Chapter 2 we mentioned wallhacks that edit game textures so the cheater can see through walls. By using pure server-side methods, it is very difficult to catch this type of cheating.

In terms of being easy to implement, there is no major difference between client-side and server-side methods. We can observe that the obfuscation of network traffic is the easiest method to implement. This is due to the fact that encryption and decryption only have to be handled in a single place in the code. On the other hand, statistical methods are usually hard to implement because they need large amounts of suitable data, as was covered in Section 5.4. Out of the client-side methods, the code encryption is the hardest to implement in such a way that it is hard for the cheaters to bypass. A naive method, like using a premade executable packing tool as demonstrated in Section 6.1, is always relatively fast and easy to apply, but actually creating a partial encryption scheme will take a skilled developer a lot of time. The ease of implementation is something that should be considered when developing anti-cheat, especially if the development team has constraints when it comes to skills or time. However, the ease of implementation should never be the main driver for developing any anti-cheat. On the contrary, the server-side methods are very tamper-resistant since the client does not have access to the server-side. The network traffic obfuscation stands out as being easy to tamper with, because if the cheater manages to figure out the encryption algorithm and finds the encryption key, all the packets can be decrypted and fake packets can also be sent with the same encryption. If a cheater manages to feed the server with fake data in large quantities, it can interfere with the statistical methods and cause issues for the anti-cheat.

The overhead of different methods can be a concern for certain types of games that are dependent on a very smooth game flow. None of the methods really stand out in this

regard, but it is clear that out of the client-side methods, code-encryption and memory obfuscation create the most overhead because they need to be performed as often as encrypted methods are run and memory is accessed. During the execution of the program, the memory is constantly being read and written and methods are executed continuously. In general, the more times the method is run and the longer it takes, the more overhead is generated. For instance, statistical methods generate no overhead when they are applied correctly. This is because they can be run on different machines and not in the same pipeline as other game actions. Even today with very powerful computers and mobile devices, overhead can become an issue; for example, if code encryption is used for a large method that is run almost constantly. None of the methods will cause excessive overhead if they are applied reasonably, but when using the selected method, the developers should pay extra attention to overhead if the method has a low score in the lack of overhead column. With code encryption, the developers should exercise special caution and try to only encrypt the most important methods.

Invasiveness of anti-cheat has been a concern in recent years, as noted in Chapter 3. Users are becoming increasingly concerned about their privacy and thus anti-cheat developers should consider the privacy concerns of the various anti-cheat methods. If we look at all the methods as a whole, we can notice that the server-side methods have close to no privacy concerns because they do not operate on the client computers. As for the client-side anti-cheat methods, the detection of known cheat programs has to rely on some kind of marker, such as process names and window titles, but these may contain personal info like credit card numbers. This information can be misused when it is sent to the server for analysis. When implementing the detection system for existing cheat programs, it is very important to pay attention to what kind of data is being scanned and transmitted to the server. It is also recommended to have policies within the company regarding the analysis of this data in the case that something very sensitive gets transmitted.

The privacy concerns become particularly serious if the anti-cheat operates on the kernel level, as kernel-level drivers can access everything on the user's computer. In some cases, file hashes can also give out information about some programs that the user uses, which demands particular attention from the developers. The European Union has introduced the General Data Protection Regulation (GDPR) which gives EU citizens a lot of rights when it comes to the usage of their data. For instance, these rights include the right to be forgotten, which means that the company is obliged to remove the user's data if the user requests it [18]. In terms of invasiveness, it is obvious from the results that the developer should prefer server-side anti-cheat whenever possible. However, as pointed out in the analysis, server-side methods have limitations that make it unfeasible to use them on their own.

Most anti-cheat methods are suitable for all types of games but various concerns need to be taken into consideration based on game type. We can notice that statistical methods are not very suitable for some types of games. As explored in Chapter 6, Section 4, statistical methods require a large amount of suitable data, which means that the playerbase of the game has to be large enough. Some games do not attract that many players or they do not necessarily have the kind of data that lends itself well to analysis by statistical methods. In Chapter 6, Section 1, we explored how to build a client-server system that does not blindly trust what comes from the client. We noted that in some cases it can be reasonable to build a system that does not verify everything on the server side. For instance, we pointed out that in some countries the internet connections are very unreliable. In Chapter 6, we used some mobile games as an example. In these types of situations, it can be reasonable to save bandwidth and server resources by simulating more things in the client. However, this will open many cheating opportunities. This is why sometimes it is not straightforward to build a totally foolproof system in terms of trusting the client data. All the other anti-cheat methods are game-type independent and do not require further consideration related to that.

9 Future of Anti-Cheat

None of the client-side anti-cheat methods are totally safe from tampering, as we have noted throughout this thesis. This is one reason why the future trends of the anti-cheat systems seem to heavily prefer server-side methods. Unless the player has access to the server, it is very difficult to tamper with the server-side methods if they are correctly implemented. This chapter takes a look into promising avenues that state-of-the-art anti-cheat systems are already starting to utilize or that they can start utilizing in the near future.

9.1 Utilizing machine learning and the cloud

In Chapter 5, Section 4, this thesis discussed the use of statistical methods in detecting cheating. Overall, these methods were found to be very effective and non-invasive, but they required large amounts of data that all games do not have. Sometimes it is hard to differentiate statistical methods from machine learning, and different groups have different definitions of these two concepts. In this thesis, techniques that utilize deep neural networks are classified as machine learning instead of pure statistical methods.

Machine learning is starting to be increasingly utilized in the anti-cheat space. For example,

Valve is utilizing deep learning to catch cheaters. According to Valve's John McDonald, Valve's current machine learning system VACnet detects cheaters and then submits the cases to the Overwatch system for the players to review and judge [29]. McDonald stated that the implementation of the VACnet boosted the conviction rate of cheaters in the Overwatch system from 15%-30% to 80%-95% [29]. It is a staggering increase that clearly shows the VACnet is detecting people who are obviously cheating. However, McDonald also noted that the current implementation of the VACnet can only catch players that are clearly cheating by a significant margin. For instance, if a cheater sets a weapon to hit on every shot, then the machine learning system will detect it. If the cheater makes the weapon only hit 60% of the time, then the system might not catch it since it is not very obvious [29]. This highlights the weakness of the current implementation. With more time and more data, the accuracy of the system should improve.

Easy Anti-Cheat is a good example of a SaaS (Software as a service) anti-cheat system that partly operates in the cloud and partly on the client machine. It is used in many popular games such as Fortnite and is somewhat similar to older programs like nProtect GameGuard and Hack Shield. Many of the older systems, such as nProtect GameGuard, acted as rootkits when they were installed on the client machine, which today is heavily criticized due to the very malware-like invasive functionality. In Chapter 5, Section 4, we briefly looked into Fairfight, an anti-cheat system that purely operates on the server-side. It is also utilized in many popular games such as Battlefield V. Easy Anti-Cheat and Fairfight represent the next generation of anti-cheats that are at least partly run in the cloud and that utilize large amounts of data to detect cheaters.

In an era when privacy is becoming more and more important, the machine learning approach is much more preferable over a system that continuously scans the client computer. If the machine-learning-based system is operated on a cloud by a third-party company, it is also a good deal in terms of development cost and scalability. The game company itself does not need specialized people who are solely focused on anti-cheat as that has been externalized to another company. The external company would be fully specialized in anti-cheat development and have deep expertise in this area compared to most game companies that have few people developing their anti-cheats. The main weakness of the machine learning approach is that there can be a situation where the system learns something wrong, and this can be hard to notice afterwards. There is also the possibility of making false decisions based on the statistics. There is no doubt that these types of fully server-side, data-science-utilizing anti-cheat systems will become more and more common as cheats on the client-side become more and more sophisticated. However, because machine-learning-based anti-cheat is totally on the server-side, it is very hard to detect client-side cheats such as wallhacks if the cheater uses them stealthily. Despite the power of the machine learning systems, it is hard to see a future where machine-learning-based

solutions overtake all other forms of anti-cheat.

9.2 Streaming games over the internet

A promising avenue for future anti-cheat is moving the game totally to the server-side, which would eliminate all possibility of using client-side cheating pathways. In practice, this would mean that video of the game is streamed to the players and the keyboard or controller inputs are relayed to the server over the internet. Now the only avenue of cheating would be to exploit the text input to cause a buffer overflow or exploit some other vulnerability (as briefly mentioned at the start of the Chapter 6). Assuming that the whole game is run on the server side and free of vulnerabilities, the system would be very close to being waterproof as long as the players did not have any access to the server. A good example of a game streaming system is Google's Stadia. Stadia runs the games on the cloud and the end user downloads the Stadia application on the chosen device. The user can then use the Stadia application to purchase and play games. The system has a free tier that allows users to stream games on 1080p and 60fps, but the monthly payment option allows streaming up to 4k and 60fps [43]. Whether Stadia succeeds remains to be seen, as there have been similar streaming services, such as OnLive, that have closed down [26].

Obviously moving the game totally to the server-side would introduce many other problems such as the requirement for having a fast internet connection in order to stream the game with high-quality video. There would also be more latency because firstly, the keyboard or controller input would have to travel to the server, and the updated video would have to be streamed back to the player. In a standard client-server situation, the game client would move the character locally instantly and the input would be sent at the same time, which would make the game appear very smooth.

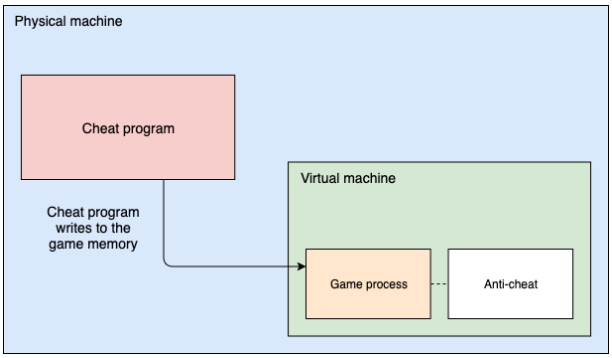


Figure 32: Cheat program writes to the memory of the game that is running inside a virtual machine

A big problem plaguing modern games is the use of virtual machines in cheating. The cheaters run the virtual machines inside their own computers. Figure 32 shows a cheat program operating inside the client machine that is running the virtual machine. The cheat program can be seen writing to the memory of the game that is run inside the virtual machine. Detecting this type of cheating is a very hard problem to solve because the host computer can access the virtual machine from the outside but the virtual machine cannot see the host machine, as it is totally isolated from it. This creates a big opportunity for cheaters to play the game on a virtual machine and use the cheats on the host machine to manipulate the state of the virtual machine. Currently games use certain methods, such as looking for virtualized environment artifacts like registry keys, hard disk names, and drivers, to detect whether the machine running the game is in fact a virtual machine. Usually, if the anti-cheat detects a virtual machine, it will block the player from the game. However, a smart cheater can still manage to bypass all these checks and fool the anti-cheat to believe that it is not being run on a virtual machine. Streaming the game would also provide a viable solution to the virtual machine problem, which can otherwise be challenging to tackle with traditional anti-cheat methods.

While streaming games over the internet makes cheating extremely difficult, in reality, it can be hard to convince players to give up control of their game clients and make them use game streaming services. Some games such as first-person shooters (FPS) require very low latency, and video streaming can make playing those games very uncomfortable because of the extra delay. Some people, especially in developing countries, still have very slow internet connections, which makes this type of system outright impossible for those players. In the future internet connections will become better and better, so the latency will be less of a problem. When services transition to online, the ownership of the products can also become a concern. For instance, when OnLive shut down, the users who had purchased

the company's PlayPass games could no longer access their games. The company gave refunds to certain eligible users [26]. The case of OnLive certainly does not increase the users' faith in game streaming services.

10 Conclusion

This thesis looked at the various anti-cheat methods that are used today and compared them across five different categories: resistance to tampering, ease of implementation, lack of overhead, non-invasiveness, and suitability for a wide variety of games.

Server-side anti-cheat methods were overall quite resistant to tampering due to the client not having physical access to the server, but at the same time, it is hard for them to detect cheats like wallhacks that rely on modifying client-side data such as textures. Network traffic obfuscation was found to be relatively easy to tamper with in comparison to the other methods. Server-side methods were found to be as difficult to implement as client-side methods overall.

When it comes to overhead, code encryption and memory obfuscation generated the most overhead due to the fact that they are run constantly. We concluded that developers should think carefully about which situations to apply these methods. In the case of code encryption, we presented an improved method where encrypted methods would be determined by a specific formula and the developers could manually exempt certain methods from being encrypted. We showed a faster method of memory obfuscation where a custom malloc method was used to allocate memory. The method tainted the memory region that it allocated to the game data. This allowed the program to know if a memory address had to be encrypted or decrypted before accessing it. Among the server-side anti-cheat methods, the tamper-resistant application protocol and the principle of not trusting the client were concluded to introduce some overhead compared to naive implementations.

In terms of non-invasiveness, the server-side methods were not found to be invasive because of their fundamental nature of acting on the server. The detection of known cheat programs stood out as the most invasive client-side anti-cheat method, as it often accesses identifiers such as window titles and process names that can contain sensitive information like credit card numbers. This can be avoided by having company policies regarding the handling of this type of information. Kernel-based anti-cheat drivers were also found to be invasive since they have full access to the user space memory. In theory, this could mean that the anti-cheat could spy on everything the user does, which places a lot of responsibility on the game developer. Despite the privacy concerns, we determined that kernel-based

anti-cheat drivers and the detection of known cheat programs are powerful methods at curbing cheating.

