Behavioural Biometrics in Anti-cheat: Evaluating Angle-Based Mouse Dynamics for Anomaly Detection

Oliver Paynter-Jones

MComp (hons) Computer Science and Mathematics

2024–2025

Behavioural Biometrics in Anti-cheat: Evaluating Angle-Based Mouse Dynamics for Anomaly Detection

submitted by Oliver Paynter-Jones

Declaration

This Dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this Dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

University of Bath

***Abstract*—Competition is a fundamental element of most modern, successful video games, but cheating software presents a significant threat to competitive integrity. This paper explores the application of mouse dynamics analysis for cheat detection in Shooter video games. It proposes and evaluates a system, that incorporates the angle-based features of Zheng and Wang [1, 2] and utilises an ensemble of SVMs to discriminate between honest and cheating behaviour.**

Contents

Acknowledgements

My sincere thanks to **Dr. Moltisanti** for their insightful guidance and encouragement throughout this dissertation. Their advice was invaluable.

I am also grateful as ever to my parents for their unwavering support.

Finally, to my partner and close friends, thank you for your unwavering encouragement and belief in me throughout this journey.

Behavioural Biometrics in Anti-cheat: Evaluating Angle-Based Mouse Dynamics for Anomaly Detection

Oliver Paynter-Jones
Department of Computer Science
University of Bath
Claverton Down, Bath BA2 7AY
Email: opj23@bath.ac.uk

1. Introduction

The primary objective is to design and evaluate a mouse dynamics-based system for accurately detecting cheating in shooter video games. It begins with a discussion of impact and motivations of cheating in gaming, followed by an overview of the cheats and their construction. It then explores the strengths and weaknesses of the anti-cheat systems used by game providers, as well as recent literature investigating the more novel solutions for cheat detection. Following this, a system for detecting cheating in shooter video games, leveraging angle-based features from the field of mouse dynamics, will be proposed, culminating in an evaluation of this system and its abilities. By doing this, I aim to contribute to the knowledge base supporting the development of improved anti-cheat systems that enhance player welfare and support game revenue for developers, as well as our current understanding as to how we play video games.

The evaluation revealed that the system, consisting of data pre-processing with action segmentation, angle-based metrics calculation and feature generation, and an ensemble of One-Class Support Vector Machines (OC-SVMs), could effectively identify anomalous cheating behaviours, including subtle aimbotting. Furthermore, a novel proposition emerged from the results regarding the way users interact with video games. Further research is needed evaluate its performance across different users, games, and cheating software; nonetheless, the approach proved to be data-efficient and resistant to changes in hardware setup and cheat configuration, indicating promising potential for real-world deployment with further refinement.

2. Cheating

2.1. Impact of Cheating Software

Cheating in multiplayer games significantly undermines player satisfaction and game revenue. Surveys indicate that a large majority of players would abandon a game if they suspected widespread cheating, with many reporting that it has "completely ruined" their experience [3, 4]. This is particularly impactful in games employing Engagement Optimised Matchmaking (EOMM), where balanced matches are crucial for player retention [5]; cheating forces players into imbalanced and frustrating encounters, especially given penalties for leaving matches prematurely.

In the current oligopolistic and live-service dominated gaming market [6], where sustained revenue relies on player engagement through in-game purchases (a model used by 60% of developers [7]), cheating is a critical threat. When players are driven away by unfair play, the loss of potential long-term revenue can be irreversible due to the abundance of alternative games. The substantial revenue generated from in-game transactions, exemplified by Counter-Strike 2's hundreds of millions in cosmetic sales [8], highlights the financial stakes. Legal actions by developers like Bungie against cheat creators, resulting in significant damages (e.g., $12 million [9]), further underscore the substantial financial and reputational harm caused by cheating, with Bungie explicitly linking anti-cheat efforts to both community well-being and sound business practice [10]. A more detailed discussion can be found in Appendix A.

2.2. Motivation

Aimware, a cheat software developer for ten games, operates on a monthly subscription model. With over 600,000 registered users [11], even a conservative 1% subscription rate at 19.99 CHF monthly yields an estimated revenue of roughly 120,000 CHF (105,653.40 GBP). Given Counter-Strike 2's large player base (over 30 million monthly active users [12]) and survey data indicating significant past and present cheating activity (43% and 12% of the population respectively [3]), this revenue estimate is likely low. The financial incentive for cheat development is substantial and grows with the expanding online gaming market.

3. Aimbots: Behaviour, Impact, and Visual Detectability

Different types of cheating software are designed to enhance various aspects of player performance. Certain cheats provide a greater advantage in specific genres, making them more commonly used in those contexts.

This research focuses exclusively on Augmented Aim cheats, specifically aimbots, which most commonly effect Shooter video games. Subsequent discussion will make minimal reference to other cheat types, although many exist and are discussed in Appendix B. Therefore, any mention of cheating software from this point onwards should be understood to refer solely to the aimbot type of Augmented Aim cheats.

Augmented Aim Cheats (aimbot, TriggerBot): These cheats enhance or automating the aiming and firing process, providing an unfair advantage. Players must typically adjust their in-game view using their mouse to align their weapon with a target before firing. This process can be complicated by addition mechanics, including bullet spread, projectile travel time, and weapon recoil.

Augmented aim cheats modify one or more aspects of this process. aimbots, for instance, might fully automate engagements—from target acquisition and aim adjustment to firing and compensating for weapon recoil. In contrast, TriggerBot cheats solely automate firing, requiring the player to manually aim but ensuring that shots are taken at the optimal moment. Many of these cheats include adjustable parameters, such as smoothing functions or speed modifiers, allowing users to make their aiming adjustments appear more human-like. As a result, they can be configured to be relatively visually subtle.

3.0.1. Mechanisms of Aimbots. Cheat software complexity varies, with sophisticated cheating-as-a-service (CaaS) offerings, often developed by teams, exceeding the capabilities of simpler, home-made cheats [9, 10]. Regardless of complexity, most cheats rely on reverse engineering. Developers use techniques like de-compilation and dynamic analysis to understand a game's data handling and processes. This knowledge enables the creation of software that grants unfair advantages through methods such as active memory manipulation or code manipulation.

For example, constructing a rudimentary aimbot may involve the following steps:

- Identify the game process using Operating System (OS) API calls.
- Retrieve the base memory address of the game process.
- Use identified memory address offsets to locate entity positions in memory and extract the cheater's current view angle.
- Calculate the necessary adjustments to the view angle to aim at the target entity.
- Emulate mouse movement to align the crosshair with the target.
- Fire the weapon.

More sophisticated aimbots incorporate additional features to evade detection (visual, software signature) and enhance effectiveness, such as:

- *Adjustable Parameters:* Allows cheaters to fine-tune cheat severity, such as the distance which is required for an aimbot to activate.
- *Smoothing Functions:* Apply artificial alterations gradually to appear more human-like rather.
- *Deliberate Imperfections:* Introduce minor errors (e.g., slight over- or undershooting, timing delays) to mimic human behaviour.
- *Code and Memory Obfuscation:* Techniques such as polymorphism alter the structure of cheat code to evade signature-based detection, while memory obfuscation minimises the cheat's footprint in system memory.
- *Kernel-Level Exploits:* Some cheats operate at the system's highest privilege level by exploiting driver vulnerabilities. This allows them to bypass most memory scans, more easily manipulate memory, and conceal their presence through more advanced evasion tactics.

The battle between cheat developers and anti-cheat systems is a continuous arms race that forces the iteration on cheating software. As one vulnerability is discovered and exploited, it quickly spreads across various cheats before developers patch it. In response, new workarounds and exploits emerge and are implemented into software. Game developers invest heavily in anti-cheat measures, but cheat developers adapt just as quickly.

4. Background and Literature Review

4.1. Existing Anti-cheat System

Game developers use multiple layers of anti-cheat measures to maximise detection rates while maintaining scalability, security, and minimising false positives. The most common approaches to combating cheating involve a combination of automated detection software and manual processes.

4.2. Automated Anti-cheat Software

The first layer of defence in most games comes from automated commercial anti-cheat programs, which are bundled with the game and required for online play. Some are third-party solutions developed by specialised developers, while others are proprietary systems built in-house by game developers.

Most automated anti-cheat programs focus on preventing cheating through various methods, such as:

- *Memory Scanning:* Detecting known cheat signatures and patterns, as well as attempts to modify memory allotted to the game.
- *Code Integrity Checks:* Verifying game file integrity using checksums or other validation methods to detect unauthorised modifications.
- *Process Monitoring:* Identifying suspicious processes running alongside the game, such as those attempting to inject code or overlay graphics on top of the game.

- *API Hooking Detection:* Flagging attempts to intercept and manipulate function calls within the game.

Some advanced automated anti-cheat systems operate at the kernel level, running from system startup to provide deeper monitoring and detection capabilities.

If an automated anti-cheat system detects an obvious violation—such as a known cheat program running or direct memory tampering—it may issue an automatic ban. However, in borderline cases where only suspicious behaviour is detected, the system may flag the player for further investigation rather than immediately banning them. Many commercial anti-cheat systems work in conjunction with manual review processes to handle these scenarios.

The remaining elements of the anti-cheat system stack are more thoroughly evaluated in Appendix C.

4.3. Prevalent Issues

While professional cheat developers have substantial financial motivation to circumvent automated anti-cheat software using sophisticated techniques like kernel-level drivers, the widespread availability of reverse-engineered methods online [13] means that even hobbyist programmers with enough perseverance can relatively easily bypass these systems. Security through obscurity offers only a temporary delay, as information on preventative measures rapidly becomes accessible, leading to swift updates in cheating software.

Automated anti-cheat software also introduces system vulnerabilities due to its often necessary "Ring 0" or kernel-level access. Kernel crashes can lead to total system failure, and the unconstrained nature of such software prevents user restriction, raising security concerns. Past controversies, such as E-Sports Entertainment's deployment of software that illicitly mined bitcoin by exploiting kernel-level access, underscore these risks [14]. This incident, detailed further in Appendix D, highlights how even purportedly legitimate kernel access can be abused for severe privacy violations, unauthorised resource use, and security breaches.

Finally, stagnant software quickly becomes overwhelmed. Such software requires a dedicated, trusted team to maintain it. This can be costly and time-consuming.

Ultimately, while the performance of automated anti-cheat software could potentially be increased, the potential for mistakenly banning innocent players (false positives) acts as a significant constraint on aggregate anti-cheat system performance. Commercial anti-cheat software is already operating at a level where pushing them further introduces a considerable risk of misidentifying and penalising honest players. Additional manual layers in system stack are used to fill the gaps, ultimately pushing the onus of heightened demand for anti-cheat efforts onto players and dedicated teams.

The cost of maintaining these dedicated teams, software, and game-security remains high. The components of an anti-cheat system must sum to a scalable, automated, and highly accurate process. The key metrics for success are true positive rates (catching real cheaters) and false positive rates (avoiding innocent bans). Future advancements should aim to improve automated detection efficacy and efficiency to minimise manual intervention.

5. Primary Research Objective

As such, the primary objective of this research is to contribute to the ongoing efforts against cheating in video games by designing and evaluating an automated system for detecting Aim Augmentation software in shooter video games. This system is intended to complement existing anti-cheat solutions by identifying cheaters who evade current detection methods. By expanding the set of tools available to developers, this work aims to support the promotion of player welfare, preserve competitive integrity, and help reduce the operational costs associated with maintaining fair online gaming environments.

6. Literature Review

Aimbots cheats manipulate the mechanics of a players aim. Detecting this category of cheat is fundamentally an authentication problem—one that focuses on identifying how a player uses their mouse. If the goal is to develop an automated detection system, the central question becomes: who is in control of the mouse—and therefore the in-game aim—a human player or cheating software?

As such, an exploration of existing literature on automated mouse-based authentication systems is essential, though other approaches to aimbot detection—including statistical and visual methods—will also be considered. The more established field of mouse dynamics offers transferable insights that can inform and strengthen the design of a aimbot cheat detection system.

6.1. Mouse Dynamics

6.1.1. Overview. Mice, despite their simple design of typically only 2 buttons, auxiliary navigation buttons, and a scroll wheel, are crucial human-computer interfaces in shooter video games. Atomic mouse movement data is represented as a tuple, (x_t, y_t), where x_t and y_t denote the x and y coordinates of the cursor in the digital system at timestamp t. Information is typically sampled at a regular frequency. Additional information, including mouse button press and release events, can also be incorporated.

From this raw data, a variety of features can be derived. Features include metrics such as speed, acceleration, and jerk. For instance, the magnitude of the mouse speed ($\mathbf{v}_t$) at time t can be calculated as:

$$\mathbf{v}_t = \sqrt{\left(\frac{dx_t}{dt}\right)^2 + \left(\frac{dy_t}{dt}\right)^2} \tag{1}$$

Mouse dynamics analyses unique patterns in mouse movements, similar to handwriting, for biometric authentication. By identifying and measuring specific movement

features, a user's unique signature can be established to verify their identity and detect different users.

6.1.2. Existing Work. The study of mouse dynamics is a relatively new area, developing in parallel with the widespread use of personal computers equipped with mice. The progress of this field has been limited predominantly by a relative lack of large, versatile, publicly accessible datasets [15, 16]. One study suggested an average of 60 minutes of user mouse movement data was for the evaluated systems designs to produce an acceptable performance [16] - Datasets containing such substantial pools of information have been rare.

