Detecting Passive Cheats in Online Games via Performance-Skillfulness Inconsistency

Daiping Liu
Univ. of Delaware
dpliu@udel.edu

Xing Gao
College of William & Mary
xgao01@email.wm.edu

Mingwei Zhang
Intel Research
mingwei.zhang@intel.com

Haining Wang
Univ. of Delaware
hnw@udel.edu

Angelos Stavrou
George Mason University
astavrou@gmu.edu

***Abstract*—As the most commonly used bots in first-person shooter (FPS) online games, aimbots are notoriously difficult to detect because they are completely passive and resemble excellent honest players in many aspects. In this paper, we conduct the first field measurement study to understand the status quo of aimbots and how they play in the wild. For data collection purpose, we devise a novel and generic technique called bait-target to accurately capture existing aimbots from the two most popular FPS games. Our measurement reveals that cheaters who use aimbots cannot play as skillful as excellent honest players in all aspects even though aimbots can help them to achieve very high shooting performance. To characterize the unskillful and blatant nature of cheaters, we identify seven features, of which six are novel, and these features cannot be easily mimicked by aimbots. Leveraging this set of features, we propose an accurate and robust server-side aimbot detector called AimDetect. The core of AimDetect is a cascaded classifier that detects the inconsistency between performance and skillfulness of aimbots. We evaluate the efficacy and generality of AimDetect using the real game traces. Our results show that AimDetect can capture almost all of the aimbots with very few false positives and minor overhead.**

I. Introduction

The online game market is one of the fastest growing entertainment industries in the world. By 2019, the size of global game market is forecasted to reach $100 billion [9]. Among the many genres of online games, first person shooter (FPS) is the second most popular. For example, two out of the five most popular games on Steam, the largest digital distribution platform, are FPS [13]. The FPS game CrossFire [3] is the most profitable, bringing in $957 million annual revenue, far more than other popular games like World of Warcraft [5]. Due to the popularity and economic importance of FPS games, it is imperative to shield them from cheating for the benefit of both game operators and players, especially considering that the competitive nature of FPS games provides strong incentives to cheat. Actually, cheating in FPS games is quite rampant. According to the official forum of CrossFire operated by Tencent [11], more than 80,000 players on average are banned for serious cheating per week. Moreover, anecdotal evidence indicates that the majority of cheaters are still at large. With regard to FPS games, the most common cheat is the use of automated tools to aim and shoot more accurately, also known as *aimbots*.

Unlike active cheating, which tampers with game executable or memory data to gain unfair advantages over other players, aimbots are completely passive and do not violate any game rules. Therefore, they are more difficult to thwart. Even worse, aimbots usually interact with humans intensively, making traditional bot detectors (which assume bots function independently) ineffective. Due to its passive and human-interactive nature, aimbot has plagued FPS games for decades, with no effective detector yet proposed.

In this paper, we first present a field measurement study of aimbots on two most popular FPS games, Counter-Strike 1.6 (CS-1.6) and Counter-Strike Global Offensive (CS:GO), to have a deep understanding of aimbots. To the best of our knowledge, our work is the first to characterize aimbots in the wild. Based on the measurement data, we conduct a systematic exploration of the feature space that characterizes aimbots and honest players from two perspectives: how they kill opponents and how they get killed. Exploiting these identified features, we propose an accurate and robust server-side aimbot detector called AimDetect. We implement a prototype of AimDetect and evaluate its performance in real scenarios.

Measurement study. Since aimbots are passive, a key challenge of such a measurement study is to obtain the ground truth from the field deployment. Neither manual inspection nor client-side detection can accurately or practically identify aimbots. To this end, we devise a novel and generic technique called *bait-target* that can accurately identify modern aimbots for the measurement purpose. The design of bait-target leverages our observation that when a player kills an opponent, it will probably shoot one or more bullets, i.e., inertial shots. Thus, once a player kills an opponent, we immediately create a *human-invisible* target high above the victim, which we call bait-target, as a trap to capture players' inertial shots. Since honest players cannot see bait-targets, their inertial shots are usually gathered around the victim and will miss our bait-target. However, as an aimbot can catch all opponents and *certainly* make an attempt to shoot at the bait-target, its inertial shots will likely hit the bait-target. In this way, bait-target can almost conclusively distinguish aimbots from honest players. Note that although bait-target is effective and reliable to capture current aimbots in FPS games for our measurement study, it cannot serve as a panacea detector in the long run as it is evasion-prone.

We then set up public game servers hosting CS-1.6 and CS:GO. Our field measurement lasts one month and two weeks for CS-1.6 and CS:GO, respectively. During the two different field measurement periods, 294 players from 131 IP addresses played in our CS-1.6 server, while 146 players from 67 IP addresses played in the CS:GO server. Bait-target successfully identifies 47 aimbots in total. After analyzing the collected game data, we observe that aimbots resemble excellent players in most metrics that are widely used by

existing aimbot detectors. We thus posit that the *key challenge* in aimbot detection is actually to differentiate aimbots from excellent players. We successfully identify seven features, of which six are novel, that can effectively address this challenge.