All of the methods except two were suitable for all types of games. Statistical methods and not trusting the client were concluded to have some limitations based on the game type and the number of players. Statistical methods need large amounts of relevant data to make conclusions. Many smaller games might not have the number of players required for this, which limits the options of applying statistical methods for anti-cheat purposes. In some situations, there might also be constraints that make it very difficult to build the server in a way that the client is never trusted directly. As an example, we explained that some mobile games that operate in areas with a bad internet connection might have to compromise on their anti-cheat systems because making the game more cheat-proof would generate much more network traffic.

We concluded that new anti-cheat methods are needed to address the fundamental shortcomings of the current methods. When we looked at the future of anti-cheat, we highlighted the use of machine learning solutions where large amounts of data would be used in sophisticated ways to catch cheaters. As an example, we looked at the Valve's new VACnet anti-cheat that managed to boost the conviction rate of cheaters from 15%-30% to 80%-95%. We also looked into Fairfight, which is a purely server-side cloud-based anti-cheat that works on game data that the developer specifies. We concluded that the cloud-based methods would become more popular because they are easy for the developers to hook into their games. By using these kinds of systems, the developers do not need to worry about maintaining the anti-cheat while they are maintaining their game. The developers do not need special skills that are required in anti-cheat development.

In terms of preventing cheating, it was concluded that streaming games over the internet is the most effective method. This would isolate the game from the players, and essentially the only thing that would be transmitted is the input from the player and the video stream of the game from the server. It was also shown that game streaming still has many obstacles and might not fundamentally be suitable for certain types of games that require very low latency. Issues of the game ownership, in the case that the service is shut down, were also raised.

The overall trend, as seen from the widely used commercial anti-cheat systems, seems to be moving towards server-side solutions as a whole. On the client-side, both cheats and anti-cheat systems are quickly moving to the kernel space. The results of the comparison give supporting evidence of this trend in the sense that server-side methods are much more tamper-resistant and often less invasive than their client-side counterparts. Many products, such as Easy-Anti Cheat and Fairfight, are being built to act as cloud-based anti-cheat solutions that the developers can easily plug into their games and let external

companies handle the anti-cheat.

Based on the comparison, it is clear that the developers should prefer server-side methods over client-side ones but they should also understand that server-side solutions cannot prevent all types of cheating, such as client-side texture modifications that, for example, could allow the cheater to see through walls.

On the server-side, the developer should ideally implement all the server-side methods that were examined in this thesis as they can work in a synergistic manner and do not interfere with each other.

On the client-side, the developer should focus on implementing good memory obfuscation and code-encryption. If possible, a kernel-based anti-cheat driver should be built to protect the user-level game executable from tampering. The kernel-level module could also do memory scanning for known cheats. File hashing was found to be redundant and susceptible to tampering so it should be placed below the other methods in terms of priority. In the future, if the internet connection speeds have developed enough and if the game does not require very low latency, the developer can also consider building the game as a fully streamed service where only the player inputs are sent to the server and a video stream of the game is returned back. This would almost totally eliminate client-side cheating. Among all anti-cheat methods we explored, both present and future options, the streaming solution was found to be the most resilient against cheating.

The fight against online cheating is more important than ever as more and more game companies earn their revenue from online games and players compete in tournaments that have price pools reaching millions of dollars. This thesis provided an overview and comparison of widely used anti-cheat methods alongside a view into the future of anti-cheat. With a better understanding of the different anti-cheat methods, the developers can spend their time more efficiently and focus on implementing an anti-cheat system that is best suitable for their game. The knowledge in this thesis can also help the game developers in purchasing an existing commercial anti-cheat service that they can integrate into their game. Without effective anti-cheat, it is almost impossible for a game to become very popular as players will quit if their game experience is hindered by unfair play. All in all, online cheating is not going to disappear any time soon, which means that new methods and a better understanding of the existing methods are required to have a chance in the fight against a phenomenon that has ruined many games.

References

[1] URL: https://www.blizzard.com/en-gb/legal/fba4d00f-c7e4-4883-b8b9-1b4500a402ea/blizzard-end-user-license-agreement (visited on 03/05/2020).

[2] *A Letter from the Anti-Cheat Team*. Feb. 2019. URL: https://www.pubg.com/2019/02/26/a-letter-from-the-anti-cheat-team/ (visited on 02/14/2020).

[3] Luigi Auriemma and Donato Ferrante. *Multiplayer Online Games Insecurity (Never Feel Safe While Playing Online)*. URL: https://media.blackhat.com/eu-13/briefings/Ferrante/bh-eu-13-multiplayer-online-games-ferrante-wp.pdf (visited on 02/14/2020).

[4] Jan Cappaert et al. "Towards Tamper Resistant Code Encryption: Practice and Experience". In: *Information Security Practice and Experience*. Ed. by Liqun Chen, Yi Mu, and Willy Susilo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 86–100. ISBN: 978-3-540-79104-1.

[5] *Cheat Engine*. URL: https://www.cheatengine.org/ (visited on 02/14/2020).

[6] *Client Server Model*. URL: https://docs.unrealengine.com/udk/Three/ClientServerModel.html (visited on 02/14/2020).

[7] Bootie Cosgrove-Mather. *Cheats Could Ruin Online Gaming*. Dec. 2002. URL: https://www.cbsnews.com/news/cheats-could-ruin-online-gaming/ (visited on 02/14/2020).

[8] *Deceiving Blizzard Warden*. URL: https://hackmag.com/uncategorized/deceiving-blizzard-warden/ (visited on 02/14/2020).

[9] *Diablo*. URL: https://diablo.elis.ugent.be/ (visited on 02/14/2020).

[10] Julian Dibbell. *The Life of the Chinese Gold Farmer*. June 2007. URL: https://www.nytimes.com/2007/06/17/magazine/17lootfarmers-t.html (visited on 02/14/2020).

[11] Wenliang Du. *Computer Security: a hands-on approach*. CreateSpace, 2017.

[12] *Easy Anti-Cheat*. URL: https://www.easy.ac/en-us/ (visited on 02/14/2020).

[13] *EnumProcesses function (psapi.h)*. URL: https://docs.microsoft.com/en-us/windows/win32/api/psapi/nf-psapi-enumprocesses (visited on 02/14/2020).

[14] *Ettercap*. URL: https://www.ettercap-project.org/ (visited on 02/14/2020).

[15] Jon Fingas. *Bungie pulls popular gun from 'Destiny 2' after discovering exploit*. Oct. 2019. URL: https://www.engadget.com/2019/10/20/bungie-pulls-telesto-from-destiny-2-after-exploit/ (visited on 02/14/2020).

[16] Lorenzo Franceschi-Bicchierai. *For 20 Years, This Man Has Survived Entirely by Hacking Online Games*. July 2017. URL: https://motherboard.vice.com/en_us/article/59p7qd/this-man-has-survived-by-hacking-mmo-online-games (visited on 02/14/2020).

[17] *GameBlocks - Server Side Anti-Cheat*. URL: https://gameblocks.com/ (visited on 02/14/2020).

[18] *GDPR Key Changes*. URL: https://eugdpr.org/the-regulation/ (visited on 02/14/2020).

[19] *Ghidra*. URL: https://www.nsa.gov/resources/everyone/ghidra/ (visited on 02/14/2020).

[20] Jill Grodt. *Epic Acquires Easy Anti-Cheat Company For Fortnite*. Oct. 2018. URL: https://www.gameinformer.com/2018/10/09/epic-acquires-easy-anti-cheat-company-for-fortnite (visited on 02/14/2020).

[21] *Guest Diary (Etay Nir) Kernel Hooking Basics*. URL: https://isc.sans.edu/forums/diary/Guest%20Diary%20Etay%20Nir%20Kernel%20Hooking%20Basics/23155/ (visited on 02/14/2020).

[22] Samuel Horti. *Elder Scrolls Blades hacks explained: what do cheats and mods do and what are the risks?* May 2019. URL: https://www.gamesradar.com/elder-scrolls-blades-hacks-cheats/ (visited on 02/14/2020).

[23] *Kernel-Mode Code Signing Requirements - Windows drivers*. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/install/kernel-mode-code-signing-requirements--windows-vista-and-later- (visited on 02/14/2020).

[24] Sheith Khidhir. *ASEAN's poor mobile internet connectivity*. Nov. 2018. URL: https://theaseanpost.com/article/aseans-poor-mobile-internet-connectivity (visited on 02/14/2020).

[25] *Look! what is Blizzard doing on your pc?* Nov. 2006. URL: http://news.mmosite.com/content/2006-11-26/20061126193343858.shtml (visited on 02/14/2020).

[26] Josh Lowensohn. *Sony buys streaming games service OnLive only to shut it down*. Apr. 2015. URL: https://www.theverge.com/2015/4/2/8337955/sony-buys-onlive-only-to-shut-it-down (visited on 02/14/2020).

[27] Sreekanth Malladi, Jim Alves-Foss, and Robert Heckendorn. "On Preventing Replay Attacks on Security Protocols". In: *Proc. International Conference on Security and Management* (June 2002).

[28] Alissa McAloon. *South Korea cracks down on cheaters with law targeting illicit game mods*. Dec. 2016. URL: https://www.gamasutra.com/view/news/286852/South_Korea_cracks_down_on_cheaters_with_law_targeting_illicit_game_mods.php (visited on 02/14/2020).

[29] John McDonald. *GDC 2018: John McDonald (Valve) - Using Deep Learning to Combat Cheating in CSGO*. Mar. 2018. URL: https://www.youtube.com/watch?time_continue=101&v=0bhK8lUfIlc (visited on 02/14/2020).

[30] *MILC*. URL: http://www.physics.utah.edu/~detar/milc/ (visited on 02/14/2020).

[31] Chris Morris. *Electronic Arts' online folly*. Mar. 2003. URL: https://money.cnn.com/2003/03/04/commentary/game_over/column_gaming/ (visited on 02/14/2020).

[32] Gabe Newell. *Valve, VAC, and trust*. Feb. 2014. URL: https://www.reddit.com/r/gaming/comments/1y70ej/valve_vac_and_trust/ (visited on 02/14/2020).

[33] David Nield. *Is Jailbreaking and Modding Still Worth It in 2019?* Aug. 2019. URL: https://gizmodo.com/is-jailbreaking-and-modding-still-worth-it-in-2019-1836857168 (visited on 02/14/2020).

[34] *Online cheaters face games ban*. Aug. 2002. URL: http://news.bbc.co.uk/2/hi/technology/2221335.stm (visited on 02/14/2020).

[35] *Overwatch FAQ*. URL: https://blog.counter-strike.net/index.php/overwatch/ (visited on 02/14/2020).

[36] Panagiotis Papadopoulos et al. "No Sugar but All the Taste! Memory Encryption Without Architectural Support". In: *Computer Security – ESORICS 2017*. Ed. by Simon N. Foley, Dieter Gollmann, and Einar Snekkenes. Cham: Springer International Publishing, 2017, pp. 362–380. ISBN: 978-3-319-66399-9.

[37] Darren Pauli. *5 ROOTKIT OF VENGEANCE defeats forces of gaming good*. Apr. 2015. URL: https://www.theregister.co.uk/2015/04/10/rootkit_tweaks_strike_anticheat_systems_dead/ (visited on 02/14/2020).

[38] *Punkbuster*. URL: http://www.evenbalance.com/punkbuster.php (visited on 02/14/2020).

[39] *RFC: 793 - TRANSMISSION CONTROL PROTOCOL*. URL: https://tools.ietf.org/html/a (visited on 02/14/2020).

[40] *Secure Connections*. URL: http://www.raknet.net/raknet/manual/secureconnections.html (visited on 02/14/2020).

[41] Eric Smith. *Valve on Anti-Cheating - Blue's News Comments*. Oct. 2001. URL: https://www.bluesnews.com/cgi-bin/board.pl?action=viewthread&boardid=1&threadid=30074&id=14017 (visited on 02/14/2020).

[42] Tom Spring. *Driver Disaster: Over 40 Signed Drivers Can't Pass Security Muster*. URL: https://threatpost.com/driver-disaster-over-40-signed-drivers-cant-pass-security-muster/147199/ (visited on 02/14/2020).

[43] *Stadia: A new generation of game development from Google*. URL: https://stadia.dev/about/ (visited on 02/14/2020).

[44] *std::hash*. URL: https://en.cppreference.com/w/cpp/utility/hash (visited on 02/14/2020).

[45] Vladislav Stolyarov and Boris Larin. *A vulnerable driver: lesson almost learned*. URL: https://securelist.com/elevation-of-privileges-in-namco-driver/83707/ (visited on 02/14/2020).

[46] Mirko Suznjevic et al. "Analyzing the Effect of TCP and Server Population on Massively Multiplayer Games". In: *International Journal of Computer Games Technology* 2014 (Jan. 2014), pp. 1–17. DOI: 10.1155/2014/602403.

[47] *System Integrity Protection Guide*. Sept. 2015. URL: https://developer.apple.com/library/archive/documentation/Security/Conceptual/System_Integrity_Protection_Guide/KernelExtensions/KernelExtensions.html (visited on 02/14/2020).

[48] Paul Tassi. *Chinese Prisoners Forced to Farm World of Warcraft Gold*. Jan. 2013. URL: https://www.forbes.com/sites/insertcoin/2011/06/02/chinese-prisoners-forced-to-farm-world-of-warcraft-gold/ (visited on 02/14/2020).

[49] Steven Templeton and Karl Levitt. "Detecting Spoofed Packets". In: (Feb. 2003).