The experiment design used for initial data collection significantly undermines the usability of any released dataset for subsequent research also. It's not yet been established when and how the specific tasks used in research, such as specialised memory exercises or "click-the-target" games, influence the way people use their mouse. This limits the dataset's overall versatility for broader research. In some instances, researchers have only shared processed data, again limiting opportunities for further experimentation, analysis, and validation. In other cases, datasets have remained completely unpublished.

Recently, there has been a concerted effort to increase the availability of mouse dynamics datasets, which is illustrated by the inclusion of mouse dynamics challenges in data science competitions, leading to the first specifically designed, publicly available mouse movement dataset [17, 18]. This dataset has encouraged the release of additional valuable datasets, with allows for more thorough evaluation of research [16, 19, 20].

Despite the challenges of a fledging field of research, a considerable body of research has demonstrated the utility of mouse data. A comprehensive survey of research published up to 2023 compared various experimental authentication systems by discussing and comparing the features employed and system architectures [15]. A comparative overview of these systems is presented in Table 1.

A significant portion of research is still focusing on exploring entirely novel approaches [15, 16], including the development of new features, classifiers, architectures, or combinations thereof. Khan et al. provide a comprehensive overview of the feature pool explored in existing research [15, p. 14]. The most distinguishing difference between features are the differing stated quantities and types of data that were required for effective performance [15]. Drawing firm conclusions on efficiency and performance is difficult due to the use of unique, private datasets in research, along with differing experimental designs. As such there is no consensus regarding which derived feature(s) are superior, however some work has been done to test systems and features in a standardised environment [21].

For example, Shen et al. [22] utilised general mouse movement data from 159 participants, extracting basic features like speed and timing for an SVM classifier, achieving low error rates (0.09% FAR, 1% FRR). Conversely, Bours & Fullu [23] analysed velocity along predefined paths from a specific maze task completed by only 28 participants (840 runs), using the complicated Edit Distance metric. Their approach yielded significantly higher error rates (27-40% EER), with the issues attributes to data variability and challenges in track detection, illustrating how differing methodologies and datasets hinder direct comparison.

Exacerbating the lack of clarity, positive results from specific controlled experiments may not reliably predict performance in diverse real-world scenarios with varied user setups. Kuric et al. reinforce this notion when they discuss the influence of configuration changes on mouse movement data and the resulting mouse dynamics [24]. It is not uncommon for users to modify parameters like pointer speed via mouse buttons or OS settings, and more advanced users might even change polling rates. Moreover, screen size and resolution often vary significantly between individuals. The authors of the survey indicated that, out of 108 mouse dynamics metrics identified in the literature, 95 and 84 were impacted by adjustable mouse parameters across two experiments they conducted [24].

A notable contribution to this field comes from the work of Zheng and Wang, who proposed a new set of features and a system design for an efficient authentication method [1, 2]:

The researchers monitored users' mouse movements and collected three datasets: two from controlled laboratory settings with 30 and 94 participants, respectively, and one from an online forum involving 1,074 participants. They then extracted three detailed angle-based metrics—direction, angle of curvature, and curvature-distance—to characterise the unique patterns in a user's mouse movements.

Their statistical analysis of their data supports other research in suggesting that typical speed-related features could characterise user signatures but were effected by configuration changes [24]. Their novel feature set demonstrated greater resilience to these changes under similar scrutiny.

The mouse movement data from a user was segmented into actions, and the metrics were computed for each action. Following common practice in mouse dynamics for noise reduction, the authors defined a "block" as a collection of these mouse actions across which statistical features were calculated for each of their three novel metrics to model the distribution of values. For classification, an ensemble of Support Vector Machines (SVMs), which have shown reliability in the domain [15, 21], were trained on these block-distributions, with the final classification determined by aggregating the individual classifier outputs.

System's performance was rigorously evaluated in terms of verification accuracy, verification time, and system overhead. Verification accuracy was assessed using the false reject rate (FRR) and the false accept rate (FAR). The results indicated that the system could verify a user with high accuracy using only a small number of clicks. With an optimal block size of 25 actions, the average false reject rate was 0.86%, and the average false accept rate was 2.96%.

Table 1: Summary of mouse dynamics user authentication systems. Type of mouse action: Mouse Movement (MM) - a continuous sequence of mouse events; Drag and Drop (DD) - a MM action starting with button press and ending with a release; Point and Click (PC) - a MM action ending with a consecutive button press and release. **Additional Abbreviations:** DM - Distance Metric, SD - Standard Deviation, FAR - False Acceptance Rate, FRR - False Rejection Rate, EDD - Elliptic Data Description, EER - Equal Error Rate, WBD - Within-Bag Distance, std - standard deviation, ANGA - Authentication Normalised Genuine Acceptance, NA - Not Applicable, C - Clicks, S - Scrolls, KNN - k-Nearest Neighbors, NN - Neural Network, SVM - Support Vector Machine, MD - Mahalanobis Distance, CV - Cross-Validation, ACC - Accuracy, AUC - Area Under the ROC Curve, NB - Naive Bayes, GBM - Gradient Boosting Machine, MLP - Multi-Layer Perceptron, CNN - Convolutional Neural Network. Adapted from [15, p. 17].

Authors	**User Tasks**	**Subjects**	**Amount of Data**	**Classifier**	**Performance Metrics**	**Training and Testing**
Revett et al.	Fixed Static Sequence of Actions	6	Click Duration (100 samples)	DM based on +/- 1.5 SD	FAR (1-4%) and FRR (1-3%)	80/20 split
Bours and Fullu	App Restricted Continuous	28	MM (Avg 45 sessions per user)	EDD	EER (40.1%)	50/50 split
Zheng, Paloski and Wang	Completely Free and App Agnostic Continuous	30 and 1k	81,218 PC actions (5801/user avg)	SVM	FRR (0.86%), FAR (2.96%) in 25 clicks	50/50 split
Gamboa and Fred	App Restricted Continuous	25	MM (5h interaction/180 strokes per user)	WBD	Mean EER (0.005) with std (0.001) for 100 strokes	50/50 split
Mondal and Bours	Completely Free	49	MM, PC, DD (5000 samples)	SVM	ANGA (NA), ANGA (94)	50/50 split
Shen, Cai and Guan	App Agnostic Semi-Controlled	28	MM, PC, DD, C (90k mouse actions over 30 sessions)	1-class SVM	FAR (0.37%), FRR (1.12%)	1-Class Classification
Shen et al.	Fixed Static Sequence of Actions	26	MM and PC (300 samples per user)	KNN, NN, SVM	EER (2.64% in 110 sec)	1-Class Classification
Shen et al.	Fixed Static Sequence of Actions	37	MM and PC (5550 samples)	1-class SVM	FAR (8.74%), FRR (7.96%) in 11.8 sec	1-Class Classification
Shen et al.	Fixed Static Sequence of Actions	58	MM and PC (17.4k samples per user)	KNN with MD & ED, 1-class SVM, K-means, MD	EER (5.68%) with std (4.12)	1-Class Classification
Shen et al.	Completely Free	159	MM, PC, DD, C (1.5m mouse operations)	1-class SVM, KNN, NN	FAR (0.09%), FRR (1%)	1-Class Classification
Shen et al.	Completely Free	20	MM, PC, DD, C, S (600 sessions)	SVM, NN	FAR (1.86%), FRR (3.46%)	50/50 split
Ma et al.	App Restricted Continuous	10	MM and C (500 sessions)	SVM	Accuracy (96.3%), FAR (1.98%), FRR (2.10%)	5-fold CV
Kaixin et al.	App Agnostic Semi-Controlled	12	MM, DD, PC, S (1000 sessions)	SVM	FAR (8.8%), FRR (5.5%) in 30 sec	70/30 split

Authors	User Tasks	Subjects	Amount of Data	Classifier	Performance Metrics	Training and Testing
Dominik et al.	App Agnostic Semi-controlled	11	MM, C, S (3283 instances)	LibSVM, ANN, DT, RF	Avg Accuracy Rate (78.1%)	10-fold CV
Almalki, Roy and Chatterjee	Completely Free	10 (Balabit)	MM, DD, PC (937 actions avg, 65 sessions)	DT, KNN, RF	ACC (99.3%), AUC (99.9%)	70/30 split
Tan and Roy	Completely Free	10 (Balabit)	MM (937 actions avg, 65 sessions)	Linear SVM	Avg EER (0.1829), AUC (0.86), FAR (0.21), FRR (0.0975)	5-fold CV
Antal and Egyed-Zsigmond	Completely Free	10 (Balabit)	MM, DD, PC (937 actions avg)	RF	Avg EER (18.80%), AUC (89.94%)	10-fold CV
Salman and Hameed	Completely Free	48	MM, DD, PC (998 sessions)	Gaussian NB	ACC (93.56%), FRR (0.822), FAR (0.009), EER (0.08), AUC (0.98)	3-fold CV
Hu et al.	Completely Free	24	PC (5000 samples per user)	RF, GBM, MLP, SVM, CNN	FRR & FAR (both $<$1%) (except CNN)	70/30 split
Aksari and Artuner	Fixed Static Sequence of Actions	10	MM (111 sessions)	DM based on +/- 1.5 SD	FAR (5.9%), FRR (5.9%), EER (5.9%)	90/10 split
Gao et al.	Completely Free	10 (Balabit)	MM (sample size not mentioned)	SVM, KNN	FAR (0.075), FRR (0.0664)	Unknown
Zheng, Paloski and Wang	Completely Free	30	MM and PC (3160 mouse actions per user)	SVM	FAR (0.86%), FRR (2.96%) after 25 clicks	50/50 split
Antal, Fejer and Buza	Fixed Static Sequence of Actions	120	MM (52 blocks per user)	CNN and 1-class SVM	AUC (0.94)	75/25 split
Pusara and Broadley	App Restricted Continuous	18	7.6k unique cursor locations	DT	Avg FAR (0.43%), FRR (1.75%)	75/25 split
Antal and Denes-Fazakas	Completely Free/App Agnostic Semi-controlled	21	MM, DD, PC (1k mouse actions)	RF	Avg AUC (0.9922) with std (0.0061)	70/30 split
Jorgensen and Yu	App Restricted Continuous	17	MM, DD, PC (325 actions per user)	NN, Logistic Regression	Avg FAR (21%), FRR (21.5%) [37]; FAR (30.3%), FRR (37.1%) [5]	60/40 split

These features also proved to be highly efficient, requiring a very small amount of less data to achieve results comparable to or better than other systems that have been tested on similar datasets (Table 1) while maintaining resilience to configuration changes [15].

To summarise, a variety of systems have shown that mouse movement data contains considerable identifying information and can be effectively used for user authentication. Despite this, performance for these systems has not yet consistently met the standards required of other biometric security systems by the EU [25]. Furthermore, many methods have not been evaluated in practical applications. Consequently, current mouse dynamics authentication systems are predominantly reserved for further investigation.

6.1.3. Applications in gaming. The application of mouse dynamics in this environment presents a distinct challenge due to the dynamic yet task-specific nature of video games. Gaming scenarios can range considerably, from 2D strategy games with repetitive on-screen button actions to 3D shooters with unrestricted mouse movement. Gamers are also more likely to adjust mouse sensitivity, screen resolution, and may even switch mice entirely during gameplay, disturbing any environmental consistency in data collection scenarios. As such, gaming tests the robustness of mouse dynamics systems and limited research has attempted to explore and address these gaming-related challenges.

Siddiqui et al. presented the first, and as of this writing, only publicly available gaming mouse movement dataset – a Minecraft mouse dynamics dataset – and reproduced the work performed on the general-use Balabit Dataset [17, 19, 26]. They collected 20 minutes of gameplay data from 10 users, segmenting it into mouse actions which were defined by 10 consecutive recorded mouse movement entries. Simple features (velocity, acceleration, direction) were extracted. The dataset was split into training and testing sets, creating user-specific datasets with equal genuine and imposter mouse actions. The goal was to train classifiers to distinguish between genuine and imposter users.

The research utilises binary Random Forest (RF) classifiers and performance was evaluated in two scenarios with differing results. In the first scenario, where the training dataset was used for both training and testing, the average accuracy rate was 92.73%, with an average false positive rate (FPR) of 14.39% and an average equal error rate (EER) of 0.1832%, suggesting that the model has the capacity to fit to the training data. However, in the second scenario, where classifiers were tested on unseen data, the average accuracy dropped to 61.60%, with an average FPR of 21.10%, a FNR of 64.80%, and average EER of 39.67%. The significant drop in performance between scenarios implies that the system may have have limitations in generalising to unseen, real-world mouse movement patterns. Overall, it's unclear as to whether the model is ineffective or simply requires more refinement. The substantial difference in observed results between this study compared to the motivating literature [19] potentially indicates that gaming environments introduce a greater level of complexity compared to general-use datasets such as the Balabit dataset.

