

# A novel approach for privacy preserving cheat detection in E-Sports using cloud- based computer vision techniques

MSc Research Project  
Cyber Security

Kartikey Sharma  
Student ID: 22128824

School of Computing  
National College of Ireland

Supervisor:     Jawad Salahuddin

**National College of Ireland**  
**MSc Project Submission Sheet**  
**School of Computing**



<b>Student Name:</b>	Kartikey Sharma
<b>Student ID:</b>	22128824
<b>Programme:</b>	MSc Cyber Security
<b>Year:</b>	2023
<b>Module:</b>	MSc Research Project
<b>Supervisor:</b>	Jawad Salahuddin
<b>Submission Due Date:</b>	14/12/2023
<b>Project Title:</b>	A novel approach for privacy preserving cheat detection in E-Sports using cloud-based computer vision techniques
<b>Word Count:</b>	6397
<b>Page Count:</b>	21

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

**Signature:** *Kartikey Sharma*

**Date:** 14/12/23

**PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST**

Attach a completed copy of this sheet to each project (including multiple copies)	<input type="checkbox"/>
<b>Attach a Moodle submission receipt of the online project submission</b> , to each project (including multiple copies).	<input type="checkbox"/>
<b>You must ensure that you retain a HARD COPY of the project</b> , both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	<input type="checkbox"/>

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

<b>Office Use Only</b>	
Signature:	
Date:	
Penalty Applied (if applicable):	

# A novel approach for privacy preserving cheat detection in E-Sports using cloud-based computer vision techniques

Kartikey Sharma  
22128824

## Abstract

The exponential growth of the e-sports industry has been shadowed by the rise of sophisticated cheating methods, posing significant threats to the integrity and fairness of competitive gaming. This research addresses the pressing need for effective, non-invasive cheat detection mechanisms by exploring the integration of cloud-based computer vision techniques. Specifically, it investigates using a fine-tuned YOLOv8 model in the cloud for real-time cheat detection in e-sports. The motivation behind this study is to establish a robust anti-cheat framework that balances stringent security measures with paramount concerns for user privacy, a balance often overlooked in conventional anti-cheat systems.

Our research involved developing a real-time object detection model trained on data from the open-source game AssaultCube. The model's performance was evaluated based on its accuracy, precision, recall, and processing speed, particularly in detecting cheats like Extra Sensory Perception (ESP) hacks. Significant findings include the model's high precision and recall rates for the classifications and its remarkable processing speed, achieving frame processing in less than 15 milliseconds on average when deployed on a GPU-enhanced cloud platform. This study not only demonstrates the feasibility of implementing AI-driven cheat detection in real-time but also opens new avenues for ethical and privacy-preserving approaches in gaming security.

The research aims to provide a groundbreaking perspective in the application of vision AI and cloud computing in cybersecurity, particularly in the domain of e-sports, setting the stage for more advanced, efficient, and ethically conscious anti-cheat systems in the future.

## 1 Introduction

The global video game industry has witnessed extraordinary growth, reaching a remarkable valuation of \$384.9 billion in 2023.<sup>1</sup> This boom, however, has been paralleled by the emergence of sophisticated cheating methods that threaten the integrity of e-sports. Anonymous groups, adept in reverse engineering, have been developing and selling unauthorized plugins and bot programs. These illicit tools offer unfair advantages to users, tarnishing the gaming experience and posing financial risks to developers by alienating legitimate players.

A variety of anti-cheat techniques have been developed and implemented to maintain the integrity of competitive gaming, ranging from software solutions to physical and procedural measures. Some of the popular software solutions are Steam's Valve Anti-Cheat (VAC)<sup>2</sup>, Epic

---

<sup>1</sup> Statista Market Insights, July 2023: <https://www.statista.com/study/39310/video-games/>

<sup>2</sup> Valve Anti-Cheat: [https://developer.valvesoftware.com/wiki/Valve\\_Anti-Cheat](https://developer.valvesoftware.com/wiki/Valve_Anti-Cheat)

Games' EasyAntiCheat (EAC)<sup>3</sup>, and the infamous Vanguard used in Valorant<sup>4</sup>. While these solutions have been proven to be effective in detection and automatic non-negotiable ban of most cheaters that use freely available cheat sources on the internet, custom made hacks and cheats which can cost thousands of dollars for the cheaters are harder to detect especially during an on-going event. The hosts or developers of the competitive game may dissect the logs and files directly to ascertain the possibility of cheating, but it costs both time and money and may take multiple days to report after the end of the event.

Despite the deployment of advanced detection techniques, however invasive or authoritative they may be, cunning cheaters continue to outmanoeuvre these systems, exploiting vulnerabilities and evading capture. This ongoing battle against cheating underscores the necessity for more robust, innovative solutions. Recognizing this, our research pivots to harnessing the potential of vision AI and cloud-based technologies. These emerging approaches promise a leap forward in cheat detection capabilities, blending near-human cognizance with efficient, scalable frameworks.

Central to our study is the exploration of cutting-edge vision AI technology, deployed in a cloud-based environment to create a formidable anti-cheat platform. This approach is designed to minimize resource usage and data collection, striking a critical balance between robust security and user privacy. By processing minimal data (frames from the game in this case) we are virtually collecting no user-data which protects our solution from the falling under data protection laws like the GDPR standards, ensuring privacy preservation alongside precise detection.

This research is driven by a pivotal question: How can we leverage the latest advancements in vision AI to develop a robust anti-cheat framework for competitive e-sports, and how can its deployment in a high-performance cloud environment enhance security and privacy? Our endeavour is a trailblazing effort to develop AI models specialized in identifying cheating in video games, thereby expanding AI's footprint in cybersecurity. We aim to demonstrate the viability and effectiveness of a cloud-based anti-cheat system. The envisioned system promises to match or even surpass existing anti-cheat mechanisms, while prioritizing ethical considerations and minimizing user data and resource usage.

The anticipated outcome of this research is the establishment of an open architecture for AI-based anti-cheat systems. This will not only mark a significant advancement in the field but also lay the groundwork for future research and development, contributing to the ongoing discourse on balancing security, privacy, and ethical practices in the digital gaming realm.

## 2 Related Work

Cheating can ruin the experience for regular players and harm the game companies financially. Effective anti-cheat systems are needed to combat this. In their comparative study of anti-cheat methods in video games Samuli Lehtonen (2020), the author, categorized anti-cheat methods into server-side and client-side. Server-side techniques operate on the game server while client-

---

<sup>3</sup> Easy Anti-Cheat: <https://www.easy.ac/en-us/>

<sup>4</sup> Vanguard: <https://support-valorant.riotgames.com/hc/en-us/articles/360046160933-What-is-Vanguard->

side methods run on the player's computer. While other authors like Zhang (2021) and Kedziora et al. (2020) explain kernel events based anti-cheats and blockchain based solutions.