Novel detector. The design rationale behind AimDetect, as well as our insight of these identified features, stems from the fact that cheaters are usually unskillful and blatant; and thus although they can perform similarly to excellent players in shooting, they resemble average players in other aspects like defending and situation awareness. Since FPS games are quite dynamic, it is challenging for aimbots to be as skillful as excellent players in all aspects.

Our design approach of AimDetect is to examine if a player's performance is consistent with its skillfulness in those aspects that cannot be easily mimicked by aimbots. For instance, many excellent honest players can kill opponents hiding behind obstacles. However, since cheaters are usually unskillful and aimbots cannot help to shoot at targets behind obstacles, most cheaters do not have such a kill. In other words, in a same FPS game session if a player is able to achieve excellent performance without the help of aimbots, highly likely it plays like an excellent player in terms of skillfulness and cautiousness.

The classification system of AimDetect includes two cascaded detectors, a Performance Oriented Detector (POD) and a Behavior Oriented Detector (BOD), to fulfill its design approach in two stages. At the first stage, POD uses one-class classification to identify those players who achieve similar performance as excellent players and aimbots. Then, at the second stage, BOD examines if an "excellent" player behaves like an honest excellent player. BOD leverages the proposed features that cannot be easily mimicked by aimbots. Therefore, AimDetect is robust against evasion in that it might be evaded only if either (1) aimbots significantly degrade their performance and thus cheaters would perform like average players; or (2) cheaters become as skillful and cautious as excellent players. However, cheaters will have no motivation or need to use aimbots anymore in either case above. We validate the efficacy of AimDetect by using the real traces collected on our public servers. Our results show that AimDetect is able to capture almost all aimbots with a false positive rate as low as 0.7%. In addition, the overhead of AimDetect is minor in terms of CPU and memory cost.

In summary, our major contributions are as follows:

- We conduct the first field measurement on two most popular FPS games and devise a novel technique, bait-target, to accurately capture modern aimbots in the wild. Bait-target can also assist to collect training dataset for aimbot detection evaluation.
- We characterize aimbots and honest players from two perspectives, how they kill opponents and how they get killed, based on the real game data. Our key observation is that cheaters cannot be as skillful as excellent honest players in all aspects.
- We propose a novel server-side aimbot detector called AimDetect that detects the inconsistency between performance and skillfulness of a player who is using aimbot in an accurate and robust manner. We evaluate AimDetect in real scenarios.

Roadmap. The remainder of this paper is organized as follows. Section II describes the background of aimbots and surveys related work. In Section III, we present a generalized model of modern FPS games. In Sections IV and V, we present the methodology and results of our measurement study, respectively. We detail the design and implementation of AimDetect in Section VI. Section VII evaluates the performance of AimDetect. Finally, we conclude in Section VIII.

II. BACKGROUND & RELATED WORK

A. Cheating in FPS Games

Cheating in FPS games falls into three basic categories: *tampering with game data*, *maphack*, and *aimbots*.

Tampering with game data. Cheaters can manipulate game data to gain unfair advantages. For instance, they could modify game executables in disk or data in memory to eliminate the recoil of weapons, to equip with infinite ammo, and to move significantly faster.

Detecting client manipulation is straightforward in FPS games. The server could track the state of each player, such as moving speed, injury, and ammo, and check against games rules. Any cheat that *actively* tampers with game rules will be revealed. Several recent works have been presented to detect client tampering in games [17] [20].

Maphack. In FPS games, players' coordinates are confidential. When a player is blocked by opaque objects like walls, it is assumed that other players are unable to see him. However, to simplify implementation and improve performance, most modern games will synchronize every player's coordinates with others. The visibility of blocked objects is mandated by client-side software. Cheaters could either eliminate the functions that enforce invisibility or directly read coordinates from memory to pinpoint all players on the map. Such a cheat is known as maphack.

Although maphack is nearly undetectable, it could be prevented. The basic idea is that the server sends player A's coordinates to player B if and only if A is visible to B. This idea has been implemented in anti-cheat software like sXe [14] for Counter-Strike. OpenConflict [18] is a similar academic work for real-time strategy games.

Aimbots. In FPS games, a player controls a virtual character to search for opponents and shoot to kill them with a variety of weapons. Several factors can determine the shooting accuracy of a player, including reaction and experience. It normally takes a long time and extensive exercises for a player to reach the expert level of shooting accuracy. However, with the help of aimbots, a newbie can shoot as accurately as a professional player does. When an aimbot is used, players only need to point the weapon at an opponent's general direction. The aimbot will then automatically aim and shoot at that opponent. Some aimbots can even lock the reticle onto a target within the cheater's field of view.

In order to aim automatically, a bot must know the exact position of an opponent in the virtual game world. Basically, aimbots have two approaches to retrieve that information.

First, cheaters can analyze game clients to locate the address of every object's coordinates stored in memory. At runtime, bots directly read the coordinates from memory and adjust the aim accordingly. The other approach scans the screen and obtains the object's position through graphics analysis. Aimbots usually exploit a simple trick, chameleon skins, to greatly simplify graphics analysis. Chameleon skins replace player model textures with brightly colored skins like red or yellow [2]. In this way, aimbots can locate opponents through scanning the screen for an area of pixels of a certain color. This trick is quite crude, but in most cases, quite effective. Once an opponent's position is obtained, aimbots simulate inputs to automatically aim and shoot via OS API.