Alkhalifa investigated cheat detection in CS:GO by classifying players as either cheaters or fair players based on gameplay data extracted from recorded game files. Recognising the absence of a public repository for cheating gameplay, the researcher generated their own dataset by recording gameplay using a purchased cheating suite. To perform classification, Hidden Markov Models (HMMs) were employed to analyze demo files using 20 defined features, which included player intersections based on view-angles. However, due to computational costs associated with their derivation, the final evaluation of the system focused on a subset of these features, specifically mouse movement speeds and view angle changes.

The evaluation of the system primarily focused on mouse movement speeds and view angle changes, showing promising results in identifying unnatural movements indicative of aimbots through their "View Difference Speed Model". While the "Signature Detection" model provided only weak evidence for detecting individual player mouse signatures, the system successfully identified recoil control cheats by recognising their consistent, repeated patterns in angle changes, a stark contrast to the varied recoil compensation of fair players. However, the system encountered difficulties in reliably distinguishing smoothed forms of cheating from the aiming skills of legitimate players. Finally, Alkhalifa emphasized the critical role of robust feature extraction methods in the context of video game data, highlighting potential discrepancies that can arise between server-recorded data and individual player perception.

Alkhalifa's work also emphasises robust feature extraction methods: there are unique complications associated with data collection from video games, for example, discrepancies might exist between server-recorded data and player perception.

Spijkerman and Ehlers took a statistical approach to mouse-dynamics based cheat detection in Counter-Strike: Global Offensive also [27]. They collected data from a single player playing offline matches against bots, capturing mouse movements and clicks, keystrokes, and in-game console logs, which included statistics like hits, damage, and targets. The captured data underwent pre-processing, including formatting, noise removal, and calculation of event timings and counts.

The authors then extracted relevant features from this data. Statistical features were derived from mouse movement data, such as the average time and distance of click-and-drag events, the approximate direction, and the average speed of movement. These features were then used to train three classifiers: Decision Trees, SVM's, and Naïve Bayes.

The results (Table 2) indicated that decision trees, particularly when pruned, achieved high accuracy in classifying honest behaviour using mouse dynamics data. While SVM performance varied with feature selection, ensemble approaches offered slight improvements for mouse dynamics. Multidimensional SVMs performed less well, possibly due to feature separability. A Naïve Bayes classifier applied to

Table 2: Performance Comparison in Terms of Accuracy of Different Methods and Feature Sets. Adapted from [27, p. 91].

Method	Keystroke Dynamics	Mouse Dynamics	Console Logs
Decision Tree			
Not Pruned	95.45%	58.33%	66%
Pruned	83.33%	100%	-
Feature Pair SVMs			
Highest Acc.	100%	75%	83.33%
Lowest Acc.	54.54%	50%	50%
Ensemble Learning (Feature Pair SVMs)			
Highest Acc.	90%	77.77%	83.33%
Lowest Acc.	100%	77.77%	83.33%
Multidimensional SVMs			
Acc.	73%	77%	83%

console logs reached 87.5% accuracy for detecting cheating. The authors concluded that AI-driven cheat detection is promising and warrants further investigation with larger datasets and across different game genres.

This research shows that various approaches to detecting cheating are possible and that the statistics of simple mouse dynamics features alone can act as an effective indicator of cheating in video games. However, the evaluation results did not address deeper performance metrics, which are crucial for any authentication system.

Willman explored the feasibility of using a Long Short-Term Memory (LSTM) recurrent neural network to detect aimbots from aim related data in Counter-Strike: Global Offensive [28]. The reseacher constructed an artificial server populated by bots, with himself as the sole human player utilising a free aimbot. 122 rounds were deemed suitable for analysis due to input data dimensionality constraints imposed by the model, with aimbotting present in approximately 70% of the recorded rounds. 510 training sequences extracted from the games and a 70/30 train-test split was employed. Notably, doubling the input data size (timeframe) led to a reported increase in accuracy from 50% to 70%. However, given the pre-existing 70:30 split of aimbots to non-aimbots data and a lack of other supporting metrics, this accuracy score is undermined. Similarly to the work of Alkhalifa, the author also acknowledges limitations stemming from data flaws due to logging interference with gameplay and reiterates the need for a larger, more authentic and balanced dataset.

Moving away from mouse dynamics related anti-cheat systems, Zhang et al. introduce HAWK, a server-side anti-cheat framework for CS:GO, designed to address limitations in existing anti-cheat solutions. HAWK uses machine learning to mimic human expert cheat detection, operating on multi-view features extracted from gameplay data. The evaluation of HAWK was conducted on real-world CS:GO datasets, which are notable for their scale. Specifically, the dataset includes 2,979 instances of aimbotting and 2,971 instances of ESP hacking, encompassing a total of 56,041 players.

HAWK achieved a maximum recall of 84% and accuracy of 80% in detecting both aimbots and ESP hacks. However, the authors also highlight limitations concerning the generalisability. While effective in CS:GO, the framework may require adaptation for optimal performance in other FPS games.

Yeung et al. presented a method for detecting cheating in multiplayer games, specifically aimbots in CS:GO, using a Dynamic Bayesian Network (DBN) [29]. This DBN models the relationships between several factors influencing aiming, including whether a player is cheating, their movement, the target's movement, changes in aiming direction, the distance to the target, and aiming accuracy. The goal is to identify cheaters by recognising unusual aiming patterns compared to legitimate players.

The DBN is trained using gameplay data from both honest and cheating players to calculate probabilities. For instance, it learns the likelihood of cheating at a given time based on past cheating behaviour, as well as the probability of achieving a certain aiming accuracy given other variables. During gameplay, the trained DBN infers the probability of a player cheating at each moment. It uses past behaviour to predict the likelihood of cheating and updates this prediction with current aiming data, resulting in a probability score that indicates the likelihood of aimbot use.

The results demonstrated the method's effectiveness. Honest players showed consistently low cheating probabilities, while cheaters' probabilities fluctuated above a threshold. The system also detected advanced aimbots and proved robust across different training/testing data, suggesting that some systems can address more advanced cheating techniques.

Image recognition has also been applied to the problem of cheating by Zhang in a short article that explored deep learning approaches for online game anti-cheat systems, focusing on binary classification models (CNNs) and one-class SVM's [20].

While acknowledging SVM relevance, the cited lack of accessible real-game data hindered their use. Instead, the authors proposed a CNN-based image recognition system to predict "reasonable" player actions with deviations flagging potential cheating. Using 300 labelled images, they

achieved 0.8-0.9 accuracy on most test images. However, object recognition limitations affected performance with small or low-resolution objects, like distant players. The authors recommended a larger, more diverse training dataset for better accuracy, and the definition of accuracy in this context remains unclear.

This article demonstrates the potential of image recognition for cheat detection, but it also underscores some clear limitations of the technology compared to a data-based approach. Visual approaches will always have to contend with visual fidelity, as well as issues relating to the size of the data, which will be larger if stored as video.

Finally, Valve, a games, hardware and software developer, and owner of some of the largest video games in history, produced the VACnet system. It was the first to employ machine learning methods to create an anti-cheat. Specifically, it relies on deep learning for cheat detection and leverages the vast collection of gameplay data generated by its user base (over 30 million monthly players [12]) to identify cheating behaviour alongside the standard VAC anti-cheat [30]. It is used to detect suspected cheating, before raising tickets for further validation.

While this approach was described as effective [30], specific details regarding long-term deployment and performance remain largely undisclosed. It has been disabled multiple times, but Valve is currently trialling VACnet 3.0, suggesting ongoing efforts to refine and improve the system.

Despite the potential of machine learning in cheat detection, to my knowledge VACnet is the only instance of widespread adoption in competitive shooters. This may be attributed to several factors, including the relative novelty of machine learning techniques, the significant resource investment required for development and deployment, and the specific challenges posed by different game genres and cheat types. For example, some games might not record gameplay data at all, or others might record it in a format not suitable for processing by applicable models.

6.2. Summary and critique

While some authentication systems based on mouse dynamics demonstrate impressive performance, they are typically evaluated in narrow, controlled contexts—often using a single dataset or limited task scope. As a result, it remains unclear how well these systems generalise to more complex and adversarial domains, such as cheat detection in video games.

In the emerging subfield of mouse dynamics-based anti-cheat systems, these challenges are even more pronounced. A major obstacle is the lack of publicly available datasets containing realistic examples of cheating gameplay. Data-intensive models (e.g. deep neural networks), require large volumes of labelled data to train effectively, which is difficult to acquire in this context. Other methods have relied on large amounts of historical data from individual players, but such historical data is rarely available in real-world scenarios and is typically only accessible in artificial experimental settings. As a result, both types of approaches face significant limitations when applied to practical cheat detection.

Additionally, many studies here do not provide comprehensive performance reporting, which makes it difficult to critically evaluate or meaningfully compare their results. Without rigorous benchmarking or more precise reporting, tentative conclusions can only be made about the general effectiveness of these systems.

The literature has nonetheless indicated the feasibility of various mouse-based approaches for player authentication in gaming [20, 26–28, 30, 31]. The most apparent challenges include:

- Data limitations: Many existing methodologies rely on access to large-scale datasets of cheating and non-cheating gameplay to train effective models which is impractical in real-world settings where historical data for specific players under suspicion is unavailable. Consequently, these approaches tend to be data inefficient and difficult to generalise.
- Performance barriers: While existing systems aim to detect cheating through mouse dynamics, few have convincingly achieved the level of performance needed for real-world deployment with no compromises. Differences in evaluation setups make direct comparison difficult, but most ultimately fall short of their intended goals, particularly in demonstrating robustness against more advanced or subtle forms of cheating.

The lack of off-the-shelf solutions in this space is not for lack of trying, instead it reflects the complexity of the problem: a system that must detect intelligent adversaries based only on behavioural input, without rich player history or clean training data. The fact that this remains an open problem is itself strong evidence of its difficulty.

7. System Design

7.1. Overview

This dissertation addresses the core challenges of data limitations, performance, and robustness highlighted in the preceding literature review by adapting techniques from mouse dynamics research to the domain of aim-based cheat detection in video games. Specifically, it incorporates the angle-based feature set proposed by Zheng and Wang—comprising direction, angle of curvature, and curvature distance—which has been shown to offer robust and efficient characterisation of user input in user authentication. While these features serve as the foundation, their integration into an anomaly-detection framework for cheat detection represents an entirely novel application. This work hypothesises that their use with an unseen processing pipeline and a purpose-built system architecture will enhance robustness against sophisticated cheating behaviours and improve data efficiency and performance.

In support of this, the system design draws on established best practices from the literature, including action

aggregation, ensemble learning, and SVMs. A key innovation lies in a novel action segmentation and processing pipeline developed specifically to improve clarity regarding the mixed-class nature of gameplay, where sequences often contain interwoven instances of both honest and cheating behaviour. Together, these components form a novel approach to cheat detection, combining established feature sets with new mechanisms designed to address the specific challenges of real-world gaming environments.

The resulting system operates as an efficient anomaly-detection framework. It builds a mouse signature from a player's typical mouse dynamics and evaluates subsequent gameplay actions for deviations from this baseline. Actions that diverge significantly from the expected signature are flagged as potentially software-controlled, enabling the detection of aim-based cheating without requiring extensive historical data or pre-labelled cheating examples.

This section starts by outlining the data collection process, followed by the preprocessing techniques used to isolate honest and potentially cheating behaviours. It then details the construction of feature samples and the system architecture that operationalises Zheng and Wang's features. Section 8 evaluates the system's performance across multiple cheat configurations and offers a novel insight into user interaction patterns in gaming contexts.

7.2. Data collection

Counter-Strike 2 (CS2) was selected as the experimental platform for this research. CS2 offers a replay functionality to consumers uncommon in many games. This feature records game information at 64 ticks per second, resulting in 64 entries of mouse movement data being logged per second into a `.dem` file format. This open-source file format is utilised across several games developed by Valve Corporation and can reconstruct gameplay events exactly. Earlier versions of this feature were used by other works detailed in the Literature Review [27–29, 31]. The game's relatively simple design also provides a good baseline for experimentation, allowing successful systems developed here to be potentially extended to more complex shooter video games.

External cheat software was obtained from the online forum UnknownCheats [13]. Given its active development for the latest CS2 version, it's advanced features, and reported use by cheating players at the time of this work [32], this software provides a good representation of cheating software in today's video game landscape. Figure 1 depicts the user interface when in use.

CS2 was always initiated using secure, offline launch parameters (Appendix 10), effectively disabling the game's anti-cheat mechanisms and online functionalities.

Each session utilised a workshop-based game mode (Appendix 11) which was played and recorded via the replay feature. This specific game mode involves non-player controlled characters spawning at pseudo-random locations behind or on top of obstacles around the map. In each session, the first recording consisted of gameplay played with the cheating software disabled. For the second recording, the aimbot feature was enabled with no smoothing (Configuration 1, Appendix 8). The third and final recording in a session was recorded with another cheating software configuration in which the advanced smoothing feature of the aimbot was turned on (Configuration 2, Appendix 9), significantly reducing the visual distinctiveness of the cheating software.

Figure 1: Screenshot of cheating software. Figure showing the user interface of the cheating software used for data collection. The software offers common cheat functionalities such as aimbot, TriggerBot, ESP, as well as other miscellaneous cheats.

Data collection occurred over multiple days, with two sessions each day. Each session utilised a distinct physical configuration (display size, mouse sensitivity, ergonomics). At the end of data collection, recordings were aggregated into three datasets each with general gameplay infused with either honest, cheating, or cheating-with-smoothing attacks, with each dataset spanning different physical configurations and temporal influences like player mood and alertness.