## **2.1 Existing Anti-Cheat Methods**

In a comparative study of existing anti-cheat methods by Samuli Lehtonen (2020), the author has emphasized that cheating has been an issue in online games since their inception and can have major impacts on game companies' revenues and reputations. Thus, having effective anti-cheat systems is critical. The paper categorizes common cheating methods into "soft cheats" that exploit game mechanics, and "hard cheats" that utilize external programs and tamper with the game client or network traffic. It then systematically analyses major anti-cheat approaches, including not trusting client data, tamper-resistant protocols, network traffic obfuscation, statistical methods, client-side protections like code encryption and memory obfuscation, and kernel-level anti-cheat drivers. The analysis examines the techniques across several key criteria: resistance to tampering, ease of implementation, performance overhead, privacy/invasiveness, and suitability for different game types. An example game was implemented to demonstrate implementations and impacts.

The paper finds that server-side methods generally provide the strongest protection against cheating but have difficulty catching certain client-side cheats. Hybrid server and client-side solutions are ideal. Among client protections, code encryption helps safeguard other methods from tampering. Memory obfuscation also proves powerful but introduces major overhead.

The analysis highlights rising privacy concerns and favourability of non-invasive solutions like server-side statistical analysis. Game streaming is flagged as an emerging approach that could virtually eliminate cheating, but latency and adoption issues persist.

Overall, the paper systematically validates strengths and weaknesses of anti-cheat techniques based on quantitative testing and multiple criteria. It provides guidance to developers on optimal implementations. The analysis points to server-side and hybrid solutions as the future, with machine learning and cloud analytics holding promise.

The paper would benefit from more comparisons with real games using the various methods. The custom game example, while illustrative, lacks the complexity of commercial titles. Analysing case studies across genres could better highlight nuances and practical impacts. The criteria used for evaluation are appropriate but could weigh the importance of tamper resistance and overhead more, given their direct impacts on anti-cheat effectiveness. The paper could also examine techniques common in mobile games, which face unique cheating threats.

Additionally, explicitly addressing legal and ethical concerns around invasive anti-cheat approaches could better cover industry best practices. Overall, the paper delivers a strong, structured analysis using apt examples and evaluation metrics like the table below.

**Table 1: Comparison of different anti-cheat methods**

	Resistance to tampering	Ease of implementation	Lack of overhead	Non-invasiveness	Suitability for wide variety of games
<b>Server-sided methods</b>					
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
<b>Client-side methods</b>					
Code encryption	2	1	2	4	4
Verifying file 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

## 2.2 Use of Artificial Intelligence

The integration of AI in anti-cheat systems offers a promising avenue and is exactly what our research project aims to achieve. One such research by Zhang (2021) proposes using artificial intelligence models to improve the timeliness of online game anti-cheat systems. It discusses the prevalence of cheating in online games and the limitations of current anti-cheat approaches, particularly the lag time in detecting and penalizing cheaters. Two AI-based solutions are presented - one analysing player behaviour based on game data, and one based on image recognition of the game screen.

**Existing Anti-Cheat Methods** - The paper summarizes common cheating methods like network packet modification and RAM cheating, which exploit vulnerabilities in game data communication and storage. It then describes current anti-cheat techniques:

- Improving packet encryption makes packets harder to intercept and crack but causes lag.
- Server-side cheat detection is computationally expensive and requires constant updating.
- Embedded anti-cheat programs are a privacy concern and can be circumvented.
- Player reporting is manually intensive, and the response is delayed.

The major weakness highlighted is the time lag between a cheat occurring and action taken, diminishing the gaming experience.

**Proposed AI Solutions** - Two AI models are proposed to judge cheating based on behaviour rationality rather than just detecting malicious software. The first involves analysing player data like health levels and mouse coordinates to identify unnatural patterns indicative of cheats. The author suggests binary classification or one-class SVM models but does not implement

them due to lacking suitable training data. The second is an image recognition model to extract information from game screens and determine if player actions align with expectations. A basic model is built and tested on CS: GO screens, showing feasibility but needing larger datasets and additional logic algorithms.

The paper makes a case for using AI to improve anti-cheat timeliness. The image recognition approach is more promising, generalizable to different games, and exactly what we hope to implement in this research project. With further development, such systems could complement existing methods and preserve multiplayer fairness.

Key limitations are the lack of actual game data for training and testing, and no implementation of the full envisioned systems. Technically, we are using this paper as our baseline and further extrapolating on it by attempting to train and implement the AI model on cloud. As mentioned in the future work we must focus on accessing suitable datasets and integrating the models into commercial games for real-world trials.

### **2.3 Risks of Kernel Anti-Cheat Software**

Existing anti-cheat systems take different approaches to detecting and preventing cheating in online games. Some client-side systems like VAC (Valve Anti-Cheat) rely on scanning for known cheat signatures and patterns in memory. If a detected cheat has checked-in with a DRM server, the user can then be banned. However, the downside with signature-based methods is that they cannot detect new or polymorphic cheats. Meanwhile, more invasive kernel-level anti-cheats run drivers at startup to block potentially vulnerable programs and drivers that cheats could exploit. But this risks stability issues, security vulnerabilities, and privacy concerns over their deep system access.

To avoid the issues with client-side anti-cheat software, Maario et al. (2021) proposed improved server-side statistical and machine learning approaches to classify player behaviour and detect cheating probabilistically. This method processes large datasets on the server to identify suspicious trends without needing kernel access. However, the computational overhead and privacy implications around mass data collection remain open questions.

In a separate study on cheat injectors, Karkallis et al. (2021) analysed online cheating communities and exploit marketplaces. They highlighted major hubs like UnknownCheats and MPGH facilitating widespread cheat distribution between hundreds of thousands of members. The advanced techniques and strong cooperation in these communities often underpinned by financial incentives were noted to technically overlap with malware development ecosystems. This illustrates the broader security risks certain game cheating tools can pose through arbitrary code execution.

Overall, each approach balances effectiveness, performance, stability, privacy, and security trade-offs differently. But advanced server-side systems present a promising direction by avoiding invasive client access while improving detection rates through large-scale behavioural modelling. Tighter analysis of exploit marketplaces also merits continued attention on the interplay between game hacking and malware ecosystems.