In summary, unlike client tampering [17] and maphack [18], aimbot could neither be easily detected as it is *passive* nor definitely prevented as it relies on *local* game data only.

B. Aimbot Type

Aimbots can be implemented with different levels of automation. (1) **OnPressed (OP-bot)**. This type of aimbot auto-aims only when the attack button is pressed by players. (2) **FollowTarget (FT-bot)**. Aimbots automatically aim once an opponent comes into sight but do not shoot automatically. (3) **AutoAtk (AA-bot)**. Aimbots automatically aim and shoot at every visible opponent. All the above three types of aimbots do not control the movement of in-game avatars. (4) **FullyAuto (FA-bot)**. Aimbots can be fully automated and require no user interaction.

Our work focuses on OP-bot, FT-bot, and AA-bot. On one hand, cheaters in FPS games rarely use FA-bot because they expect aimbots to assist but not replace themselves for playing. By contrast, FA-bot is preferred in MMORPGs since players often need to complete tedious and repeated tasks. In fact, among the most popular FPS games like CrossFire, CS-1.6, TF 2, CS:Source (CS:S), and CS:GO, we find only few instances of FA-bot for CS-1.6 and CS:S. On the other hand, a major headache players face in FPS games is that even though someone looks suspiciously like an aimbot, they cannot manually confirm with high confidence. However, when an aimbot is highly automated, it becomes significantly easier for manual detection. Moreover, FPS games are quite dynamic, interactive, and complicated. The automated software can hardly handle every scenario naturally. That is why cheaters prefer to use OP-bot in practice and actually OP-bot accounts for more than 60% real-world instances in our collected real game data.

C. Related Work

Game cheating has received wide attention. A systematic survey and taxonomy can be found in [23]. Modern commercial anti-cheat systems like PunkBuster and Valve Anti-Cheat (VAC) augment client hosts with monitoring functionality to perform cheat detection. These commercial solutions exclusively rely on signature matching, a method commonly considered as ineffective. Kaiser et al. [24] went one step further and proposed anomaly-based cheat detection that requires no signature. However, it requires reliable monitoring on client hosts, which is not always guaranteed in reality. In contrast, we solely rely on data that can be collected on the server side. Moreover, all of these methods are designed to be versatile, while ours targets a specific type of cheat.

A number of previous works target specific types of cheating other than aimbots. To detect cheats that tamper with game clients, Bethea et al. [17] and Cochran et al. [20] employed symbolic execution to verify on the server side if the received messages could be explained by authentic games rules. Some cheaters hack the event ordering protocols used in online games. To prevent such cheats, Baughman et al. [16] proposed the lockstep protocol. OpenConflict [18] leverages oblivious set intersections to prevent disclosing confidential game data to unauthorized players, i.e. map-hacking.

The works closest to ours are those of detecting bots in online games. Several detect game bots in MMORPGs [22] [25] [26]. These approaches leverage the repeated behaviors and self-similarity of game bots for detection. However, aimbots require intensive human cooperation and rarely exhibit repeated and self-similar patterns. There are also works on aimbot detection in FPS games [15] [21] [27] [28]. However, these works are either prone to evasions or ineffective in differentiating aimbots from excellent players, which is the key challenge in aimbot detection. Instead, we identify seven features, of which six are novel, that characterize the unskillful and blatant nature of aimbots. We also design a more effective cascaded classifier that detects the inconsistency between performance and skillfulness of cheaters.

III. GENERALIZING FPS GAMES

FPS games feature a first-person point of view with which a player witnesses the action through the eyes of the in-game avatar [7]. There are three basic elements in almost all modern FPS games:

Virtual World. All FPS games simulate a virtual world that usually complies with physical laws in the real world. For instance, it regulates how fast a player can move and how a player reacts to the recoil of weapons.

Avatars. An avatar is an in-game entity that is controlled by a human player. In FPS games, a player usually controls only one avatar.

Weapons. FPS games typically give players a choice of weapons, either realistic or imaginative. Most primary weapons in FPS are firearms that can attack opponents at a distance.

The runtime game state can be abstracted into two sets:

$$\mathbb{A} = \{A_i^t\} \quad and \quad \mathbb{E} = \{E_j\}. \tag{1}$$

A_i^t = <pos, aiming, fov, health, attacking> denotes the state of avatar i at time t. The position of an avatar is defined by a three-dimension coordination pos = (x, y, z). The aiming describes the direction that a player is aiming at, and it is denoted in two forms. One is angle = (pitch, yaw, roll), where pitch, yaw, and roll are commonly used in games to model a player's view as illustrated in Figure 1. The other is a direction vector $\vec{f}$ = (a, b, c). The field of view (fov) specifies the visible angle of a player. The health denotes how much damage is required to kill the player. The attacking state is 1 if the attack button is held and 0 otherwise. In games, the time t is

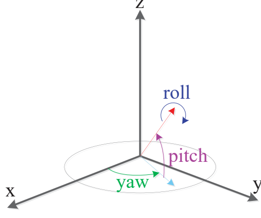


Fig. 1: **Pitch**: degree of deviation from a horizontal plane. **Yaw**: a deflection from an intended course. **Roll** is rarely used in games.

usually measured in ticks or frames rather than the wall clock time.

