XGUARDIAN: Towards Generalized, Explainable and More Effective Server-side Anti-cheat in First-Person Shooter Games

Jiayi Zhang
The University of Hong Kong
brucejiayi@connect.hku.hk

Chenxin Sun
The University of Hong Kong
roniny@connect.hku.hk

Chenxiong Qian
The University of Hong Kong
cqian@cs.hku.hk

Abstract

Aim-assist cheats are the most prevalent and infamous form of cheating in First-Person Shooter (FPS) games, which help cheaters illegally reveal the opponent's location and auto-aim and shoot, and thereby pose significant threats to the game industry. Although a considerable research effort has been made to automatically detect aim-assist cheats, existing works suffer from unreliable frameworks, limited generalizability, high overhead, low detection performance, and a lack of explainability of detection results. In this paper, we propose XGUARDIAN, a server-side generalized and explainable system for detecting aim-assist cheats to overcome these limitations. It requires only two raw data inputs, pitch and yaw, which are all FPS games' must-haves, to construct novel temporal features and describe aim trajectories, which are essential for distinguishing cheaters and normal players. XGUARDIAN is evaluated with the latest mainstream FPS game CS2, and validates its generalizability with two different games. It achieves high detection performance and low overhead compared to prior works across different games with real-world and large-scale datasets, demonstrating wide generalizability and high effectiveness. It is able to justify its predictions and thereby shorten the manual review latency. We make XGUARDIAN and our datasets publicly available.

1 Introduction

The global online gaming market is projected to grow to approximately $250.2 billion by the end of 2025, with PC gaming leading as the fastest-growing segment, anticipated to expand by 8.1% in 2025 [36]. Within this landscape, First-Person Shooter (FPS) games hold a significant 20.9% market share and are expected to maintain a compound annual growth rate of 8.1% from 2025 to 2033 [13, 56]. Despite their popularity, FPS games are particularly vulnerable to cheating, as developers of malicious software exploit their competitive and multiplayer nature for monetary gain [63]. This pervasive issue not only undermines fair play but also poses substantial economic and reputational risks to the gaming industry [18].

Among the various forms of cheating, aim-assist cheats are the most prevalent and notorious. These cheats enable players to illegally reveal opponents' locations, auto-aim and shoot, providing unfair advantages even to seasoned or professional players [7]. This has spurred significant interest from both academia and industry in developing effective anti-cheat solutions. Numerous approaches have been proposed, leveraging statistical features and machine learning techniques to identify cheaters [2, 22, 30, 61, 62], while others have employed advanced time-series or ensemble models to detect temporal anomalies [7, 63]. However, existing solutions are hampered by several critical limitations, which we categorize as **L1** through **L4**.

(L1) Client-side Dependency and Resource Constraints. Many recent solutions rely on client-side data collection and cheat detection [7, 61, 62]. However, this approach is fraught with challenges, including the ease of bypassing anti-cheat mechanisms, risks of memory tampering, hardware limitations, and system overhead in highly concurrent environments. Cheaters can circumvent most current anti-cheat systems by using obfuscated cheats and higher privileges [28], or by directly accessing memory to tamper with detection models or processes [7]. Although the latest design, BOTSCREEN [7], addresses some of these issues using Intel SGX, a Trusted Execution Environment (TEE), SGX was deprecated in 2021 [5, 26], and its limited compatibility and practicality hinder widespread industrial adoption. Moreover, client-side solutions must operate in highly concurrent environments, where the client handles game logic, rendering, voice chat, and anti-cheat data extraction and analysis simultaneously. This consumption of scarce client-side resources increases hardware requirements, complicates optimization, and degrades the overall gaming experience.

(L2) Limited Real-world Performance. The real-world efficacy of most previous server-side [2, 22, 30] and client-side [7, 61, 62] solutions remains uncertain, as they have only been tested on small-scale, simulated datasets. This lack of real-world validation could lead to performance degradation or failure in complex scenarios. While the latest server-side

Table 1: ML-based anti-cheat detection works comparison.

Side	Method	Work(s)	Availability[1]	Generalizability			Interpretability	Real-world Functionality		
				Data Collection	Feature[2]	Evaluated[3]		Dataset	System Constraint	Performance
Client	Unsupervised Angle Anomaly Detection	[7]	T, C, D	✗	⚔	✗	✗	SSL[4]	TEE (Intel SGX)	Unknown[5]
	Statistical Classification	[61, 62]	T, C, D	✗	✓	✗	✗	SSL	Concurrent overhead	Unknown
Server	Statistical Classification	[2, 22, 30]	T, C, D	✗	✓	✗	✗	SSL	Concurrent overhead	Unknown
	Multi-view Ensemble Learning	[63]	T, C, D	✓	⚔	✗	✗	Real-world	None	Low
	Temporal Aim Trajectory Classification	XGUARDIAN	T, C, D	✓	✓	✓	✓	Real-world	None	High

[1] **Availability**: Public availability of technical details (T), source code (C), and datasets (D), red and green indicate unavailable and available; [2] **Feature**: ⚔ denotes part or all of the features are not generalized; [3] **Evaluated**: If the method's generalizability is evaluated through experiment(s); [4] **SSL**: Simulated, small-scale, low complexity; [5] **Unknown**: Have not been evaluated.

solution [63] is evaluated on a large-scale, real-world dataset and avoids client-side limitations (detailed in Section 2.1), its prediction performance remains relatively low due to the complexity of the data. Additionally, increasing community grievances about FPS anti-cheat systems indicate low recall rates in current industrial solutions [27, 41, 51, 63].

(L3) Limited Generalizability. Existing solutions often lack generalizability. For instance, the feature design of the most advanced server-side solution [63] is tailored to a specific FPS subgenre, tactical shooters [57]. It requires features like economy- and grenade-related features, making it difficult to generalize across different sub-genres of FPS games, as its authors acknowledge. While some statistical features in other works, such as playtime and winning rates [23], are generalizable, they are not directly relevant to aim-assist cheats [7], reducing prediction accuracy. Other features, like the angle between the line of sight and a target used in the latest client-side solution [7], require specific engine modifications for different games, as such data may not exist in log or replay files (introduced in Section 2.3), limiting their applicability.

(L4) Lack of Explainability. A critical drawback in all prior solutions is the lack of interpretability. A ban requires manual verification, which involves reviewers reviewing hours of the suspect's gameplay replays. This process is expensive and time-consuming for the industry. None of the existing anti-cheat works can explain how or why a prediction is made, which is essential for any practical anti-cheat system to pinpoint the exact suspicious segments so that it can justify the ban and shorten the review time.

In this paper, we introduce **Ex**plainable **Guardian** (XGUARDIAN) to address the aforementioned limitations. Deployed on the server side, XGUARDIAN avoids the inherent limitations of client-side solutions and leverages replay files as the data source to perform detection asynchronously during server idle periods, minimizing overhead **(L1)**. XGUARDIAN achieves state-of-the-art performance on a large, complex real-world dataset **(L2)** and is the first work to demonstrate wide generalizability by validating its effectiveness on large datasets from different sub-genres of games **(L3)**. XGUARDIAN addresses the explainability bottleneck by instantly pinpointing the exact suspicious frames and features within a match. This drastically reduces manual review time, making our system a powerful and practical tool for game moderators. We also conduct case studies to illustrate XGUARDIAN's interpretability **(L4)**. Table 1 lists the high-level comparison between XGUARDIAN and prior works.

The key innovation of XGUARDIAN lies in its ability to standardize and generalize the representation of player aiming trajectories across all FPS games, while also providing an interpretable analysis of these trajectories. XGUARDIAN is generalized by requiring only two raw inputs, pitch and yaw (introduced in Section 4.1.1), which are must-haves in all FPS games. We discover three fundamental aspects, velocity, acceleration, and angular change, to effectively represent a player's aiming trajectory in a generalized manner, making it applicable to detecting cheaters. XGUARDIAN employs a GRU-CNN model to effectively classify time-series features, enhances the explainability framework SHAP (introduced in Section 2.4), and designs the corresponding visualizations to provide interpretable detection results.

We implemented and evaluated XGUARDIAN on the well-known FPS game Counter-Strike 2 (CS2), using a large-scale real-world dataset from an active platform with millions of users. The dataset is two orders of magnitude larger than that of the most advanced client-side solution [7] and matches the data volume of the leading server-side solution [63], both analyzing approximately 3,000 matches. Specifically, XGUARDIAN analyzes 3,069,216 aiming operations across 31,971 aim trajectories from 5,486 players in real-world matches. It achieves up to 90.7% recall while maintaining a false positive rate (FPR) of 4.1%. Compared to previous anti-cheat methods, XGUARDIAN improves detection accuracy by 12.5% over the state-of-the-art method. Importantly, to the best of our knowledge, XGUARDIAN is the first work in anti-cheat detection to prove its generalizability through a large-scale experiment on different games in real practice. We demonstrate that XGUARDIAN's performance remains consistent across different sub-genres of games on different platforms (PC and mobile). We empirically show that XGUARDIAN remains somewhat robust under adversarial attacks. XGUARDIAN also achieves interpretability by illustrating each elimination trajectory explanation from different feature aspects. Additionally, XGUARDIAN incurs marginal overhead on the server. Our contributions are as follows:

- We propose novel, generalized, and effective features, a

time-series GRU-CNN model, and an explainable detection framework for identifying aim-assist cheats in games.

- Our approach demonstrates strong performance and evaluated with the real-world data.
- We open-source XGUARDIAN and release two new datasets (CS2, Farlight84) in Open Science.

2 Background

2.1 Server-side Anti-cheat

Server-side anti-cheat systems are defined by their architectural design to collect, process, and analyze gameplay data exclusively on secure server infrastructure. While client-side telemetry remains inherently vulnerable to tampering (as discussed in **L1**), server-side detection offers a critical advantage: cheaters must ultimately manifest their advantages through observable server-authoritative actions to impact gameplay outcomes [7, 63]. By aggregating final-state player inputs and game-state snapshots, these systems establish a ground-truth dataset resistant to client-side manipulation.

Earlier server-side anti-cheat systems [2, 22, 30, 60] relied on simplistic threshold-based heuristics or single-model classifiers using coarse-grained metrics, e.g., hit-accuracy, win-rate distributions. However, the proliferation of replay systems in modern FPS games (see Section 2.3) has enabled richer server-side data pipelines, characterized by modularity and loose coupling between game engines and analytical frameworks. The state-of-the-art anti-cheat design, HAWK [63], leverage this paradigm shift through multi-view feature engineering, combining spatial-temporal behavioral patterns with deep learning architectures to detect sophisticated cheats. Nevertheless, two critical limitations persist: (a) existing systems lack interpretable decision mechanisms, and (b) detection models exhibit poor generalizability across all FPS games.

2.2 Aim-Assist Cheats in FPS Games

Modern cheating tools have evolved into integrated platforms that combine multiple exploit modalities. For example, in most modern FPS games, **aim-assist** cheats commonly bundle aimbots (auto-aiming) with wallhacks (see-through walls), creating compounded unfair advantages [44]. In terms of the form, aim-assist cheats have evolved from pure aimbots (direct reticle manipulation to target centroids) to triggerbots (automated firing conditioned on crosshair overlap) and computer-vision-based aimbots (real-time object detection via screen scraping) [49]. Moreover, cheaters dynamically adjust sophistication levels of using aim-assist cheats across two axes: (1) functional granularity (e.g., targeting specific body part vs. full-body locks) and (2) stealth tradeoffs (blatant play vs. behavioral mimicry of legitimate players). Thus, anti-cheat systems have shifted from signature-based detection of individual cheat types to behavioral analysis of cheat categories [15, 16]. As aim-assist cheats constitute a significant portion of all cheating behaviors for FPS games [7,38,63], our work focuses on addressing the aim-assist cheats. Our evaluations in Section 5 addresses this through multi-dimensional sophistication modeling across large-scale real-world datasets.