## 2.4 Why secure video games?

Online video games have become immensely popular, with millions of people around the world playing games like World of Warcraft, Fortnite, and League of Legends. However, as Yan and Randell (2009) discuss in their paper “An Investigation of Cheating in Online Games”, cheating has emerged as a major issue that threatens the integrity and enjoyment of these games. Securing online games is critical to maintain fairness and prevent cheating that ruins the gaming experience for honest players. A core appeal of video games is the sense of fair competition they create. As Yan and Randell explain, cheating enables dishonest players to gain unfair advantages over opponents. This destroys the spirit of fair play that makes games worthwhile. Cheating can allow players to exploit gameplay bugs, hack the game code, or use external tools to reveal hidden information. Effective security is necessary to ensure all players compete on a level playing field. Lax security enables cheating that gives cheaters unearned victories while honest players feel the game is rigged against them. Cheating scandals can seriously damage the reputation and trustworthiness of a game developer. If cheating is seen as rampant and unchecked, players lose faith that the developer cares about integrity. This causes loss of loyal customers and revenue. Robust security reassures players the developer takes fairness seriously and reduces situations where cheating undermines trust. Ongoing security also protects the brand reputation developers depend on for continued success.

Most importantly, preventing cheating enhances enjoyment for gamers. As Yan and Randell make clear, cheating degrades the gaming experience for typical players. Facing opponents who cheat takes away the fun and satisfaction derived from playing by the rules and winning fairly. Effective security locks out cheaters so honest players can immerse themselves in the game world and derive fulfilment from legitimate competition and achievement. Stopping cheating enables developers to maximize entertainment value for customers.

The authors effectively argue that securing online games is vital because cheating sabotages fair play, erodes developer trust, and diminishes gaming enjoyment. Game developers must prioritize robust security protections to foster gaming experiences that attract and delight players over the long term. Preventing cheating through ongoing security efforts helps online games reach their full potential as captivating entertainment.

## 3 Research Methodology

During the inception of this research project, it was understood that our experience with game development and C++ could prove as a strength to be able to write a proof of concept as the ability to write directly to a process memory is available in C/C++ using the Windows memory API<sup>5</sup>. Moreover, choosing an open-source game that closely resembles most popular titles was deemed to be the ideal tool for collection of datasets for training of the AI to simulate real-life performance in terms of functionality.

---

<sup>5</sup> Windows memoryapi.h header: <https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/>



### 3.1 Game Selection

We selected the open-source first-person shooter game AssaultCube as our test environment. AssaultCube (Rabid Viper Productions, 2006) is an ideal choice due to its accessibility, modifiability, and simplicity of implementation that can be leveraged.

### 3.2 Data Collection

To generate training data, we developed an ESP (wallhack) cheat for AssaultCube using C++ (Abramov, 2022). The ESP cheat provides players with unfair advantages by revealing the locations of other players within the game world. With the cheat enabled, we automated the collection of in-game screenshots capturing various scenarios of the cheat in action, such as displaying outlines around enemy players.

### 3.3 Data Annotation

The collected screenshots were uploaded to Roboflow<sup>6</sup> for annotation. Each image was manually labelled, marking the visual elements added by the wallhack cheat such as player outlines and position markers. Precise annotation of these cheating artifacts is essential for the model to learn to detect them.

### 3.4 Model Development

We trained a real-time object detection model using the YOLOv8<sup>7</sup> architecture on our annotated dataset. Training was performed for 100 epochs with an input resolution of 640x640. Extensive parameter tuning was conducted to optimize model accuracy and inference speed for real-time cheat detection during gameplay.

### 3.5 Model Deployment

The trained model was deployed locally where it was given a recorded gameplay session with ESP hack enabled. The next step in deployment was to containerize the model using docker and deploy on a cloud platform equipped with GPUs for efficient inferencing performance. A REST API endpoint was to be created to accept streams of game frames, run cheat detection on each frame using the model, and return predictions indicating potential cheating.

### 3.6 Testing and Results

Rigorous tests were conducted by playing AssaultCube with cheats enabled and monitoring warning outputs. Metrics gathered included model accuracy, inference lag, and impact on gameplay performance. Testing provided quantified estimates of model viability and areas for improvement.

## 4 Design Specification

### 4.1 Game Selection: AssaultCube

AssaultCube, an open-source, first-person shooter game, was chosen as the test environment for this study. Its open-source nature allows for controlled testing conditions and the creation of a cheat program specifically for this research.

---

<sup>6</sup> Roboflow: <https://roboflow.com/>

<sup>7</sup> YOLOv8: <https://yolov8.com/>



Figure 1: AssaultCube open-source game

## 4.2 Development of ESP (Extra Sensory Perception) Hack

An ESP hack was developed in C++ for AssaultCube. ESP hacks provide players with unfair advantages by displaying extra information within the game, such as the locations of other players. This hack in general is designed specifically to visually augment the game environment, making it suitable for detection via computer vision techniques. The development process involved the following:

- Reverse engineering the game's memory.
- Understanding the data structure used to describe in-game elements.
- Parsing that data to isolate the segment describing player and enemy entities.
- Fetching coordinates of the player model in the game world.
- Converting the 3D coordinates to a 2D rectangle scaling with the players location and size of the screen using a special WorldToScreen function.
- Rendering the rectangle graphics overlaying the player positions in real-time.

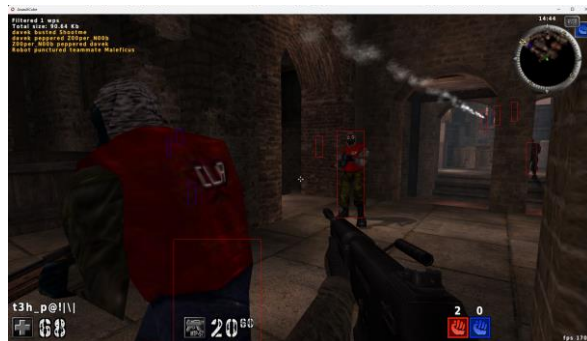


Figure 2: ESP cheat enabled on AssaultCube

### 4.2.1 Reverse engineering AssaultCube

The process of reverse engineering AssaultCube begins with using a tool called CheatEngine (Heijnen, 2008) which is a memory scanning and editing tool. With the help of cheat engine, we attach a debugger to the game's process "ac\_client.exe" and start scanning the memory to find an address that stores an in-game value like player health, ammo, or name.

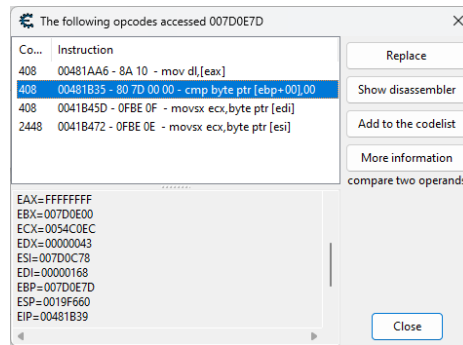


Figure 3: Attaching debugger to game process

The debugger attached to a found address helps us in finding the base address which is needed to enumerate through the entity list in the game. In this case, we can see that the value in EBP is being compared (cmp) to 00 which is probably a check if the object is null. This could be a function checking the player status, reload status or existence of a player entity. The instruction with the highest execution counter is copying the content of register ECX, which can either have a value or pointer to a location in memory, to the destination ESI (essentially reading data from the memory). Hence, we conclude that the value in ESI (007D0C78) is the location of our entity list.