$E_j \in \{E^i_{Shoot},\ E^{i\to j}_{Hit},\ E^{i\to j}_{Kill}\}$ represents three basic events involving avatar `i`. $i \to j$ denotes avatar `i` hits or kills `j`. Shoot events are generated when an avatar is holding the attack button (i.e., in attacking state). Players can shoot at any time, even when there is no visible opponent. Therefore, we only log the shoot events for an avatar if there is at least one opponent in the sight of the avatar. In addition, a shoot event does not always imply one shot bullet. For instance, there could be multiple shots in one shoot event for some weapons in CS games.

In order to be general, our modeling of a game state is quite simplified. A specific game may extend $\mathbb{A}$ and $\mathbb{E}$ to define a larger state space, which may improve the detection accuracy. However, our evaluation results show that the generalized modeling already performs very well in the real world deployment.

IV. Measurement Infrastructure

A. Bait-target Design

To shed light on the characteristics of real-world aimbots, we set up public game servers to collect traces. The key challenge is to obtain ground truth due to the passive nature of aimbots. It is widely hold that manual inspection cannot reliably confirm the existence of aimbots. Also, it is impractical in our deployment to resort to client-side detectors. To this end, we devise *bait-target* that can accurately identify modern aimbots.

The basic idea is to simulate an opponent using a human-invisible bot, which is referred to as a bait-target, as a trap to catch aimbots. The design rationale behind bait-target lies in two aspects: (1) honest players rarely hit a *transparent* bait-target, thus incurring fewer false positives; (2) because an aimbot is able to catch all different kinds of targets, it will *certainly* shoot at the bait-target and likely make the hit, thus ensuring to achieve a high detection rate. Although the idea looks simple, we need to address several challenging problems.

When to spawn bait-targets? Obviously, it does not make sense to spawn bait-targets casually. For example, if we spawn a bait-target whenever a player fires, we could falsely flag many honest players as aimbots. To address this problem, we leverage the following key observation:

[Inertial shots] When a player kills an opponent, he will highly likely shoot one or more extra bullets.

Fig. 2: An example of bait-target. The bait-target is rendered as visible for illustration purpose.

Inertial shots guarantee both low false positive rate and high detection rate. On one hand, inertial shots are *transient*, lasting less than one second. If bait-targets are properly placed, such short-lived inertial shots from honest players are unlikely to hit them. However, aimbots attempt to make a hit for every shot and thus they will probably hit bait-targets. On the other hand, inertial shots are *inevitable*. Most modern aimbots require humans to press the attack button. Since all humans cannot react instantly to a kill event, they usually shoot several more bullets after the target is killed.

Note that while this finding is valid in most cases, it may not hold for those slow-shooting weapons like a sniper. This is a minor limitation of bait-target. Our experience of playing several popular FPS games shows that aimbots in the wild rarely use these weapons.

Where to place bait-targets? The position where a bait-target is placed can also affect the accuracy. If bait-targets are placed very close to a player's crosshair or it is placed near a real opponent, many false alarms could be raised. However, if they are placed far away from a player's crosshair, we may miss most aimbots. We decide to place a bait-target *exactly above the victim* upon a kill event. When a player kills an opponent, we can assume that his crosshair is close to the victim and therefore honest players' inertial shots are usually gathered around the victim. However, aimbots can *immediately* catch next valid target, causing their inertial shots to hit the bait-target. Figure 2 illustrates the placement of a bait-target. The distance between bait-target and ground is computed in Equation 2.

$$h = \tan\theta \cdot dist \tag{2}$$

where $dist$ is the distance between victim and attacker and θ is the angle that induces h larger than the potential radius of normal inertial shots from honest players as shown in Figure 2. θ should be set empirically for different games. The unit of $dist$ does not matter and different games can define their own world coordinations.

How long do bait-targets last? Another important parameter is how long a bait-target lasts. The longer it lasts, the more false positives we generate. However, if bait-target lasts too short, we may miss inertial shots from aimbots. Our implementation empirically chooses one second, after which we move the bait-target away and change its team

Parameter	Points
Head	10
Body	5
Other Parts	3
th_score	15
th_tests	2

TABLE I: A sample of parameters for bait-target tests.

Game	Ranking among all games	Ranking among FPS games	Release
CS:GO	2	1	2012
CS-1.6	12	4	2003

TABLE II: Ranking on Steam [13] of the two studied games.

to unassigned. Our measurement shows that one second is sufficient to capture an aimbot.

How does bait-targets flag aimbot? Our bait-target adopts a simple scoring scheme to identify aimbots. In most FPS games, the difficulty to hit different parts of opponents varies. Generally, the more difficult, the more damages incurred. Aimbots usually prefer to shooting at the parts with more damages, like head. In our design, a direct hit on different parts of bait-target will count certain points. When the score reaches a threshold th_score, we consider a positive test is observed. The score will be reset if it is still below th_score after three consecutive tests, avoiding false alarms caused by accumulated accidental shots on the bait-target. When a predefined number of positive tests th_tests are observed, the attacker will be logged as an aimbot.