2.3 Replay Systems and Replay Files

Modern FPS engines incorporate replay systems that serialize match state transitions into game-agnostic replay files (e.g., CS2's *.dem* format). These files provide a complete spatiotemporal record of player inputs, game events, and entity states, decoupled from runtime engine dependencies. XGUARDIAN operates on server-authoritative post-match records, which provide three practical benefits for behavioral cheat analysis: (1) resistance to client-side manipulation, (2) avoiding introducing real-time overhead constraints, (3) access to a unified global game state for camera-independent behavior reconstruction. Considering server-side solutions require direct cooperation from game developers, which is a resource rarely accessible to the broader research community, replay files as an anti-cheat data source can be further leveraged for offline detection methods, as replay files can be parsed and analyzed independently of the game server to extract relevant data. Broader researchers can likewise use XGUARDIAN to assess the presence of cheaters in a given match.

2.4 SHAP Explainers

Model interpretability is critical for anti-cheat systems for adversarial robustness and fairness auditing. SHAP [32, 33] addresses this by quantifying feature contributions through coalitional game theory. It establishes a connection between optimal credit allocation and local explanations by calculating classic Shapley values, which explain the ML model's output by distributing the difference between the actual prediction and the model's baseline prediction across all input features. A positive Shapley value indicates that the feature significantly contributes to increasing the prediction and vice versa. *GradientExplainer* and *TreeExplainer* are SHAP's implementations used for interpreting complex ML models and are customized and used in XGUARDIAN for the decision explanation. *GradientExplainer* is suited for deep learning models and leverages model gradients to explain how input features impact the prediction. *TreeExplainer* is designed for tree-based models to compute optimized Shapley values.

3 Overview

3.1 Threat Model

To reflect realistic anti-cheat deployment scenarios, we assume that adversaries utilize aim-assist cheats to gain an un-

fair competitive advantage in FPS games. We model the specific type, configuration, and level of sophistication of these cheats as black boxes, focusing on their observable behavioral impact. While we cannot exhaustively evaluate every possible cheat variant beyond our dataset, this assumption captures the diversity and unpredictability of real-world cheating behaviors, where attackers may use commercially available or custom-developed tools with varying degrees of subtlety or aggression. For instance, one adversary may employ a wallhack from vendor X, deliberately disabling visual enhancements such as enemy model highlighting while enabling minimap position overlays to mimic legitimate play and avoid detection. Conversely, another may use a combination of a wallhack and a hard-lock aimbot from vendor Y, bypassing visual obfuscations (e.g., flashbangs or smoke grenades in CS2) and engaging in overt, easily noticeable cheating. Our approach aims to detect a wide spectrum of aim-assist behavior, from stealthy to blatant, without relying on prior knowledge of the cheat's implementation details.

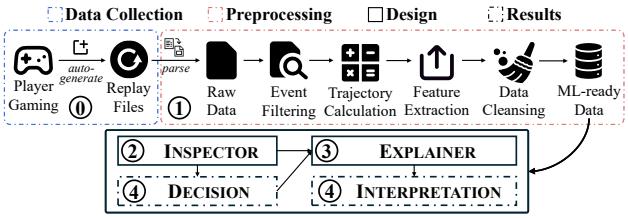


Figure 1: Workflow of XGUARDIAN.

3.2 Workflow

Figure 1 presents the overall workflow of XGUARDIAN, which operates in four main stages following gameplay.

⓪ **Data Collection**. Players engage in matches as usual, during which the game engine automatically records gameplay into replay files upon match completion. These replay files are collected from our partner platform, which also provides ground truth labels indicating whether the player has cheated.

① **Preprocessing**. We begin by parsing raw time-series data from the replay files (referred to *demos* in CS2) into structured tabular form. For each player, we identify the *ticks*, the smallest time unit in the game, corresponding to *elimination events* (conduct a kill on an opponent), and extract a fixed-length time window surrounding each event. Within this window, we compute the player's crosshair trajectory in 2D screen space, representing their real-world mouse-controlled aiming path. From these trajectories, we extract a comprehensive set of features as detailed in Section 4.1, thereby constructing a dataset for model training and evaluation.

② **Inspection**. The preprocessed samples that correspond to an elimination event are then passed to INSPECTOR. INSPECTOR employs an ensemble of temporal classification models to analyze the samples and detect anomalous patterns indicative of aim-assist behavior. Its core responsibility is to dynamically assess whether suspicious behavior occurs during specific eliminations within a match. Further architectural and implementation details are provided in Section 4.2.

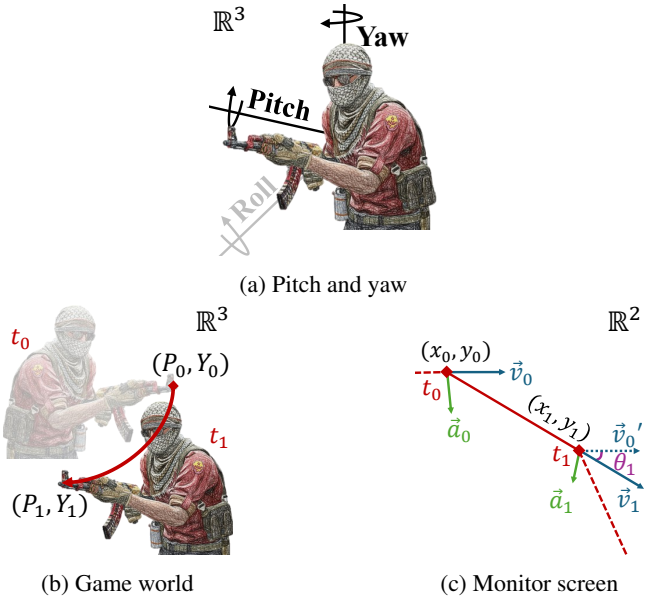
③ **Explanation**. After classification, INSPECTOR's outputs are combined with the training data to serve as background context for EXPLAINER. EXPLAINER quantifies the influence of input features on each classification result, offering interpretable insights into the detected aiming behaviors and the broader in-match performance. A detailed discussion of the explanation module is provided in Section 4.3.

④ **Decision and Explainability Output**. For each player in each match, XGUARDIAN produces a binary decision indicating whether cheating was detected. Each decision is accompanied by a visualized explanation at feature and match level respectively, providing transparency into how and why the system arrived at its conclusion.

4 XGUARDIAN

4.1 Trajectory Extraction and Feature Design

4.1.1 Dimensions of movement in FPS games



(a) Pitch and yaw

(b) Game world

(c) Monitor screen

Figure 2: Pitch and yaw, the dimensions of movement (a); aiming trajectory extraction in 3D game world (b) and feature constructions in 2D monitor screen (c).

Figure 2a illustrates the three rotational dimensions commonly used to describe player view orientation in 3D first-person shooter (FPS) environments: *pitch*, *yaw*, and *roll*. All these angles are defined in the game engine's world coordinate system. Among these, pitch and yaw are the primary angular measurements used to represent a player's aiming

direction and camera orientation. **Pitch** refers to the vertical angle of the player's view. It captures the up-and-down rotation of the camera, with values typically ranging from -90° to 90°. A pitch of -90° indicates that the player is looking directly downward, 0° corresponds to a level forward gaze, and 90° indicates a view directly upward. **Yaw** represents the horizontal angle of the player's view, measuring the left-to-right rotation. Yaw typically ranges from -180° to 180°. A yaw of 0° indicates a forward-facing direction, -90° indicates a leftward view, 90° indicates a rightward view, and ±180° corresponds to looking directly behind. Roll describes rotation around the forward axis of the camera, i.e., the tilt of the view to the left or right. While roll is a valid component of 3D orientation, it is largely irrelevant in the context of FPS gameplay, where camera tilt does not impact aiming or navigation. Consequently, XGUARDIAN focuses exclusively on pitch and yaw for modeling player aiming behavior.

4.1.2 Extraction scope

Throughout an entire match, players spend most of their time outside of combat, meaning their aiming trajectories often contain little useful information and may instead introduce noise and unnecessary computational overhead. To address this, we define an *elimination window*, which is created when a player triggers an *elimination event*. Specifically, we locate the corresponding *tick* of the event and extract data from m *ticks* before and n *ticks* after it. The window size is $(m-1)+n+1 = m+n$. These selected *ticks* form an elimination window and serve as individual data samples. Figure 3 illustrates the elimination window selection process. To prevent adversarial attacks, XGUARDIAN focuses on analyzing elimination, an unavoidable and attack-resistant game-critical event, rather than other events like firing that cheaters may easily manipulate (e.g., spam random shots to introduce noise and poison the model).

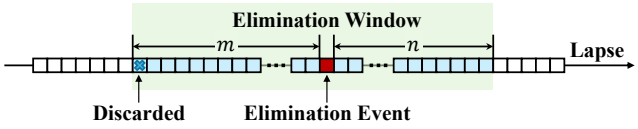


Figure 3: Elimination window selection.

4.1.3 3D aiming trajectory extraction and 2D mapping

In all FPS games, players use input peripherals, typically a mouse or controller, to control their in-game avatar's aim within a 3D virtual environment. This aiming behavior is governed by two angular measurements: *pitch* and *yaw*, which represent vertical and horizontal orientations. The player's aiming trajectory in the 3D world can thus be expressed as a sequence of pitch and yaw values over time. As illustrated in Figure 2b, the red curve denotes a trajectory beginning at time tick t_0 with angles (P_0, Y_0) and ending at t_1 with angles (P_1, Y_1), where P and Y denote pitch and yaw, respectively.

To enable effective visualization and downstream analysis, these 3D world-space angles must be mapped into 2D screen coordinates, simulating how the player's crosshair moves across the display. The raw data extracted from the *demo* files includes pitch and yaw values, which are converted to 2D coordinates using the transformation process detailed in Algorithm 1. Specifically, yaw values, which range from [-180°, 180°], are first normalized to [0°, 360°] by adding 180° and then scaled according to the screen width. Similarly, pitch values, ranging from [-90°, 90°], are normalized to [0°, 180°] by adding 90° and scaled to the screen height. To conform to the screen coordinate system, where the origin is at the top-left corner, the Y-axis is then inverted.

Algorithm 1 Convert Pitch and Yaw to 2D Coordinates

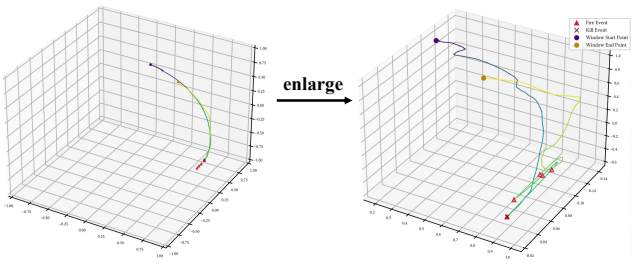
function PITCHYAWTOXY(pitch, yaw, width, height)
  $x \leftarrow \left(\frac{yaw+180}{360}\right) \times width$
  $y \leftarrow \left(\frac{pitch+90}{180}\right) \times height$
  $y \leftarrow height - y$
  return (x, y)
end function

The conversion result is shown in Figure 2c, where the solid red line represents the corresponding 2D crosshair trajectory mapped from the 3D trajectory shown in Figure 2b, spanning from t_0 to t_1. Importantly, pitch and yaw are standardized components across all FPS games, enabling XGUARDIAN to generalize across a broad range of titles in this genre.