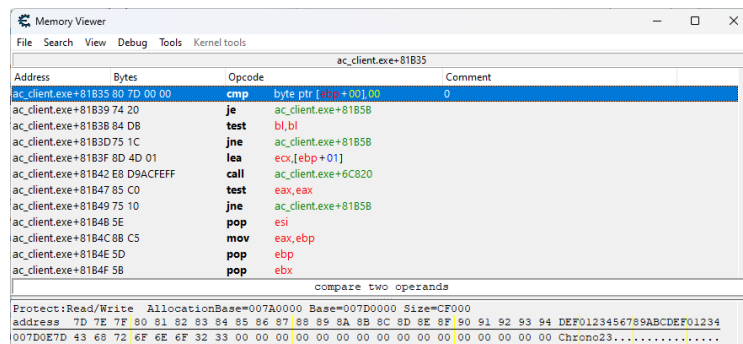


Figure 4: Memory viewer

Diving deep into the process memory, we begin dissecting the data structures from to generate the entity list we could use in writing the ESP. Essentially what we need is the game's base address and the offset to the entity list.

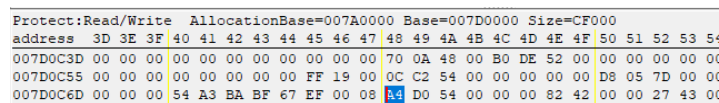


Figure 5: Hexadecimal representation of 8 byte segments of memory

We start by using the data structure dissection tool and then try to figure out the values that exist in that region.

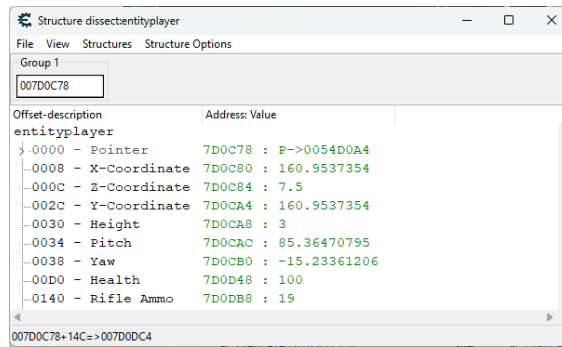


Figure 6: Data structure dissection

With the player entity offsets found, we proceed to writing our code.

#### 4.2.2 Coding our ESP Cheat

We pre-define our offsets into a header file as they will be needed multiple times throughout the code.

```
#define BASE_ADDRESS 0x00400000
#define BASE_PLAYER_OFFSET 0x18AC00
#define BASE_ENTITY_ARRAY_OFFSET 0x191FCC
#define PLAYER_OFFSET (BASE_ADDRESS + BASE_PLAYER_OFFSET)
#define ENTITY_ARRAY_OFFSET (BASE_ADDRESS + BASE_ENTITY_ARRAY_OFFSET)
```

Figure 7: Pre-defining base address and offsets

We define a handle object to attach to the game's process and store the window's width and height.

```
HANDLE hProcess;
int width, height;
```

Figure 8: Handle definition

Then we read the process memory using the ReadProcessMemory function available in memoryapi.h which is already included in Windows.h header file. The function returns the entity array read from the memory to the defined variable and sets the BOOL rpmEntityArray to true, signifying that a successful read has been executed. Otherwise, the BOOL value will be false.

```
uintptr_t p_entityArray;
BOOL rpmEntityArray = ReadProcessMemory(hProcess, (LPCVOID) ENTITY_ARRAY_OFFSET, &p_entityArray, sizeof(int), NULL);
```

Figure 9: ReadProcessMemory function

We loop through the array to individually store the x, y, and z coordinates of the enemy players in a Vec3 object which is a typedef of three float values to store the three coordinates.

```
Vec3 vector;
vector.x = x;
vector.y = y;
vector.z = z;
```

Figure 10: Vec3 typedef

The coordinates and playerMatrix are then passed over to the WorldToScreen function for conversion to a flat rectangle that needs to be drawn onto the screen as an overlay.

```

bool WorldToScreen(Vec3 pos, Vec3 *screen, float matrix[16], int windowHeight, int windowWidth)
{
    clipCoords.x = pos.x * matrix[0] + pos.y * matrix[4] + pos.z * matrix[8] + matrix[12];
    clipCoords.y = pos.x * matrix[1] + pos.y * matrix[5] + pos.z * matrix[9] + matrix[13];
    clipCoords.z = pos.x * matrix[2] + pos.y * matrix[6] + pos.z * matrix[10] + matrix[14];
    clipCoords.w = pos.x * matrix[3] + pos.y * matrix[7] + pos.z * matrix[11] + matrix[15];

    if (clipCoords.w < 0.1f) return false;

    NDC.x = clipCoords.x / clipCoords.w;
    NDC.y = clipCoords.y / clipCoords.w;
    NDC.z = clipCoords.z / clipCoords.w;

    screen->x = (windowWidth / 2 * NDC.x) + (NDC.x + windowHeight / 2);
    screen->y = -(windowHeight / 2 * NDC.y) + (NDC.y + windowHeight / 2);

    return true;
}

```

Figure 11: WorldToScreen function

Using the wm-paint library in windows, we draw the overlay and update it at a little over 60 frames per second to achieve real-time display of enemy locations on screen.

```

void esp()
{
    for (int i = 0; i < entityCount; i++)
    {
        bool world = WorldToScreen(entityArray[i]->vector, &screen, pData.matrix, width, height);
        if (world && entityArray[i]->health > 0) {
            if (true) {
                const COLORREF colorOfTeam = entityArray[i]->entityTeam == pData.team ? teamColor : enemyColor;
                const HBRUSH brushOfTeam = entityArray[i]->entityTeam == pData.team ? hBrushTeam : hBrushEnemy;
                DrawESP(screen.x, screen.y, GetDistance3D(pData.pos, entityArray[i]->vector), entityArray[i]->health, entityArray[i]->name, brushOfTeam, colorOfTeam);
            }
        }
    }
}

```

Figure 12: Main ESP function

After compilation and execution, the result will highlight enemy players in red and allies in blue.