7.3. Data pre-processing

The open-source demo file parsing library, Demoparser2 [33], was used to process the recorded data. Following the query stage, the majority of the extracted data was discarded, with the exception of two primary data categories: the timestamps of each non-player character elimination event and the player's aiming-related information.

A representative sample of the raw data remaining is illustrated in Table 3 and Table 4.

Table 3: A data sample of a bot-hurt query showing the tick and their health after the attack. Health of 0 indicates elimination.

tick	health
83	0
695	04
529	0
695	0
554	74

Table 4: A data sample as a result of querying the parser for player pitch, yaw.

tick	pitch	yaw
⋮	⋮	⋮
5000	1.677475	11.320724
5001	1.677475	11.320724
5002	1.677475	11.320724
5003	1.677475	11.320724
5004	1.787338	10.358398
⋮	⋮	⋮
9996	0.137329	79.382385
9997	-0.165131	75.422180
9998	-0.440140	72.149628
9999	-0.577469	69.894684
10000	-0.604935	68.327408
⋮	⋮	⋮

7.4. Action Segmentation

A well-established process for segmenting a continuous stream of mouse movement data into discrete actions is lacking. For instance, Siddiqui et al. [26] segment the stream into blocks of 10 consecutive events. In contrast, as shown in Table 1, other experiments define actions based on button presses, such as Drag and Drop or Point and Click actions. In this experiment, a novel two-step process was implemented to identify different mouse action types for subsequent feature extraction and analysis. This was necessary because gameplay can be categorised into "Attack Actions" (defined here by the elimination of a bot) and "General Actions" (navigating the map to find opponents).

Engaging aimbot functionality during general map navigation offers no discernible advantage and effectively negates playing the game entirely. As such, Attack Actions are assumed to contain the entirety of any aim-based cheating employed by the player throughout gameplay, while the aiming data outside these intervals is highly likely to represent only authentic player behaviour. In other words:

- **Attack Actions (AA):** These actions originated from time-frames around bot eliminations, further categorised as Cheating AAs (involving cheating) and Honest AAs (without cheating).
- **General Actions (GA):** These actions were derived from gameplay segments outside calculated AAs and are assumed to be free of cheating.

First, Attack Actions were identified by examining the timestamps of bot eliminations and applying a tick padding before and after each elimination tick. A `right-padding` interval extends after the elimination. This is to examine whether relevant attack information continues briefly post-elimination, which might reflect the player's reaction time in acknowledging the kill and disengaging their aim. A `left-padding` of 25 ticks (approximately 390ms) was applied before the attack action began, chosen because it matches the determined average time taken to transition from stationary aim to achieving the elimination. It also slightly exceeds typical human reaction times [34]. Figure 2 illustrates how this padding separates the gameplay timeline into focused Attack Actions.

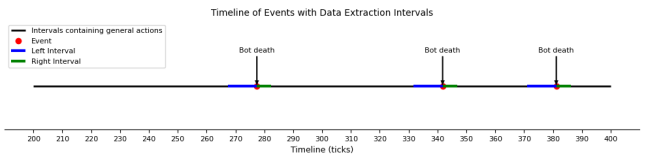

Figure 2: Decomposition of Tick Timeline into Attack Actions. The diagram illustrates how a raw tick timeline is segmented with `left-` and `right-padding` parameters to form Attack Actions encompassing each bot elimination.

The intervals defined by Attack Actions were removed from the complete aiming information to leave a dataset entirely devoid of aim-based cheating.

Then, the honest mouse movement data was then segmented further into General Actions based on the speed of mouse movement (Equation (1)). Segmentation points were identified where the aiming speed fell below a configurable threshold. To prevent the destruction of continuous mouse movements due to brief drops in speed, a `lull-period` parameter was implemented. If the aiming speed temporarily dropped below the threshold for a duration shorter than `lull-period` before subsequently increasing above the threshold again, the drop in speed is ignored. Conversely, if the speed remained below the threshold for a duration equal to or exceeding `lull-period`, a break was introduced to the data, segmenting the data into distinct, honest, General Actions. This process was applied to the honest honest data to create General Actions that are intended to encompass all of the distinct phases of intentional mouse movement: stationary, sustained movement, and return to stationary.

7.5. Metric calculation

Three angle-based metrics and corresponding probability distributions, originally proposed by Zheng and Wang in 2011 [1], are calculated from the mouse movement data within an action:

- **Direction (d_t):** For two consecutive points $A = (x_t, y_t)$ and $B = (x_{t+1}, y_{t+1})$, the direction at time t is defined as

$$d_t = \arctan 2(y_{t+1} - y_t,\, x_{t+1} - x_t)$$

If $d_t < 0$, then adjust by

$$d_t = d_t + 2\pi$$

ensuring that $d_t \in [0, 2\pi]$.
Note: The offset adjustment for d_t does not affect the final results, but it aids in visualising the data in a more intuitive way.

Table 5: Comparison of Action Quantities Across Categories.
Note: The total number of actions presented here differs from the final number of samples available for testing and training, which is determined by the block size used during data processing.

Action Category	Quantity
General Actions (All Demos)	4898
Honest Attack Actions	468
Cheating Attack Actions (Configuration 1)	414
Cheating Attack Actions (Configuration 2)	316

- **Angle of Curvature (θ_t):** For three consecutive points $A = (x_{t-1}, y_{t-1})$, $B = (x_t, y_t)$, and $C = (x_{t+1}, y_{t+1})$, define the vectors:

$$\vec{AB} = (x_t - x_{t-1},\, y_t - y_{t-1})$$

$$\vec{BC} = (x_{t+1} - x_t,\, y_{t+1} - y_t)$$

Then the angle of curvature at B is

$$\theta_t = \arccos\left(\frac{\vec{AB} \cdot \vec{BC}}{\|\vec{AB}\|\|\vec{BC}\|}\right)$$

ensuring $\theta_t \in [0, \pi]$.

- **Curvature Distance (δ_t):** For three points $A = (x_{t-1}, y_{t-1})$, $B = (x_t, y_t)$, and $C = (x_{t+1}, y_{t+1})$, first, define the perpendicular distance from B to line AC is:

$$d(B, AC) = \frac{|ax_t + by_t + c|}{\sqrt{a^2 + b^2}}.$$

$$a = y_{t+1} - y_{t-1}, \quad b = x_{t-1} - x_{t+1},$$

$$c = x_{t+1}y_{t-1} - y_{t+1}x_{t-1}.$$

Then the curvature distance at time t is

$$\delta_t = \frac{d(B, AC)}{\|\vec{AC}\|}$$

Note: Curvature distance is a ratio with a theoretical range of ($[0, \infty]$). In practice, values are limited to a maximum as the majority fall within a finite interval.

7.6. Final Feature Vector Calculation

7.6.1. Sample Exclusion. Actions were then checked for tick discontinuities against a specific threshold. If substantial time elapsed between consecutive entries, impacted metric values were removed. Actions ending with less than 10 valid values for any of the three metrics were completely excluded from the datasets, as they lacked sufficient data for accurate action depiction.

The parameter set used for data processing, unless specifically stated otherwise, can be seen in Appendix 11. The final count of each action type can be seen in Table 5. Approximately 16 attack actions per minute, and approximately 85 general actions per minute were generated in the chosen experimental task.

7.6.2. Block Creation. Let a block $B = \{A_1, A_2, \ldots, A_n\}$ consist of n unique actions, where each A_i corresponds to a specific mouse action and its three associated metrics: direction angle d_t, angle of curvature θ_t, and curvature distance δ_t.

For each metric, the values from all actions in the block are pooled together. The metric values are separated into m bins, and the probability of a value belonging to each bin is calculated to form empirical discrete probability distributions for each metric. Specifically, for each metric:

$$P_d = \{p_{d_1}, p_{d_2}, \ldots, p_{d_m}\},$$

$$P_\theta = \{p_{\theta_1}, p_{\theta_2}, \ldots, p_{\theta_m}\},$$

$$P_\delta = \{p_{\delta_1}, p_{\delta_2}, \ldots, p_{\delta_m}\}$$

where p_{d_i}, p_{θ_i}, and p_{δ_i} represent the probability of a value for the respective metric (denoted by the subscript) falling into the i-th bin, and $\sum_{i=1}^{m} p_{d_i} = \sum_{i=1}^{m} p_{\theta_i} = \sum_{i=1}^{m} p_{\delta_i} = 1$. The value of m effects the resolution of the final feature vector, which is evaluated in Section 8.3.1.

Averaging these features over a block of actions reduces noise and improve performance of some systems [21]. The resulting three averaged distributions together constitute a single sample in the training and testing datasets.

The final feature vector of length $3 \cdot m$ for block B, compatible with model input format, is obtained by concatenating the three probability distributions:

$$\mathbf{f}(B) = [P_d, P_\theta, P_\delta]$$

7.7. Model Training

Support Vector Machines (SVMs) are powerful supervised machine learning models renowned for their effectiveness in classification and regression tasks. Within the realm of game data analysis and cheat detection SVMs have demonstrated the unique ability to identify patterns indicative of cheating behaviour [20, 27]. SVMs handle non-linearly separable data by transforming it into a higher-dimensional feature space using a kernel function. This increases dimensionality, potentially allowing a clear separation of classes via an optimal hyperplane, which isn't feasible in the original space. The kernel function, implicitly defining this mapping (e.g., polynomial, RBF, sigmoid), is crucial. Kernel choice, influencing the decision boundary's complexity, depends on data characteristics and desired complexity.

Consistent with other work [21], empirical testing demonstrated the RBF kernel's superior performance (Appendix 12), likely attributable to the other kernels' insufficient capacity for the data's complexity.

Unlike other SVMs, the unsupervised One-Class SVM (OC-SVM) learns to identify outliers without labelled anomalies, which is especially useful when labelling cheaters is difficult or time-consuming. The model appears in the literature on mouse dynamics [15], with Shen et al.

reporting it as one of the highest performing models [21] along with it's two-class variant.

Given the inherent randomness in dataset generation from block formations (different action groupings yielding varied blocks), a single classifier may fail to accurately capture mouse dynamics due to an unfavourable ordering of actions. An ensemble of classifiers was constructed instead, using an odd number of OC-SVM models from the scikit-learn Python library [35]. Generating distinct training datasets for each ensemble classifier mitigates this risk of inaccurate classification as a result of a single bad dataset.

Each constituent OC-SVM within the ensemble was trained on a distinct dataset Honest AAs. These datasets were generated by randomly permuting the set of original Honest AA's before block calculation. This methodology ensures that each sequence of actions constituting a block, and thus the block itself, is unique both within and across the datasets utilised for training the different models in the ensemble. The hyper-parameters for each model's training were optimised using a Tree-structured Parzen Estimator search (Appendix 10) to create a tight decision boundary around the baseline class. This optimisation aimed to classify unseen samples from the baseline class within that boundary.

OC-SVMs are particularly susceptible to noisy training data, where outliers can significantly impair effective decision boundary creation. Therefore, data preprocessing to isolate potential cheating instances and separate the different data classes, as well as remove discontinuities is crucial. Block utilisation also minimises noise impact by averaging probability distributions across actions, stabilizing player signatures. The effect of block size on system performance is evaluated in Section 8.2.

7.8. Decision Making

Majority voting classifies samples. Individual model decision scores above a given threshold suggest an anomaly, while scores below indicate a non-anomalous classification, with the final decision based on the majority vote. Adjusting this threshold can bias the classification, potentially skewing results; for example, a high threshold might reduce false positives but at the cost of potentially missing more anomalies. Section 8.2 examines various threshold values through Receiver Operating Characteristic (ROC) curves.

7.9. Evaluation Methods

Trained on Honest AAs, the system generated decision scores for entirely unseen samples, indicating anomalous or non-anomalous classifications by the OC-SVMs. To evaluate the impact of threshold selection bias on performance metrics, Receiver Operating Characteristic (ROC) curve analysis was employed (detailed in Section 8.2). These ROC curves provide a comprehensive view of the system's potential performance across various threshold values, accommodating different operational needs (e.g., more or less aggressive detection strategies).

A general indicator of the system's discriminatory power is provided by the Area Under the Receiver Operating Characteristic curve (AUC-ROC). A perfect classifier, capable of establishing a precise decision boundary between classes, achieves an AUC-ROC of 1.0. In contrast, the random classifier that assigns classifications randomly yields an AUC-ROC of 0.5.

8. Results

8.1. Overview

With a training dataset of Honest Attacks derived from approximately 12 minutes of gameplay, the system demonstrated good-to-excellent discriminatory power. The classification of unseen Honest Attack Actions as non-anomalous aligned with the learned baseline. Furthermore, Cheating Attack Actions, including the subtle configuration, were consistently identified as anomalous. Unexpectedly, through qualitative analysis it is clear the system considered General Actions (GAs) to be the most anomalous relative to the other action types, despite GAs being entirely user-controlled, suggesting a substantial discernible difference from the learned baseline. This implies that the mouse dynamics of users change during aiming versus non-aiming gameplay in first-person shooters, an entirely novel proposition, however further validation is required to confirm.