We further visualize the player's aiming trajectory in the 3D game environment in Figure 4a. It reflects how the player adjusts their view direction, via pitch and yaw, to target opponents. The trajectory begins at the purple point and ends at the yellow point, with the color gradient indicating temporal progression. Triangle markers denote the moments when the player fired their weapon, while cross markers represent confirmed eliminations. Figure 4b shows the corresponding 2D crosshair trajectory, capturing the physical input behavior mapped to screen space. To reduce visual clutter from overlapping lines, the temporal color gradient is omitted, while all other markers retain their meanings from Figure 4a.

4.1.4 Feature construction

After obtaining the 2D coordinates, we encounter a challenge: despite aligning the coordinates based on each elimination event, the aiming trajectories remain highly scattered and patternless. As a result, directly using the raw trajectory coordinates as input is impractical. To address this, we design features illustrated in Figure 2c that effectively describe a player's aiming trajectory. These features must be uncorrelated with the actual spatial distribution of coordinates while



(a) Game world aiming trajectory

(b) Monitor screen aiming trajectory

Figure 4: The example of 3D trajectory (a) and 2D screen trajectory mapping (b). The enlarged 3D trajectory (a-right) uses different measurement units for each coordinate axis for better illustration. 2D trajectory visualization is mapped onto a 1920×1080 resolution, and the coordinates of the elimination event are relocated to the center of the screen.

providing a standardized, quantifiable representation of the information contained in the aiming trajectory. These features include the instantaneous velocity and acceleration of trajectory points in the x- and y-directions, as well as the angular change in velocity direction.

Velocity ($\vec{v}$) is calculated as the rate of change of position with respect to time. Its components in the x and y directions are computed using the differences in the x and y coordinates divided by the differences in the *tick* values shown in Equation 1 and Equation 2.

$$\vec{v}_x = \frac{dx}{dt} \tag{1}$$

$$\vec{v}_y = \frac{dy}{dt} \tag{2}$$

Acceleration ($\vec{a}$) is calculated as the rate of change of velocity with respect to time. The acceleration components in the x and y directions are computed using the differences in the velocity components divided by the differences in the *tick* values shown in Equation 3 and Equation 4.

$$\vec{a}_x = \frac{d\vec{v}_x}{dt} = \frac{d^2x}{dt^2} \tag{3}$$

$$\vec{a}_y = \frac{d\vec{v}_y}{dt} = \frac{d^2y}{dt^2} \tag{4}$$

Angle of the velocity (α) is calculated using the 2-argument arctangent, which calculates the angle between the positive x-axis and the point given by the coordinates $(\vec{v}_x, \vec{v}_y)$ shown in Equation 5. **Angular change** (θ) is calculated as the difference between the above consecutive angle values. This represents the change in the direction of the velocity vector over time shown in Equation 6.

$$\alpha = \arctan 2(\vec{v}_y, \vec{v}_x) \tag{5}$$

$$\theta = \frac{d\alpha}{dt} \tag{6}$$

Figure 2c illustrates an exemplary case. The two diamond-shaped points represent the trajectory positions at *ticks*, t_0 and t_1. The solid red line represents the segment of the aiming trajectory between these two points, while the dashed red line represents the trajectory before and after these points. At *tick* t_0, the instantaneous velocity is $\vec{v}_0$, with its direction determined by the line connecting the previous trajectory point to t_0. The velocity components in the x- and y-directions are then computed, allowing us to further calculate the angle. The same process applies to $\vec{v}_1$ at t_1. Using the instantaneous velocities at t_0 and t_1, we can compute the angular change θ_1 between them. Furthermore, we can calculate the instantaneous acceleration and its corresponding components in the x- and y-directions. It is important to note that since computing the above features requires information from the previous time step, we always discard the first data tuple when calculating features, as its values are not applicable.

4.1.5 ML-ready data

After the aforementioned preprocessing, the dataset constructed and utilized in this paper follows the tuple structure $[t, I_f, I_e, \vec{v}_x, \vec{v}_y, \vec{a}_x, \vec{a}_y, \theta]$, where t represents the absolute *tick* of the observation, I_f and I_e are Boolean values indicating whether the player is firing (I_f) or conducting an elimination (I_e). The remaining values correspond to the features described in the previous section. Each valid tuple will be combined into an array to become a sample (i.e., an elimination window) and used as input to XGUARDIAN.

4.2 INSPECTOR

INSPECTOR's architecture is shown in Figure 5a. INSPECTOR processes samples composed of $m+n$ ML-ready tuples, as described in Section 4.1.2 and Section 4.1.5. We empirically set the size of each elimination window to 96, with the parameter selection process detailed in Section 5.4. A sliding window mechanism is employed to segment each input sample into $m+n-w+1$ sub-sequences, where w denotes the size of the sliding window. Based on empirical evaluation, we set $w = 6$, and the selection rationale is also discussed in Section 5.4.

GRU is a well-established recurrent architecture known for its training efficiency and competitive performance relative to LSTM [24], while CNNs are effective for pattern recognition and classification tasks. Each sub-sequence is processed by a hybrid architecture consisting of stacked GRU and CNN layers, as illustrated in Figure 5, which jointly perform temporal embedding and local pattern classification. This GRU-CNN design demonstrates superior performance over alternative baselines, as presented in Section 5.4. For each elimination window, the model produces $m+n-w+1$ predictions—one for each sub-sequence generated by the sliding window. These

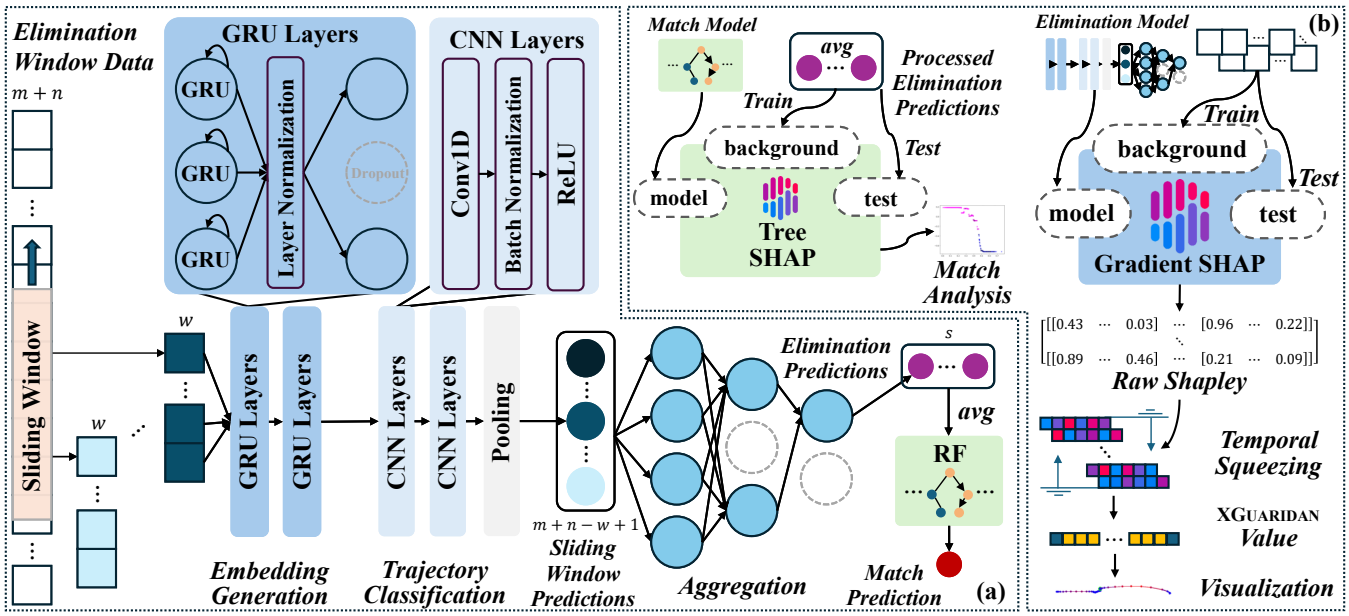


Figure 5: XGuardian framework. The subfigures illustrate the architectures of (a)-Inspector and (b)-Explainer.

predictions are then embedded and fed into stacked fully connected layers, with dropout applied to reduce overfitting. Assume a player has s elimination windows within a match. The s sets of predictions are averaged and passed into a random forest classifier. We adopt a random forest due to its flexible decision boundaries, which are well-suited to handling the nonlinear structure of the aggregated prediction embeddings, enabling robust and dynamic classification. Its final binary output, indicating whether the player is cheating, constitutes the decision produced by XGuardian.

4.3 Explainer

We choose SHAP over other candidates below for Explainer considering its (a) loose coupling and (b) explanation stability. Attention weight is model-specific and often correlates loosely with causal feature importance [45]. LIME [40] can be unstable and inconsistent due to random sampling, especially in time-series data with nonlinear dependencies and strong feature interactions (e.g., FPS games). Instead, SHAP's additive decomposition compares the influence of temporal features across model variants in unified frameworks. Its model-agnostic nature provides loose coupling with the detection framework and thereby makes it easy to evaluate or replace different detection architectures. Explainer's architecture is shown in Figure 5b. Explainer consists of two main components: the *Elimination Trajectory Explainer* (Figure 5b left) and the *Match Explainer* (Figure 5b right). The *Elimination Trajectory Explainer* analyzes the feature importance contribution for each *tick* within an input elimination window. Meanwhile, the *Match Explainer* provides an interpretation of the decision regarding whether a player engaged in cheating during a match.

Both explainers leverage *GradientExplainer* or *TreeExplainer* from SHAP, as introduced in Section 2.4. These explainers are initialized using the trained model and the training dataset (serving as the background for baseline comparison). We then apply the explainers to instances from the test dataset to compute their corresponding Shapley values. However, since SHAP is primarily designed for structured data, it cannot be directly applied to time-series trajectory data, particularly in the context of the *Elimination Trajectory Explainer*. To address this, we propose an interpretability enhancement method specifically tailored for temporal data. For each sample, every feature at each time step within a sliding window receives corresponding Shapley values. Time steps at the boundaries of an elimination window (i.e., the first and last) yield only one set of Shapley values, while intermediate time steps, due to overlapping sliding windows, may produce between two and six sets. After applying *GradientExplainer*, the resulting Shapley values for each elimination window form a matrix of shape $(m+n, f)$, where f is the number of features. Since each sliding window is treated with equal importance in explaining elimination trajectories in an anti-cheat scenario, we introduce a mechanism called *temporal squeezing*. *Temporal squeezing* aggregates the Shapley values for each feature across all overlapping time steps in the elimination window and computes their mean, producing a set of values referred to as XGuardian *values*. These values quantify feature importance at each time step and address the challenge of applying SHAP to temporal data in FPS anti-cheat applications. Equation 7 formalizes the computation

of the XGUARDIAN *value* under various conditions within an elimination window, where $SP(f)$ denotes the Shapley value of feature f at a given time step i. The XGUARDIAN *value* represents the average marginal contribution of a feature toward the model's prediction. A higher XGUARDIAN *value* indicates a greater contribution to a positive prediction (i.e., cheating), whereas a lower or negative value implies influence toward a non-cheating prediction. For the *Match Explainer*, we adopt a simpler approach by directly using SHAP's built-in Shapley values for match-level interpretation and visualization. Section 5.7 presents and visualizes both the trajectory-wise and match-wise explanation results.

$$\mathbf{V}_{\text{XGUARDIAN}}(f,i) = \begin{cases} SP(f), & i = 1 \text{ or } i = m+n \\ \frac{\sum SP(f)}{i}, & 2 \le i \le w \\ \frac{\sum SP(f)}{w}, & w+1 \le i \le m+n-w \\ \frac{\sum SP(f)}{m+n-i}, & m+n-w+1 \le i \le m+n-1 \end{cases} \tag{7}$$

5 Experiments

5.1 Experimental Setups

5.1.1 Dataset Construction

Data Collection. This study is conducted in collaboration with *5EPlay*[1], a world-class CS2 gaming platform. Players engage in regular gameplay on the platform, and after each match, *demos* are automatically generated. The proprietary anti-cheat system is used only for preliminary filtering and does not directly determine the final labels. Each player is labeled to indicate their cheating status per match. All labels are manually re-verified before being shared with us. Consistent with industry standards, labels are binary (i.e., cheater/normal player) and do not distinguish between specific cheat subtypes. We released all raw *demos* used in this study to ensure long-term reproducibility. A carefully curated dataset, representing the most accurately labeled data currently achievable in the industry, is then used for our evaluations.