Figure 13: Successful execution of the ESP

### 4.3 Data Collection: Screenshot Capturing

With the ESP hack enabled, a large dataset of screenshots was captured from the game. These screenshots represent the game environment as seen by a player using the ESP cheat, providing the necessary data for training the computer vision model. The capture process was automated to ensure a wide variety of game scenarios and player positions were represented in the dataset.





Figure 14: Sample of collected dataset for training

#### 4.4 Data Annotation: Roboflow

Roboflow, an annotation and dataset management tool, was employed to annotate the collected screenshots. This involved manually labelling the elements added by the ESP hack, such as player outlines or position markers. Precise annotation is crucial for training the model to accurately identify the signs of cheating in the game environment.

The following images showcase how the annotations were drawn on top of the training images. The drawn boxes are exported as coordinate values for the top left and bottom right corner of the annotation rectangles. These coordinates can be read by the AI model while training to determine exactly what to look for when predicting the presence of enabled cheats.



Figure 15: Sample Annotation on Training Images

The dataset was further enhanced by adding noise to the subjects that are to be identified. This helps in producing a much more robust training dataset where the model can be made efficient in case of compression algorithms reducing the visual fidelity of input while prediction is being performed.

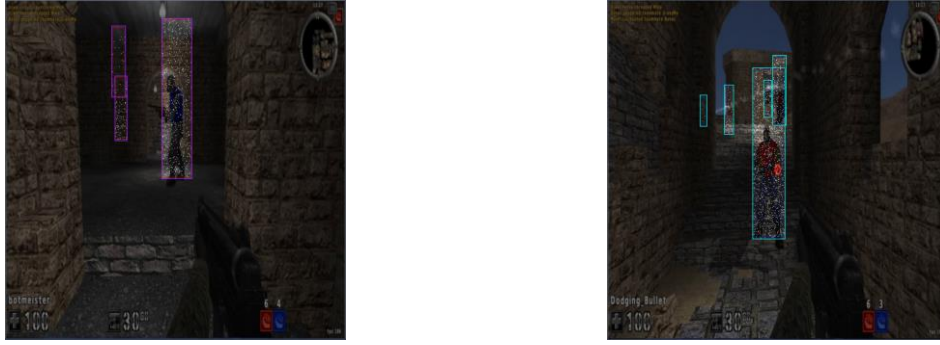


Figure 16: Sample of Adding Noise to Dataset

## 4.5 Model Training: YOLOv8

The YOLOv8 architecture was selected to develop the computer vision model for its state-of-the-art performance in real-time object detection tasks. As detecting visual cheating elements like ESP graphics requires accurately localizing multiple objects in real-time, YOLOv8 was an ideal fit. Its key advantages over alternative object detection models are:

- **Speed:** YOLOv8 is extremely fast, achieving higher FPS (frames per second) than other accurate detectors like RetinaNet (Li and Ren, 2019) or Mask R-CNN (Bharati and Pramanik, 2020), which enables real-time cheat analysis.
- **Accuracy:** Despite its high speed, YOLOv8 matches or exceeds state-of-the-art accuracy results as measured by common metrics like mAP. This ensures cheating cues are detected reliably.
- **Scalability:** The model design scales well to large datasets and transfer learning scenarios like our novel anti-cheat task. This adaptability aids quick training.

Other alternatives like Single-Shot Detectors (SSDs) were not selected due to marginally slower speeds while classic algorithms like R-CNNs were far too slow for real-time usage (Liu *et al.*, 2016). Thus, YOLOv8 struck the best balance of speed, accuracy, and adaptability for the anti-cheat system. The small trade-off in precision versus higher-precision models was acceptable for proof-of-concept purposes.

The annotated dataset was used to train a computer vision model using YOLOv8 (You Only Look Once, version 8). YOLOv8 was selected for its superior performance in real-time object detection tasks. The training process involved fine-tuning the model on the specific features of the ESP cheat in AssaultCube. This step required careful adjustment of model parameters to optimize for both accuracy and speed, ensuring that the system could operate effectively in real-time scenarios.

The first iteration of model training was run for 100 epochs and image size parameter set to 640. This was done to ensure a fast and efficient training on a basic consumer grade GPU that was used to train this model.

### 4.5.1 Confusion Matrix

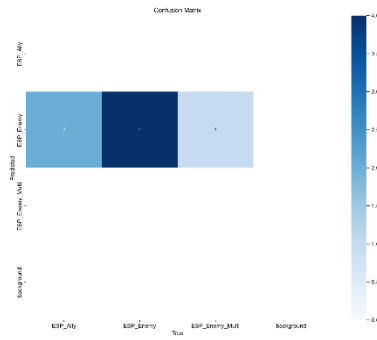


Figure 17: Confusion Matrix

- The x-axis represents the true labels (actual classification).
- The y-axis represents the predicted labels (model prediction).

Each cell in the matrix represents the count of predictions for a pair of true and predicted labels.

#### 4.5.1.1 Classes

- ESP\_Ally: ESP box that is detecting an Ally.
- ESP\_Enemy: ESP box that is detecting an Enemy.
- ESP\_Enemy\_Multi: A broader detection class where multiple ESP\_Enemy classifications are clustered together.
- background: All parts of the training image except the area bound by above classifications.

The confusion matrix suggests that ESP\_Enemy is being predicted relatively well, but there is confusion when distinguishing between ESP\_Ally and ESP\_Enemy, or when multiple enemies are present (ESP\_Enemy\_Multi). This can guide further refinement of our model; however, it is fundamentally justified for the current proof-of-concept presented in this paper.

### 4.5.2 F1-Confidence Curve

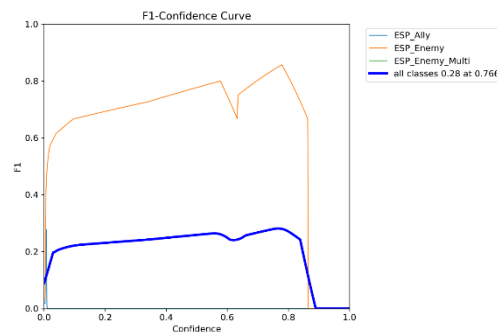


Figure 18: F1-Confidence Curve

The F1 score is a harmonic mean of precision and recall. It is a measure of a test's accuracy and considers both the false positives and the false negatives. The F1 score reaches its best value at 1 (perfect precision and recall) and worst at 0. The confidence threshold is the probability above which a prediction is considered positive by the model. Adjusting this



threshold affects both precision (the model's ability to avoid labeling a negative sample as positive) and recall (the model's ability to find all the positive samples).