8.2. Quantitative Analysis

To evaluate the system's capacity to discriminate between honest and cheating behaviour, classifiers were trained exclusively on Honest Attack Actions (AAs) and tested on both unseen Honest AAs and Cheating AAs. The system demonstrated strong discriminative performance, with results improving consistently as block size increased. This suggests that aggregating a larger number of mouse actions per block enhances the stability and separability of behavioural signatures.

Table 6 presents the average Receiver Operating Characteristic Area Under Curve (AUC-ROC) and Equal Error Rate (EER) scores across different block sizes. Figure 3 shows the corresponding ROC curves at these different block sizes also. At larger block sizes, the system achieved near-perfect classification performance — an average AUC-ROC of 0.9877 and a corresponding average EER of 6.08% (Figure 3) — indicating a clear distinction between Cheating AAs (Configuration 1) and unseen Honest AAs. These results support the hypothesis that cheating software in its most extreme configuration, operating without smoothing, exhibits a mouse signature sufficiently distinct from the users' normal mouse signature.

When tested against the more visually subtle cheating configuration, which applied advanced smoothing techniques, the system showed improved performance across all block sizes compared to Configuration 1 of the cheating software, highlighting the desired resilience of the feature

Table 6: Performance Metrics by Block Size (Configuration 2). Performance metrics on system trained on different block sizes, and asked to classify Configuration 2 Cheating AAs.

Block Size	Average AUC ROC	Average EER
1	0.7209	0.3424
5	0.9071	0.1781
10	0.9703	0.0945
14	0.9877	0.0608

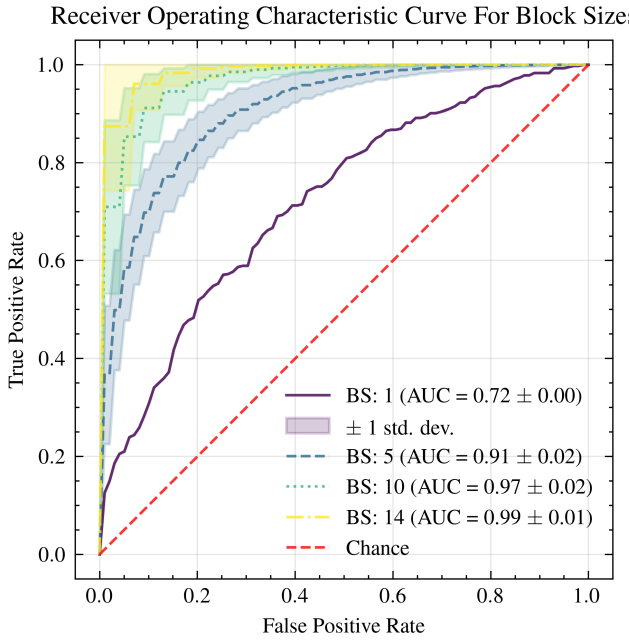

Figure 3: Honest vs. Cheating (Configuration 1) AA Discrimination Performance at Varying Block Sizes. ROC curves illustrating the performance of the ensemble classifier in distinguishing unseen Honest from Configuration 1 Cheating Attack Actions over 1000 iterations at block sizes (BS), 1, 5, 10, and 14.

set. Although not perfect, average AUC-ROC and EER scores improved steadily with block size (Table 7). At block size 1, performance was relatively weak (AUC: 0.7382, EER: 32.9%), but by block size 14, the system achieved an AUC of 0.9999 and an EER of just 0.06%, indicating that this configuration of the cheating software continues to produce a distinct signature.

Block size significantly impacts model performance and

Table 7: Performance Metrics by Block Size (Configuration 2). Performance metrics on system trained on different block sizes, and asked to classify Configuration 2 Cheating AAs.

Block Size	Average AUC ROC	Average EER
1	0.7382	0.3290
5	0.9850	0.0593
10	0.9993	0.0080
14	0.9999	0.0006

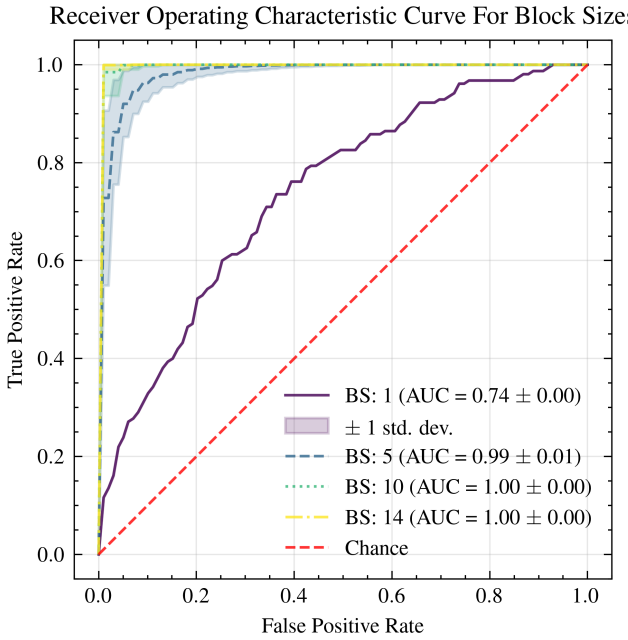

Figure 4: Honest vs. Cheating (Configuration 2) AA Discrimination Performance at Varying Block Sizes. ROC curves illustrating the performance of the ensemble classifier in distinguishing unseen Honest from Configuration 2 Cheating Attack Actions over 1000 iterations at block sizes (BS), 1, 5, 10, and 14.

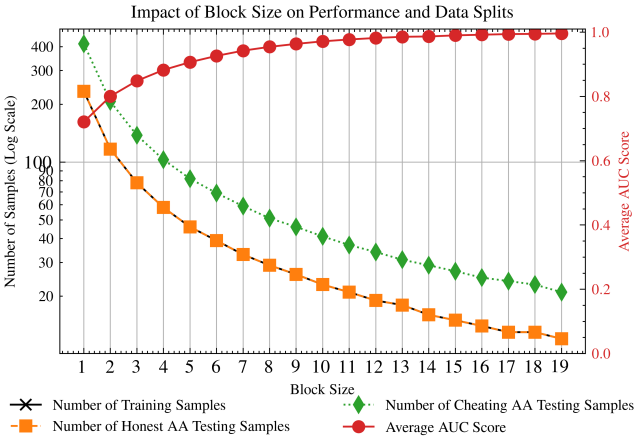

Figure 5: Block Size vs. Average AUC and Dataset Sizes. The figure depicts the relationship between block size used on both training and testing datasets, the average AUC-ROC score, and the resulting sizes of the Honest AA test split, and the Cheating AA (Configuration 1) dataset.

total available training/evaluation data. Figure 5 details the effect of block size on dataset sizes and average AUC-ROC. This analysis trained the model on 50% of the Honest AA dataset and evaluated it on the remaining unseen 50% and the Cheating AA (Configuration 1) dataset.

Noisy single-action probabilities yielded a 0.72 average

AUC-ROC. Increasing the block size to 4 improved AUC-ROC to over 0.90 (a 25% gain) but required 400% more actions per block, thus increasing the time to collect sufficient data for classification by the same. Achieving the desired AUC-ROC > 0.99 necessitated block sizes ≥ 14. This suggests a progressive classification approach for unseen samples, where iteratively appending attack actions to a block refines accuracy. Further testing showed that with sufficient training data (40-50 blocks), larger block sizes were only needed for testing samples to maintain performance.

8.3. Qualitative Analysis

The original in-game mouse data is inherently noisy and inconsistent, shaped by changing player intent, physical configurations, and dynamic game contexts—conditions under which SVMs (particularly OC-SVMs) typically struggle to establish effective decision boundaries given their sensitivity to distributional overlap and high levels of variance. However, at least superficially, the system's strong performance suggests that the preprocessing pipeline — particularly the novel action segmentation, chosen feature set, and block-level aggregation — has successfully reduced this complexity.

For models trained solely on Honest Attack Actions (AAs), unseen Honest AAs consistently received decision scores near zero (Figure 6), indicating their proximity to the learned decision boundary. In contrast, both Cheating AAs (Configurations 1 and 2) and General Actions (GAs) were confidently classified as anomalous. Notably, the decision values assigned to both cheating configurations were similar, despite substantial differences in visual behaviour (e.g., raw vs. smoothed aimbot movements). This suggests that the model is correctly detecting the underlying mouse signature produced by the cheating software and that this signature persists across obfuscation attempts, confirming the system's success in achieving one of its objectives: maintaining robustness against advanced cheating software techniques designed to mask the software.

The most substantial deviation from Honest AAs came from GAs — mouse actions performed by the same user outside of engagements. While both GAs and Honest AAs were recorded under full human control, their separation in feature space implies a task-induced shift in user mouse signature.

Under a second assessment, the models were trained on both GAs and Honest AAs (Figure 7). Here, the ensemble produced wide, non-discriminative decision boundaries — failing to flag any group, including cheating behaviours, as anomalous. This collapse in performance suggests that the model could not reconcile samples from the two different action categories. This leads to a crossroads of conclusions:

First, and most problematically, it may be that the user maintains a single, coherent mouse signature across GAs and AAs, but the pre-processing, feature set, or models do not facilitate its capture. If the system cannot unify both behaviour types into a single mouse signature, this suggests a deeper limitation of the system. Specifically, it

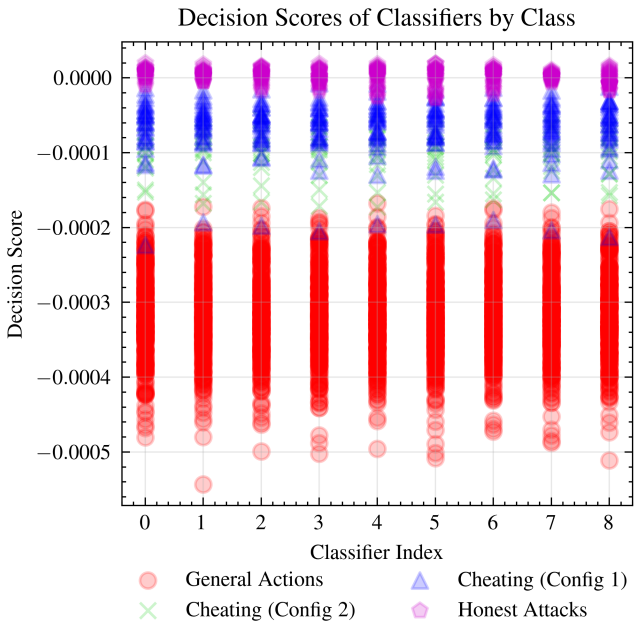

Figure 6: Decision Score Comparison: Honest AAs vs. Cheating AAs vs. GAs. The figure depicts decision scores produced by an ensemble (trained on Honest AAs) for unseen Honest AAs, Cheating AAs (Configuration 1 & 2), and GAs.

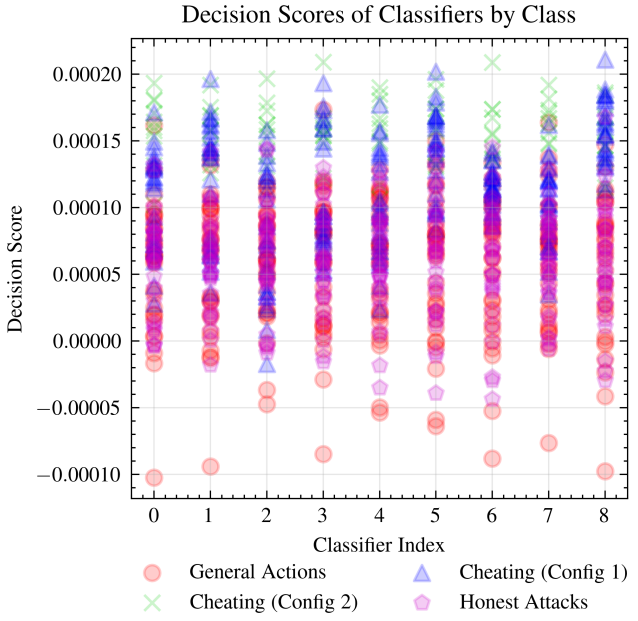

Figure 7: Decision Score Comparison: GAs & Honest AA's vs. Cheating AAs. The figure depicts decision scores produced by an ensemble (trained on equal numbers of GAs and AAs) for unseen GAs, unseen Honest AAs, and Cheating AAs (Configuration 1 & 2).

would indicate that previous strong results-when training on just a single action class-may not reflect the model's ability to learn a genuine user-specific mouse signature. Instead, they may reflect the exploitation of simpler, structural differences, or even unintended artefacts in the data, rather than the intended mouse signatures. Despite these considerations, the system's successful performance outside this test indicates that some aspect of this system's approach is effective and warrants further investigation to pinpoint the underlying mechanism.