Demo Parsing. We use an open-source library, *awpy2* [59], which extracts raw time-series data from the provided CS2 *demos* and parses them into tabular data.

Trajectory Extraction. In terms of CS2, we use its default resolution 1920 × 1080 as Algorithm 1's inputs. Noted that the default resolution is used for normalizing and visualizing the trajectory. Different resolutions on the client side would not affect the server-side trajectory conversion, since we are using the server-side pitch-yaw for the reconstruction.

Data Cleansing. When constructing the dataset as described in Section 4.1.4, we filter out certain edge cases to minimize noise. First, data from players who have no eliminations are removed, as it is impossible to extract elimination windows from such cases. Second, elimination windows with a length shorter than $m+n$ are discarded. This rare condition arises when, within n ticks after an elimination, one of the following occurs: (1) the player disconnects from the server, (2) the match ends immediately, and (3) the match enters a side-switch phase. Additionally, within the elimination window, we filter out cases where: (4) a server glitch led to missing or duplicated tuples. The first three cases prevent the player from moving their avatar's view within n ticks after the elimination, while the last one disrupts the temporal sequence of the data. Both could contaminate the dataset and thereby be discarded.

Dataset Description. Table 2 provides a description of the dataset we constructed, including three data splits. When we obtain the *demos*, we randomly divide all demos into three equal parts and apply the aforementioned preprocessing steps. This process results in the three data splits, which are used as training, validation, and test sets in the following evaluations. We publicize our repository and dataset. To our knowledge, this is the first publicly available large-scale aiming trajectory dataset for aim-assist cheats detection obtained from a real-world active modern FPS game.

Table 2: Dataset description.

Data Split	A	B	C
#Tick	992,064	1,014,624	1,062,528
#Total Elimination Window	10,334	10,569	11,068
#Cheating Elimination Window	2,385	2,760	2,900
#Normal Elimination Window	7,949	7,809	8,168
#Match	971	944	988
#Cheater	108	115	127
#Normal Player	1,711	1,671	1,754
#Total Player	1,819	1,786	1,881

5.1.2 Training Configurations

INSPECTOR's hyperparameters and configurations for the models are listed in Appendix B. The GRU-CNN model is trained with parallelized computation using the mirrored strategy. Three callbacks are employed during training to improve performance and generalization: (1) *ModelCheckpoint*, which saves the model with the lowest validation loss, (2) *ReduceLROnPlateau*, which reduces the learning rate by half if the validation loss plateaus (with a patience of 20 epochs and a minimum learning rate of 0.0001), and (3) *EarlyStopping*, which halts training if no improvement is observed after 50 consecutive epochs to prevent overfitting. The aggregation model also utilizes the same *EarlyStopping* mechanism for consistency. Both models are optimized using the Adam optimizer and trained with a *binary crossentropy* loss function. We use a batch size of 256 and train for up to 500 epochs, applying class weight balancing to address the class imbalance inherent in the cheating detection task. The training and validation loss curves are provided in Appendix C.

[1] 5EPlay: https://arena.5eplay.com/.

5.2 Detection Performace

Can the INSPECTOR in XGUARDIAN effectively identify cheaters? To evaluate XGUARDIAN's anti-cheat performance in a real-world setting, we conduct experiments using the collected dataset. We apply 3-fold cross-validation to assess XGUARDIAN's effectiveness and robustness. Table 3 presents the overall anti-cheat performance of XGUARDIAN. The three data splits are used as the training, validation, and test sets in different orders to eliminate any potential bias that might arise from specific dataset compositions. Since the dataset has an imbalanced distribution of cheaters and legitimate players, all evaluation metrics are computed using weighted averages to more accurately reflect the system's real-world performance. A detailed introduction to the metrics is presented in Appendix A. In Table 3, we observe that XGUARDIAN achieves high average performance across all metrics, with low variance and standard deviation, indicating that it maintains stable performance across different data splits. In the following evaluations, we use the data split combination of ABC as default for the training, validation, and test sets, respectively.

Table 3: XGUARDIAN weighted average overall performance with different set combinations. **Blue**, **green**, and **red** in the combination rows indicate the use of the set as training, validation, and test set, respectively.

		Accuracy(↑)	FPR(↓)	Recall(↑)	Precision(↑)	F1-Score(↑)
Combination	ABC	0.889	0.062	0.889	0.938	0.907
	ACB	0.876	0.049	0.876	0.951	0.901
	BAC	0.882	0.101	0.882	0.899	0.890
	BCA	0.907	0.041	0.907	0.959	0.924
	CAB	0.872	0.050	0.872	0.950	0.898
	CBA	0.879	0.048	0.879	0.952	0.904
mean (μ)		0.884	0.059	0.884	0.942	0.904
var (σ^2)		1.32E-04	4.00E-04	1.32E-04	4.00E-04	1.08E-04
std (σ)		1.15E-02	2.00E-02	1.15E-02	2.00E-02	1.04E-02

5.3 Feature Ablation Study

Can the features used in XGUARDIAN be further simplified or improved? As discussed in Section 4.1.5, XGUARDIAN relies on a set of input features. Among these features, I_f and I_e represent whether the player fired or achieved an elimination at a given tick. Both features are critical to aiming trajectory to representation and therefore cannot be further modified or removed. To verify the remaining features constitute the optimal configuration, we conduct ablation experiments (Table 4) and comparative experiments (Table 5).

Table 4 presents the results of the feature ablation study of XGUARDIAN. XGUARDIAN demonstrates consistently superior performance across all metrics compared to individual feature combinations (first three rows). The results of pairwise feature combinations (next three rows) indicate that no single combination achieves optimal performance across all metrics, and this holds when compared to XGUARDIAN as well. For example, while the combination of velocities and angle change yields the highest accuracy, its FPR and precision are worse than those of the other two combinations. Therefore, to comprehensively evaluate the performance of XGUARDIAN against pairwise combinations, we need to compare the average performance across all pairwise combinations with XGUARDIAN. By averaging the results of the pairwise combinations, XGUARDIAN outperforms the average performance of all pairwise combinations across all metrics. Consequently, we choose to utilize the full set of features.

Table 5 presents the comparative results of different modifications or removals of the tick feature. Specifically, the raw tick index refers to the original tick values from the dataset (e.g., within a given elimination window, raw tick indices might range from 5000 to 5095); the reset tick index replaces the raw tick index with a fixed range of 0 to 95. Evaluation results show that using the raw tick index outperforms the alternative approaches. This is because some cheaters exhibit delayed activation behaviors, where cheats are enabled only after falling behind during a match. Using raw ticks preserves such mid-match temporal context, allowing the model to better capture phase-dependent behavioral shifts.

Table 4: Feature ablation study weighted average results.

Feature			Metrics				
$\vec{a}_x$,$\vec{a}_y$	$\vec{v}_x$,$\vec{v}_y$	θ	Accuracy	FPR	Recall	Precision	F1-Score
✓	✗	✗	0.838	0.059	0.838	0.941	0.873
✗	✓	✗	0.884	0.060	0.884	0.940	0.904
✗	✗	✓	0.877	0.057	0.877	0.943	0.900
✓	✓	✗	0.886	0.062	0.886	0.938	0.905
✓	✗	✓	0.880	0.058	0.880	0.942	0.902
✗	✓	✓	0.901	0.063	0.901	0.937	0.915
✓	✓	✓	0.890	0.060	0.890	0.940	0.908

Table 5: XGUARDIAN input feature *tick* indexing comparative analysis weighted average results. 'w/o' and 'w/' denote without and with. 'Reset index' indicates re-allocating each elimination window's tick index from 0 to $m+n-1$.

Tick Indexing	Accuracy	FPR	Recall	Precision	F1-Score
w/o tick index	0.801	0.078	0.801	0.922	0.846
w/ reset index	0.784	0.073	0.784	0.927	0.834
XGUARDIAN (w/ raw index)	**0.889**	**0.062**	**0.889**	**0.938**	**0.907**

5.4 INSPECTOR Model and Parameters

Are the models and parameters in XGUARDIAN optimal for the aim-assist cheats detection? To justify our choice, we evaluated the performance of XGUARDIAN using different models and parameters. Recapture in Section 4.1.2, Section 4.2, Figure 3, and Figure 5, XGUARDIAN selects an elimination window of size $m+n$, capturing m ticks before and n ticks after the elimination tick. We investigate the impact of the elimi-

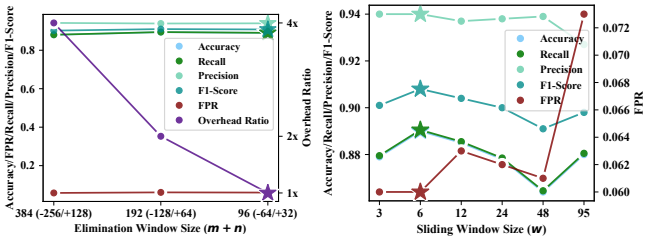
nation window size on XGUARDIAN's performance in Section 5.4.1. XGUARDIAN employs a length of w sliding window for segmenting elimination information. We discover the different performance of XGUARDIAN with different lengths of the sliding window in Section 5.4.2. XGUARDIAN leverages the GRU-CNN model for the embedding generation and trajectory classification. We evaluate the different performance of XGUARDIAN with different models in Section 5.4.3 and Section 5.4.4. XGUARDIAN integrates the elimination predictions through the averaging operation to integrate and generate a match prediction. We investigate the influence of using different representations of elimination predictions embedding on the performance of XGUARDIAN in Section 5.4.5.

5.4.1 Different Elimination Window Sizes

In Figure 6a, variations in the elimination window size have a minimal effect on cheat detection performance (a maximum 0.4% difference between the optimal and the chosen value). Considering that larger windows exponentially increase system overhead (including storage, computation, memory, and processing time for preprocessing, training, and inference), we choose a window size of 96 as XGUARDIAN's parameter. Since CS2's average time-to-kill is approximately 0.5 seconds, a 96-tick (≈ 1.5 seconds) window ensures full coverage of pre-aiming, firing, and elimination behaviors. Smaller windows would truncate critical firing events and thus lead to the loss of essential trajectory information.

5.4.2 Different Sliding Window Sizes

In Figure 6b, for different evaluation metrics, we empirically obtain that the optimal sliding window size is 6.

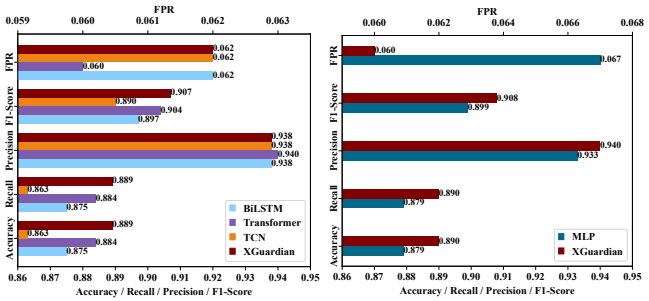


(a) Elimination window (b) Sliding window

Figure 6: INSPECTOR sliding window and elimination window size comparative analysis weighted average results. The star (⋆) indicates the parameters used by XGUARDIAN.