- The blue line for **ESP\_Ally** shows a relatively stable but low F1 score across confidence thresholds, suggesting a consistent but potentially low precision and/or recall for this class.
- The orange line for **ESP\_Enemy** shows a high F1 score across a range of confidence thresholds, indicating that the model is more effective at identifying this class with both high precision and recall.
- The green line for **ESP\_Enemy\_Multi** is not visible in the graph, which means that its F1 score is consistently low or that the data for this class is not properly displayed.
- The aggregate line (labeled "all classes") shows the combined F1 score for all classes at different thresholds. The peak of this line indicates the best overall confidence threshold for the model, which in this case is around 0.28 with an F1 score of approximately 0.766.

Our priority in this proof-of-concept is to produce an AI model with the ability to output a yes or no if an ESP cheat is enabled which is viable even in the model's current accuracy.

### 4.5.3 Precision-Confidence Curve

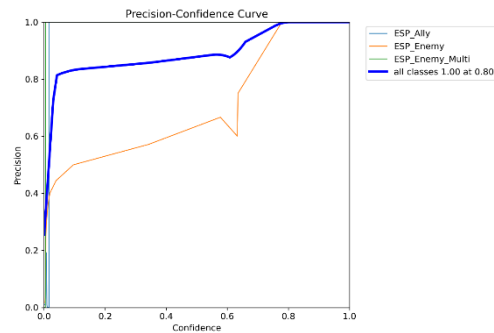


Figure 19: Precision-Confidence Curve

Precision is a metric that quantifies accuracy of the positive predictions made by the model. It is the ratio of true positive predictions to the total number of positive predictions (both true positives and false positives). A higher precision means that when the model predicts a class, it is more likely to be correct.

- The blue line for **ESP\_Ally** starts with high precision at low confidence thresholds but experiences a drop, which suggests that the model initially has high accuracy for this class, but as the confidence threshold increases, the model struggles to maintain precision.
- The orange line for **ESP\_Enemy** shows that the model maintains high precision across a wide range of confidence thresholds, indicating reliable performance for this class.
- The green line for **ESP\_Enemy\_Multi** appears to be mostly flat, which may indicate that there are not enough data points or variability in predictions for this class to assess a trend.
- The bold blue line going across the top of the graph and labeled "all classes" indicates the aggregate precision for all classes combined. This line shows a high precision level maintained across the confidence thresholds, peaking at a threshold of approximately 0.804, where it reaches a precision of 1.00, implying no false positives at this point.

The Precision-Confidence Curve is crucial for evaluating the trade-off between having a confident model and maintaining a high precision rate. For an anti-cheat system where it's important to minimize incorrect bans, a high precision is desirable, and this curve helps in determining the appropriate confidence threshold to achieve that.

#### 4.5.4 Precision-Recall Curve

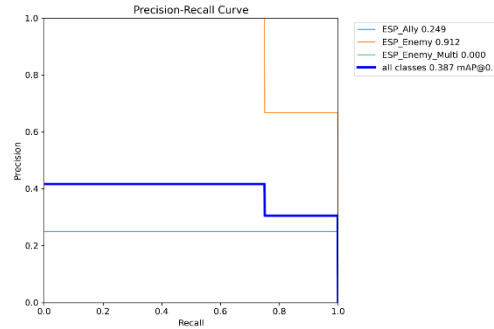


Figure 20: Precision-Recall Curve

Recall (also known as sensitivity or true positive rate) is defined as the number of true positives divided by the number of true positives plus the number of false negatives. It reflects the model's ability to detect all actual positives.

- The **blue line** for **ESP\_Ally** starts with a precision just above 0.2 for all recall levels, which is relatively low. This suggests that the model is not very precise in predicting **ESP\_Ally**; when it does predict **ESP\_Ally**, it is correct only about 20% of the time.
- The **orange line** for **ESP\_Enemy** indicates much higher precision at almost all levels of recall, starting at a precision of about 0.9. This suggests that the model is quite accurate in predicting **ESP\_Enemy** – it is correct 90% of the time when it predicts this class.
- The **green line** for **ESP\_Enemy\_Multi** is not visible, which typically indicates a precision and recall of zero. This suggests that the model was not able to identify any true positives for this class, or the class was not present in the test set.
- The **bold blue line** represents the model's overall precision-recall performance across all classes, with a mean Average Precision (mAP) at a particular IoU (Intersection over Union) threshold (often 0.5). The mAP for this model is 0.387, which is an aggregate measure over all classes.
- The annotations on the graph lines indicate the Average Precision (AP) score for each class, which summarizes the precision-recall curve as the weighted mean of precisions achieved at each threshold, with the increase in recall from the previous threshold used as the weight:
  - **ESP\_Ally**: AP = 0.249
  - **ESP\_Enemy**: AP = 0.912
  - **ESP\_Enemy\_Multi**: AP = 0.000
- The aggregate mAP score across all classes is given as 0.387 at an IoU threshold of 0.5, which is the standard for object detection tasks. This score reflects the overall performance across all classes and suggests that while the model is quite good at detecting **ESP\_Enemy**, it struggles with **ESP\_Ally** and practically fails at **ESP\_Enemy\_Multi**.

## 4.6 Testing and validation

To test our trained model, we recorded a session of AssaultCube with the ESP cheat turned on and then fed the recording to the model to see if it can detect the visual cues.

We can easily do that in Python using the OpenCV library to load the video and feed it frame by frame to our model and draw its findings on the screen frame by frame using the plot function.

As seen in the screenshots below, when loaded to be run on the CPU, the model works as expected and generates a frame-by-frame plot of detected classifications with the confidence of prediction alongside the labels.



Figure 21: ESP\_Energy Classification plot

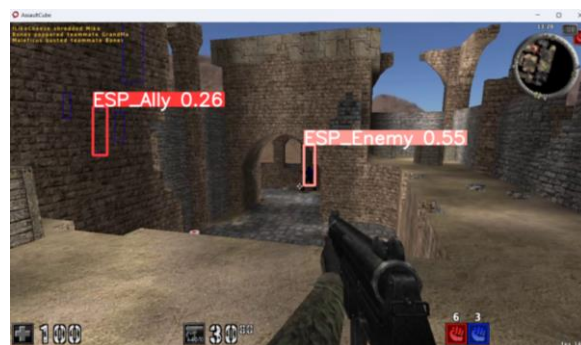


Figure 22: Mixed classification plot

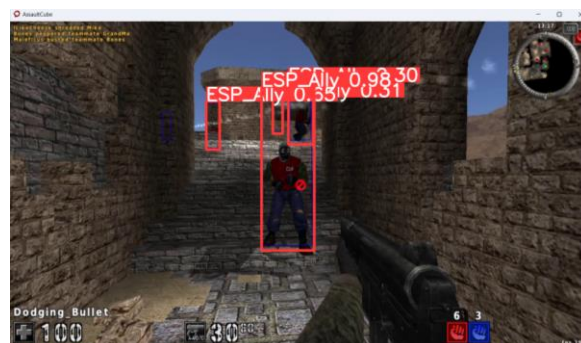


Figure 23: ESP\_Ally classification plot

Looking at the console output, we can see the time it takes per frame to complete a detection cycle.