Alternatively, an entirely novel hypothesis is presented: it may be the case that the user exhibits genuinely distinct behavioural signatures for GAs and Honest AAs, inside and outside of engagements. Under this hypothesis, the model cannot reasonably construct a tight decision boundary around two distinct mouse signatures without encompassing a large portion of feature space, effectively rendering it insensitive to deviations like those introduced by cheating. If true, this suggests that the system is successfully learning task-specific mouse signatures from the data, but expectedly struggles when asked to consolidate them given the choice of classification model - SVMs, particularly the OC-SVM variants, are known for their unsuitability when tasked with establishing a boundary around more noisy or varied data. Under this hypothesis, the action segmentation process used plays a pivotal role in making the data learnable: it isolates behaviourally coherent segments, allowing the model to learn task-specific patterns that would otherwise be obscured by overlapping distributions.

GAs being classified as the most anomalous is not unexplainable under this hypothesis: the aimbot used likely generated user input with directional patterns statistically similar to legitimate aiming when conveyed through this feature set. In contrast, GAs likely consist of more habitual, continuous, and broader directional changes, like 180-degree turns, employed for map navigation. These smoother, less abrupt shifts differ fundamentally from the rapid, target-focused adjustments in attacks. Consequently, the system, trained on AAs, potentially considers the mouse signature of Cheating AAs less anomalous compared to general actions because they share an underlying task and intention.

8.3.1. Impact of Feature Vector Resolution and Value Clipping. Figure 8 demonstrates that a minimum of ten bins is necessary for adequate model performance with angle-based metrics; fewer bins likely obscure critical details for signature identification. Sufficient granularity is achieved with at least twenty bins, with optimal performance at forty bins (vector length $\geq$ 120), underscoring the need to preserve data complexity through for model effectiveness.

As discussed in Section 7.5, the unbounded curvature-distance ratio metric was clipped. This process is employed to prevent extreme values from unduly influencing the distribution and subsequent binning. Figure 9 shows that increasing the curvature-distance clipping beyond 1.5 yields marginal performance improvements with key information concentrated within [0,1.5].

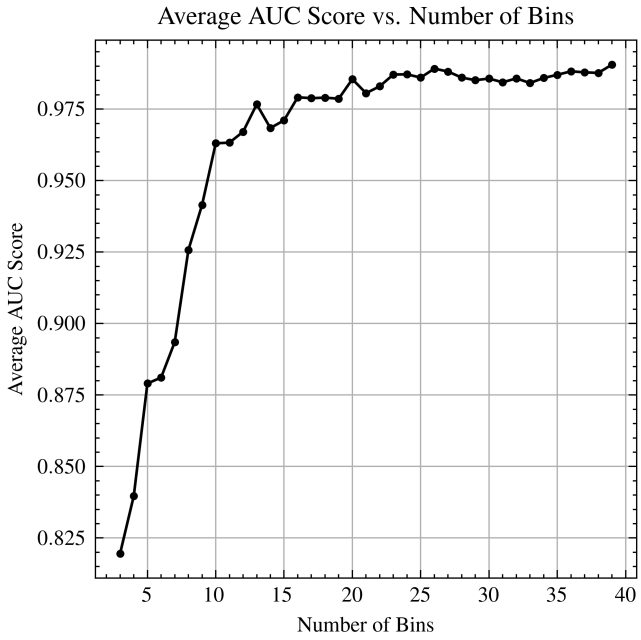

Figure 8: Quantity of Bins vs AUC-ROC score. The figure depicts system performance for different quantities of bins.

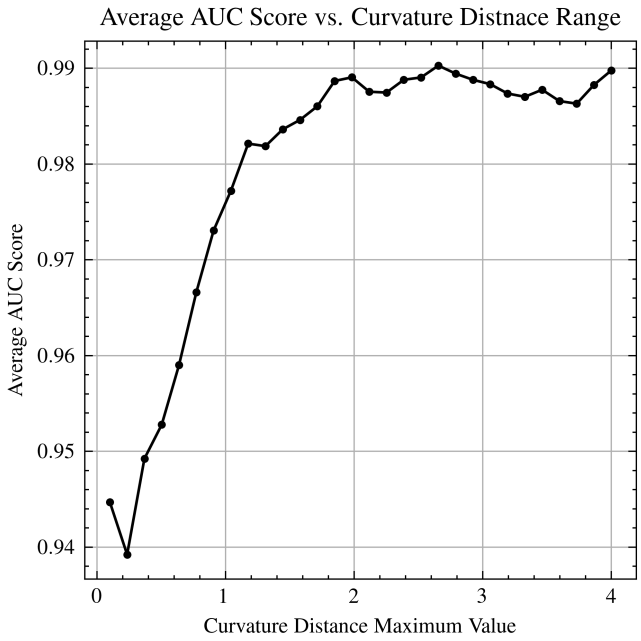

Figure 9: Impact of Curvature Distance Clipping on AUC-ROC Score. The figure depicts the impact on the upper value used to clip values of the curvature distance metric.

8.4. Limitations

The system demonstrated near-perfect classification performance when trained on a single category of mouse action with sufficient block size, with extremely low false acceptance and rejection rates, in Section Sections 8.2 and 8.3. While this is encouraging, these results are uncommon in the current landscape of mouse dynamics authentication systems (Table 1).

It's possible to rule out various causes of this performance, including artefacts introduced by physical configuration (display size, mouse sensitivity, ergonomics), as well as user biases such as mood or alertness, as these were dispersed between all the datasets. Additionally, any data down to the action used for training and testing during evaluation was completely separate and unseen. This removes any chance of accidental overlap between training and test data means the model has no memorisation bias.

That said, important limitations must be considered regarding the performance of the system:

Firstly, and most significantly, the results presented in Section 8.2 are highly likely to reflect a best-case scenario, and not a realistic depiction of the system's performance when generalised. Both the data processing pipeline (e.g., block size, clipping thresholds, lull period, left- and right-padding for extracting actions) and the SVM hyper-parameters were carefully optimised to promote a tight and optimal decision boundary, as discussed in Section 7.6 and 7.7, for this single user and this specific aimbot. This tightly coupled optimisation is likely to result in over-fitting to the particular characteristics of the collected data relative to the fit required in any broader deployment—where parameters would need to be fixed or generalised to be applicable to many more users' mouse dynamics and cheat types performance would almost certainly degrade.

Additionally, it's possible that some artefacts were introduced to the data as a result of the CS2 workshop used during data collection. For example, as the bots are sought out, it's common practice for the aim to be moved around whilst keeping the crosshair at head height. The bots were sometimes suspended above or below this horizontal plane on platforms or in troughs in the training arena. The angle-based metrics serve as good indicators as to whether the aim is being adjusted up or down off this horizontal plane. As such, it is possible that simply through this artefact to distinguish between a majority of general and attack actions.

9. Discussion

9.1. Reflection on initial research question

The primary objective of this research was to design and evaluate an automated system for detecting Aim Augmentation software in shooter video games. This objective has been achieved: throughout the course of this project a wide variety of designs were found, examined, evaluated for what can be learned, and what can be improved. The major problems were identified as: being related to data limitations pertaining the amount of data needed for successful performance, or that it needed to be labelled in the first place. The other problem pertained to performance, namely achieving performance required for commercial deployment and demonstrating robustness in the face of more advanced cheating methods.

Through the thorough first evaluation in this work, the proposed system demonstrated performance and efficiency sufficient for commercial deployment as a part of an anti-cheat system. Given that the system's design relies solely on mouse movement data, it possesses the versatility needed for it to be adapted to other shooter video games with relative ease, as well as significantly reduces, if not eliminates, permissions required on the user's system. It requires as little as 12 minutes of training data, easily acquirable through a simple scenario, and has demonstrated very good performance on a readily used and abused form of aim aimbot cheating, including both extreme and very subtle forms of cheating. Whilst there is much to be done to validate the distinctiveness of these results, namely a more diverse suite of aimbots, more advanced features (such as "magnetism" aimbots), and a study on a wider range of users to confirm the findings of this work, the system was successfully and efficiently able to detect aimbots in CS2.

9.2. Comparison to existing works

While direct quantitative comparisons with prior systems are inappropriate due to differences in experimental setup, cheat types, and datasets, the system proposed in this work addresses key shortcomings highlighted in the literature, and while not benchmarked numerically against prior models due to these differences, it demonstrates clear methodological improvements that address key limitations in the field.

Notably, unlike Siddiqui et al.'s system, which required extensive 20-minute data collection periods from multiple users and suffered significant accuracy drops with unseen data [26], this system achieves effective performance on unseen data requiring almost half the time. Furthermore, by relying solely on mouse movement data, this work avoids the server-client inconsistencies in perception and feature tractability issues encountered by Alkhalifa's approach [31].

A core contribution of this work is the introduction of a segmentation pipeline that isolates meaningful actions, enabling targeted analysis of different types of behaviour. This approach contrasts with the fixed or arbitrary segmentation strategies employed by other systems [26, 31], which potentially blur distinct mouse signatures.

9.3. Concluding Remarks

The field of mouse-based cheat detection currently echoes the landscape of mouse dynamics analysis in 2011, when Zheng and Wang's angle-based feature set demonstrated exceptional performance and efficiency for authentication. Their work elegantly distilled complex mouse movement data into three potent features, establishing a bench-

mark for effective mouse signature extraction. Recognising the enduring power of this approach, this research integrates their feature set into a novel system designed specifically for cheat detection. Initial evaluations suggest that, similar to the success in authentication, a well-designed system leveraging these features can achieve desirable performance within the subdomain of mouse-based cheat detection. This system proposal and initial evaluation lay a crucial foundation for future machine learning-driven advancements in this area, encouraging further validation, stress testing, and iterative refinement.

A novel proposition also emerged from this system's approach to segmenting actions: the distinct behavioural patterns exhibited by players inside and outside of engagements. This fundamental observation holds significant potential for future development of more sophisticated cheat detection systems. By focusing on these context-dependent signatures within gameplay mouse data, future research can aim for greater specificity in system design and ultimately lead to more robust and accurate cheat detection methodologies.

10. Robustness and Adversarial Analysis

Testing across various cheating software configurations indicates the system's robustness against advanced aimbot techniques, specifically smoothing. Given that smoothing is one of the most common advanced features used for cheat obfuscation, this suggests a strong likelihood of the system's resilience against this category of aimbot. However, it's important to note that aimbots employing subtle input manipulation, such as "aim-assist" or "magnetism"-like cheats, present a distinct challenge. The system's effectiveness against these types of cheats, which only subtly modify a user's mouse signature rather than fully overriding control, hinges on the ability to differentiate the user's genuine mouse movements from the subtle influence of the cheat. This dependency on signature separability represents a potential weakness in the current design.

Obtaining accurate and honest training data is also critical to system success. If users can substitute cheating software signatures for their own during the data acquisition phase, the system's utility is likely negated. Successful deployment requires exceptionally secure initial data collection before relaxing restrictions for general use.

11. Overall System Effectiveness and Feasibility

Using a further refined and validated version of the system put forward in this work, I propose a conceptual integration into the anti-cheat stack. Upon game installation, a mandatory 15-minute tutorial on secure servers will collect initial AA data while introducing gameplay. Following this, the system is trained remotely, and the model stored. Standard gameplay then commences. To select an appropriate threshold that minimises false positives, a pool of pre-saved cheating attack actions, representative of various cheating software, can be run through the trained model before deployment.

During public matches, the system continuously gathers Attack Actions, forming incrementally larger blocks for analysis. Once 14 actions are accumulated, the pre-trained system delivers a verdict. Players exhibiting cheat-assisted engagements will show deviations from their initial baseline and be flagged for review.

To account for long-term aiming variations, the system may periodically request (e.g., bi-weekly) 15 minutes of additional gameplay data, potentially through an optional aim-training mode. This minimally invasive system is easily adaptable to Shooter video games due to its low requirements and offers minimal running costs by analysing in-game behaviour rather than direct cheat software detection. Successful implementation can enhance existing automated anti-cheat filters, reduce manual review, and improve cheater detection rates.

12. Future work

While the method is clearly viable, future work and wider testing is needed to thoroughly analyse and evaluate performance in the aforementioned areas. The results suggest that the angle-based features proposed by Zheng and Wang are effective in categorising mouse movements in games. While the system has also shown resilience against advanced smoothing features, it should also be tested against more examples of sophisticated cheating software to ensure it's truly robust. A real-time implementation in a larger study would be a crucial step in understanding its real-world effectiveness.

Incorporating other techniques already successful in mouse dynamics should also be considered. More advanced feature engineering like curve fitting, for example, has shown to have a positive impact on performance in some situations [36].

13. Thoughts

13.1. Expectations & Personal Experience

I found this project incredibly engaging. While the theory and existing research on mouse dynamics in general contexts gave an indication of potential success, the outcome significantly exceeded my expectations. I had, however, anticipated GAs and Honest AAs would be categorised similarly, as I assumed the process of honest mouse movement in gaming would be homogenous across both action types such that one signature would be present, which could be differentiated from the Cheating AA signature. This is perhaps the most interesting revelation to me, personally, as it speaks to how we engage with video games as a medium. Overall, I'm very pleased with the system's final performance and believe the next logical step is to involve more people to assess the system's

robustness further. I hope it serves as a foundation for future improvements that ultimately reduce cheating for game providers, developers, and players and I'm excited about its potential for understanding game interaction and improving anti-cheat measures.

Word Count: 10,685

References

[1] Nan Zheng, Aaron Paloski, and Haining Wang. An efficient user verification system via mouse movements. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, page 139–150, New York, NY, USA, 2011. Association for Computing Machinery.