5.4.3 Different Models in Embedding Generation

As shown in Figure 7a, we compare various state-of-the-art time-series models, BiLSTM [64], Transformer [55], and TCN [3]. While the Transformer achieves a slightly higher precision than the GRU used in XGUARDIAN (by 0.2%),



(a) Embedding generation stage (b) Trajectory classification stage

Figure 7: INSPECTOR embedding generation and trajectory classification stages model selection weighted average results across SOTA models [3, 42, 55, 64].

XGUARDIAN outperforms all other models across the remaining metrics. Additionally, the computational overhead of the Transformer is significantly higher than that of GRU [46]. Therefore, we retain the current GRU-based structure in the embedding generation stage of XGUARDIAN.

5.4.4 Different Models in Trajectory Classification

As shown in Figure 7b, we compare the CNN used in XGUARDIAN with MLPs [42] (i.e., fully connected layers). The result demonstrates that XGUARDIAN achieves the best performance across all evaluation metrics.

5.4.5 Different Input Styles in Elimination Predictions

In Table 6, we compare all possible representations of elimination predictions. Among these, *binary* refers to the binary classification of elimination predictions. Specifically, the binary classification is determined by a dynamic threshold, the value of which is learned from the training and validation sets. The results show that using the original predicted values as input to the match prediction model yields better overall performance than the other combinations.

Table 6: INSPECTOR elimination predictions stage embedding style comparison. 'Binary' denotes embedding binary classified predictions as match prediction's input. 'Original' denotes embedding raw predictions as match prediction input. XGUARDIAN's configuration is bolded.

Input Style	Accuracy	FPR	Recall	Precision	F1-Score
Binary	0.881	0.061	0.881	0.939	0.902
Binary+Original	0.881	0.061	0.881	0.939	0.902
XGUARDIAN (Original)	**0.889**	**0.062**	**0.889**	**0.938**	**0.907**

5.5 Comparative Study

Is XGUARDIAN performing better compared to prior works? We compare XGUARDIAN with seven other server-side de-

tection methods, including comparisons with two other reproduced methods using the dataset utilized by XGUARDIAN. Due to the lack of available open-source solutions, we reproduced two server-side aim-assist cheat detection methods based on simple statistical features related to the aiming or elimination used in prior works. Both methods are related to XGUARDIAN but use non-temporal features, data, and approach. Below is a summary of each method:

- TH_HITACC: Filters cheaters based on the weapon accuracy of the player, using a threshold [2, 22].
- TH_ACCA: Filters cheaters when the player's field-of-view acceleration exceeds a threshold [62].

Furthermore, we compare XGUARDIAN with the state-of-the-art server-side solution, HAWK, and its three subsystems using the aimbot dataset provided by the same solution [63]. Below is a summary of the method and its sub-systems:

- REVPOV: Uses LSTM to classify a wide range of temporal operation data of the player for detecting cheaters [63].
- REVSTATS: Designs structured features to represent the player's behavior and classify with ensemble learning [63].
- EXSPC: Uses deep learning to examine the consistency of a player's gaming sense and their combat performance [63].
- HAWK: Combines the judgments from the above three subsystems and dynamically classifies cheaters using a threshold that can quickly adapt to anti-cheat requirements [63].

The results are shown in Table 7. XGUARDIAN outperforms previous works on both datasets. The reproduction results of the prior works are aligned with their paper performance (if applicable on the large dataset). A direct comparison of HAWK with XGUARDIAN's dataset is infeasible, as HAWK's features are highly game-specific, as discussed in Section 1, and some of the raw data (e.g., whether the opponents are in sight) required for feature extraction are not available in CS2's *demo*.

Table 7: XGUARDIAN comparative study to prior works using different data sources. The optimal results are bolded.

Dataset	Method	Accuracy	FPR	Recall	Precision	F1-Score
XGUARDIAN	TH_ACCA [62]	0.738	0.455	0.738	0.545	0.627
	TH_HITACC [2, 22]	0.644	0.086	0.644	0.914	0.711
	XGUARDIAN	**0.889**	**0.062**	**0.889**	**0.938**	**0.907**
HAWK [63]	REVPOV [63]	0.616	0.151	0.616	0.849	0.694
	REVSTATS [63]	0.814	0.109	0.814	0.891	0.842
	EXSPC [63]	0.793	0.114	0.793	0.886	0.827
	HAWK [63]	0.716	0.112	0.716	0.888	0.772
	XGUARDIAN	**0.841**	**0.056**	**0.841**	**0.944**	**0.878**

5.6 Generalizability Study

Can XGUARDIAN be generalized to different games developed with different game engines? Table 8 demonstrates XGUARDIAN's generalizability evaluation results. We observe that XGUARDIAN's performance remains consistent on different subgenres of games, engines, and platforms. Note that even though CS:GO and CS2 are from the same series, they use entirely different engines with different replay systems and parsers, and thereby should be treated as two different games. We note a slight decrease in the effectiveness of XGUARDIAN on the Farlight84 dataset due to the fact that the precision of its playback system (i.e., the frequency of operations recording) is much lower than that of the CS2's (approximately half that of the CS2's). Therefore, the trajectory description is vaguer than the CS2 dataset and thereby leads to a decrease. Farlight84 data is provided by its developer *Lilith Games*[2]. XGUARDIAN's performance slightly decreases on HAWK's dataset due to the existence of inaccurate negative labels in the dataset, as mentioned by their authors. Overall, the performance of XGUARDIAN in this experiment is acceptable in terms of evaluating the generalizability.

Table 8: XGUARDIAN's generalizability evaluation results on different sub-genre games, platforms, and engines. The descriptions of the other two datasets are shown in Appendix D.

Dataset	HAWK [63]	FARLIGHT84	XGUARDIAN
Game (Release Year)	CS:GO (2012)	Farlight84 (2023)	CS2 (2023)
Sub-genre	Tactical Shooter	Battle Royal	Tactical Shooter
Engine	Source	Unreal Engine 4	Source 2
Platform	PC	Mobile	PC
Accuracy	0.841	0.816	0.889
FPR	0.056	0.144	0.062
Recall	0.841	0.816	0.889
Precision	0.944	0.856	0.938
F1-Score	0.878	0.829	0.907

5.7 Explainability Case Study

The elimination-scope cases analyze how XGUARDIAN arrives at individual elimination predictions. The match-scope case illustrates how XGUARDIAN generates a final match-level prediction, using a representative cheater case for demonstration. Video demonstrations and interactive analyses for all case studies are available on our project website[3]. To ensure the selected cases are representative and insightful, they are curated by two FPS game experts whose skill levels are detailed in Table 9. Case A, C, and E are blatant cheaters; Case D is a disguised cheater; Case B is a normal player. A detailed discussion on the cases can be found either on the website or in Appendix E. Particularly, in Case D, the cheater only employs wallhack. Legitimate players scan and react to visual cues. A wallhack user, however, knows an opponent's location in advance. XGUARDIAN detects wallhacks by identifying the anomalous aiming behavior they produce. This often manifests as low-valued features just before an engagement; the cheater simply waits for the target to cross their pre-positioned crosshair. XGUARDIAN detects this unnatural lack of movement, which is counterintuitive compared to high-

[2]Lilith Games: https://ancient.lilith.com/.
[3]Case study website: https://xguardian-anti-cheat.github.io/.

valued movements while using aimbots, but is an indicative signal of wallhacking.

Table 9: Experts' skill level for case study selection and review, adversarial sample selection, and user study case selection. All stats are retrieved from [52].

Expert	Playtime	Skill Level
A	>7,000hrs	CS2 - *Premier* 19,000 (Top 2.62%) Overwatch2 - *Role* Grand Master (Top 1.46%) Apex Legend - *BR* Master (Top 0.6%)
B	>5,000hrs	CS2 - *Premier* 18,000 (Top 4.59%)

5.8 Robustness Under Adversarial Attack

Suppose that cheaters know XGUARDIAN's mechanism and attempt to evade detection, will XGUARDIAN be dysfunctional? Theoretically, cheaters must fundamentally alter their cheating behaviors, such as prolonging aim-locking durations, smoothing out jerky movements, or avoiding unnaturally perfect pre-aiming. These lead to two outcomes: (1) If they mimic normal behavior but still desire illegal advantages, as in Case A and D, they will still be detected by XGUARDIAN; (2) If they perfectly mimic legitimate player behaviors, the cheat itself becomes ineffective, as normal players can outperform '*normal players*' with such limited assistance. This defeats the purpose of cheating. This is a cornerstone of behavioral-based security and has been noted in prior works [7, 63].

To validate XGUARDIAN's robustness in detecting cheaters who attempt to conceal their behavior by mimicking legitimate players, we first computed the mean values of each feature for all non-cheaters on the test set. These averages served as a baseline representation of normal gameplay behavior. Next, for each cheater per match, we evaluated whether its feature value fell within ±10% of the corresponding non-cheating averages. The final adversarial dataset contains cheaters per match that meet this similarity condition under one or more features in the test set. We further discuss the parameter selections in Appendix F. We have collected 22 cheaters with 54,816 ticks of data. We further invite the two game experts to verify and confirm the stealthiness of all selected samples. XGUARDIAN identifies 18 among 22 cheaters, achieves 0.818 recall and 0.900 F1-Score. Both the two-proportion Z-test [37] and Fisher's exact test [19] indicate that the observed decrease in recall under adversarial attack is statistically insignificant ($p > 0.05$). Note that due to our threat model settings, we are unable to acquire the cheater's specific (a) cheat functions and (b) parameter configurations.

5.9 Overhead

XGUARDIAN is designed as an asynchronous system that analyzes post-game *replay files* during periods of low server activity. It operates independently of active game servers and can be deployed on a separate server. We conduct overhead assessments for reference. We utilize an Ubuntu 22.04.5 LTS server equipped with 24-core Intel(R) Xeon(R) Gold 5418Y and 503 GB RAM, a real dedicated game server configuration [14]. We use two NVIDIA GeForce RTX 4090 (24GB) for the model training and testing. XGUARDIAN can train with one GPU with over 12 GB memory, but we adopt the mirror strategy in all experiments for synchronous distributed training. Leveraging the pre-trained model, XGUARDIAN requires only CPU resources to function efficiently. Table 10 shows the overhead statistics on the entire dataset. We further evaluate XGUARDIAN's server overhead in a real-world deployment scenario. Based on data from our partner platform, the system handles a maximum of 150,000 matches daily, with each individual server processing up to 573 matches per day. Analysis of platform metrics and public server statistics [48] reveals that player activity reaches its lowest point between 1:00 AM and 5:00 AM local time. During this off-peak window, the system must process a maximum of 144 matches per hour. Our performance measurements indicate that XGUARDIAN can complete all required analyses within this timeframe. Based on the statistics in Table 10, the total prediction time for the entire dataset (2,903 matches across three splits from Table 2) is 28,966.87 seconds, so it takes only less than 9.98 seconds of prediction time per match with CPU-only. This represents a remarkable improvement of 4,104.31% compared to the state-of-the-art server-side solution HAWK [63], which requires over 400 seconds. Furthermore, XGUARDIAN demonstrates superior memory efficiency, consuming less than 8.6GB RAM for the entire dataset, whereas HAWK demands over 5GB RAM per single match. Therefore, each server has more than sufficient resources for deploying XGUARDIAN.

Table 10: XGUARDIAN's overheads tested with the entire dataset. The numbers are retrieved by the Python libraries time [21] and resource [20].

Overheads	Preprocessing	Prediction	Training
User time (sec)	52.4	28966.87	28960.42
System time (sec)	30.21	2905.86	2905.14
Elapsed time (sec)	48.03	3.95	11423.11
Max. resident set size (GB)	3.226	8.596	8.596

5.10 User Study on EXPLAINER's Bias

Will EXPLAINER-generated explanations introduce decision-making bias in human reviewers, potentially increasing false positive judgments in cheat detection? To answer this research question, we conduct a user study including four former reviewers and six cases. In terms of the case selection, we select cheaters with similar cheating sophistications to compare the decision correctness and time efficiency with and without the use of XGUARDIAN. 1A and 1B are blatant cheaters

Table 11: User study results of the EXPLAINER's potential bias questionnaire.