We then ban the aimbot after ten more kills, allowing our server to collect more data. As an example, a typical configuration is shown in Table I. Different games may change the parameters and even define different hit parts.

B. Design Clarification

Transparent Targets. Bait-target requires aimbots shoot at transparent targets. In order to assess to what extent this requirement can be met, we collected and analyzed more than 300 real aimbot instances for several games including CrossFire, TF 2, CS:GO, CS-1.6 and CS:S. These are currently the most popular FPS games [13] [5]. We find that modern aimbots *rarely* avoid shooting at transparent targets. In particular, only some FA-bot does not shoot at transparent targets. This is reasonable because modern games usually implement rich game features and modes. In many cases, e.g., Ghost Mode, the targets can be indeed invisible. In order to work reliably, aimbots should not avoid shooting at transparent targets.

Bait-target is *not* a long-term detector. We do not expect bait-target to serve as a promising detector in the long run. Bait-targets per se are a form of security through obscurity and thus are evasion-prone. Once aimbots become aware of their existence, they could evade by shooting only at visible targets or switching off for second after a kill, even though this sacrifices the reliability of aimbots.

Actually, bait-target is originally designed as an effective measurement tool rather than a robust detector. Besides, it can also assist game operators to obtain the training set for machine learning based detectors. We believe bait-target successfully achieves these two goals. On one hand, almost all identified aimbots are true positives. On the other hand, honest players' data can be randomly sampled to mitigate potential influence of evasions.

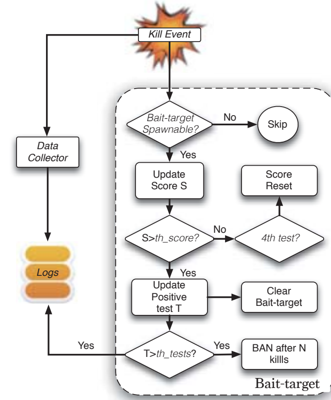


Fig. 3: Flow chart of our experimental protocol.

Generality of Bait-target. We posit that bait-target is generic and can be applied to all FPS games. First, bait-target does not depend on any game/platform specific features. It relies only on the fact that inertial shots are inevitable due to the delay between when the target is killed and when the attacker receives the death event and stops shooting. This is true in all games. We have validated the prototypes for games on two games, CS-1.6 and CS:GO.

C. Data Collection

Server setup. We host CS:GO and CS-1.6 on our public servers. These two represent the most popular modern FPS games, as shown in Table II. Our servers are published on ww w.gametracker.com. The servers cycle on three most popular maps, dust2, iceworld, and inferno, with each lasting day.

Prototype of bait-target. We implement the bait-target using Amxmod [1] for CS-1.6 and SourceMod [12] for CS:GO. We evaluate its effectiveness using our collected aimbots and the traces from our public servers. The results are presented in §IV-D. θ in Equation 1 is empirically set as $\frac{\pi}{18}$ for both games.

Handling FA-bot. In §IV-B, we have mentioned that the `FA-bot` of CS-1.6 does not shoot at bait-targets. Even worse, there is no practical and reliable approach to identify FA-bot. A further analysis of FA-bot reveals that all of them avoid shooting at transparent targets through checking the rendering mode of an object. Based on this observation, we re-render *all* players as opaque with a transparency mode. This ad-hoc technique is simple but quite effective to disable all FA-bot.

Note that this trick does not reduce the generality or accuracy of our measurement results. On one hand, our work does not target FA-bot. On the other hand, disabling `FA-bot` does not introduce further false detections. Meanwhile, this trick does not induce any negative effects upon honest players.

Experimental protocol. Our public server reserves two slots for bait-targets because FPS games usually limit the number of players that can join a server. The limit is 32 and 64 for CS-1.6 and CS:GO, respectively. Bait-targets and a behavioral monitor run simultaneously to log results. Figure 3 shows our experimental protocol.

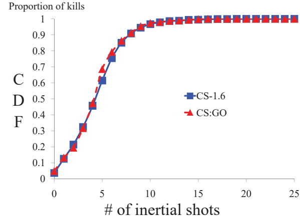


Fig. 4: Statistics of inertial shots.

Game	Category	Connections	IPs	Countries
CS-1.6	Honest	256	93	25
	Aimbot	38	38	15
CS:GO	Honest	137	58	13
	Aimbot	9	9	5

TABLE III: Dataset of real-world deployment. The IP addresses of aimbots are banned. Therefore, there is only one connection for each aimbot.

Data collector. Our in-game data collector leverages Metamod [10] and SourceMod [12]. MetaMod and SourceMod provide comprehensive hooks to in-game events for CS-1.6 and CS:GO respectively. They enable us to develop third-party plugins to monitor all players' actions on server side at every frame.

Dataset. Our dataset contains one-month and two-week traces for CS-1.6 and CS:GO, respectively. Table III summarizes the collected data. During our deployment, 294 connections from 131 distinct IP addresses are logged on our CS-1.6 server. Bait-targets identify 38 aimbots. 146 connections from 67 distinct IP addresses are logged on the CS:GO server, among which five are aimbots. The IP addresses are also mapped to countries using GeoLite2-City [8].