[2] Nan Zheng, Aaron Paloski, and Haining Wang. An efficient user verification system using angle-based mouse movement biometrics. *ACM Trans. Inf. Syst. Secur.*, 18(3), apr 2016.

[3] Irdeto. Global gaming report 2018.

[4] time2play. Is it cheating? exploring the grey areas in gaming.

[5] netease. Optmatch: Optimized matchmaking via modeling the high-order interactions on the arena.

[6] Newzoo. Pc & console gaming report 2024, 2024.

[7] Irdeto. Irdeto survey reveals increased worry of cheating and tampering, 2022.

[8] CSGO Case Tracker. 2023 year review, 2023.

[9] Bungie, inc. v. claudiu florentin.

[10] Stephen Totilo. Bungie faces lawsuits against cheaters and harassers. *Axios*, Aug 2022.

[11] Aimware. Member list - aimware forums.

[12] Valve Corporation. Counter-strike 2.

[13] UnKnoWnCheaTs. Unknowncheats - multiplayer game hacking and cheats.

[14] Giles Turnbull. Gaming co esea hit by $1m fine for hidden bitcoin mining enslaver. *The Register*, nov 2013.

[15] Simon Khan, Charles Devlen, Michael Manno, and Daqing Hou. Mouse dynamics behavioral biometrics: A survey. *ACM Comput. Surv.*, 56(6), feb 2024.

[16] Margit Antal and Lehel Denes-Fazakas. User verification based on mouse dynamics: a comparison of public data sets. In *2019 IEEE 13th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pages 143–148, 2019.

[17] A. Fulop, L. Kovacs, T. Kurics, and E. Windhager-Pokol. Balabit mouse dynamics challenge data set, 2016.

[18] Balabit. Releasing the balabit mouse dynamics challenge data set, 2016.

[19] Margit Antal and Elöd Egyed-Zsigmond. Intrusion detection using mouse dynamics. *IET Biometrics*, 8(5):285–294, 2019.

[20] Jiayi Zhang, Chenxin Sun, Yue Gu, Qingyu Zhang, Jiayi Lin, Xiaojiang Du, and Chenxiong Qian. Identify as a human does: A pathfinder of next-generation anti-cheat framework for first-person shooter games, 2024.

[21] Chao Shen, Zhongmin Cai, Xiaohong Guan, and Roy Maxion. Performance evaluation of anomaly-detection algorithms for mouse dynamics. *Computers & Security*, 45:156–171, 2014.

[22] Chao Shen, Yufei Chen, Xiaohong Guan, and Roy A. Maxion. Pattern-growth based mining mouse-interaction behavior for an active user authentication system. *IEEE Transactions on Dependable and Secure Computing*, 17(2):335–349, March 2020.

[23] Patrick Bours and Christopher Johnsrud Fullu. A login system using mouse dynamics. In *2009 Fifth International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, pages 1072–1077, 2009.

[24] Eduard Kuric, Peter Demcak, Matus Krajcovic, and Peter Nemcek. Is mouse dynamics information credible for user behavior research? an empirical investigation. *Computer Standards & Interfaces*, 90:103849, 2024.

[25] European Commission. Commission Implementing Decision (EU) 2019/329 of 25 February 2019 laying down the specifications for the quality, resolution and use of fingerprints and facial image for biometric verification and identification in the Entry/Exit System (EES). Official Journal of the European Union, L 57/18, 2019.

[26] Nyle Siddiqui, Rushit Dave, and Naeem Seliya. Continuous authentication using

mouse movements, machine learning, and minecraft, 2021.

[27] Ruan Spijkerman and Elizabeth Marie Ehlers. Cheat detection in a multiplayer first-person shooter using artificial intelligence tools. In *Proceedings of the 2020 3rd International Conference on Computational Intelligence and Intelligent Systems*, CIIS '20, page 87–92, New York, NY, USA, 2021. Association for Computing Machinery.

[28] Martin Willman. Machine learning to identify cheaters in online games, 2020.

[29] S.F. Yeung, J.C.S. Lui, Jiangchuan Liu, and J. Yan. Detecting cheaters for multiplayer games: theory, design and implementation. In *CCNC 2006. 2006 3rd IEEE Consumer Communications and Networking Conference, 2006.*, volume 2, pages 1178–1182, 2006.

[30] Valve John McDonald. How deep learning is used to combat cheating in csgo.

[31] Salman Alkhalifa. Machine learning and anti-cheating in fps games. Master's thesis, University College London, 09 2016.

[32] hitcommunity. Cheat legit external update tkz credits.

[33] LaihoE. Counter-strike 2 replay parser for python and javascript, 2025.

[34] Human Benchmark. Reaction time statistics, 2007-2025.

[35] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[36] Yi Xiang Marcus Tan, Alexander Binder, and Arunava Roy. Insights from curve fitting models in mouse dynamics authentication systems. In *2017 IEEE Conference on Application, Information and Network Security (AINS)*, pages 42–47, 2017.

Appendix

1. Impact of Cheating Software

Player Welfare: Irdeto conducted a survey of 9,436 players in 2018 to assess the prevalence and impact of cheating. The survey revealed that 77% of respondents indicated they would likely stop playing a multiplayer game if they suspected other players were cheating [3]. A separate survey, focusing on over 1,000 American gamers, found that a significant majority (74%) reported that cheating had "completely ruined a game" for them [4].

In modern multiplayer gaming, the implementation of Engagement Optimised Matchmaking (EOMM) has become increasingly common. EOMM algorithms aim to maximise player engagement by strategically placing gamers in matches. EOMM research has unsurprisingly indicated that balanced and fair matches, for example those with close wins or losses, are critical for enhancing player satisfaction and retention [5].

Matches involving honest and cheating players can range from marginally skewed to overwhelmingly imbalanced in outcome. In some environments, players who encounter cheaters may opt to leave the match and join another, thereby avoiding most of the impact. However, many games have introduced penalties for leaving prematurely to discourage players from abandoning matches when facing defeat. Consequently, cheating in video games can "ruin a game" for players, as they are now often forced to endure imbalanced matches, which are key to player satisfaction and welfare.

Revenue: The video game market's consolidation into an oligopoly, where a mere 48 titles accounted for 90% of new playtime in the US and UK in 2023 (including 25 live-service games [6]), has created an intensely competitive environment for developers. Simultaneously, the increasing shift towards sustained revenue through player engagement and in-game purchases, rather than upfront sales, means that initial game development is a sunk cost, and financial success hinges on player retention. This revenue model, utilized by 60% of developers as of 2022 [7], is critically vulnerable to cheating. When cheating drives players away in this concentrated market, they are less likely to return, given the abundance of established and popular alternatives, effectively leading to an irreversible loss of the game's primary revenue stream.

The specific quantity of revenue at stake is far from negligible: In 2023 alone, CSGOcasetracker, a website tracking in-game items for Counter-Strike 2, stated that the game generated a minimum of 980 million USD from the sale of in-game "cases" that unlock cosmetic weapon skins [8]. This figure underscores the immense financial potential tied to the in-game transactions threatened by cheating.

Bungie's aggressive legal pursuit of cheat developers has revealed the financial cost required to protect a game from cheating. These lawsuits, whilst focused on "tortious interference" with Bungie's contractual relationships with its players, revealed the significant damage inflicted by cheating. In one instance, Bungie was awarded 12 million USD in damages "not withstanding its anti-cheat efforts" [9], with their general counsel stating separately, "We believe very strongly that most people do not want to be in communities where cheating or harassment is allowed to thrive" and "In our view, removing harassment and abuse from our community is not only the right thing to do, it is also good business" [10].

2. Additional Types of Cheating in Video games

Continued from Section 3.

Augmented Movement Cheats (Fly-hacks, Scripts): This class of cheating software manipulate in-game movement mechanics through direct input manipulation or scripted automation. Augmented

movement cheats grant an unfair advantage by automating complex input sequences that would otherwise require extensive time and effort to master, reducing or eliminating the associated skill requirement.

The impact of these cheats fluctuates. In shooter video games, for example, where aiming accuracy is paramount, movement augmentations provide comparatively little benefit. In other cases they fundamentally disrupt intended game behaviour—for example, bypassing the game's physics engine to enable flight. While these more extreme applications provide a more substantial advantage, they are also overt and therefore more easily detectable. Their detectability is not necessarily due to their underlying programming but rather the clear, rule-breaking behaviour they produce - a simple check—such as detecting whether a player has remained airborne for an extended period without any valid in-game mechanics—can reliably flag and ban offenders. In consequence of the above, augmented movement cheats tend to be either subtle and difficult to identify or blatantly detectable.

Augmented Awareness Cheats (Extra-sensory Perception or ESP cheats, X-ray cheats): These cheats provide players with additional information that would otherwise be inaccessible to them, fundamentally altering their decision-making capabilities. By granting players access to this hidden information, these cheats artificially heighten situational awareness, enabling superhuman levels of judgment. This can manifest in various ways, such as pre-emptively firing before an opponent enters view or making seemingly perfect tactical decisions based on an opponent's position or remaining health. These cheats often take the form of an overlay on the game's display output, highlighting the position and attributes of entities.

The effectiveness of these cheats is defined by their versatility. Unlike more blatant forms of cheating, augmented awareness does not control the behaviour of players, instead they can choose how to leverage the information. While some may engage in obvious behaviours, such as repeatedly shooting through walls with perfect accuracy, others may use the knowledge more subtly to appear within the bounds of legitimate high-level play. Highly refined intuition and experience can sometimes give the illusion of the all-knowing decision-making ability provided by these cheats, making it hard to determine if a player is cheating or just exceptionally experienced. Sometimes accusations have been even levelled against professional players. Generally, the visibility of augmented awareness cheats is up to the discretion of the cheater, making them particularly insidious within competitive environments.

3. Anti-Cheat System Stack

Many online games incorporate **manual reporting** systems that allow players to flag suspected cheaters. Players can submit reports along with comments detailing their concerns, which helps prioritise cases for further investigation. The weight of a report can be influenced by factors such as the number of reports a player receives and the consistency of those reports over time. For example, if a player is reported multiple times across different matches for aimbot usage, their case may be escalated more quickly.

Manual review is an essential component of anti-cheat systems, used to address the tickets raised by automated anti-cheat software and manual reports. Review teams—either in-house or out-sourced—oversee these nuanced situations and determine whether disciplinary action is warranted. Some games also leverage community-driven review systems to build these teams of reviewers. For example, Counter-Strike 2 previously used the Overwatch system, where experienced players could review flagged gameplay and collectively decide if the player was cheating.

When a player is caught cheating, **penalties** vary based on factors like the severity of the cheat, prior infractions, and game context:

- *Repeat Infractions:* First-time offenders might receive temporary bans, while repeat offenders face permanent bans.
- *Cheat Type:* Severe cheats like aimbots typically result in instant, harsher bans, whereas exploiting in-game bugs may lead to lighter penalties.
- *Context:* Cheating in ranked matches or professional tournaments is punished more severely than in casual play, sometimes leading to suspensions or fines.
- *Hardware/Network Bans:* Some bans target a player's hardware or network, making it harder to evade punishment with a new account.

In professional gaming, consequences extend beyond in-game bans. Players caught cheating may be barred from tournaments, face fines, or have their organisation penalised.

The largest issues with in-game reporting systems are also flawed. For them to be effective, several conditions must be considered:

- *Ease of Use:* If the reporting system is clunky, intrusive, or un-intuitive, players won't use it.
- *Abuse Prevention:* Players can misuse the system, mass-reporting innocent users in an attempt to get them falsely penalised.
- *Scalability:* Each report generates a ticket that must be reviewed. If reports pile up faster than they can be processed, cheaters will persist unchecked.

4. Anti-Cheat Controversy

In the case of the company E-Sports Entertainment, software was deployed that not only monitored user activity beyond the scope of its intended anti-cheat services but also exploited users' computers for illicit bitcoin mining. The mining functionality was achieved by an employee embedding malicious code deep within the operating system, a level of access granted by kernel-level privileges [14].

The ability to operate at this level allowed E-Sports to bypass typical user-level security measures, enabling them to create a botnet consisting of over 14,000 users' computers. This incident highlights the significant risks associated with granting such extensive access to third-party software. Even when intended for legitimate purposes, kernel-level access can be abused, leading to severe privacy violations, unauthorised resource utilisation, and potential security breaches.

5. Game and Cheat Configurations

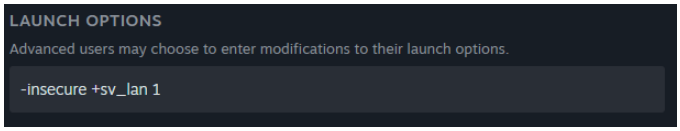

Figure 10: Counter-Strike 2 Launch Options: Disabling Steam and VAC. Screenshot of the launch configurations for Counter-Strike 2, with the `sv_lan` flag disabling Steam authentication and the `insecure` flag preventing VAC protected server connections.