Reviewer	Decision[1]						Confidence[2]						Time Consumption					
	1A	1B	2A	2B	3A	3B	1A	1B	2A	2B	3A	3B	1A	1B	2A	2B	3A	3B
#1	1	1	1	1	1	1	5	5	5	4	5	5	3′36″	1′45″	4′14″	1′08″	2′54″	1′03″
#2	1	1	1	1	**0**	1	5	5	4	5	4	3	1′17″	0′40″	4′59″	2′31″	3′59″	2′43″
#3	1	1	1	1	1	1	5	5	4	5	4	3	0′37″	1′19″	4′27″	2′22″	7′45″	7′04″
#4	1	1	1	1	1	1	5	5	5	5	5	5	2′30″	0′14″	2′49″	0′34″	2′47″	1′49″

[1]**Decision**: 1 and **0** denote the review's decision is correct and incorrect, respectively; [2]**Confidence**: **1**, **2**, **3**, **4**, **5** represent reviewer's decision confidence from the least confident to the most confident; [3] *Grey background* denotes the current case is applying XGUARDIAN;

(1A and 1B are Case C and Case E in the case study); 2A and 2B are disguised cheaters (2A is newly selected, 2B is Case D in the case study); 3A and 3B are seasoned players that are misclassified by XGUARDIAN, i.e., the false positives (both cases are newly selected). The case selections are conducted by two experts listed in Table 9. The selected cases are reviewed by the reviewers following the procedures in Table 20. The reviewer's skill level is listed in Table 12. The user study results on this research question are demonstrated in Table 11. Based on the results, we observe that a reviewer without XGUARDIAN makes an incorrect decision. Reviewer's confidences on the false positives might be influenced by XGUARDIAN (lower confidence on 3B). However, their decisions are not misdirected (correct decisions on 3B). XGUARDIAN primarily accelerates the review process by reducing the time consumption for each case. Reviewers with lower confidence tend to spend more time reviewing; with XGUARDIAN, this additional time caused by lower confidence is effectively eliminated, resulting in faster review times compared to cases with higher-confidence reviewers.

Table 12: User study reviewers' CS2 skill level, where *5EPlay* [1] and *Perfect World* [39] are the platform (introduced in Ethical Considerations) ranks, *Premier* is the official rank. All invited reviewers are the former reviewers of CS:GO Overwatch Investigators [53].

Reviewer	CS2 Skill Level	Playtime (hrs)
#1	*5EPlay* - **A**, *Perfect World* - **S**	3,563
#2	*5EPlay* - **A**, *Perfect World* - **A**, *Premier* - 18,491	2,020
#3	*5EPlay* - **A**, *Perfect World* - **S**, *Premier* - 19,486	2,400
#4	*5EPlay* - **A+**, *Perfect World* - **S**, *Premier* - 20,000	2,917

6 Discussion

Data Collection Generalizability. While most FPS games incorporate replay systems, certain niche titles or those developed by smaller studios may lack this functionality. However, since XGUARDIAN only requires pitch and yaw data, which are fundamental parameters that all FPS games must track to build the game (i.e., the control of the in-game avatar's view directions), developers can easily collect them for XGUARDIAN as input. Notably, XGUARDIAN is game-agnostic, and any FPS game providing temporal pitch and yaw data can leverage the system for subsequent feature extraction, model training, cheat detection, and interpretable analysis.

Reducing and Expediting Review. Given the current anti-cheat development, manual review of the positive suspects is inevitable across all FPS games [63]. Given that the in-game paid items are tied to a player's account, a ban should not be issued solely based on the anti-cheat system decision. It is imperative to provide substantial evidence before initiating such an action. Therefore, the final decision requires manual review to avoid false bans. According to the statistics of our partner platform, their in-use proprietary anti-cheat obtains 47.5% recall and 10.1% FPR after the manual re-verification on the label. In contrast, XGUARDIAN consistently outperforms both the industrial baseline and prior works in Section 5.5 across all evaluation metrics. Regarding manual involvement, since XGUARDIAN reduces the FPR from 10.1% to 5.9% (Table 4), this corresponds to a 1.71× reduction in the number of false positives that require manual review, thereby substantially alleviating the human inspection workload. Additionally, XGUARDIAN shortens the review time on cheating samples by providing explainable analyses. Regarding the ability to identify cheaters, XGUARDIAN's recall excels among all tested and reported anti-cheat methods. Considering the worrying performance of the current anti-cheat in most FPS games [8, 27, 41, 51, 63], XGUARDIAN is able to significantly reduce the FNR and FPR at the same time.

Undetectable Cases. XGUARDIAN is designed around elimination events and therefore cannot detect players who do not secure kills. For example, purely support-oriented heroes, such as *Mercy* in Overwatch, may not contribute to eliminations, making XGUARDIAN inapplicable for determining whether they are cheating. However, since reducing the opposing team's player count is a necessary step toward victory in FPS games, cheating without contributing to eliminations has a comparatively limited impact on overall gameplay outcomes. In addition, if a cheater employs techniques that do not influence the aiming trajectory (e.g., temporarily toggling wallhacks during encounters, illegally sharing information with teammates), XGUARDIAN is not applicable as such behaviors do not affect the player's aiming trajectory; nevertheless, the vast majority of cheaters do not rely on these atypical

strategies. Regarding genre applicability, XGUARDIAN is theoretically applicable to any FPS game whose engine uses a pitch–yaw representation for avatar navigation to detect aim-assist–based cheats. If a game does not employ a pitch–yaw representation, XGUARDIAN cannot be directly applied; however, it can still be adapted if an alternative means of obtaining the player's aiming trajectory is available, by modifying the coordinate transformation in Algorithm 1.

7 Related Works

Statistics-based Detection. Yu *et al.* [61,62] detect aim-assist cheats using two statistical features, cursor acceleration and target lock duration, to distinguish between legitimate players and cheaters via the Kolmogorov-Smirnov test. However, the approach requires modifications to the client-side game engine, limiting its generalizability across different games. Moreover, their features are aggregated over entire matches rather than temporally fine-grained like XGUARDIAN's, making them less effective at capturing gameplay details. Additionally, since their detection operates solely on the client side, it remains vulnerable to memory tampering. Liu *et al.* [30] exploit behavioral discrepancies in cheaters, such as unusually low skill levels relative to their performance metrics (e.g., kill counts). However, their method struggles with highly skilled or motivated players, such as seasoned gamers carefully exploiting the aim-assist cheats, as well as sophisticated cheats (e.g., Cases A and D in our case study). Galli *et al.* [22] and Alayed *et al.* [2] propose supervised learning approaches that rely on complex features, such as aiming angles, rather than XGUARDIAN's more generalizable angular change metric. These features require modifications to the server-side game engine. For instance, aiming angles depend on target visibility, which varies with client-specific field-of-view settings or is not directly available for some games, such as PUBG. Consequently, these methods suffer from limited generalizability. Beyond these limitations, none of the aforementioned approaches have been validated on large-scale real-world datasets or have considered interpretability, leaving their practical effectiveness uncertain and unreliable.

Behavioral-based Detection. Server-side HAWK [63] and client-side BOTSCREEN [7] are the state-of-the-art solutions for aim-assist cheats detection. BOTSCREEN [7] leverages one temporal feature, aiming angles, which is similar to the one in the previous statistical methods [2, 22] but in a time-series manner. Therefore, the generalizability issue inherently exists. Besides, it is tested with a small dataset constructed by 14 hired players and implemented in a single game. Thus, its real-world performance and generalizability remain unclear. Additionally, it leverages unsupervised learning to detect anomalies with only the aforementioned normal players' data as model input, and is tested with only one open-source aimbot. Considering the diverse cheating sophistication discussed in Section 2.2 and demonstrated in the previous case studies, BOTSCREEN's real-world applicability remains questionable. HAWK [63] requires massive game-specific features (e.g., flash bang affection duration on the opponents and teammates, incendiary damages, etc.) as mentioned in Section 1 and Section 5.5. Therefore, their generalizability is confined to CS:GO, with some features failing to reproduce even on its successor, CS2. Importantly, its performance is lower compared to XGUARDIAN. AntiCheatPT [31] leverages LLMs for cheat detection. However, its overhead, generalizability, and deployability on the real game server remain unknown. Additionally, none of the above works are explainable.

Client-side Prevention. BlackMirror [38] leverages SGX to prevent wallhacks by supplying the GPU with only visible entities based on local predictions. However, it focuses solely on prevention without detection capabilities, and is limited by hardware requirements, overhead, and its narrow scope targeting a single cheat modality. Invisibility Cloak [49] thwarts computer-vision-based aimbots by obstructing object detection in screen captures, but remains ineffective against most aim-assist cheats that exploit memory access.

Industrial Proprietary Anti-Cheat. Commercial anti-cheat systems such as VAC [54], Vanguard [25], EAC [17], and BattlEye [4] provide both detection and prevention across various games via custom integrations [47]. These systems monitor process privileges, scan DNS caches and cookies for browsing histories and cheat purchase traces [29, 54], and utilize kernel-level drivers [41]. However, they raise significant security and privacy concerns due to hard drive scanning and root privilege requirements, which risk unauthorized access and data leakage [34, 35, 58]. Known vulnerabilities, including RCE exploits [6] and several CVEs [9–12], further expose users to system instability and privacy breaches [63]. These systems also rely heavily on cheat blacklists and malware signatures [7, 28, 30, 41], which are ineffective against novel or obfuscated cheats with altered signatures [7, 30, 63]. Modern paid cheat services frequently auto-update to evade detection, causing industry anti-cheats to exhibit low recall and lag behind emerging threats [63]. Moreover, commercial solutions are embedded pre-release and run during gameplay [41], introducing client-side overhead and limiting generalizability. While these systems often combine multiple detection strategies to cross-verify suspicious activity, this extends the ban cycle [41]. Finally, being proprietary and closed-source [41], they hinder independent research and comparative evaluations without direct collaboration with game developers.

Explainable Detection in Other Genres. Tao *et al.* proposed a four-pronged explainable design for detecting real money trading in MMORPGs like World of Warcraft [50]. The interpretability of their work on FPS games is limited to client-side screenshots' anomaly overlays identification (e.g., cheat's bounding box or UI). This may raise privacy concerns and can be easily bypassed by managing process privileges or disabling the visible overlays. Hence, it is less practical for real-world deployment.

8 Conclusion

We presented XGUARDIAN, a generalized, explainable, and efficient server-side anti-cheat system for detecting aim-assist cheats in FPS games. XGUARDIAN introduces novel detection features and a hybrid framework to effectively identify and explain cheating behaviors. We demonstrate that XGUARDIAN achieves state-of-the-art performance in reliably detecting aim-assist cheaters on different games.

Acknowledgments

We thank *5EPlay* and *Lilith Games* for sharing and labeling their raw data and their support for verifying ground truth labels. We appreciate the voluntary reviews from *Mr.K*, *z1an*, *mrh929*, and *Demons* (disclosed with their IDs) on the user study cases. We acknowledge the constructive review comments by the anonymous reviewers from USENIX Security and NDSS. This work is partially supported by the NSFC for Young Scientists of China (No.62202400) and the RGC for Early Career Scheme (No.27210024). Any opinions, findings, or conclusions in this paper are those of the authors and do not necessarily reflect the views of NSFC and RGC.