[50] Ramakrishna Thurimella and William Mitchell. "Cloak and Dagger: Man-In-The-Middle and Other Insidious Attacks". In: *International Journal of Information Security and Privacy (IJISP)* 3.3 (July 2009), pp. 55–75. URL: https://ideas.repec.org/a/igg/jisp00/v3y2009i3p55-75.html.

[51] Ty. *Linux Users Banned From World of Warcraft?* Nov. 2006. URL: https://web.archive.org/web/20080216041937/http://www.linuxlookup.com/2006/nov/15/linux_users_banned_from_world_of_warcraft (visited on 02/14/2020).

[52] *Ultima Online*. URL: https://mmo-population.com/r/ultimaonline/ (visited on 02/14/2020).

[53] *UPX*. URL: https://upx.github.io/ (visited on 02/14/2020).

[54] *User Datagram Protocol*. URL: https://tools.ietf.org/html/rfc768 (visited on 02/22/2020).

[55] *Valve challenged over anti-cheating tools*. Feb. 2014. URL: https://www.bbc.com/news/technology-26240140 (visited on 02/14/2020).

[56] Michael VanKuiper. *Riot's Approach to Anti-Cheat*. July 2018. URL: https://technology.riotgames.com/news/riots-approach-anti-cheat (visited on 02/14/2020).

[57] Mark Ward. *Warcraft game maker in spying row*. Oct. 2005. URL: http://news.bbc.co.uk/2/hi/technology/4385050.stm (visited on 02/14/2020).

[58] *Widespread Cheating in Multiplayer Online Games Drives Gamers in Asia Pacific Away*. June 2018. URL: https://irdeto.com/news/widespread-cheating-in-multiplayer-online-games-drives-gamers-in-asia-pacific-away/ (visited on 02/14/2020).

[59] Windows-Sdk-Content. *CreateRemoteThread function (processthreadsapi.h) - Win32 apps*. URL: https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createremotethread (visited on 02/14/2020).

[60] Windows-Sdk-Content. *SetWindowsHookExA function (winuser.h) - Win32 apps*. URL: https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setwindowshookexa (visited on 02/14/2020).

[61] Windows-Sdk-Content. *WriteProcessMemory function (memoryapi.h) - Win32 apps*. URL: https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-writeprocessmemory (visited on 02/14/2020).

[62] *Wireshark*. URL: https://www.wireshark.org/ (visited on 02/14/2020).