D. Inevitable Inertial Shots

We create a bait-target to identify aimbots on our server. In order to assess the effectiveness of bait-target, we need to answer the following two questions: (1) to what extent does our observation of inertial shots hold? and (2) how likely is that honest players hit bait-targets?

Validity of Inertial Shots. We record the inertial shots of every player connected to our servers. Since our bait-target test lasts one second, we define intertial shots as those that are fired within one second after a kill event. Also, to make the results more accurate, a shot is counted only if there is no enemy left in sight of the attacker. The statistics from our deployment are presented in Figure 4.

We find that only 3.6% kills from CS-1.6 and 4.8% from CS:GO come with no inertial shots. Meanwhile, for each player, most of their kills have at least one inertial shot. Thus, our observation would hold in most cases. Also, in both games, about 80% kills are followed with two to seven inertial shots. This result demonstrates that our parameter configuration in Table I is reasonable.

False alarms. Our data shows that only $\sim$0.8% inertial shots from honest players hit a bait-target in both games. Thus, by setting an appropriate hitting threshold, we can prevent bait-targets from issuing a false alarm. To verify this, we recruit six participants who do not know the existence of bait-target to conduct an in-lab experiment. During the 3-hour testing, no honest player is flagged as an aimbot.

$$\begin{aligned}
&\mathbb{F} ::= \bigcup op(\mathbb{R}_i \mid cond)\\
&op ::= \{VAL,\ AVG,\ STDV,\ COUNT,\ TIME,\ ELAPSE\}\\
&cond ::= \{(ON)SEE,\ (ON)ATK,\ (ON)AIM,\ \ MOV,\ KILL\}\\
&\mathbb{R}_1 ::= \mathbb{A} = \bigcup A_i^t\\
&\mathbb{R}_2 ::= \mathbb{E} = \bigcup E_j^i\\
&\mathbb{R}_3 ::= \mathbb{A}^{\Delta t} = \bigcup (A_i^t - A_i^{t-n})\\
&\mathbb{R}_4 ::= \mathbb{A}_i^t \diamond \mathbb{A}_j^t
\end{aligned}$$

Fig. 5: Rules to systematically derive our feature set.

V. Characterization of Aimbots

In this section, we characterize the behaviors of aimbots from two perspectives, i.e., how they kill opponents and how they get killed, based on the data collected in the field.

A. Systematic Exploration of Feature Space

The raw data of $\mathbb{A}$ and $\mathbb{E}$ is insufficient to characterize a player's behaviors. In particular, it cannot be fed to a machine learning classifier that does not know the physical meaning of the data. Existing works [28] [27] [21] [15] have empirically defined their own feature sets. However, these features are either ineffective in the real world or prone to evasions.

Two existing works [21] [15] rely on players' position, fighting time, and aiming accuracy to detect cheaters. Unfortunately, these features are more likely to differentiate excellent from average players rather than aimbots from honest players. Their evaluations achieve promising results but only two or three human players are involved. Moreover, aimbots can easily evade the high ranked features without degrading performance. For instance, aiming accuracy, defined as the time aiming at a target divided by the total time when the target is visible, is ranked the first in their evaluations. However, aimbots do not need to aim constantly. There is a relatively long interval between two shots. Aimbots only need to aim upon a shot and then move away during the intervals. Actually, many average players can also have a high aiming accuracy although they cannot aim accurately upon a shot. For the same reason, we exclude the shooting miss ratio, an intuitive feature that many people consider to be reliable. However, we find that aimbots can fire more shots to increase the miss ratio while achieving the same number of hits within the same time period.

In [28] [27], the authors define two features, the cursor acceleration and time on target (i.e., aiming accuracy). Although the cursor acceleration can effectively differentiate excellent players from aimbots, it can be evaded by simply mimicking the cursor movement of humans. We instead decide to derive our own feature set by systematically exploring the feature space, in which we successfully identify seven features, six of which are novel, for play behavior characterization. Given the generalized model of FPS games in §III and the aimbot types in §II-B, we derive the feature set $\mathbb{F}$ using the rules in Figure 5.

The first three operators of `op` return the value, average, and standard deviation of a feature. `COUNT` returns the number of values. `TIME` computes the duration of a feature. Finally, `ELAPSE` returns the elapsed time between two game states. Note that both `TIME` and `ELAPSE` operators return the wall time, rather than the frames as `t` for avatar state.

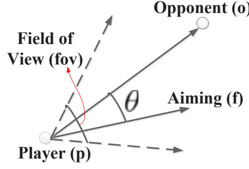


Fig. 6: Geometric meaning of divergence.

We also specify when a feature is collected using `cond`. As described in §II-B, aimbots function only during a specific time period. We may collect a lot of noise data if we do not constrain when a feature should be collected. We currently define eight conditions: `SEE(i,j)` denotes avatar j is visible to i and `ONSEE(i,j)` means the moment when avatar j becomes visible to i; `ATK(i)`, `ONATK(i)`, `AIM(i,j)`, and `ONAIM(i,j)` are defined for the attacking and aiming states of an avatar, respectively; `MOV(i)` means avatar i is moving, and `KILL(i,j)` specifies the event when avatar i kills j. All these conditions can be easily computed based on either avatars' state ($\mathbb{A}$) or events ($\mathbb{E}$). These eight conditions will play a critical role to filter out noise data in the feature collection. Note that only one condition, i.e., `ATK(i)`, is considered in previous research.