Ethical Considerations

Data Source and Acquisition. All data used in this study were provided by a third-party competitive gaming platform *5EPlay* that hosts CS2 servers independently of *Valve Corporation* (the game's developer). These servers operate under the platform's own user agreements and privacy policies, while authentication occurs via Steam. *5EPlay* directly provided the de-identified dataset to the research team, in compliance with their platform policies[4]. No private information or personally identifiable data was disclosed to the authors. Players' re-identifiable information was removed prior to data transfer.

Data Usage Agreement. The dataset is governed by the *5EPlay Personal Information Protection Policy*, which permits sharing de-identified information for academic research purposes conducted in the public interest. While the data usage agreement acknowledged by players was not specific to this research, it explicitly allows the platform to share anonymized gameplay data for research. All data were handled with strict confidentiality and in accordance with applicable legal and ethical standards.

EXPLAINER Potential Impact. EXPLAINER could theoretically influence reviewers' decisions, potentially increasing the risk of false decisions. To mitigate this risk, EXPLAINER is strictly a supportive interpretability tool intended to help experienced game masters (GMs) quickly identify anomalous gameplay segments. All decisions are made independently by trained GMs, and bans are only enforced when multiple reviewers reach the same conclusion, minimizing the influence of any single reviewer or algorithm. GMs should be trained to know that XGUARDIAN can make mistakes. Players affected by false positives retain the right to appeal, triggering review by a separate moderation team.

IRB Review. This research was reviewed and approved by the Human Research Ethics Committee (Reference Number: EA250702) at the author's affiliated institution. The study design, data acquisition, and handling procedures were deemed compliant with human subjects research standards.

[4] *5EPlay Personal Information Protection Policy*, translated by the authors, original in Simplified Chinese: https://arena.5eplay.com/page/privacy.

Open Science

We open-source XGUARDIAN's repository, datasets, and artifacts at https://doi.org/10.5281/zenodo.17845614, as well as its raw CS2 data listed in Table 13.

Table 13: XGUARDIAN's raw data.

Part	URL
1	https://doi.org/10.5281/zenodo.17838584
2	https://doi.org/10.5281/zenodo.17838763
3	https://doi.org/10.5281/zenodo.17838827
4	https://doi.org/10.5281/zenodo.17838865
5	https://doi.org/10.5281/zenodo.17839153
6	https://doi.org/10.5281/zenodo.17844922
7	https://doi.org/10.5281/zenodo.17844928
8	https://doi.org/10.5281/zenodo.17844938
9	https://doi.org/10.5281/zenodo.17844942
10	https://doi.org/10.5281/zenodo.17844950
11	https://doi.org/10.5281/zenodo.17844964
12	https://doi.org/10.5281/zenodo.17844969
13	https://doi.org/10.5281/zenodo.17844977

References

[1] 5EPlay. 5EPlay — Home. https://www.5eplay.com/, 2025. Accessed: 2025-12-08.

[2] Hashem Alayed, Fotos Frangoudes, and Clifford Neuman. Behavioral-based cheating detection in online first person shooters using machine learning techniques. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8, Niagara Falls, ON, Canada, Aug 2013. IEEE.

[3] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling, 2018.

[4] BattlEye Innovations. Interested in using BattlEye? https://www.battleye.com/, 2025.

[5] BleepingComputer. New intel chips won't play blu-ray disks due to sgx deprecation. https://www.bleepingcomputer.com/news/security/new-intel-chi

ps-wont-play-blu-ray-disks-due-to-sgx-deprecation/, 2022. Accessed: 2025-03-31.

[6] bright, IDontCode, irql0. EasyAntiCheat Exploit to inject unsigned code into protected processes. https://blog.back.engineering/10/08/2021/, August 2021. Accessed: 2025-04-21.

[7] Minyeop Choi, Gihyuk Ko, and Sang Kil Cha. Botscreen: trust everybody, but cut the aimbots yourself. In *Proceedings of the 32nd USENIX Conference on Security Symposium*, SEC '23, USA, 2023. USENIX Association.

[8] Sam Collins, Alex Poulopoulos, Marius Muench, and Tom Chothia. Anti-cheat: Attacks and the effectiveness of client-side defences. In *Proceedings of the 2024 Workshop on Research on Offensive and Defensive Techniques in the Context of Man At The End (MATE) Attacks*, CheckMATE '24, page 30–43, New York, NY, USA, 2024. Association for Computing Machinery.

[9] CVE-2019-16098. CVE-2019-16098: Vulnerability in MSI Afterburner. https://nvd.nist.gov/vuln/detail/CVE-2019-16098, 2019. Accessed: 2023-12-07.

[10] CVE-2020-36603. CVE-2020-36603: Vulnerability in Genshin Impact Anti-Cheat Software. https://nvd.nist.gov/vuln/detail/CVE-2020-36603, 2022. Accessed: 2023-12-07.

[11] CVE-2021-3437. CVE-2021-3437: HP OMEN Gaming Hub Privilege Escalation Bug Hits Millions of Gaming Devices. https://www.sentinelone.com/labs/cve-2021-3437-hp-omen-gaming-hub-privilege-escalation-bug-hits-millions-of-gaming-devices/, 2021. Accessed: 2023-12-07.

[12] CVE-2023-38817. CVE-2023-38817: Vulnerability in Minecraft Anti-Cheat Tool. https://nvd.nist.gov/vuln/detail/CVE-2023-38817, 2023. Accessed: 2023-12-07.

[13] DataHorizzon Research. Fps game market size, share, trends, and forecast 2025-2033, 2025. Accessed: 2025-04-01.

[14] DatHost. CS2 Server Hosting. https://dathost.net/cs2-server-hosting, 2024. Accessed: 2024-06-05.

[15] Delta Force. Delta force - g.t.i security, 2025. Accessed: 2025-02-18.

[16] Delta Force - G.T.I Security. Delta force - notice of penalties for violations, 2025. Accessed: 2025-02-18.

[17] Epic Games, Inc. Easy Anti-Cheat. https://www.easy.ac/, 2025.

[18] Exponential Era. The million dollar video game cheating market, 2024. Accessed: 2025-04-01.

[19] R. A. Fisher. On the interpretation of χ^2 from contingency tables, and the calculation of p. *Journal of the Royal Statistical Society*, 85(1):87–94, 1922.

[20] Python Software Foundation. *resource — Resource usage information*. Python Software Foundation, 2025. Accessed: 2025-04-11.

[21] Python Software Foundation. *time — Time access and conversions*. Python Software Foundation, 2025. Accessed: 2025-04-11.

[22] Luca Galli, Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. A cheating detection framework for unreal tournament iii: A machine learning approach. In *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, pages 266–272, Seoul, Korea (South), 2011. IEEE.

[23] Mee Lan Han, Jung Kyu Park, and Huy Kang Kim. Online game bot detection in fps game. In *Proceedings of the 18th Asia Pacific Symposium on Intelligent and Evolutionary Systems-Volume 2*, pages 479–491, Singapore, 2015. Springer.

[24] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[25] Riot Games Inc. What is vanguard? https://support-valorant.riotgames.com/hc/en-us/articles/360046160933-What-is-Vanguard-, January 2024.

[26] Intel Community. Rising to the challenge: Data security with intel confidential. https://community.intel.com/t5/Blogs/Products-and-Solutions/Security/Rising-to-the-Challenge-Data-Security-with-Intel-Confidential/post/1353141, 2022. Accessed: 2025-03-31.

[27] JARVIS THE NPC. Warzone Reddit Outcry: Anti-Cheat Failures Fuel Frustration Among Players. https://www.zleague.gg/theportal/warzone-reddit-outcry-anti-cheat-failures-fuel-frustration-among-players/, April 2024. Accessed: 2025-04-20.

[28] Anssi Kanervisto, Tomi Kinnunen, and Ville Hautamäki. Gan-aimbots: Using machine learning for cheating in first person shooters. *IEEE Transactions on Games*, 15(4):566–579, 2023.

[29] Samuli Lehtonen et al. Comparative study of anti-cheat methods in video games. *University of Helsinki, Faculty of Science*, 2020.

[30] Daiping Liu, Xing Gao, Mingwei Zhang, Haining Wang, and Angelos Stavrou. Detecting passive cheats in online games via performance-skillfulness inconsistency. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 615–626, Denver, CO, USA, 2017. IEEE.

[31] Mille Mei Zhen Loo, Gert Luẑkov, and Paolo Burelli. Anticheatpt: A transformer-based approach to cheat detection in competitive computer games. In *2025 IEEE Conference on Games (CoG)*, pages 1–4, 2025.

[32] Scott M. Lundberg, Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M. Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. From local explanations to global understanding with explainable ai for trees. *Nature Machine Intelligence*, 2(1):2522–5839, 2020.

[33] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017.

[34] Anton Maario, Vinod Kumar Shukla, A. Ambikapathy, and Purushottam Sharma. Redefining the risks of kernel-level anti-cheat in online gaming. In *2021 8th International Conference on Signal Processing and Integrated Networks (SPIN)*, pages 676–680, Noida, India, 2021. IEEE.

[35] Kevin Kjelgren Mikkelsen. Information security as a countermeasure against cheating in video games. Master's thesis, NTNU, 2017.

[36] Mobile Marketing Reads. Global gaming market will rebound to $250 billion in 2025, 2025. Accessed: 2025-04-01.

[37] Jerzy Neyman and Egon S. Pearson. On the problem of the most efficient tests of statistical hypotheses. *Philosophical Transactions of the Royal Society of London, Series A*, 231:289–337, 1933.

[38] Seonghyun Park, Adil Ahmad, and Byoungyoung Lee. Blackmirror: Preventing wallhacks in 3d online fps games. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 987–1000, New York, NY, USA, 2020. Association for Computing Machinery.

[39] Perfect World. Perfect World Esports. https://pvp.wanmei.com/, 2025. Accessed: 2025-12-08.

[40] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?": Explaining the predictions of any classifier, 2016.

[41] RYAN K. RIGNEY. The gamers do not understand anti-cheat. https://www.pushtotalk.gg/p/the-gamers-do-not-understand-anti-cheat, Febuary 2024. Accessed: 2025-04-20.

[42] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[43] scikit-learn developers. scikit-learn: Randomforestclassifier, 2025. Accessed: 2025-03-23.

[44] Secure Cheats. Call of duty: Warzone hacks, 2025. Accessed: 2025-02-18.

[45] Sofia Serrano and Noah A. Smith. Is attention interpretable?, 2019.

[46] Tang Shi and Kazuya Shide. A comparative analysis of lstm, gru, and transformer models for construction cost prediction with multidimensional feature integration. *Journal of Asian Architecture and Building Engineering*, 2025.

[47] José Nuno Silva. Towards automated server-side video game cheat detection. https://repositorio-aberto.up.pt/bitstream/10216/142935/2/572983.pdf, July 2022.

[48] SteamDB. Counter-Strike 2 Steam Charts. https://steamdb.info/app/730/charts/#48h, 2024. Accessed: 2024-06-05.

[49] Chenxin Sun, Kai Ye, Liangcai Su, Jiayi Zhang, and Chenxiong Qian. Invisibility cloak: Proactive defense against visual game cheating. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 3045–3061, Philadelphia, PA, August 2024. USENIX Association.

[50] Jianrong Tao, Yu Xiong, Shiwei Zhao, Yuhong Xu, Jianshi Lin, Runze Wu, and Changjie Fan. Xai-driven explainable multi-view game cheating detection. In *2020 IEEE Conference on Games (CoG)*, pages 144–151, Osaka, Japan, 2020. IEEE.

[51] Bill Toulas. Apex Legends players worried about RCE flaw after ALGS hacks. https://www.bleepingcomputer.com/news/security/apex-legends-players-worried-about-rce-flaw-after-algs-hacks/, March 2024. Accessed: 2025-04-08.

[52] Tracker Network. Tracker Network: Find your stats for your favorite games. https://tracker.gg/, 2025. Accessed: 2025-04-20.