Appendices

A Guessing game server

```
1 //
2 //  GameManager.hpp
3 //  Thesis-game-server
4 //
5 //  Created by Samuli Lehtonen on 21/04/2019.
6 //  Copyright © 2019 Samuli Lehtonen. All rights reserved.
7 //
8
9 #ifndef GameManager_hpp
```

```
10 #define GameManager_hpp
11
12 #include <stdio.h>
13 #include <SFML/Network.hpp>
14 #include <string>
15 #include <random>
16 #include "MatchMaker.hpp"
17 #include "PacketHandler.hpp"
18 #include "OpCodes.hpp"
19 #include "Player.hpp"
20
21 enum AuthenticationStatus {
22     WAITING_FOR_AUTHENTICATION = false,
23     AUTHENTICATED = true,
24 };
25
26 class GameManager {
27 public:
28     void SendAuthenticationHash(sf::Packet packet, sf::IpAddress sender,
    int port);
29     void PlayerAuthenticated(sf::Packet packet, sf::IpAddress sender, int
    port);
30     void PlayerEnterMatchMaking(sf::Packet packet, sf::IpAddress sender,
    int port);
31     void PlayerGuessNumber(sf::Packet packet, sf::IpAddress sender, int
    port);
32     void ConfirmGameIntegrity(sf::Packet packet, sf::IpAddress sender, int
    port);
33
34     void MatchFound(std::shared_ptr<Match> match);
35     void HandleGameMessage();
36     GameManager(PacketHandler * packetHandler);
37
38 private:
39     std::string GenerateAuthenticationHash();
40     bool VerifyAuthenticationHash(std::string hash);
41     void SetAuthenticationStatusForHash(std::string hash,
    AuthenticationStatus status);
42     void RegisterOpCodes();
43     void AddPlayerToMatchMaking();
44     void RemovePlayerFromMatchMaking();
45     void SendMatchFoundPacket(std::shared_ptr<Match> match);
46     void SendGameIntegrityConfirmedPacket(sf::IpAddress address, int port);
47     void SendNewRoundPacket(std::shared_ptr<Match> match);
48     bool ValidateAuthenticationHashFromPacket(sf::Packet * packet);
49     sf::Packet CreatePacketWithOpCode(OpCodes opCode);
50     std::string GetAuthenticationHash(sf::Packet packet);
```

```
51
52      sf::Packet CreateAuthenticationPacket(std::string authenticationHash);
53
54      std::string mGameIntegrityHashOSX = "15546534240171485050";
55      std::string mGameIntegrityHashWindows = "53451231231231241";
56
57      std::unordered_map<std::string, int> mPlayerPorts;
58      std::unordered_map<std::string, AuthenticationStatus>
        mAuthenticationMap;
59      std::unordered_map<std::string, std::shared_ptr<Player>> mPlayers;
60      std::unordered_map<int, std::shared_ptr<Match>> mOngoingMatches;
61
62      std::shared_ptr<Match> GetMatchWithId(int id);
63
64      PacketHandler * mPacketHandler;
65      MatchMaker mMatchMaker;
66  };
67
68  #endif /* GameManager_hpp */
```

```
1   //
2   //  GameManager.cpp
3   //  Thesis-game-server
4   //
5   //  Created by Samuli Lehtonen on 21/04/2019.
6   //  Copyright © 2019 Samuli Lehtonen. All rights reserved.
7   //
8
9   #include "GameManager.hpp"
10
11  GameManager::GameManager(PacketHandler * packetHandler):mPacketHandler(
        packetHandler), mMatchMaker(3, std::bind(&GameManager::MatchFound, this,
         std::placeholders::_1)) {
12      this->RegisterOpCodes();
13  }
14
15  sf::Packet GameManager::CreatePacketWithOpCode(OpCodes opCode) {
16      sf::Packet packet;
17      packet << opCode;
18      return packet;
19  }
20
21  // Sends packet to all players in match, indicating that a match has began
22  void GameManager::SendMatchFoundPacket(std::shared_ptr<Match> match) {
23      sf::Packet packet = this->CreatePacketWithOpCode(OpCodes::
        SERVER_MATCH_STARTED);
24      packet << match->GetMatchId();
```

```
25      packet << match->GetNumberToGuess();
26      packet << match->GetCurrentTurnPlayer()->GetName();
27
28      for (const auto & player : match->GetPlayers()) {
29          packet << player->GetName();
30      }
31
32      for (const auto & player : match->GetPlayers()) {
33          sf::Packet packetToSend = packet;
34          if (match->GetCurrentTurnPlayer() == player) {
35              packetToSend << true;
36          } else {
37              packetToSend << false;
38          }
39          this->mPacketHandler->SendPacket(packetToSend, player->GetAddress()
        , player->GetPort(), player->GetAuthenticationHash());
40      }
41  }
42
43  std::shared_ptr<Match> GameManager::GetMatchWithId(int id) {
44      try {
45          std::shared_ptr<Match> match = this->mOngoingMatches[id];
46          return match;
47      } catch (std::exception & exception) {
48          return nullptr;
49      }
50  }
51
52  void GameManager::MatchFound(std::shared_ptr<Match> match) {
53      // Add match
54      this->mOngoingMatches[match->GetMatchId()] = match;
55      this->SendMatchFoundPacket(match);
56  }
57
58  bool GameManager::VerifyAuthenticationHash(std::string hash) {
59
60  }
61
62  void GameManager::HandleGameMessage() {
63
64  }
65
66  void GameManager::SendNewRoundPacket(std::shared_ptr<Match> match) {
67      match->SetNextTurnPlayer();
68      match->SetNewNumberToGuess();
69      sf::Packet newRoundPacket = this->CreatePacketWithOpCode(OpCodes::
    SERVER_NEW_ROUND);
```

```
70      newRoundPacket << match->GetNumberToGuess();
71      for (const auto & player : match->GetPlayers()) {
72          sf::Packet packetToSend = newRoundPacket;
73          if (player == match->GetCurrentTurnPlayer()) {
74              packetToSend << true;
75          } else {
76              packetToSend << false;
77          }
78          this->mPacketHandler->SendPacket(packetToSend, player->GetAddress()
    , player->GetPort(), player->GetAuthenticationHash());
79      }
80  }
81
82  void GameManager::PlayerGuessNumber(sf::Packet packet, sf::IpAddress sender
    , int port) {
83      if (this->ValidateAuthenticationHashFromPacket(&packet)) {
84          int matchId;
85          int number;
86          packet >> matchId;
87          packet >> number;
88          std::shared_ptr<Match> match = this->GetMatchWithId(matchId);
89          if (match != nullptr) {
90              std::shared_ptr<Player> guesser = match->
    GetPlayerWithAddressAndPort(sender, port);
91              if (guesser == match->GetCurrentTurnPlayer()) {
92              {
93                  if (number == match->GetNumberToGuess()) {
94                      // Start new round
95                      this->SendNewRoundPacket(match);
96
97                  } else {
98                      sf::Packet wrongPacket = this->CreatePacketWithOpCode(
    OpCodes::SERVER_WRONG_NUMBER);
99                      wrongPacket << number;
100                     if (number < match->GetNumberToGuess()) {
101                         wrongPacket << -1;
102                     } else if (number > match->GetNumberToGuess()) {
103                         wrongPacket << 1;
104                     } else {
105                         wrongPacket << 0;
106                     }
107                     for (const auto & player : match->GetPlayers()) {
108                         this->mPacketHandler->SendPacket(wrongPacket,
    player->GetAddress(), player->GetPort(), player->GetAuthenticationHash()
    );
109                     }
110                     // Change turn, also inform guesser about the wrong
```

```
    result
111                     match->SetNextTurnPlayer();
112                     sf::Packet packet = this->CreatePacketWithOpCode(
    OpCodes::SERVER_SET_TURN);
113                     packet << match->GetCurrentTurnPlayer()->GetName();
114                     for (const auto & player : match->GetPlayers()) {
115                         sf::Packet packetToSend = packet;
116                         if (player == match->GetCurrentTurnPlayer()) {
117                             packetToSend << true;
118                         } else {
119                             packetToSend << false;
120                         }
121                         this->mPacketHandler->SendPacket(packetToSend,
    player->GetAddress(), player->GetPort(), player->GetAuthenticationHash()
    );
122                     }
123                 }
124             }
125         }
126         }
127     }
128 }
129
130 void GameManager::SendGameIntegrityConfirmedPacket(sf::IpAddress address,
    int port) {
131     sf::Packet packet = this->CreatePacketWithOpCode(OpCodes::
    SERVER_CONFIRM_GAME_INTEGRITY);
132     this->mPacketHandler->SendPacket(packet, address, port, false);
133 }
134
135 void GameManager::ConfirmGameIntegrity(sf::Packet packet, sf::IpAddress
    sender, int port) {
136     std::string hash;
137     packet >> hash;
138     if (hash == this->mGameIntegrityHashOSX || hash == this->
    mGameIntegrityHashWindows) {
139         this->SendGameIntegrityConfirmedPacket(sender, port);
140     } else {
141         std::cout << "Verification failed";
142     }
143 }
144
145 void GameManager::RegisterOpCodes() {
146     // Register all opcodes to handlers here
147     using namespace std::placeholders;
148
149     this->mPacketHandler->SubscribeToOpCode(OpCodes::
```

```
    USER_REQUEST_AUTHENTICATION_HASH, std::bind(&GameManager::
    SendAuthenticationHash, this, _1, _2, _3));
150     this->mPacketHandler->SubscribeToOpCode(OpCodes::
    USER_MATCHMAKING_ENTER_QUEUE, std::bind(&GameManager::
    PlayerEnterMatchMaking, this, _1, _2, _3), true);
151     this->mPacketHandler->SubscribeToOpCode(OpCodes::USER_AUTHENTICATED,
    std::bind(&GameManager::PlayerAuthenticated, this, _1, _2, _3));
152     this->mPacketHandler->SubscribeToOpCode(OpCodes::USER_GUESS_NUMBER, std
    ::bind(&GameManager::PlayerGuessNumber, this, _1, _2, _3), true);
153     this->mPacketHandler->SubscribeToOpCode(OpCodes::USER_SEND_FILE_HASH,
    std::bind(&GameManager::ConfirmGameIntegrity, this, _1, _2, _3));
154 }
155
156 bool GameManager::ValidateAuthenticationHashFromPacket(sf::Packet * packet)
    {
157     std::string hash;
158     *packet >> hash;
159     try {
160         if (this->mAuthenticationMap.at(hash) == AUTHENTICATED) {
161             return true;
162         }
163     } catch (std::exception & exception) {
164         std::cout << "Authentication error";
165         return false;
166     }
167     return false;
168 }
169
170 std::string GameManager::GetAuthenticationHash(sf::Packet packet) {
171     std::string hash;
172     packet >> hash;
173     return hash;
174 }
175
176 void GameManager::PlayerEnterMatchMaking(sf::Packet packet, sf::IpAddress
    sender, int port) {
177     std::string hash = this->GetAuthenticationHash(packet);
178
179     if (this->ValidateAuthenticationHashFromPacket(&packet)) {
180         // Create player object
181         std::string name;
182         packet >> name;
183         std::shared_ptr<Player> player (new Player(sender, name, hash, this
    ->mPlayerPorts[hash]));
184         this->mPlayers[hash] = player;
185         this->mMatchMaker.AddPlayer(player);
186     }
```

```
187 }
188
189 void GameManager::AddPlayerToMatchMaking() {
190
191 }
192
193 void GameManager::SetAuthenticationStatusForHash(std::string hash,
    AuthenticationStatus status) {
194     if (status == AUTHENTICATED) {
195         try {
196             if (this->mAuthenticationMap.at(hash) ==
    WAITING_FOR_AUTHENTICATION) {
197                 this->mAuthenticationMap[hash] = AUTHENTICATED;
198             }
199         } catch (std::exception & exception) {
200             // Error with authentication
201         }
202     } else {
203         this->mAuthenticationMap[hash] = WAITING_FOR_AUTHENTICATION;
204     }
205 }
206
207 void GameManager::PlayerAuthenticated(sf::Packet packet, sf::IpAddress
    sender, int port) {
208     std::string hash;
209     packet >> hash;
210     this->SetAuthenticationStatusForHash(hash, AUTHENTICATED);
211     this->mPacketHandler->SetEncryptionKeyForAddress(sender, port, hash);
212 }
213
214 sf::Packet GameManager::CreateAuthenticationPacket(std::string
    authenticationHash) {
215     sf::Packet packet;
216     packet << OpCodes::SERVER_GIVE_AUTHENTICATION_HASH;
217     packet << authenticationHash;
218     return packet;
219 }
220
221 std::string GameManager::GenerateAuthenticationHash() {
222     std::mt19937_64 gen{std::random_device{}()};
223     std::uniform_int_distribution<short> dist{'a', 'z'};
224     int length = 30;
225     std::string str(length, '\0');
226     for (auto& c: str) {
227         c = dist(gen);
228     }
229     return str;
```

```
230 }
231
232 void GameManager::SendAuthenticationHash(sf::Packet packet, sf::IpAddress
        sender, int port) {
233     // Generate secret hash here and send to client
234     std::string hash = this->GenerateAuthenticationHash();
235     this->mPlayerPorts[hash] =  port;
236     sf::Packet authenticationPacket = this->CreateAuthenticationPacket(hash
        );
237
238     this->SetAuthenticationStatusForHash(hash, WAITING_FOR_AUTHENTICATION);
239     this->mPacketHandler->SendPacket(authenticationPacket, "127.0.0.1", (
        unsigned short)port);
240 }
```

```
1  //
2  //  GameServer.hpp
3  //  Thesis-game-server
4  //
5  //  Created by Samuli Lehtonen on 21/04/2019.
6  //  Copyright © 2019 Samuli Lehtonen. All rights reserved.
7  //
8
9  #ifndef GameServer_hpp
10 #define GameServer_hpp
11
12 #include <stdio.h>
13 #include <iostream>
14 #include <SFML/Network.hpp>
15 #include <thread>
16 #include <chrono>
17 #include "GameManager.hpp"
18 #include "PacketHandler.hpp"
19
20
21 class GameServer {
22 public:
23     GameServer();
24     void StartServer();
25
26 private:
27     std::thread mSocketThread;
28     const int mReceivePort = 5500;
29     const int mSendPort = 5600;
30     bool mIsServerRunning;
31     sf::UdpSocket mUdpSocket;
32     PacketHandler mPacketHandler;
```

```
33      GameManager mGameManager;
34
35      void ProcessSockets();
36      void ShowServerControlMenu();
37  };
38
39  #endif /* GameServer_hpp */
```

```
1   //
2   //  GameServer.cpp
3   //  Thesis-game-server
4   //
5   //  Created by Samuli Lehtonen on 21/04/2019.
6   //  Copyright © 2019 Samuli Lehtonen. All rights reserved.
7   //
8
9   #include "GameServer.hpp"
10
11  GameServer::GameServer():mPacketHandler(this->mSendPort, &this->mUdpSocket)
        ,mGameManager(&this->mPacketHandler) {
12
13  }
14
15  void GameServer::StartServer() {
16      this->mUdpSocket.setBlocking(false);
17      if (this->mUdpSocket.bind(this->mReceivePort) != sf::Socket::Done) {
18          throw std::runtime_error(std::string("Failed to bind socket to port
        ") + std::to_string(this->mReceivePort) );
19      }
20
21      // Run the processing of sockets in another thread
22      this->mIsServerRunning = true;
23      this->mSocketThread = std::thread(&GameServer::ProcessSockets, this);
24
25      // Run control menu in main thread
26      this->ShowServerControlMenu();
27  }
28
29  void GameServer::ProcessSockets() {
30      while (this->mIsServerRunning) {
31          std::this_thread::sleep_for(std::chrono::milliseconds(1));
32          sf::Packet packet;
33          sf::IpAddress sender;
34          unsigned short port;
35          sf::Socket::Status socketStatus = this->mUdpSocket.receive(packet,
        sender, port);
36
```

```
        if (socketStatus == sf::Socket::Done) {