We observe a 1.86ms preprocessing time on average followed by an average inference time of 42.62ms and finally an average 1.1ms of postprocessing time.

It is safe to say that the model takes less than 50ms of processing time per frame on a regular consumer grade computer with 16 Gigabytes of RAM and an 8-core processor.

```
0: 384x640 3 ESP_Enemy, 43.0ms
Speed: 2.3ms preprocess, 43.0ms inference, 1.0ms postprocess per image at shape (1, 3, 384, 640)

0: 384x640 2 ESP_Enemy, 42.0ms
Speed: 1.5ms preprocess, 42.0ms inference, 1.0ms postprocess per image at shape (1, 3, 384, 640)

0: 384x640 2 ESP_Enemy, 42.4ms
Speed: 1.0ms preprocess, 42.4ms inference, 1.2ms postprocess per image at shape (1, 3, 384, 640)

0: 384x640 2 ESP_Enemy, 41.6ms
Speed: 2.0ms preprocess, 41.6ms inference, 1.0ms postprocess per image at shape (1, 3, 384, 640)

0: 384x640 1 ESP_Enemy, 44.5ms
Speed: 2.0ms preprocess, 44.5ms inference, 1.4ms postprocess per image at shape (1, 3, 384, 640)

0: 384x640 2 ESP_Enemy, 42.2ms
Speed: 2.4ms preprocess, 42.2ms inference, 1.0ms postprocess per image at shape (1, 3, 384, 640)
```

Figure 24: CPU powered local execution

## 5 Implementation

The final implementation of the testing was conducted by running the model on an Nvidia GPU instead of a CPU as they are significantly more efficient at running machine learning models since they can utilize the power of CUDA engine which has highly optimized ML pipelines to speed up the inference.

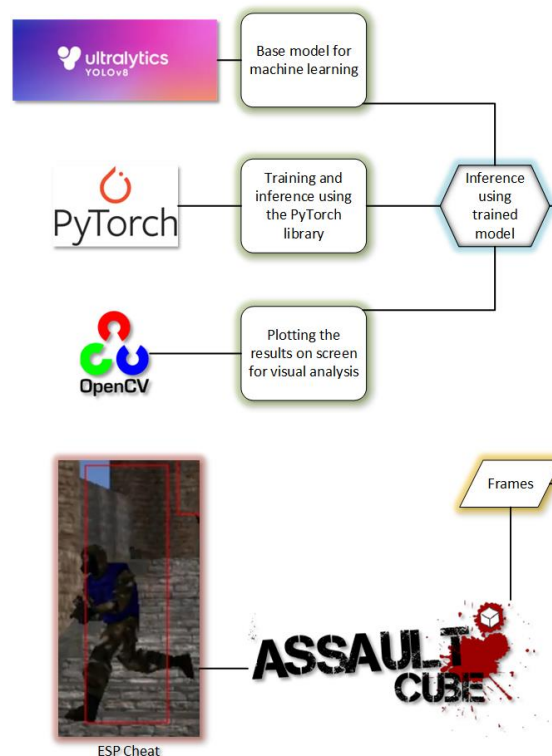


Figure 25: Basic implementation schematic

Now in the final implementation, the base YOLOv8 model was trained, and run on an Nvidia RTX 3060 GPU with 6 Gigabytes of VRAM. We used PyTorch library to feed the dataset to

the YOLOv8 model for training and inference and finally processed each frame in real-time by using display capture from OpenCV and running our model on each frame.

As shown in section 4.6 Testing and Validation, the detections were plotted on the screen frame-by-frame in real-time while the console log showed a significant improvement over the instance running on CPU.

The following console log was captured when running the model on a GPU.

```
0: 384x640 2 ESP_Enemys, 7.2ms
Speed: 1.0ms preprocess, 7.2ms inference, 3.0ms postprocess per image at shape (1, 3, 384, 640)

0: 384x640 2 ESP_Enemys, 7.0ms
Speed: 2.1ms preprocess, 7.0ms inference, 2.0ms postprocess per image at shape (1, 3, 384, 640)

0: 384x640 2 ESP_Enemys, 7.0ms
Speed: 3.0ms preprocess, 7.0ms inference, 1.0ms postprocess per image at shape (1, 3, 384, 640)

0: 384x640 2 ESP_Enemys, 7.9ms
Speed: 1.1ms preprocess, 7.9ms inference, 1.0ms postprocess per image at shape (1, 3, 384, 640)

0: 384x640 2 ESP_Enemys, 8.1ms
Speed: 1.0ms preprocess, 8.1ms inference, 2.0ms postprocess per image at shape (1, 3, 384, 640)

0: 384x640 4 ESP_Enemys, 7.0ms
Speed: 2.0ms preprocess, 7.0ms inference, 2.0ms postprocess per image at shape (1, 3, 384, 640)
```

Figure 26: GPU powered local execution

It can be observed that running the model on a GPU powered pipeline significantly reduces the amount of processing time taken per frame. In statistical terms, 1.7ms average preprocessing time, 7.37ms average inference time, and 1.83ms of average postprocessing time per frame.

In conclusion, we can expect an average processing time of less than 15ms per frame which is about 30% of the processing time we needed on a CPU.

The model is highly efficient and when deployed on a cloud service which can support multiple GPUs and much higher VRAM, this time can be virtually removed to achieve a real-time response.

```
--- 38.732967376708984 seconds ---
```

Figure 27: Local execution time on GPU


Further testing and statistical analysis revealed that when tested the trained model powered by a GPU (RTX 3060) on our test video which had 2428 frames captured at 60 fps, the model finished processing all the frames within 38.73 seconds which is slightly faster than real-time as the video duration was 40.46 seconds. Albeit with the same level of accuracy we observed in our past tests.

## 5.1 Cloud deployment

To test the deployment of our model on a cloud environment and compare the performance with the local instance we decided to use a private cloud service provider called RunPod.

The reason we chose this service over services like Google Collab was because in initial testing it was found that storage is relatively slow on a Google Collab instance and when our model accessed the video file from the instance storage provided on Google Collab it suffered from a large overhead in I/O. This caused the model to perform worse than the local instance which was unexpected but informative.

Moving on to deploying our model on RunPod, we used an instance equipped with an Nvidia RTX 4090 which is a state-of-the-art GPU with 24 Gigabytes of VRAM.



```
--- 33.38959527015686 seconds ---
```

Figure 28: Cloud execution time on GPU

On testing it was observed that the model completed execution on the same data input in about 33.4 seconds which is about 86.2% of the time taken by the local deployment.