[53] Valve Corporation. Overwatch — Counter-Strike Blog. https://blog.counter-strike.net/overwatch/, May 2013. Accessed: 2025-12-08.

[54] Valve Corporation. Valve Anti-Cheat (VAC) System. https://help.steampowered.com/en/faqs/view/571A-97DA-70E9-FF74, 2025.

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[56] WePC. Video Game Industry Statistics, Trends and Data In 2023. https://www.wepc.com/news/video-game-statistics/, 2023.

[57] Wikipedia. Tactical shooter. https://en.wikipedia.org/wiki/Tactical_shooter, 2024. Accessed: 2025-03-31.

[58] Tim Witschel and Christian Wressnegger. Aim low, shoot high: Evading aimbot detectors by mimicking user behavior. In *Proceedings of the 13th European Workshop on Systems Security*, EuroSec '20, page 19–24, New York, NY, USA, 2020. Association for Computing Machinery.

[59] Peter Xenopoulos. Awpy - awpy 2.0.0b4 documentation. https://pypi.org/project/awpy/, 2025. Accessed: 2023-12-07.

[60] 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, Las Vegas, NV, USA, 2006. IEEE.

[61] Su-Yang Yu, Nils Hammerla, Jeff Yan, and Peter Andras. Aimbot detection in online fps games using a heuristic method based on distribution comparison matrix. In *Proceedings of the 19th International Conference on Neural Information Processing - Volume Part V*, ICONIP'12, page 654–661, Berlin, Heidelberg, 2012. Springer-Verlag.

[62] Su-Yang Yu, Nils Hammerla, Jeff Yan, and Peter Andras. A statistical aimbot detection method for online fps games. In *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, Brisbane, QLD, Australia, 2012. IEEE.

[63] 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. *IEEE Transactions on Information Forensics and Security*, pages 1–1, 2025.

[64] Peng Zhou, Wei Shi, Jun Tian, Zhenyu Qi, Bingchen Li, Hongwei Hao, and Bo Xu. Attention-based bidirectional long short-term memory networks for relation classification. In Katrin Erk and Noah A. Smith, editors, *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 207–212, Berlin, Germany, August 2016. Association for Computational Linguistics.

A Evaluation Metrics

Accuracy measures the proportion of correct predictions among all predictions, calculated as Equation 8. Precision represents the ratio of true positives to all positive predictions with Equation 9. Recall quantifies the fraction of actual positives correctly identified as Equation 10. F1 Score provides a harmonic mean of precision (Equation 9) and recall (Equation 10) through Equation 11. False Positive Rate (FPR) indicates the proportion of negatives incorrectly classified as positives using Equation 12.

$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN}. \tag{8}$$

$$\text{Precision} = \frac{TP}{TP+FP}, \tag{9}$$

$$\text{Recall} = \frac{TP}{TP+FN}. \tag{10}$$

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{11}$$

$$\text{FPR} = \frac{FP}{FP+TN}. \tag{12}$$

Weighted averages adjust for class imbalance by weighting each class's contribution proportionally to its sample size. The weighted average is computed as Equation 13, where w_i is the number of instances of class i and Metric_i is the metric value for class i. Notably, weighted accuracy (Equation 8) equals weighted recall (Equation 10) in multi-class settings because both metrics converge when accounting for class weights. This occurs because true negatives (TN) become negligible in multi-class calculations, and the class-weighting causes the denominators to align, making both metrics effectively measure the proportion of correctly identified instances relative to each class's prevalence. The equality specifically holds for weighted averages and not for macro averages or binary classification scenarios.

$$\text{Weighted Avg} = \frac{\sum_{i=1}^{n} w_i \times \text{Metric}_i}{\sum_{i=1}^{n} w_i}, \tag{13}$$

B Model Parameters

The hyperparameters and configurations for the models in INSPECTOR are detailed in Table 14, Table 15, and Table 16.

C Training and Validation Loss

The training and validation loss are illustrated in Figure 8.

D Datasets in Generalizability Evaluation

The datasets used in generalizability evaluation in Section 5.6 is descripted in Table 18 and Table 19.

E Case Study Details

As discussed in Section 2.2, modern aim-assist cheats are highly refined and subtle, making them difficult for novice players to detect. For instance, cheaters in Cases A and D used covert techniques that would likely go unnoticed without expert analysis. Nonetheless, XGUARDIAN identified their abnormal behaviors and provided interpretable explanations, such as the rapid, unnatural aim adjustments across multiple eliminations between 0:23–0:25 in Case A, and the combination of suspicious actions in Case D, including pre-aiming at unseen enemies, achieving kills with minimal crosshair movement, and quickly flicking away post-shot to mask intent. In contrast, some cheaters exhibited overt behavior, as in Cases C and E, leaving clear mechanical patterns inconsistent with human play. Examples include unnatural flicking and tracking from 0:42–0:43 in Case C and 0:58 in Case E, as well as implausibly accurate enemy tracking without visual confirmation at 0:46, 0:51, 1:03, and 1:08 in Case C. Finally, the match-scope case shows how XGUARDIAN aggregates elimination-level predictions into a match-level decision, visualizing the final verdict by averaging elimination outputs with SHAP values highlighting each feature's contribution.

Figure 10 shows XGUARDIAN's elimination-level interpretation for Case C's cheater, selected due to the prominence of their cheating indicators. Within the 7 ticks (109.375 ms) around the elimination, the player's aiming trajectory exhibits significant anomalies across multiple features, including aimbot-driven oscillations and implausible round-trip travel distances—patterns infeasible for a human in such a brief period. Figure 9 illustrates XGUARDIAN 's match-level explainable decision-making. The model classifies a sample as a cheater when its input exceeds the learned decision threshold, with the x-axis showing the sample's input value and the y-axis its SHAP value. A dotted line marks the decision boundary. Other match samples are gray, ground-truth cheaters are circular, legitimate players are triangular, and the highlighted case study is a purple star. Further analysis is available on the our website.

F Adversarial Attack Parameter Selection

Table 17 lists different parameters for the adversarial sample selection. The sample is selected in a way that a certain feature value is highly similar to the average of the normal player, meanwhile ensuring a relatively large sample size to ensure the results are as unbiased as possible. The reason for not choosing the deviation in the ±15% interval is that the deviation is too large, and it does not meet the requirements for the nature of the sample in adversarial attack. The reason for not choosing the deviation in the ±5% interval is that the sample size was too small and may be prone to bias in the results.

G User Study Reviewer's Checklist

The reviewer's checklist for the user study is listed in Table 20.

Table 14: Embedding generation and trajectory classification stage model architecture. None indicates the variable length of the input batch.

Type	Output Shape	Param #
InputLayer	(None, 6, 8)	0
GRU	(None, 6, 64)	14,208
LayerNormalization	(None, 6, 64)	128
Dropout	(None, 6, 64)	0
GRU	(None, 6, 32)	9,408
LayerNormalization	(None, 6, 32)	64
Dropout	(None, 6, 32)	0
Conv1D	(None, 6, 64)	6,208
BatchNormalization	(None, 6, 64)	256
ReLU	(None, 6, 64)	0
Conv1D	(None, 6, 64)	12,352
BatchNormalization	(None, 6, 64)	256
ReLU	(None, 6, 64)	0
Conv1D	(None, 6, 64)	12,352
BatchNormalization	(None, 6, 64)	256
ReLU	(None, 6, 64)	0
GlobalAveragePooling1D	(None, 64)	0
Dense	(None, 32)	2,080
Dropout	(None, 32)	0
Dense	(None, 1)	33
Total params:	57,601 (225.00 KB)	
Trainable params:	57,217 (223.50 KB)	
Non-trainable params:	384 (1.50 KB)	

Table 15: Aggregation stage model architecture. None indicates the variable length of the input batch.

Type	Output Shape	Param #
Dense	(None, 32)	6,304
Dropout	(None, 32)	0
Dense	(None, 16)	528
Dropout	(None, 16)	0
Dense	(None, 1)	17
Total params:	6,849 (26.75 KB)	
Trainable params:	6,849 (26.75 KB)	
Non-trainable params:	0 (0.00 B)	

Table 16: Match prediction stage random forest classifier [43] parameters.

Parameter	Value
bootstrap	True
ccp_alpha	0.0
class_weight	{0: 0.5316, 1: 8.4213}
criterion	gini
max_depth	None
max_features	sqrt
max_leaf_nodes	None
max_samples	None
min_impurity_decrease	0.0
min_samples_leaf	100
min_samples_split	10
min_weight_fraction_leaf	0.0
monotonic_cst	None
n_estimators	100
n_jobs	None
oob_score	False
random_state	None
verbose	0
warm_start	False

Table 17: Different parameters for pre-filtering the candidate adversarial samples.

Non-cheater deviation	#Feature meet the deviation	#Compliant sample
±5%	≥ 1	9
	≥ 2	None
±10%	≥ 1	22
	≥ 2	None
±15%	≥ 1	30
	≥ 2	1
	≥ 3	None

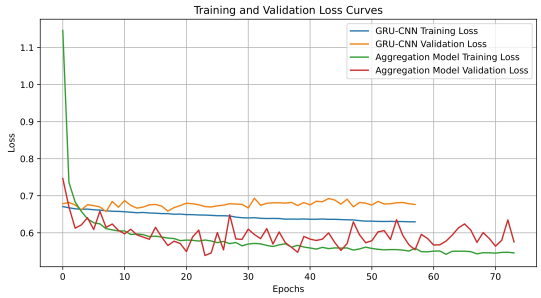


Figure 8: Training and validation loss for GRU-CNN and aggregation model.

Table 18: FARLIGHT84 dataset descriptions.

Scale	Train		Validation		Test	
	#Normal	#Cheater	#Normal	#Cheater	#Normal	#Cheater
Sample	11,402	2,601	8,552	1,951	8,553	1,952
Match	7,554	1,493	6,147	1,232	6,149	1,256
Player	5,513	64	4,678	64	4,691	66

Table 19: HAWK dataset descriptions.

Scale	Train		Validation		Test	
	#Normal	#Cheater	#Normal	#Cheater	#Normal	#Cheater
Sample	152,113	12,273	146,543	11,970	146,887	12,313
Match	1,007	539	975	528	993	541
Player	8,971	357	8,660	336	8,800	336

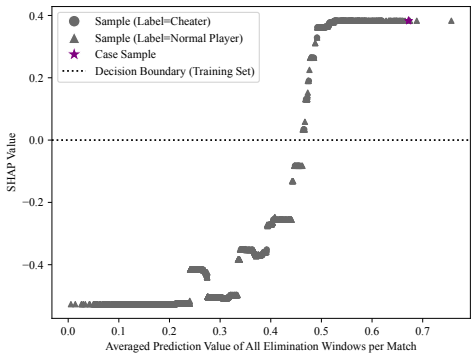


Figure 9: Explainable match-scope visualizations of an identified aim-assist cheater.

Figure 10: Explainable elimination-scope visualizations of an identified aim-assist cheater.

Table 20: User study reviewers' checklist.

Cases end with A	Cases end with B
☐ Open the demo	☐ Open the demo
☐ Start the timer	☐ Open PDFs in `vis` subfolder
☐ Begin the evaluation	☐ Start the timer
☐ Make a decision	☐ Review abnormal segments in PDFs
☐ Stop the timer	☐ Begin the evaluation
☐ Complete the questionnaire	☐ If the information is sufficient: ☐ Make a decision
	☐ If the information is insufficient: ☐ Continue evaluating by yourself until you make a decision

	☐ Compare with the XGUARDIAN's decision in `result.txt`
	  ☐ If different, you may:
	  ☐ Keep your decision
	  ☐ Re-evaluate, change the decision

	☐ Stop the timer
	☐ Complete the questionnaire