            this->mPacketHandler.HandlePacket(packet, sender, port);
        } else if (socketStatus == sf::Socket::NotReady) {
            // No data
        } else {
            // Error
        }

    }
}

void GameServer::ShowServerControlMenu(){
    std::cout << "Server control panel" << std::endl;
    std::cout << "Server is running on port " << this->mReceivePort << std
    ::endl;
    std::cout << std::endl;

    std::string input;
    while (this->mIsServerRunning) {
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }

    std::cout << "Server shutting down" << std::endl;
    this->mSocketThread.join();
}
```

```
#include <SFML/Audio.hpp>
#include <SFML/Graphics.hpp>
#include "GameServer.hpp"

int main(int argc, char const** argv)
{
    GameServer gameServer;
    gameServer.StartServer();
    return 1;
}
```

```
//
//  Match.hpp
//  Thesis-game-server
//
//  Created by Samuli Lehtonen on 22/04/2019.
//  Copyright © 2019 Samuli Lehtonen. All rights reserved.
//

#ifndef Match_hpp
#define Match_hpp
```

```
11
12 #include <stdio.h>
13 #include <iostream>
14 #include <vector>
15 #include <random>
16 #include "Player.hpp"
17
18 class Match {
19 private:
20     int mMatchId;
21     int mCurrentTurnPlayer;
22     int mNumberToGuess;
23     std::vector<std::shared_ptr<Player>> mPlayers;
24 public:
25     Match(int matchId, std::vector<std::shared_ptr<Player>> players);
26     Match(const Match &m2);
27     int GetMatchId();
28     std::vector<std::shared_ptr<Player>> GetPlayers();
29     std::shared_ptr<Player> GetCurrentTurnPlayer();
30     std::shared_ptr<Player> GetPlayerWithAddressAndPort(sf::IpAddress
       address, int port);
31     void SetNextTurnPlayer();
32     void SetNewNumberToGuess();
33     int GetNumberToGuess();
34 };
35
36 #endif /* Match_hpp */
```

```
1 //
2 //  Match.cpp
3 //  Thesis-game-server
4 //
5 //  Created by Samuli Lehtonen on 22/04/2019.
6 //  Copyright © 2019 Samuli Lehtonen. All rights reserved.
7 //
8
9 #include "Match.hpp"
10
11 Match::Match(int matchId, std::vector<std::shared_ptr<Player>> players):
      mMatchId(matchId), mPlayers(players), mCurrentTurnPlayer(0) {
12     this->SetNewNumberToGuess();
13 }
14
15 Match::Match(const Match &m2) {
16     this->mCurrentTurnPlayer = m2.mCurrentTurnPlayer;
17     this->mMatchId = m2.mMatchId;
18     this->mPlayers = m2.mPlayers;
```

```
}

std::shared_ptr<Player> Match::GetPlayerWithAddressAndPort(sf::IpAddress address, int port) {
    for (const auto & player : this->mPlayers) {
        if (player->GetAddress() == address && player->GetPort() == (unsigned short)port) {
            return player;
        }
    }
    return nullptr;
}

void Match::SetNewNumberToGuess() {
    std::random_device rd;
    std::mt19937_64 gen(rd());
    std::uniform_int_distribution<> dist(1, 100);
    this->mNumberToGuess = dist(gen);
    std::cout << "Number to guess is: " << this->mNumberToGuess << std::endl;
}

int Match::GetNumberToGuess() {
    return this->mNumberToGuess;
}

std::shared_ptr<Player> Match::GetCurrentTurnPlayer() {
    return this->mPlayers[this->mCurrentTurnPlayer];
}

void Match::SetNextTurnPlayer() {
    if (this->mCurrentTurnPlayer == this->mPlayers.size() - 1) {
        this->mCurrentTurnPlayer = 0;
    } else {
        this->mCurrentTurnPlayer++;
    }
}

int Match::GetMatchId() {
    return this->mMatchId;
}

std::vector<std::shared_ptr<Player>> Match::GetPlayers() {
    return this->mPlayers;
}
```

```
//
```

```
//  MatchMaker.hpp
//  Thesis-game-server
//
//  Created by Samuli Lehtonen on 21/04/2019.
//  Copyright © 2019 Samuli Lehtonen. All rights reserved.
//

#ifndef MatchMaker_hpp
#define MatchMaker_hpp

#include <stdio.h>
#include <queue>
#include "Match.hpp"
#include "Player.hpp"

class MatchMaker {
private:
    Match CreateMatchFromPlayer();
    std::function<void(std::shared_ptr<Match>)> mMatchFoundDelegate;
    std::queue<std::shared_ptr<Player>> mQueuedPlayers;
    int mMatchSize;
    void TryToMatchMake();
    int mMatchId;
public:
    MatchMaker(int matchSize, std::function<void(std::shared_ptr<Match>)>
    matchFoundDelegate);
    void AddPlayer(std::shared_ptr<Player> player);
    void RemovePlayer(Player *player);
};

#endif /* MatchMaker_hpp */
```

```
//
//  MatchMaker.cpp
//  Thesis-game-server
//
//  Created by Samuli Lehtonen on 21/04/2019.
//  Copyright © 2019 Samuli Lehtonen. All rights reserved.
//

#include "MatchMaker.hpp"

MatchMaker::MatchMaker(int matchSize, std::function<void(std::shared_ptr<
    Match>)> matchFoundDelegate): mMatchSize(matchSize), mMatchFoundDelegate
    (matchFoundDelegate), mMatchId(0) {

}
```

```
14
15 void MatchMaker::AddPlayer(std::shared_ptr<Player> player) {
16     this->mQueuedPlayers.push(player);
17     this->TryToMatchMake();
18 }
19
20 void MatchMaker::TryToMatchMake() {
21     if (this->mQueuedPlayers.size() >= this->mMatchSize) {
22         // Match found
23         std::vector<std::shared_ptr<Player>> players;
24         for (int i = 0; i < this->mMatchSize; i++) {
25             players.push_back(this->mQueuedPlayers.front());
26             this->mQueuedPlayers.pop();
27         }
28         // Call delegate with match
29         std::shared_ptr<Match> match (new Match(this->mMatchId, players));
30         this->mMatchId++;
31         this->mMatchFoundDelegate(match);
32     }
33 }
```

```
1
2 //
3 //  OpCodes.hpp
4 //  Thesis-game-server
5 //
6 //  Created by Samuli Lehtonen on 21/04/2019.
7 //  Copyright © 2019 Samuli Lehtonen. All rights reserved.
8 //
9
10 #ifndef OpCodes_hpp
11 #define OpCodes_hpp
12
13 enum OpCodes {
14     USER_MATCHMAKING_ENTER_QUEUE,
15     USER_MATCHMAKING_LEAVE_QUEUE,
16     USER_PING,
17     USER_REQUEST_AUTHENTICATION_HASH,
18     USER_AUTHENTICATED,
19     USER_GUESS_NUMBER,
20     USER_SEND_FILE_HASH,
21     SERVER_GIVE_AUTHENTICATION_HASH,
22     SERVER_MATCH_STARTED,
23     SERVER_SET_TURN, // Turn changes
24     SERVER_WRONG_NUMBER, // Tell if number is lower or higher
25     SERVER_CORRECT_NUMBER, // End game
26     SERVER_CONFIRM_GAME_INTEGRITY,
```

```
    SERVER_NEW_ROUND
};

#endif /* OpCodes_hpp */
```

```
//
//  PacketHandler.hpp
//  Thesis-game-server
//
//  Created by Samuli Lehtonen on 21/04/2019.
//  Copyright © 2019 Samuli Lehtonen. All rights reserved.
//

#ifndef PacketHandler_hpp
#define PacketHandler_hpp

#include <stdio.h>
#include <SFML/Network.hpp>
#include <iostream>
#include <unordered_map>

class PacketHandler {
public:
    PacketHandler(int sendPort, sf::UdpSocket * socket);
    sf::Socket::Status SendPacket(sf::Packet packet, sf::IpAddress receiverIp, bool encrypt = false);
    sf::Socket::Status SendPacket(sf::Packet packet, sf::IpAddress receiverIp, unsigned short receiverPort, bool encrypt = false);
    sf::Socket::Status SendPacket(sf::Packet packet, sf::IpAddress receiverIp, unsigned short receiverPort, std::string key);
    void HandlePacket(sf::Packet packet, sf::IpAddress senderIp, int port);
    void SubscribeToOpCode(int opCode, std::function<void(sf::Packet, sf::IpAddress, int)> function);
    void SubscribeToOpCode(int opCode, std::function<void(sf::Packet, sf::IpAddress, int)>, bool decrypt);
    void SetReceivePort(int receivePort);
    void SetDecryptionKey(std::string key);
    void SetEncryptionKeyForAddress(sf::IpAddress address, unsigned short port, std::string key);
    void SetSendAddress(sf::IpAddress address);
    sf::IpAddress GetSendAddress();
    int GetReceivePort();

private:
    bool mUseEncryption;
    int mSendPort;
    int mReceivePort;
```

```
37      std::string mCryptoKey;
38      sf::IpAddress mSendAddress;
39      sf::UdpSocket * mUdpSocket;
40      std::unordered_map<int, std::function<void(sf::Packet, sf::IpAddress,
        int)>> mPacketHandlers;
41      std::unordered_map<int, bool> mNeedsDecryption;
42      std::unordered_map<std::string, std::string> mEncryptionKeys;
43      std::string CreateAddresString(sf::IpAddress address, unsigned short
        port);
44      sf::Packet EncryptData(sf::Packet packet);
45      sf::Packet EncryptData(sf::Packet packet, std::string key);
46      sf::Packet DecryptData(sf::Packet packet, std::string key);
47      sf::Packet DecryptData(sf::Packet packet);
48  };
49
50  #endif /* PacketHandler_hpp */
```

```
1   //
2   //  PacketHandler.cpp
3   //  Thesis-game-server
4   //
5   //  Created by Samuli Lehtonen on 21/04/2019.
6   //  Copyright © 2019 Samuli Lehtonen. All rights reserved.
7   //
8
9   #include "PacketHandler.hpp"
10
11  PacketHandler::PacketHandler(int sendPort, sf::UdpSocket * socket):
        mSendPort(sendPort), mUdpSocket(socket), mUseEncryption(true) {
12      //this->mUseEncryption = false;
13  }
14
15  sf::Packet PacketHandler::EncryptData(sf::Packet packet) {
16      return this->EncryptData(packet, this->mCryptoKey);
17  }
18
19  sf::Packet PacketHandler::EncryptData(sf::Packet packet, std::string key) {
20      // If encryption is not in use, just return the packet as normal
21      if (!this->mUseEncryption) {
22          return packet;
23      }
24
25      // Extract opcode from the packet
26      sf::Packet encryptedPacket;
27
28      int opCode;
29      packet >> opCode;
```

```
30      encryptedPacket << opCode;
31
32      // Get data of the packet into buffer
33      const char *charBuffer = (char*)packet.getData()+sizeof(char)*4;
34      int n = packet.getDataSize()-sizeof(char)*4;
35
36      // Convert the char* data buffer to std::vector data buffer
37      std::vector<char> data(charBuffer, charBuffer + n);
38      std::size_t dataSize = n;
39      std::vector<char> encryptedData;
40
41      // Encrypt by XOR:ing with encryption key
42      for (std::size_t i = 0; i < dataSize; i++) {
43          encryptedData.push_back(data[i] ^ key[i % key.length()]);
44      }
45
46      // Create the encrypted packet and append the data
47
48      encryptedPacket.append(encryptedData.data(), dataSize);
49      return encryptedPacket;
50  }
51
52  sf::Packet PacketHandler::DecryptData(sf::Packet packet, std::string key) {
53      sf::Packet decryptedPacket = this->EncryptData(packet, key);
54      int opCode;
55      decryptedPacket >> opCode;
56      return decryptedPacket;
57  }
58
59  sf::Packet PacketHandler::DecryptData(sf::Packet packet) {
60      return this->DecryptData(packet, this->mCryptoKey);
61  }
62
63  void PacketHandler::SetSendAddress(sf::IpAddress address) {
64      this->mSendAddress = address;
65  }
66
67  sf::IpAddress PacketHandler::GetSendAddress() {
68      return this->mSendAddress;
69  }
70
71  void PacketHandler::SetEncryptionKeyForAddress(sf::IpAddress address,
        unsigned short port, std::string key) {
72      std::string stringAddress = this->CreateAddresString(address, port);
73      this->mEncryptionKeys[stringAddress] = key;
74  }
75
```

```
76 std::string PacketHandler::CreateAddresString(sf::IpAddress address,
       unsigned short port) {
77     return address.toString() + ":" + std::to_string(port);
78 }
79
80 sf::Socket::Status PacketHandler::SendPacket(sf::Packet packet, sf::
       IpAddress receiverIp, unsigned short receiverPort, std::string key) {
81     packet = this->EncryptData(packet, key);
82     return this->mUdpSocket->send(packet, receiverIp, receiverPort);
83 }
84
85 sf::Socket::Status PacketHandler::SendPacket(sf::Packet packet, sf::
       IpAddress receiverIp, bool encrypt) {
86     return this->SendPacket(packet, receiverIp, this->mSendPort, encrypt);
87 }
88
89 sf::Socket::Status PacketHandler::SendPacket(sf::Packet packet, sf::
       IpAddress receiverIp, unsigned short receiverPort, bool encrypt) {
90     if (encrypt) {
91         packet = this->EncryptData(packet);
92     }
93     sf::Socket::Status res = this->mUdpSocket->send(packet, receiverIp,
       receiverPort);
94     return res;
95 }
96
97 void PacketHandler::SetReceivePort(int receivePort) {
98     this->mReceivePort = receivePort;
99 }
100
101 int PacketHandler::GetReceivePort() {
102     return this->mReceivePort;
103 }
104
105 void PacketHandler::HandlePacket(sf::Packet packet, sf::IpAddress senderIp,
        int port) {
106     sf::Packet copyPacket = packet;
107     int opCode = -1;
108     packet >> opCode;
109
110     if (this->mNeedsDecryption[opCode]) {
111         std::string stringAddress = this->CreateAddresString(senderIp, port
       );
112         if (this->mEncryptionKeys.find(stringAddress) != this->
       mEncryptionKeys.end()) {
113             packet = this->DecryptData(copyPacket, this->mEncryptionKeys[
       stringAddress]);
```

```
114         } else {
115             packet = this->DecryptData(copyPacket);
116         }
117     }
118 
119     if (packet) {
120         // Data extraction succesfull
121         if (this->mPacketHandlers.find(opCode) != this->mPacketHandlers.end
    ()) {
122             // Found key
123             this->mPacketHandlers[opCode](packet, senderIp, port);
124         } else {
125             // Discard the packet with invalid opcode
126             std::cout << "Invalid opcode " << opCode << " received";
127         }
128     }
129 }
130 
131 void PacketHandler::SubscribeToOpCode(int opCode, std::function<void(sf::
    Packet, sf::IpAddress, int)> function) {
132     this->mNeedsDecryption[opCode] = false;
133     this->mPacketHandlers[opCode] = function;
134 }
135 
136 void PacketHandler::SubscribeToOpCode(int opCode, std::function<void (sf::
    Packet, sf::IpAddress, int)> function, bool decrypt) {
137     this->mNeedsDecryption[opCode] = decrypt;
138     this->mPacketHandlers[opCode] = function;
139 }
140 
141 void PacketHandler::SetDecryptionKey(std::string key) {
142     this->mCryptoKey = key;
143 }
```

```
1 //
2 //  Player.hpp
3 //  Thesis-game-server
4 //
5 //  Created by Samuli Lehtonen on 22/04/2019.
6 //  Copyright © 2019 Samuli Lehtonen. All rights reserved.
7 //
8 
9 #ifndef Player_hpp
10 #define Player_hpp
11 
12 #include <stdio.h>
13 #include <SFML/Network.hpp>
```

```
14
15 class Player {
16 public:
17     Player(sf::IpAddress ipAddress, std::string name, std::string
       authenticationHash, int port);
18     std::string GetName();
19     sf::IpAddress GetAddress();
20     unsigned short GetPort();
21     std::string GetAuthenticationHash();
22 private:
23     std::string mAuthenticationHash;
24     sf::IpAddress mIpAddress;
25     std::string mName;
26     int mPort;
27 };
28
29 #endif /* Player_hpp */
```

```
1 //
2 //  Player.cpp
3 //  Thesis-game-server
4 //
5 //  Created by Samuli Lehtonen on 22/04/2019.
6 //  Copyright © 2019 Samuli Lehtonen. All rights reserved.
7 //
8
9 #include "Player.hpp"
10
11 Player::Player(sf::IpAddress ipAddress, std::string name, std::string
       authenticationHash, int port): mIpAddress(ipAddress), mName(name),
       mAuthenticationHash(authenticationHash), mPort(port) {
12
13 }
14
15 std::string Player::GetName() {
16     return this->mName;
17 }
18
19 sf::IpAddress Player::GetAddress() {
20     return this->mIpAddress;
21 }
22
23 unsigned short Player::GetPort() {
24     return (unsigned short)this->mPort;
25 }
26
27 std::string Player::GetAuthenticationHash() {
```

```
28      return this->mAuthenticationHash;
29  }
```