$\mathbb{R}_1$ collects the states of an avatar and $\mathbb{R}_2$ collects all events occurred during the game. $\mathbb{R}_3$ computes the state change of an avatar between two frames. $\mathbb{R}_4$ describes the interaction between two players at frame t.

The rules in Figure 5 can generate almost infinite number of features. To make our analysis tractable, we only consider the features that are geometrically or statistically meaningful. For instance, it is obviously meaningless to compute the relationship between the health and position of a player. In total, we test more than a hundred features, among which seven are finally selected. A feature is selected if (1) it can differentiate aimbots from either excellent or average players and (2) it is difficult for aimbots to evade without extensive efforts from cheaters or degrading performance significantly. We next present these features in details.

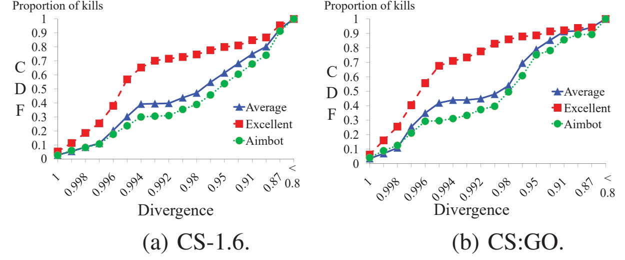
B. How Players Kill Opponents?

A player's goal in FPS games is to kill the opponents. Aimbots, as well as all other cheats, assist cheaters in this sense. Therefore, our first subset of features characterize how players kill opponents. To single out excellent players, we propose a new metric, Expertise-Index (EI) as defined in Equation 3, which computes the difference between the number of kills a player made and the number of times itself being killed. In our analysis, the top 5% honest players in terms of EI are labeled as excellent and the remaining as average.

$$EI_i = \Sigma E_{Kill}^{i \to *} - \Sigma E_{Kill}^{* \to i}. \quad (3)$$

Although EI can retroactively assess the performance of a player, it is not suitable to serve as a feature in our detector because it highly depends on how long a player plays. In addition, the detection can be dramatically delayed.

Divergence of aiming upon the opponent comes into sight (F_1). For each kill event, we first look at the game state when the victim comes into the sight of the attacker. In particular, we only consider the cases when there are no other opponents in the sight of the attacker. We find that the divergence of aiming is an effective feature to help identify aimbots from excellent players. Figure 6 illustrates the geometrical meaning of this feature, which is computed as:



(a) CS-1.6. (b) CS:GO.

Fig. 7: Divergence of aiming.

$$\cos\theta(i,j) = \frac{a_i a_j + b_i b_j + c_i c_j}{\sqrt{a_i^2 + b_i^2 + c_i^2}\sqrt{a_j^2 + b_j^2 + c_j^2}}. \quad (4)$$

Figure 7 shows the cumulative distribution of the results. The x axis represents the interval of the divergence. The larger $\cos\theta$ is, the smaller θ is. Excellent players are more skillful and experienced. They usually aim at the point where an opponent is more likely to appear so that they can react quickly and shoot accurately. However, most cheaters are unskillful and blatant. Although aimbots can help aiming at opponents during attacks, they cannot help predicate the opponents' positions, which heavily depends on the experience and cautiousness of a player. We define this feature as the percentage of kills that have a divergence smaller than a threshold α:

$$F_1 = \frac{\Sigma(E_{Kill}^{i \to *} \wedge (\cos\theta \leq \alpha))}{\Sigma E_{Kill}^{i \to *}}. \quad (5)$$

For different games, α may need to be adjusted accordingly. Based on the results in Figure 7, we set α to 0.994 for both games.

Suspiciousness of critical hits (F_2). Most commercial FPS games try to accurately model the real world. In particular, hits on different parts of opponents cause varying damages. For instance, hits on head usually incur the most damages in almost all games and thus aimbots usually lock the head for simplicity and efficiency. Our survey shows that more than 90% of modern aimbot instances lock the head as target by default. In addition, 18% of aimbot tools allow users to change shooting target to other parts of body. Our traces present a similar result. Table IV shows the ratio of headshots for aimbots and honest players. It can be seen that both aimbots and excellent players can achieve higher headshot ratio than average players.

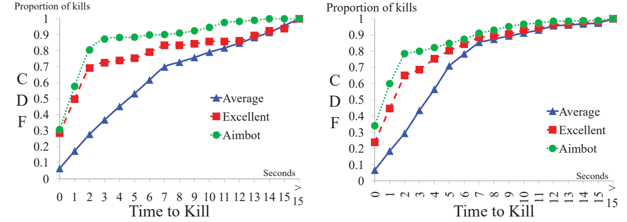
Moreover, aimbots usually try to achieve headshots as fast as possible. Table V lists the percentage of headshots made by the first hit. Combining the above results, we therefore define the suspiciousness as:

Game	Aimbot	Excellent	Average
CS-1.6	80.2%	68.2%	30.3%
CS:GO	77.3%	62.5%	32.7%

TABLE IV: Ratio of headshots.