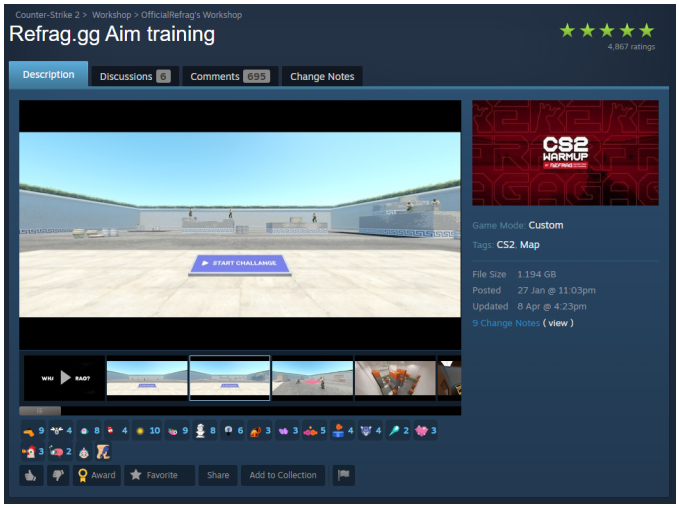

Figure 11: Shop Page of the Experimental Workshop Game-mode. Screenshot of the in-game shop interface of the custom workshop game-mode/map used for data collection.

Table 8: Cheating Configuration 1. Table showing the configuration of cheating software with no smoothing.

Setting	Value
ShowBoneESP	1
TriggerDelay	90
ShowBoxESP	1
TriggerHotKey	0
TriggerMode	0
RCSBullet	1
ShowHealthBar	1
AimFov	6.4
FovLineSize	60
AimBotHotKey	5
ShowLineToEnemy	0
RCSScale.x	1
RCSScale.y	1
ShowWeaponESP	1
ShowDistance	1
Smooth	0
ShowFovLine	1
ShowEyeRay	1
ShowPlayerName	1
AimBot	1
AimPosition	0
AimPositionIndex	6
HealthBarType	0
BoneColor	1 1 1 1
FovLineColor	0.215686 0.215686 0.215686 0.862745
LineToEnemyColor	1 1 1 0.862745
BoxColor	1 1 1 1
EyeRayColor	1 0 0 1
RadarCrossLineColor	0.862745 0.862745 0.862745 1
HeadShootLineColor	1 1 1 1
ShowMenu	1
ShowRadar	1
RadarRange	150
RadarPointSizeProportion	1
ShowCrossLine	1
RadarType	2
Proportion	2230
BoxType	0
TriggerBot	1
TeamCheck	1
VisibleCheck	1
ShowHeadShootLine	1
ShowCrossHair	1
CrossHairColor	0.176471 0.176471 0.176471 1
CrossHairSize	150
ShowAimFovRange	1
AimFovRangeColor	0.901961 0.901961 0.901961 1
OBSBypass	1
BunnyHop	0
ShowWhenSpec	1
AntiFlashbang	1

Table 9: Cheating Configuration 2. Table showing configuration of cheating software with smoothing.

Setting	Value
ShowBoneESP	1
TriggerDelay	90
ShowBoxESP	1
TriggerHotKey	0
TriggerMode	0
RCSBullet	1
ShowHealthBar	1
AimFov	5
FovLineSize	60
AimBotHotKey	5
ShowLineToEnemy	0
RCSScale.x	1
RCSScale.y	1
ShowWeaponESP	1
ShowDistance	1
Smooth	0.7
ShowFovLine	1
ShowEyeRay	1
ShowPlayerName	1
AimBot	1
AimPosition	0
AimPositionIndex	6
HealthBarType	0
BoneColor	1 1 1 1
FovLineColor	0.215686 0.215686 0.215686 0.862745
LineToEnemyColor	1 1 1 0.862745
BoxColor	1 1 1 1
EyeRayColor	1 0 0 1
RadarCrossLineColor	0.862745 0.862745 0.862745 1
HeadShootLineColor	1 1 1 1
ShowMenu	1
ShowRadar	1
RadarRange	150
RadarPointSizeProportion	1
ShowCrossLine	1
RadarType	2
Proportion	2230
BoxType	0
TriggerBot	1
TeamCheck	1
VisibleCheck	1
ShowHeadShootLine	1
ShowCrossHair	1
CrossHairColor	0.176471 0.176471 0.176471 1
CrossHairSize	150
ShowAimFovRange	1
AimFovRangeColor	0.901961 0.901961 0.901961 1
OBSBypass	1
BunnyHop	0
ShowWhenSpec	1
AntiFlashbang	1

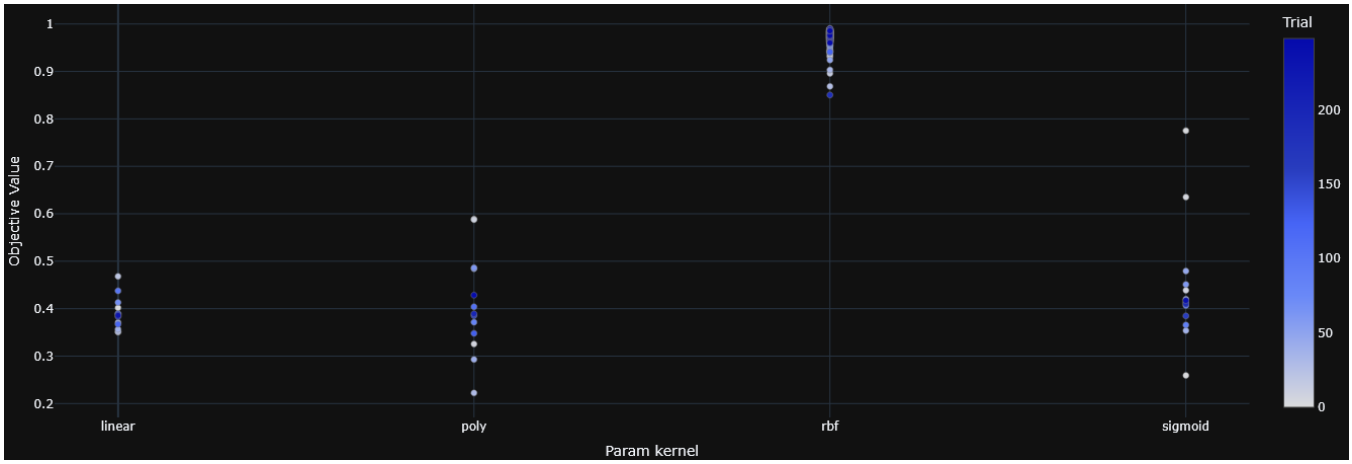

Figure 12: Tree-structured Parzen Estimator for Kernel Selection: The results of a Tree-structured Parzen Estimator search showing the RBF kernel as the superior choice within the explored search space.

Table 10: Optimised Hyper-parameters 1: The hyper-parameters selected for the scikit-learn One-Class SVM model using a Tree-structured Parzen Estimator search.

hyper-parameter	Value
ν	0.00782
kernel	rbf
γ	0.00907
degree	3
coef0	0.08249
tol	0.000001
shrinking	True
cache_size	300
verbose	False

Table 11: Optimised Hyper-parameters 2: The hyper-parameters selected for data processing using empirical testing and a Tree-structured Parzen Estimator search.

Hyper-Parameter	Value
train_test_split	0.50
vote_threshold	5
block_size	15
allow_duplicate_blocks	False
n_bins	39
direction_angle_range	(0, 360)
angle_of_curvature_range	(0, 180)
curvature_distance_range	(0, 2.75)
left_padding	25
right_padding	0
speed_threshold	0.27
lull_allowance	3
discontinuity_threshold	2
min_valid_entries	10

6. Interaction Interval Calculation

7. Metric Calculation

8. Probability Density Function Metric Calculation

9. General Interval Calculation

```python
def calculate_interaction_intervals_aim_arena(
    events: pd.DataFrame,
    left_padding: int = 25,
    right_padding: int = 0,
) -> pd.DataFrame:
    required_columns = {"tick", "health"}

    if not required_columns.issubset(events.columns):
        raise ValueError(
            f"Input DataFrame must contain columns: {required_columns}")

    intervals = []

    events = events.sort_values("tick")

    end_tick = None

    for i in range(len(events)):
        tick = events.iloc[i]["tick"]
        health = events.iloc[i]["health"]

        if health <= 0:
            end_tick = tick
            intervals.append(
                {
                    "attacker_id": 0,
                    "victim_id": 1,
                    "start_tick": max(0, end_tick - left_padding),
                    "end_tick": min(end_tick + right_padding, events['tick'].max()),
                }
            )

            end_tick = None

    return pd.DataFrame(intervals)
```

```python
def calculate_metrics(aim_records: pd.DataFrame, rads=False, make_positive=True):
    def _calculate_direction_angle_metric(
        aim_records: pd.DataFrame, make_positive=True, rads=False
    ):
        required_columns = {"tick", "pitch", "yaw"}

        if not required_columns.issubset(aim_records.columns):
            raise ValueError(
                f"Aim records dataframe missing: {required_columns - aim_records}"
            )

        angles_rads = np.arctan2(
            aim_records["pitch"].diff(), aim_records["yaw"].diff())

        if make_positive:
            angles_rads = np.where(
                angles_rads < 0, angles_rads + 2 * np.pi, angles_rads
            )

        df = None
        if rads:
            df = pd.DataFrame(
                {"tick": aim_records["tick"], "direction_angle": angles_rads}
            )
        else:
            df = pd.DataFrame(
                {
                    "tick": aim_records["tick"],
                    "direction_angle": np.rad2deg(angles_rads),
                }
            )

        df["direction_angle"] = df["direction_angle"].shift(-1)

        return df

    required_columns = {"tick", "pitch", "yaw"}

    if not required_columns.issubset(aim_records.columns):
        raise ValueError(
            f"Aim records dataframe missing: {required_columns - aim_records}"
        )

    metrics_df = pd.DataFrame({"tick": aim_records["tick"]})

    direction_angle_metrics = _calculate_direction_angle_metric(
        aim_records, make_positive=make_positive, rads=rads
    )

    metrics_df = metrics_df.merge(
        direction_angle_metrics, on="tick", how="outer")

    angle_of_curvatures = [None]
    curvature_distances = [None]

    for i in range(1, len(aim_records) - 1):
        x1, y1 = aim_records.iloc[i - 1][["yaw", "pitch"]]  # A
        x0, y0 = aim_records.iloc[i][["yaw", "pitch"]]  # B
        x2, y2 = aim_records.iloc[i + 1][["yaw", "pitch"]]  # C

        bc = np.hypot(x2 - x0, y2 - y0)
        ab = np.hypot(x0 - x1, y0 - y1)
        ac = np.hypot(x2 - x1, y2 - y1)

        if bc == 0 or ab == 0:
            angle_of_curvatures.append(None)
            curvature_distances.append(0.0)
            continue

        if ac == 0:
            angle_of_curvatures.append(0.0)
            curvature_distances.append(None)
            continue

        D_perp = abs((y2 - y1) * x0 - (x2 - x1) * y0 + x2 * y1 - y2 * x1) / ac

        curvature_distances.append(D_perp / ac)

        cos_theta = (bc**2 + ab**2 - ac**2) / (2 * bc * ab)
        cos_theta = np.clip(cos_theta, -1.0, 1.0)

        theta = np.arccos(cos_theta)
        if rads:
            angle_of_curvatures.append(theta)
        else:
            angle_of_curvatures.append(np.rad2deg(theta))

    if aim_records.shape[0] > 1:
        angle_of_curvatures.append(None)
        curvature_distances.append(None)

    metrics_df["angle_of_curvature"] = angle_of_curvatures
    metrics_df["curvature_distance"] = curvature_distances

    return metrics_df
```

```python
def calculate_pdfm(metric_samples, n_bins, data_range: tuple[float, float], clip=False):
    if clip:
        metric_samples = np.clip(
            metric_samples, min=data_range[0], max=data_range[1])

    counts, bin_edges = np.histogram(
        metric_samples, bins=n_bins, range=data_range)

    pdf = counts / len(metric_samples)

    return pdf
```

```python
def calculate_general_intervals(
    non_attack_dataframes,
    speed_threshold=0.06,
    lull_allowance=3,
):
    intervals = []
    for df in non_attack_dataframes:
        # Calculate movement deltas
        delta_yaw = df["yaw"].diff()
        delta_pitch = df["pitch"].diff()

        # Compute overall movement speed
        df["speed"] = np.sqrt(delta_yaw**2 + delta_pitch**2)

        in_interval = False
        lull_ticks = 0
        start_tick = None
        last_tick = None

        for idx, row in df.iterrows():
            tick = row["tick"]
            speed = row["speed"]

            if speed >= speed_threshold:
                if not in_interval:
                    # Start of a new interval
                    in_interval = True
                    start_tick = tick
                lull_ticks = 0  # Reset lull when speed is good
            else:
                if in_interval:
                    if last_tick is not None:
                        lull_ticks += tick - last_tick  # Add tick difference to lull
                    if lull_ticks > lull_allowance:
                        # Too long of a lull, end interval at the last good tick
                        end_tick = last_tick
                        intervals.append(
                            {"start_tick": start_tick, "end_tick": end_tick}
                        )
                        in_interval = False
                        lull_ticks = 0
                        start_tick = None

            last_tick = tick

        # If we end while still in a good interval
        if in_interval:
            intervals.append({"start_tick": start_tick, "end_tick": last_tick})

    return pd.DataFrame(intervals)
```