While this improvement may not appear groundbreaking, it is certainly a move in the right direction and with further optimization of the execution pipeline this time could possibly be brought down further.

Ultimately, the complete processing of a 40 second video input, which is quite large compared to the real-life requirement of the model, we can conclude that this novel methodology can be reliably used to detect and ban cheaters in E-Sports without much delay.

## 6 Evaluation

### 6.1 Objective and Methodology

The primary objective of this research was to develop and assess a cloud-based computer vision system for real-time cheat detection in e-sports using YOLOv8. We employed statistical tools to evaluate the model's performance, focusing on accuracy, precision, recall, and processing times.

#### 6.1.1 Findings

- **Model Performance:** The model demonstrated high precision and recall rates, particularly for detecting ESP\_Enemy classification, with a mean Average Precision (mAP) of 0.387 at an IoU threshold of 0.5.
- **Processing Speed:** When deployed on a GPU (Nvidia RTX 3060), the model processed frames in less than 15ms on average. This increased efficiency was further enhanced in a cloud environment, where a 40-second video was processed in 33.4 seconds, faster than real-time.
- **Comparison with CPU Performance:** The GPU deployment showed a significant reduction in processing time per frame compared to the CPU deployment, highlighting the advantage of cloud-based deployment for real-time applications.

#### 6.1.2 Implications

- **Academic:** The research contributes to the field of AI in cybersecurity by demonstrating the effectiveness of vision AI in cheat detection, addressing a growing need in the e-sports industry.
- **Practical:** From a practical standpoint, the system's ability to process data quickly and accurately makes it a viable solution for live e-sports events, offering a non-invasive, privacy-preserving alternative to current anti-cheat methods.

## 7 Conclusion and Future Work

### 7.1 Recap

Our research aimed to answer how vision AI can be leveraged to develop a robust anti-cheat framework for competitive e-sports and enhance security and privacy in a high-performance cloud environment. The study successfully demonstrated the viability and effectiveness of a cloud-based anti-cheat system using YOLOv8 for real-time cheat detection.

### 7.2 Key Findings

- High accuracy in detecting specific types of cheat (ESP).
- The feasibility of real-time processing in a cloud-based environment.
- Significant improvement in processing times when using GPU compared to CPU.

### 7.3 Limitations

- The study focused on a single game environment (AssaultCube), which may limit the generalizability of the findings.
- The current model is only effective for ESP cheats which result in visual difference in the game. It is not trained to for other types of cheats, indicating a need for further refinement to add a full suite of detection abilities.

### 7.4 Future Work

- **Expanding Game Environments:** Testing the model across various game genres to validate its applicability in diverse gaming scenarios.
- **Model Enhancement:** Incorporating advanced AI techniques like deep learning or reinforcement learning to improve accuracy and adaptability.
- **Dataset Enrichment:** Acquiring more diverse and extensive training data to enhance the model's robustness and reduce classification errors.
- **Commercialization Potential:** Exploring partnerships with game developers for real-world implementation and further optimization for commercial use.
- **API Development:** After deployment on a cloud hosted environment, a REST API with secure access could be developed that will allow game developers to directly integrate the service into their code.

### 7.5 Closing Thoughts

This research marks a significant step towards developing effective, privacy-preserving cheat detection systems in e-sports, contributing to the integrity of competitive gaming, and offering insights for future advancements in the field.

## References

Abramov, A. (2022) *Creating an ESP: Introduction (1/4)* « RCE Endeavors. Available at: <https://www.codereversing.com/archives/529> (Accessed: 17 November 2023).

Bharati, P. and Pramanik, A. (2020) 'Deep Learning Techniques—R-CNN to Mask R-CNN: A Survey', *Advances in Intelligent Systems and Computing*, 999, pp. 657–668. Available at: [https://doi.org/10.1007/978-981-13-9042-5\\_56/FIGURES/6](https://doi.org/10.1007/978-981-13-9042-5_56/FIGURES/6).



Heijnen, E. (2008) *Cheat Engine*. Available at: <https://www.cheatengine.org/> (Accessed: 12 December 2023).

Karkallis, P. *et al.* (2021) ‘Detecting Video-Game Injectors Exchanged in Game Cheating Communities’, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 12972 LNCS, pp. 305–324. Available at: [https://doi.org/10.1007/978-3-030-88418-5\\_15/TABLES/5](https://doi.org/10.1007/978-3-030-88418-5_15/TABLES/5).

Kedziora, M. *et al.* (2020) ‘Anti-Cheat tool for detecting unauthorized user interference in the unity engine using blockchain’, *Lecture Notes on Data Engineering and Communications Technologies*, 40, pp. 191–209. Available at: [https://doi.org/10.1007/978-3-030-34706-2\\_10/TABLES/2](https://doi.org/10.1007/978-3-030-34706-2_10/TABLES/2).

Lehtonen, S. (2020) ‘Comparative Study of Anti-cheat Methods in Video Games’, *University of Helsinki* [Preprint]. Available at: <http://urn.fi/URN:NBN:fi:hulib-202003241639> (Accessed: 14 July 2023).

Li, Y. and Ren, F. (2019) ‘Light-Weight RetinaNet for Object Detection’. Available at: <https://arxiv.org/abs/1905.10011v1> (Accessed: 12 December 2023).

Liu, W. *et al.* (2016) ‘SSD: Single shot multibox detector’, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9905 LNCS, pp. 21–37. Available at: [https://doi.org/10.1007/978-3-319-46448-0\\_2/FIGURES/5](https://doi.org/10.1007/978-3-319-46448-0_2/FIGURES/5).

Maario, A. *et al.* (2021) ‘Redefining the Risks of Kernel-Level Anti-Cheat in Online Gaming’, *Proceedings of the 8th International Conference on Signal Processing and Integrated Networks, SPIN 2021*, pp. 676–680. Available at: <https://doi.org/10.1109/SPIN52536.2021.9566108>.

Rabid Viper Productions (2006) *AssaultCube*. Available at: <https://assault.cubers.net/> (Accessed: 11 December 2023).

Yan, J. and Randell, B. (2009) ‘An investigation of cheating in online games’, *IEEE Security and Privacy*, 7(3), pp. 37–44. Available at: <https://doi.org/10.1109/MSP.2009.60>.

Zhang, Q. (2021) ‘Improvement of Online Game Anti-Cheat System based on Deep Learning’, *Proceedings - 2021 2nd International Conference on Information Science and Education, ICISE-IE 2021*, pp. 652–655. Available at: <https://doi.org/10.1109/ICISE-IE53922.2021.00153>.