B Guessing game client

```
1   //
2   //  Game.hpp
3   //  Thesis-game
4   //
5   //  Created by Samuli Lehtonen on 22/04/2019.
6   //  Copyright © 2019 Samuli Lehtonen. All rights reserved.
7   //
8
9   #ifndef Game_hpp
10  #define Game_hpp
11
12  #include <stdio.h>
13  #include <SFML/Audio.hpp>
14  #include <SFML/Graphics.hpp>
15  #include <SFML/Network.hpp>
16  #include <random>
17  #include <thread>
18  #include <chrono>
19  #include "ResourcePath.hpp"
20  #include "../../Thesis-game-server/Thesis-game-server/OpCodes.hpp"
21  #include "PacketHandler.hpp"
22  #include "SceneHandler.hpp"
23  #include "InputNameScene.hpp"
24  #include "GameScene.hpp"
25
26  class Game {
27  public:
28      Game();
29      void Run();
30  private:
31      sf::RenderWindow mGameWindow;
32      PacketHandler mPacketHandler;
33      sf::UdpSocket mUdpSocket;
34      SceneHandler mSceneHandler;
35
36      int mReceivePort;
37      const int mSendPort = 5500;
38
39      void Render();
40      void ReceivePackets();
41      void RegisterOpCodes();
```

```
42      void RegisterScenes();
43      void ReceivedAuthenticationHash(sf::Packet packet, sf::IpAddress sender
        , int port);
44  };
45
46  #endif /* Game_hpp */
```

```
1   //
2   //  Game.cpp
3   //  Thesis-game
4   //
5   //  Created by Samuli Lehtonen on 22/04/2019.
6   //  Copyright © 2019 Samuli Lehtonen. All rights reserved.
7   //
8
9   #include "Game.hpp"
10
11  Game::Game():mGameWindow(sf::VideoMode(800, 600), "Number guessing game"),
        mPacketHandler(5500, &this->mUdpSocket), mSceneHandler(this->mGameWindow
        ) {
12      std::random_device rd;
13      std::mt19937_64 gen(rd());
14      std::uniform_int_distribution<> dist(6000, 7000);
15
16      this->mReceivePort = dist(gen);
17      this->mPacketHandler.SetReceivePort(this->mReceivePort);
18      this->mPacketHandler.SetSendAddress("127.0.0.1");
19      this->mUdpSocket.bind(this->mReceivePort);
20      this->mUdpSocket.setBlocking(false);
21      this->RegisterOpCodes();
22      this->RegisterScenes();
23      this->mSceneHandler.SetVariable("isRunning", "YES");
24  }
25
26  void Game::Render() {
27      this->mSceneHandler.DrawCurrentScene();
28  }
29
30  void Game::RegisterScenes() {
31      this->mSceneHandler.RegisterScene(1, new InputNameScene(&this->
        mSceneHandler, &this->mPacketHandler));
32      this->mSceneHandler.RegisterScene(2, new GameScene(&this->mSceneHandler
        , &this->mPacketHandler));
33
34      this->mSceneHandler.SetScene(1);
35  }
36
```

```
37 void Game::RegisterOpCodes() {
38     using namespace std::placeholders;
39     this->mPacketHandler.SubscribeToOpCode(OpCodes::
   SERVER_GIVE_AUTHENTICATION_HASH, std::bind(&Game::
   ReceivedAuthenticationHash, this, _1, _2, _3));
40 }
41
42 void Game::ReceivedAuthenticationHash(sf::Packet packet, sf::IpAddress
   sender, int port) {
43     std::string hash;
44     packet >> hash;
45     std::cout << hash;
46     this->mSceneHandler.SetVariable("HASH", hash);
47 }
48
49 void Game::ReceivePackets() {
50     sf::Packet packet;
51     sf::IpAddress sender;
52     unsigned short port;
53     sf::Socket::Status socketStatus = this->mUdpSocket.receive(packet,
   sender, port);
54
55     if (socketStatus == sf::Socket::Done) {
56         std::cout << "Received packet" << std::endl;
57         this->mPacketHandler.HandlePacket(packet, sender, port);
58     } else if (socketStatus == sf::Socket::NotReady) {
59         // No data
60     } else {
61         // Error
62         std::cout << "error with packet" << std::endl;
63     }
64 }
65
66 void Game::Run() {
67
68     // Set the Icon
69     sf::Image icon;
70     if (!icon.loadFromFile(resourcePath() + "icon.png")) {
71         return EXIT_FAILURE;
72     }
73     this->mGameWindow.setIcon(icon.getSize().x, icon.getSize().y, icon.
   getPixelsPtr());
74
75
76     while (this->mGameWindow.isOpen() && this->mSceneHandler.GetVariable("
   isRunning") == "YES")
77     {
```

```
78          std::this_thread::sleep_for(std::chrono::milliseconds(50));
79          this->ReceivePackets();
80          this->Render();
81      }
82  }
```

```
1   //
2   //  GameScene.hpp
3   //  Thesis-game
4   //
5   //  Created by Samuli Lehtonen on 27/04/2019.
6   //  Copyright © 2019 Samuli Lehtonen. All rights reserved.
7   //
8
9   #ifndef GameScene_hpp
10  #define GameScene_hpp
11
12  #include <stdio.h>
13  #include "Scene.hpp"
14
15  enum TurnType {
16      TURN_YOU_CHOOSE_NUMBER,
17      TURN_OTHER_PLAYER_CHOOSE_NUMBER,
18      TURN_OTHER_PLAYER_GUESS_NUMBER,
19      TURN_YOU_GUESS_NUMBER,
20      TURN_UNDEFINED
21  };
22
23  class GameScene : public Scene {
24  public:
25      virtual int Run (sf::RenderWindow & window);
26      GameScene(SceneHandler *sceneHandler, PacketHandler *packetHandler);
27
28      void ReceiveMatchStartPacket(sf::Packet packet, sf::IpAddress sender,
        int port);
29      void ReceiveSetTurnMessage(sf::Packet packet, sf::IpAddress sender, int
         port);
30      void ReceiveWrongGuessMessage(sf::Packet packet, sf::IpAddress sender,
        int port);
31      void ReceiveCorrectGuessMessage(sf::Packet packet, sf::IpAddress sender
        , int port);
32      void ReceiveNewRoundMessage(sf::Packet packet, sf::IpAddress sender,
        int port);
33
34  private:
35      sf::Font mFont;
36      sf::Text mPlayer1Text;
```

```
37      sf::Text mPlayer2Text;
38      sf::Text mPlayer3Text;
39      sf::Text mInfoText;
40      std::string mPlayer1Name;
41      std::string mPlayer2Name;
42      std::string mPlayer3Name;
43      std::string mGameStateInfo;
44
45      sf::Text mGuessNumberTitleText;
46
47      sf::RectangleShape mGuessNumberBox;
48      sf::Text mGuessNumberText;
49      sf::String mGuessNumber;
50
51      InputAction GetTextInput(sf::Event event);
52
53
54      int mMatchId;
55      int mNumberToGuess;
56      TurnType mTurnType;
57
58      void Draw(sf::RenderWindow &window);
59      void SetSelectNumberToGuess();
60      void SetGuessNumber();
61      void SetWaitForOtherToSelectNumber();
62      void SetTurnForPlayerWithName(std::string name);
63      void SendGuessToServer(int guess);
64      void SetWaitingForOtherPlayersGuess();
65      void SetYourTimeToGuess();
66  };
67
68  #endif /* GameScene_hpp */
```

```
1   //
2   //  GameScene.cpp
3   //  Thesis-game
4   //
5   //  Created by Samuli Lehtonen on 27/04/2019.
6   //  Copyright © 2019 Samuli Lehtonen. All rights reserved.
7   //
8
9   #include "GameScene.hpp"
10
11  GameScene::GameScene(SceneHandler *sceneHandler, PacketHandler *
        packetHandler): Scene(sceneHandler, packetHandler) {
12      this->mTurnType = TURN_UNDEFINED;
13
```

```
14      if (this->mFont.loadFromFile(resourcePath() + "sansation.ttf"))
15      {
16          // error...
17      }
18
19      this->mGuessNumberTitleText.setFont(this->mFont);
20      this->mGuessNumberTitleText.setString("Waiting for other player's guess
        ");
21      this->mGuessNumberTitleText.setFillColor(sf::Color::White);
22
23      this->mGameStateInfo = "";
24      this->mInfoText.setFont(this->mFont);
25      this->mInfoText.setFillColor(sf::Color::White);
26      this->mInfoText.setString(this->mGameStateInfo);
27
28      this->mPlayer1Text.setFont(this->mFont);
29      this->mPlayer2Text.setFont(this->mFont);
30      this->mPlayer3Text.setFont(this->mFont);
31      this->mPlayer1Text.setFillColor(sf::Color::White);
32      this->mPlayer2Text.setFillColor(sf::Color::White);
33      this->mPlayer3Text.setFillColor(sf::Color::White);
34
35      this->mGuessNumberBox.setSize(sf::Vector2f(450, 80));
36      this->mGuessNumberBox.setFillColor(sf::Color::Magenta);
37
38      this->mGuessNumberText.setFont(this->mFont);
39      this->mGuessNumberText.setCharacterSize(50);
40      this->mGuessNumberText.setFillColor(sf::Color::Black);
41
42      using namespace std::placeholders;
43
44      this->mPacketHandler->SubscribeToOpCode(OpCodes::SERVER_MATCH_STARTED,
        std::bind(&GameScene::ReceiveMatchStartPacket, this, _1, _2, _3), true);
45      this->mPacketHandler->SubscribeToOpCode(OpCodes::SERVER_SET_TURN, std::
        bind(&GameScene::ReceiveSetTurnMessage, this, _1, _2, _3), true);
46      this->mPacketHandler->SubscribeToOpCode(OpCodes::SERVER_WRONG_NUMBER,
        std::bind(&GameScene::ReceiveWrongGuessMessage, this, _1, _2, _3), true)
        ;
47      this->mPacketHandler->SubscribeToOpCode(OpCodes::SERVER_CORRECT_NUMBER,
         std::bind(&GameScene::ReceiveCorrectGuessMessage, this, _1, _2, _3),
        true);
48      this->mPacketHandler->SubscribeToOpCode(OpCodes::SERVER_NEW_ROUND, std
        ::bind(&GameScene::ReceiveNewRoundMessage, this, _1, _2, _3), true);
49  }
50
51  void GameScene::SetTurnForPlayerWithName(std::string name) {
52
```

```
53 }
54
55 void GameScene::ReceiveNewRoundMessage(sf::Packet packet, sf::IpAddress
       sender, int port) {
56     packet >> this->mNumberToGuess;
57     bool isYou;
58     packet >> isYou;
59     if (isYou) {
60         this->SetYourTimeToGuess();
61     } else {
62         SetWaitingForOtherPlayersGuess();
63     }
64 }
65
66 void GameScene::ReceiveWrongGuessMessage(sf::Packet packet, sf::IpAddress
       sender, int port) {
67     int direction;
68     int guessedNumber;
69     packet >> guessedNumber;
70     packet >> direction;
71     if (direction == -1) {
72         this->mGameStateInfo = "The guessed number " + std::to_string(
       guessedNumber) + " was too low!";
73     } else if (direction == 1) {
74         this->mGameStateInfo = "The guessed number " + std::to_string(
       guessedNumber) + " was too high!";
75     } else {
76         this->mGameStateInfo = "The guessed number " + std::to_string(
       guessedNumber) + " was correct!";
77     }
78     this->mInfoText.setString(this->mGameStateInfo);
79 }
80
81 void GameScene::ReceiveSetTurnMessage(sf::Packet packet, sf::IpAddress
       sender, int port) {
82     std::string userTurn;
83     bool isYou;
84     packet >> userTurn;
85     packet >> isYou;
86     if (isYou) {
87         this->SetYourTimeToGuess();
88     } else {
89         this->SetWaitingForOtherPlayersGuess();
90     }
91 }
92
93 void GameScene::ReceiveCorrectGuessMessage(sf::Packet packet, sf::IpAddress
```

```
    sender, int port) {
94      // Close the game
95      this->mSceneHandler->SetVariable("isRunning", "NO");
96  }
97
98  void GameScene::Draw(sf::RenderWindow &window) {
99
100     this->mGuessNumberTitleText.setOrigin(this->mGuessNumberTitleText.
        getLocalBounds().left/2.0f, this->mGuessNumberTitleText.getLocalBounds()
        .top/2.0f);
101     this->mGuessNumberTitleText.setPosition(290,230);
102
103     this->mGuessNumberText.setOrigin(this->mGuessNumberText.getLocalBounds
        ().left/2.0f, this->mGuessNumberText.getLocalBounds().top/2.0f);
104     this->mPlayer1Text.setOrigin(this->mPlayer1Text.getLocalBounds().left
        /2.0f, this->mPlayer1Text.getLocalBounds().top/2.0f);
105     this->mPlayer2Text.setOrigin(this->mPlayer2Text.getLocalBounds().left
        /2.0f, this->mPlayer2Text.getLocalBounds().top/2.0f);
106     this->mPlayer3Text.setOrigin(this->mPlayer3Text.getLocalBounds().left
        /2.0f, this->mPlayer3Text.getLocalBounds().top/2.0f);
107     this->mGuessNumberText.setPosition(190,290);
108
109     // Input box
110     this->mGuessNumberBox.setPosition(180, 280);
111
112     this->mPlayer1Text.setPosition(150, 100);
113     this->mPlayer2Text.setPosition(350, 100);
114     this->mPlayer3Text.setPosition(550, 100);
115
116     this->mInfoText.setPosition(150, 500);
117
118     window.draw(this->mGuessNumberBox);
119     window.draw(this->mGuessNumberText);
120
121     window.draw(this->mPlayer1Text);
122     window.draw(this->mPlayer2Text);
123     window.draw(this->mPlayer3Text);
124
125     window.draw(this->mInfoText);
126 }
127
128 InputAction GameScene::GetTextInput(sf::Event event) {
129     if (event.type == sf::Event::TextEntered && this->mTurnType ==
        TURN_YOU_GUESS_NUMBER) {
130         if (event.text.unicode == '\b') {
131             if (this->mGuessNumber.getSize() > 0) {
132                 this->mGuessNumber.erase(this->mGuessNumber.getSize() - 1,
```

```
    1);
133             }
134         } else if (event.text.unicode == '\n') {
135             return InputAction::INPUT_SUBMIT_GUESS;
136         } else {
137             if (this->mGuessNumber.getSize() <= 10 && event.text.unicode <
    128 ) {
138                 mGuessNumber += event.text.unicode;
139             }
140         }
141         mGuessNumberText.setString(mGuessNumber);
142     }
143     return InputAction::INPUT_NO_ACTION;
144 }
145
146 void GameScene::ReceiveMatchStartPacket(sf::Packet packet, sf::IpAddress
    sender, int port) {
147     packet >> this->mMatchId;
148     packet >> this->mNumberToGuess;
149     std::string numberSelectorName;
150     packet >> numberSelectorName;
151     packet >> this->mPlayer1Name;
152     packet >> this->mPlayer2Name;
153     packet >> this->mPlayer3Name;
154
155     this->mPlayer1Text.setString(this->mPlayer1Name);
156     this->mPlayer2Text.setString(this->mPlayer2Name);
157     this->mPlayer3Text.setString(this->mPlayer3Name);
158
159     bool yourTurn;
160     packet >> yourTurn;
161     if (yourTurn) {
162         this->SetYourTimeToGuess();
163     }
164     this->mSceneHandler->SetScene(2);
165 }
166
167 void GameScene::SendGuessToServer(int guess) {
168     sf::Packet packet = this->CreatePacketWithOpCode(OpCodes::
    USER_GUESS_NUMBER, true);
169     packet << this->mMatchId;
170     packet << guess;
171     this->mPacketHandler->SendPacket(packet, this->mPacketHandler->
    GetSendAddress(), true);
172 }
173
174 int GameScene::Run(sf::RenderWindow &window) {
```

```
175     sf::Event Event;
176 
177         while (window.pollEvent(Event))
178         {
179             InputAction action = this->GetTextInput(Event);
180             if (action == InputAction::INPUT_SUBMIT_GUESS) {
181                 // Send guess to server
182                 try {
183                     int number = std::stoi(this->mGuessNumber.toAnsiString
    ());
184                     this->SendGuessToServer(number);
185                 } catch (std::exception &exception) {
186                     // error
187                     std::cout << "guess is not a number" << std::endl;
188                 }
189             }
190         }
191 
192         window.clear(sf::Color::Black);
193         this->Draw(window);
194         window.display();
195 }
196 
197 void GameScene::SetYourTimeToGuess() {
198     this->mTurnType = TURN_YOU_GUESS_NUMBER;
199     this->mGuessNumberTitleText.setString("Your turn to guess");
200     this->mGuessNumberBox.setFillColor(sf::Color::White);
201     this->mPlayer1Text.setFillColor(sf::Color::Blue);
202 }
203 
204 void GameScene::SetWaitingForOtherPlayersGuess() {
205     this->mTurnType = TURN_OTHER_PLAYER_GUESS_NUMBER;
206     this->mGuessNumber = "";
207     this->mGuessNumberTitleText.setString("Waiting for other player's guess
    ");
208     this->mGuessNumberBox.setFillColor(sf::Color::Magenta);
209 }
```

```
1 //
2 //  InputNameScene.hpp
3 //  Thesis-game
4 //
5 //  Created by Samuli Lehtonen on 27/04/2019.
6 //  Copyright © 2019 Samuli Lehtonen. All rights reserved.
7 //
8 
9 #ifndef InputNameScene_hpp
```

```
#define InputNameScene_hpp

#include <stdio.h>
#include "Scene.hpp"
#include "../../Thesis-game-server/Thesis-game-server/OpCodes.hpp"
#include <SFML/Network.hpp>
#include <functional>
#include <fstream>



class InputNameScene : public Scene {
private:
    void VerifyGameIntegrity();
    void DrawInputNameBox(sf::RenderWindow &window);
    void RequestAuthentication(std::string name);
    InputAction GetTextInput(sf::Event event);
    sf::Font mFont;
    sf::Text mInputTitleText;
    sf::Text mInputText;
    sf::String mPlayerNameText;
    sf::RectangleShape mInputBox;

public:
    InputNameScene(SceneHandler *sceneHandler, PacketHandler *packetHandler
    );
    virtual int Run(sf::RenderWindow &window);
    void JoinMatchMaking(std::string hash);
    void ReceiveAuthenticationHash(sf::Packet packet, sf::IpAddress sender)
    ;
    void ReceiveMatchFoundMessage(sf::Packet packet, sf::IpAddress sender);
    void ReceiveGameIntegrityMessage(sf::Packet packet, sf::IpAddress
    sender);
    void SendAuthenticatedStatus(std::string hash);
};

#endif /* InputNameScene_hpp */
```

```
//
//  InputNameScene.cpp
//  Thesis-game
//
//  Created by Samuli Lehtonen on 27/04/2019.
//  Copyright © 2019 Samuli Lehtonen. All rights reserved.
//

#include "InputNameScene.hpp"
```

```
10
11 InputNameScene::InputNameScene(SceneHandler *sceneHandler, PacketHandler *
       packetHandler): Scene(sceneHandler, packetHandler) {
12     if (this->mFont.loadFromFile(resourcePath() + "sansation.ttf"))
13     {
14         // error...
15     }
16     this->mInputTitleText.setFont(this->mFont);
17     this->mInputTitleText.setString("Enter your name:");
18     this->mInputTitleText.setFillColor(sf::Color::White);
19
20     this->mInputBox.setSize(sf::Vector2f(450, 80));
21     this->mInputBox.setFillColor(sf::Color::White);
22
23     this->mInputText.setFont(this->mFont);
24     this->mInputText.setCharacterSize(50);
25     this->mInputText.setString("");
26     this->mInputText.setFillColor(sf::Color::Black);
27
28     // Subscribe opcodes
29     using namespace std::placeholders;
30     this->mPacketHandler->SubscribeToOpCode(OpCodes::
       SERVER_GIVE_AUTHENTICATION_HASH, std::bind(&InputNameScene::
       ReceiveAuthenticationHash, this, _1, _2));
31     this->mPacketHandler->SubscribeToOpCode(OpCodes::
       SERVER_CONFIRM_GAME_INTEGRITY, std::bind(&InputNameScene::
       ReceiveGameIntegrityMessage, this, _1, _2));
32 }
33
34 void InputNameScene::ReceiveGameIntegrityMessage(sf::Packet packet, sf::
       IpAddress sender) {
35     this->RequestAuthentication(this->mPlayerNameText);
36 }
37
38 void InputNameScene::VerifyGameIntegrity() {
39     try {
40         std::ifstream in(getPathForResource(), std::ios::in | std::ios::
       binary);
41
42         std::ifstream::pos_type pos = in.tellg();
43         std::vector<char> buffer(pos);
44         in.seekg(0, std::ios::beg);
45         in.read(&buffer[0], pos);
46
47         std::hash<char*> hash;
48         size_t resultHash = hash(buffer.data());
49         std::cout << resultHash;
```

```
50
51          sf::Packet gameIntegrityPacket = this->CreatePacketWithOpCode(
    OpCodes::USER_SEND_FILE_HASH);
52          std::string hashAsString = std::to_string(resultHash);
53          gameIntegrityPacket << hashAsString;
54          this->mPacketHandler->SendPacket(gameIntegrityPacket, this->
    mPacketHandler->GetSendAddress());
55
56      } catch (std::exception &e) {
57          std::cout << "Failed to verify game integrity, aborting";
58      }
59  }
60
61  void InputNameScene::ReceiveMatchFoundMessage(sf::Packet packet, sf::
    IpAddress sender) {
62
63  }
64
65  void InputNameScene::SendAuthenticatedStatus(std::string hash) {
66      sf::Packet packet;
67      packet << OpCodes::USER_AUTHENTICATED;
68      packet << hash;
69      this->mPacketHandler->SendPacket(packet, this->mPacketHandler->
    GetSendAddress());
70  }
71
72  void InputNameScene::JoinMatchMaking(std::string hash) {
73      sf::Packet packet;
74      packet << OpCodes::USER_MATCHMAKING_ENTER_QUEUE;
75      packet << hash;
76      packet << this->mPlayerNameText.toAnsiString();
77      this->mPacketHandler->SendPacket(packet, this->mPacketHandler->
    GetSendAddress(), true);
78  }
79
80
81  void InputNameScene::ReceiveAuthenticationHash(sf::Packet packet, sf::
    IpAddress sender) {
82      std::string hash;
83      packet >> hash;
84      std::cout << "Received hash: ";
85      std::cout << hash;
86
87      this->mPacketHandler->SetDecryptionKey(hash);
88      this->SendAuthenticatedStatus(hash);
89      this->mSceneHandler->SetVariable("HASH", hash);
90      this->JoinMatchMaking(hash);
```

```
91 }
92
93 void InputNameScene::DrawInputNameBox(sf::RenderWindow &window) {
94
95     // Header text
96     this->mInputTitleText.setOrigin(this->mInputTitleText.getLocalBounds().
    left/2.0f, this->mInputTitleText.getLocalBounds().top/2.0f);
97     this->mInputTitleText.setPosition(290,230);
98
99     this->mInputText.setOrigin(this->mInputTitleText.getLocalBounds().left
    /2.0f, this->mInputTitleText.getLocalBounds().top/2.0f);
100    this->mInputText.setPosition(190,290);
101
102    // Input box
103    this->mInputBox.setPosition(180, 280);
104
105    window.draw(this->mInputTitleText);
106    window.draw(this->mInputBox);
107    window.draw(this->mInputText);
108 }
109
110 InputAction InputNameScene::GetTextInput(sf::Event event) {
111     if (event.type == sf::Event::TextEntered) {
112         if (event.text.unicode == '\b') {
113             if (this->mPlayerNameText.getSize() > 0) {
114                 this->mPlayerNameText.erase(this->mPlayerNameText.getSize()
    - 1, 1);
115             }
116         } else if (event.text.unicode == '\n') {
117             return InputAction::INPUT_SUBMIT_NAME;
118         } else {
119             if (this->mPlayerNameText.getSize() <= 10 && event.text.unicode
    < 128 ) {
120                 mPlayerNameText += event.text.unicode;
121             }
122         }
123         mInputText.setString(mPlayerNameText);
124     }
125     return InputAction::INPUT_NO_ACTION;
126 }
127
128 void InputNameScene::RequestAuthentication(std::string name) {
129     sf::Packet packet = this->CreatePacketWithOpCode(OpCodes::
    USER_REQUEST_AUTHENTICATION_HASH);
130     this->mPacketHandler->SendPacket(packet, this->mPacketHandler->
    GetSendAddress());
131 }
```

```
132
133
134 int InputNameScene::Run(sf::RenderWindow &window) {
135     sf::Event Event;
136
137     while (window.pollEvent(Event))
138     {
139         InputAction action = this->GetTextInput(Event);
140         if (action == InputAction::INPUT_SUBMIT_NAME) {
141             this->VerifyGameIntegrity();
142         }
143     }
144
145     window.clear(sf::Color(255,123,1,0));
146     this->DrawInputNameBox(window);
147     window.display();
148 }
```

```
1
2 //
3 // Disclaimer:
4 // ----------
5 //
6 // This code will work only if you selected window, graphics and audio.
7 //
8 // Note that the "Run Script" build phase will copy the required frameworks
9 // or dylibs to your application bundle so you can execute it on any OS X
10 // computer.
11 //
12 // Your resource files (images, sounds, fonts, ...) are also copied to your
13 // application bundle. To get the path to these resources, use the helper
14 // function `resourcePath()` from ResourcePath.hpp
15 //
16
17 #include <SFML/Audio.hpp>
18 #include <SFML/Graphics.hpp>
19 #include <SFML/Network.hpp>
20 #include "ResourcePath.hpp"
21 #include "../../Thesis-game-server/Thesis-game-server/OpCodes.hpp"
22 #include "Game.hpp"
23
24 int main(int, char const**)
25 {
26     Game game;
27     game.Run();
28
29     return EXIT_SUCCESS;
```

```
}
```

```
//
//  PacketHandler.hpp
//  Thesis-game-server
//
//  Created by Samuli Lehtonen on 21/04/2019.
//  Copyright © 2019 Samuli Lehtonen. All rights reserved.
//

#ifndef PacketHandler_hpp
#define PacketHandler_hpp

#include <stdio.h>
#include <SFML/Network.hpp>
#include <iostream>
#include <unordered_map>

class PacketHandler {
public:
    PacketHandler(int sendPort, sf::UdpSocket * socket);
    sf::Socket::Status SendPacket(sf::Packet packet, sf::IpAddress receiverIp, bool encrypt = false);
    sf::Socket::Status SendPacket(sf::Packet packet, sf::IpAddress receiverIp, unsigned short receiverPort, bool encrypt = false);
    sf::Socket::Status SendPacket(sf::Packet packet, sf::IpAddress receiverIp, unsigned short receiverPort, std::string key);
    void HandlePacket(sf::Packet packet, sf::IpAddress senderIp, int port);
    void SubscribeToOpCode(int opCode, std::function<void(sf::Packet, sf::IpAddress, int)> function);
    void SubscribeToOpCode(int opCode, std::function<void(sf::Packet, sf::IpAddress, int)>, bool decrypt);
    void SetReceivePort(int receivePort);
    void SetDecryptionKey(std::string key);
    void SetEncryptionKeyForAddress(sf::IpAddress address, unsigned short port, std::string key);
    void SetSendAddress(sf::IpAddress address);
    sf::IpAddress GetSendAddress();
    int GetReceivePort();

private:
    bool mUseEncryption;
    int mSendPort;
    int mReceivePort;
    std::string mCryptoKey;
    sf::IpAddress mSendAddress;
    sf::UdpSocket * mUdpSocket;
```

```
40      std::unordered_map<int, std::function<void(sf::Packet, sf::IpAddress,
        int)>> mPacketHandlers;
41      std::unordered_map<int, bool> mNeedsDecryption;
42      std::unordered_map<std::string, std::string> mEncryptionKeys;
43      std::string CreateAddresString(sf::IpAddress address, unsigned short
        port);
44      sf::Packet EncryptData(sf::Packet packet);
45      sf::Packet EncryptData(sf::Packet packet, std::string key);
46      sf::Packet DecryptData(sf::Packet packet, std::string key);
47      sf::Packet DecryptData(sf::Packet packet);
48  };
49
50  #endif /* PacketHandler_hpp */
```

```
1   //
2   //  PacketHandler.cpp
3   //  Thesis-game-server
4   //
5   //  Created by Samuli Lehtonen on 21/04/2019.
6   //  Copyright © 2019 Samuli Lehtonen. All rights reserved.
7   //
8
9   #include "PacketHandler.hpp"
10
11  PacketHandler::PacketHandler(int sendPort, sf::UdpSocket * socket):
        mSendPort(sendPort), mUdpSocket(socket), mUseEncryption(true) {
12      //this->mUseEncryption = false;
13  }
14
15  sf::Packet PacketHandler::EncryptData(sf::Packet packet) {
16      return this->EncryptData(packet, this->mCryptoKey);
17  }
18
19  sf::Packet PacketHandler::EncryptData(sf::Packet packet, std::string key) {
20      // If encryption is not in use, just return the packet as normal
21      if (!this->mUseEncryption) {
22          return packet;
23      }
24      // Extract opcode from the packet
25      sf::Packet encryptedPacket;
26
27      int opCode;
28      packet >> opCode;
29      encryptedPacket << opCode;
30
31      // Get data of the packet into buffer
32      const char *charBuffer = (char*)packet.getData()+sizeof(char)*4;
```

```
33      int n = packet.getDataSize()-sizeof(char)*4;
34
35      // Convert the char* data buffer to std::vector data buffer
36      std::vector<char> data(charBuffer, charBuffer + n);
37      std::size_t dataSize = n;
38      std::vector<char> encryptedData;
39
40      // Encrypt by XOR:ing with encryption key
41      for (std::size_t i = 0; i < dataSize; i++) {
42          encryptedData.push_back(data[i] ^ key[i % key.length()]);
43      }
44
45      // Create the encrypted packet and append the data
46
47      encryptedPacket.append(encryptedData.data(), dataSize);
48      return encryptedPacket;
49  }
50
51  sf::Packet PacketHandler::DecryptData(sf::Packet packet, std::string key) {
52      sf::Packet decryptedPacket = this->EncryptData(packet, key);
53      int opCode;
54      decryptedPacket >> opCode;
55      return decryptedPacket;
56  }
57
58  sf::Packet PacketHandler::DecryptData(sf::Packet packet) {
59      return this->DecryptData(packet, this->mCryptoKey);
60  }
61
62  void PacketHandler::SetSendAddress(sf::IpAddress address) {
63      this->mSendAddress = address;
64  }
65
66  sf::IpAddress PacketHandler::GetSendAddress() {
67      return this->mSendAddress;
68  }
69
70  void PacketHandler::SetEncryptionKeyForAddress(sf::IpAddress address,
        unsigned short port, std::string key) {
71      std::string stringAddress = this->CreateAddresString(address, port);
72      this->mEncryptionKeys[stringAddress] = key;
73  }
74
75  std::string PacketHandler::CreateAddresString(sf::IpAddress address,
        unsigned short port) {
76      return address.toString() + ":" + std::to_string(port);
77  }
```

```
78
79 sf::Socket::Status PacketHandler::SendPacket(sf::Packet packet, sf::
       IpAddress receiverIp, unsigned short receiverPort, std::string key) {
80     packet = this->EncryptData(packet, key);
81     return this->mUdpSocket->send(packet, receiverIp, receiverPort);
82 }
83
84 sf::Socket::Status PacketHandler::SendPacket(sf::Packet packet, sf::
       IpAddress receiverIp, bool encrypt) {
85     return this->SendPacket(packet, receiverIp, this->mSendPort, encrypt);
86 }
87
88 sf::Socket::Status PacketHandler::SendPacket(sf::Packet packet, sf::
       IpAddress receiverIp, unsigned short receiverPort, bool encrypt) {
89     if (encrypt) {
90         packet = this->EncryptData(packet);
91     }
92     sf::Socket::Status res = this->mUdpSocket->send(packet, receiverIp,
       receiverPort);
93     return res;
94 }
95
96 void PacketHandler::SetReceivePort(int receivePort) {
97     this->mReceivePort = receivePort;
98 }
99
100 int PacketHandler::GetReceivePort() {
101     return this->mReceivePort;
102 }
103
104 void PacketHandler::HandlePacket(sf::Packet packet, sf::IpAddress senderIp,
        int port) {
105     sf::Packet copyPacket = packet;
106     int opCode = -1;
107     packet >> opCode;
108
109     if (this->mNeedsDecryption[opCode]) {
110         std::string stringAddress = this->CreateAddresString(senderIp, port
       );
111         if (this->mEncryptionKeys.find(stringAddress) != this->
       mEncryptionKeys.end()) {
112             packet = this->DecryptData(copyPacket, this->mEncryptionKeys[
       stringAddress]);
113         } else {
114             packet = this->DecryptData(copyPacket);
115         }
116     }
```

```
117
118     if (packet) {
119         // Data extraction succesfull
120         if (this->mPacketHandlers.find(opCode) != this->mPacketHandlers.end
    ()) {
121             // Found key
122             this->mPacketHandlers[opCode](packet, senderIp, port);
123         } else {
124             // Discard the packet with invalid opcode
125         }
126     }
127 }
128
129 void PacketHandler::SubscribeToOpCode(int opCode, std::function<void(sf::
    Packet, sf::IpAddress, int)> function) {
130     this->mNeedsDecryption[opCode] = false;
131     this->mPacketHandlers[opCode] = function;
132 }
133
134 void PacketHandler::SubscribeToOpCode(int opCode, std::function<void (sf::
    Packet, sf::IpAddress, int)> function, bool decrypt) {
135     this->mNeedsDecryption[opCode] = decrypt;
136     this->mPacketHandlers[opCode] = function;
137 }
138
139 void PacketHandler::SetDecryptionKey(std::string key) {
140     this->mCryptoKey = key;
141 }
```

```
1 //
2 //  Player.hpp
3 //  Thesis-game
4 //
5 //  Created by Samuli Lehtonen on 27/04/2019.
6 //  Copyright © 2019 Samuli Lehtonen. All rights reserved.
7 //
8
9 #ifndef Player_hpp
10 #define Player_hpp
11
12 #include <stdio.h>
13 #include <iostream>
14
15 class Player {
16     Player(std::string name);
17 private:
18     std::string mName;
```

```
};

#endif /* Player_hpp */
```

```
//
//  Player.cpp
//  Thesis-game
//
//  Created by Samuli Lehtonen on 27/04/2019.
//  Copyright © 2019 Samuli Lehtonen. All rights reserved.
//

#include "Player.hpp"

Player::Player(std::string name):mName(name) {

}
```

```
////////////////////////////////////////////////////////////
//
// SFML - Simple and Fast Multimedia Library
// Copyright (C) 2007-2018 Marco Antognini (antognini.marco@gmail.com),
//                         Laurent Gomila (laurent@sfml-dev.org)
//
// This software is provided 'as-is', without any express or implied warranty.
// In no event will the authors be held liable for any damages arising from the use of this software.
//
// Permission is granted to anyone to use this software for any purpose,
// including commercial applications, and to alter it and redistribute it freely,
// subject to the following restrictions:
//
// 1. The origin of this software must not be misrepresented;
//    you must not claim that you wrote the original software.
//    If you use this software in a product, an acknowledgment
//    in the product documentation would be appreciated but is not required.
//
// 2. Altered source versions must be plainly marked as such,
//    and must not be misrepresented as being the original software.
//
// 3. This notice may not be removed or altered from any source distribution.
//
////////////////////////////////////////////////////////////
```

```
25
26 #ifndef RESOURCE_PATH_HPP
27 #define RESOURCE_PATH_HPP
28
29 ////////////////////////////////////////////////////////////
30 // Headers
31 ////////////////////////////////////////////////////////////
32 #include <string>
33
34 ////////////////////////////////////////////////////////////
35 /// \brief Return the path to the resource folder.
36 ///
37 /// \return The path to the resource folder associate
38 /// with the main bundle or an empty string is there is no bundle.
39 ///
40 ////////////////////////////////////////////////////////////
41 std::string resourcePath(void);
42 std::string currentPath(void);
43 std::string getPathForResource(void);
44
45 #endif
```

```
1 ////////////////////////////////////////////////////////////
2 //
3 // SFML - Simple and Fast Multimedia Library
4 // Copyright (C) 2007-2018 Marco Antognini (antognini.marco@gmail.com),
5 //                         Laurent Gomila (laurent@sfml-dev.org)
6 //
7 // This software is provided 'as-is', without any express or implied
    warranty.
8 // In no event will the authors be held liable for any damages arising from
     the use of this software.
9 //
10 // Permission is granted to anyone to use this software for any purpose,
11 // including commercial applications, and to alter it and redistribute it
    freely,
12 // subject to the following restrictions:
13 //
14 // 1. The origin of this software must not be misrepresented;
15 //    you must not claim that you wrote the original software.
16 //    If you use this software in a product, an acknowledgment
17 //    in the product documentation would be appreciated but is not required
    .
18 //
19 // 2. Altered source versions must be plainly marked as such,
20 //    and must not be misrepresented as being the original software.
21 //
```

```
22 // 3. This notice may not be removed or altered from any source
       distribution.
23 //
24 ////////////////////////////////////////////////////////////
25
26 ////////////////////////////////////////////////////////////
27 // Headers
28 ////////////////////////////////////////////////////////////
29 #include "ResourcePath.hpp"
30 #import <Foundation/Foundation.h>
31
32 ////////////////////////////////////////////////////////////
33 std::string resourcePath(void)
34 {
35     NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];
36
37     std::string rpath;
38     NSBundle* bundle = [NSBundle mainBundle];
39
40     if (bundle == nil) {
41 #ifdef DEBUG
42         NSLog(@"bundle is nil... thus no resources path can be found.");
43 #endif
44     } else {
45         NSString* path = [bundle resourcePath];
46         rpath = [path UTF8String] + std::string("/");
47     }
48
49     [pool drain];
50
51     return rpath;
52 }
53
54 std::string currentPath(void)
55 {
56     NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];
57     NSFileManager *fileManager;
58     fileManager = [[NSFileManager alloc] init];
59     [pool drain];
60     NSURL *appFolder = [[[NSBundle mainBundle] bundleURL]
       URLByDeletingLastPathComponent];
61     return [[fileManager currentDirectoryPath] UTF8String] + std::string("/
       ");
62 }
63
64 std::string getPathForResource()
65 {
```

```
66      NSString *filePath = [[NSBundle mainBundle] pathForResource:@"test"
        ofType:@"txt"];
67      return std::string([filePath UTF8String]);
68  }
```

```
1   //
2   //  Scene.hpp
3   //  Thesis-game
4   //
5   //  Created by Samuli Lehtonen on 27/04/2019.
6   //  Copyright © 2019 Samuli Lehtonen. All rights reserved.
7   //
8
9   #ifndef Scene_hpp
10  #define Scene_hpp
11
12  #include <stdio.h>
13  #include <SFML/Graphics.hpp>
14  #include "PacketHandler.hpp"
15  #include "SceneHandler.hpp"
16  #include "ResourcePath.hpp"
17  #include "../../Thesis-game-server/Thesis-game-server/OpCodes.hpp"
18
19  enum InputAction {
20      INPUT_NO_ACTION = 0,
21      INPUT_SUBMIT_NAME = 1,
22      INPUT_SUBMIT_GUESS = 2
23  };
24
25  class SceneHandler;
26
27  class Scene {
28  public:
29      Scene(SceneHandler *sceneHandler, PacketHandler *packetHandler);
30      virtual int Run (sf::RenderWindow & window) = 0;
31
32  protected:
33      SceneHandler *mSceneHandler;
34      PacketHandler *mPacketHandler;
35      sf::Packet CreatePacketWithOpCode(OpCodes opCode, bool
        setAuthenticationHash = false);
36  };
37
38  #endif /* Scene_hpp */
```

```
1   //
2   //  Scene.cpp
```

```
//  Thesis-game
//
//  Created by Samuli Lehtonen on 27/04/2019.
//  Copyright © 2019 Samuli Lehtonen. All rights reserved.
//

#include "Scene.hpp"

Scene::Scene(SceneHandler *sceneHandler, PacketHandler *packetHandler):
    mSceneHandler(sceneHandler), mPacketHandler(packetHandler) {

}

sf::Packet Scene::CreatePacketWithOpCode(OpCodes opCode, bool
    setAuthenticationHash) {
    sf::Packet packet;
    packet << opCode;
    if (setAuthenticationHash) {
        if (this->mSceneHandler->HasVariable("HASH")) {
            packet << this->mSceneHandler->GetVariable("HASH");
        } else {
            throw std::runtime_error("HASH not found in map");
        }
    }
    return packet;
}
```

```
//
//  SceneHandler.hpp
//  Thesis-game
//
//  Created by Samuli Lehtonen on 27/04/2019.
//  Copyright © 2019 Samuli Lehtonen. All rights reserved.
//

#ifndef SceneHandler_hpp
#define SceneHandler_hpp

#include <stdio.h>
#include <iostream>
#include <unordered_map>
#include <SFML/Graphics.hpp>
#include "Scene.hpp"

class Scene;

class SceneHandler {
```

```
21 private:
22     std::unordered_map<int, Scene*> mScenes;
23     std::unordered_map<std::string, std::string> mVariables;
24 
25     sf::RenderWindow &mWindow;
26     int mCurrentScene = 0;
27 public:
28     SceneHandler(sf::RenderWindow &window);
29     ~SceneHandler();
30     void DrawCurrentScene();
31     void RegisterScene(int sceneId, Scene * scene);
32     void SetScene(int sceneId);
33     void SetVariable(std::string key, std::string variable);
34     std::string GetVariable(std::string key);
35     bool HasVariable(std::string key);
36 };
37 
38 #endif /* SceneHandler_hpp */
```

```
1  //
2  //  SceneHandler.cpp
3  //  Thesis-game
4  //
5  //  Created by Samuli Lehtonen on 27/04/2019.
6  //  Copyright © 2019 Samuli Lehtonen. All rights reserved.
7  //
8  
9  #include "SceneHandler.hpp"
10 
11 SceneHandler::SceneHandler(sf::RenderWindow &window):mWindow(window) {
12 
13 }
14 
15 SceneHandler::~SceneHandler() {
16     // Delete all scenes here
17 }
18 
19 void SceneHandler::RegisterScene(int sceneId, Scene * scene) {
20     this->mScenes[sceneId] = scene;
21 }
22 
23 void SceneHandler::DrawCurrentScene() {
24     this->mScenes[this->mCurrentScene]->Run(this->mWindow);
25 }
26 
27 void SceneHandler::SetScene(int sceneId) {
28     this->mCurrentScene = sceneId;
```

```
}

void SceneHandler::SetVariable(std::string key, std::string variable) {
    this->mVariables[key] = variable;
}

std::string SceneHandler::GetVariable(std::string key) {
    return this->mVariables[key];
}

bool SceneHandler::HasVariable(std::string key) {
    try {
        this->mVariables.at(key);
        return true;
    } catch (std::exception & exception) {
        return false;
    }
}
```

C Memory obfuscation benchmarking code

```
#include <iostream>
#include <chrono>

using namespace std::chrono;

template <typename T>
T *SetVariable(T *currentVariable, T value) {
    delete currentVariable;
    T *newValue = new T(value);
    return newValue;
}

void TestWithRelocation() {
    int *a = new int(50);
    high_resolution_clock::time_point t1 = high_resolution_clock::now();
    for (int i = 0; i < 1000000; i++) {
        a = SetVariable(a, i);
    }
    high_resolution_clock::time_point t2 = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>( t2 - t1 ).count();
    std::cout << "Test with relocation: " << duration << std::endl;
}

void TestWithRelocationAndEncryption() {
    int *a = new int(50);
```

```
26      high_resolution_clock::time_point t1 = high_resolution_clock::now();
27      for (int i = 0; i < 1000000; i++) {
28          *a = *a ^ 10; // Simple XOR operation to simulate light encryption
29          a = SetVariable(a, i);
30      }
31      high_resolution_clock::time_point t2 = high_resolution_clock::now();
32      auto duration = duration_cast<microseconds>( t2 - t1 ).count();
33      std::cout << "Test with relocation and encryption: " << duration << std
        ::endl;
34  }
35
36  void TestWithoutRelocation() {
37      int a = 50;
38      high_resolution_clock::time_point t1 = high_resolution_clock::now();
39      for (int i = 0; i < 1000000; i++) {
40          a = i;
41      }
42      high_resolution_clock::time_point t2 = high_resolution_clock::now();
43      auto duration = duration_cast<microseconds>( t2 - t1 ).count();
44      std::cout << "Test without relocation: " << duration << std::endl;
45  }
46
47  int main(int argc, const char * argv[]) {
48      TestWithRelocation();
49      TestWithoutRelocation();
50      TestWithRelocationAndEncryption();
51
52      return 0;
53  }
```