Game	Aimbot	Excellent	Average
CS-1.6	65.4%	54.7%	32.5%
CS:GO	66.1%	53.3%	33.1%

TABLE V: Ratio of headshots achieved by the first hit.



(a) CS-1.6. (b) CS:GO.

Fig. 8: Cumulative distribution of F_4.

$$F_2 = \frac{1}{n}\sum_{i=1}^{n}(-s_i), \quad s_i = \begin{cases} \frac{1}{j_{th}\text{hit}} & \text{if critical hit,} \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

where s_i denotes that the first critical hit is the j_{th} hit in i_{th} kill event. If there is no critical hit during a kill event, s_i is 0. We use the negative value of s_i to make it easier to present our classifier in §VI-B.

Ratio of hits when the attacker is moving (F_3). Most FPS games also conform to the physical laws. A common rule regulates that the faster an attacker moves, the more difficult for it to aim accurately. Therefore, honest players tend to stand still during attacks. This feature computes the ratio of hits when an attacker is moving, i.e.,

$$F_3 = \frac{\Sigma(E_{Hit}^{i\rightarrow *} \wedge MOV(i))}{\Sigma E_{Hit}^{i\rightarrow *}}. \tag{7}$$

The results of our traces are shown in Table VI. As we can see, most excellent players are cautious while the majority of aimbots and many average players rush blatantly. Meanwhile, we also check 10 playing demos from 30 professional players for CrossFire, CS-1.6, and CS:GO. We find that these expert players also move cautiously and they tend to stand still during most kills. By contrast, aimbots are quite blatant and they commonly have little intention to play cautiously.

Game	Aimbot	Excellent	Average
CS-1.6	85.7%	42.3%	57.7%
CS:GO	74.1%	46.8%	54.6%

TABLE VI: Ratio of hits when the attacker is moving.

Time to kill (F_4). During an attack, a player must kill opponents as quickly as possible; otherwise, it will probably get killed. We measure how long it takes for an attacker to kill an opponent using Equation 8. This is the only feature that has been used in one existing work [15]. Intuitively, excellent players and aimbots usually are able to kill opponents faster.

$$TTK_i = ELAPSE(ONSEE(i,j),\ KILL(i,j)). \tag{8}$$

Figure 8 shows the cumulative distribution of F_4 for CS-1.6 and CS:GO. Most aimbots and excellent players can kill opponents within two seconds. By contrast, average players generally take longer time to kill an opponent. F_4 is effective to differentiate aimbots from average players. Similar to F_1, we define F_4 as:

$$F_4 = -\frac{\Sigma(E_{Kill}^{i\rightarrow *} \wedge (TTK \leq \alpha))}{\Sigma E_{Kill}^{i\rightarrow *}}. \tag{9}$$

According to Figure 8, the threshold α can be set as two seconds. For the same reason as F_2, this feature also has the negative value.

Local impact ranking (F_5). During the games, a player has a higher impact if it can kill more opponents. At time t of a game, assume K is the number of kills made by player p and N is the total number of opponents spawned since p joined the game. We define a new metric, impact index, as $\frac{K}{N}$. Since players can join and leave during the games, we compute the local impact ranking for each player separately. When a player joins the game, we starts to recompute the impact index for all players. This feature then is the local ranking of each player. As expected, after a short bootstrapping, both aimbots and excellent players will rank and remain in the top. Although intuitive, this feature is selected because if cheaters attempt to evade our detection, they have to deliberately miss many targets to lower their rankings.

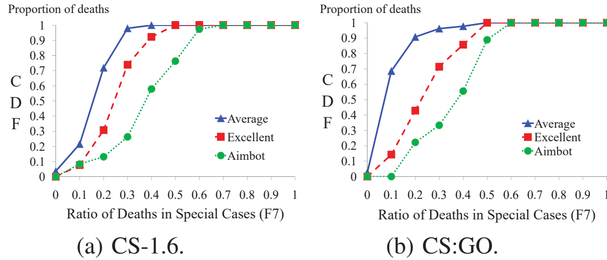
Ability to kill opponents behind obstacles (F_6). We also find that many excellent players can kill opponents behind obstacles like walls. In the two games together, about 45% of excellent players at least kill one invisible opponent, while the ratio of aimbots is only 14.9%. A common scenario is that the victim moves behind some obstacles to protect itself during attacks. An excellent player is still able to kill the opponents in such a scenario. However, most aimbots cannot work when the opponents are invisible. Actually, with the help of commercial anti-wallhack techniques like sXe [14], game servers can make aimbots absolutely useless in face of invisible opponents. Therefore, we define this feature as:

$$F_6 = \begin{cases} 1 & \text{if killing opponents behind obstacles,} \\ 0 & \text{otherwise.} \end{cases} \tag{10}$$

C. How Players Get Killed?

We next examine the characteristics of aimbots from the opposite side, i.e., how they get killed. A look into the deaths of aimbots and excellent players reveals two interesting observations.

First, a non-trivial portion of aimbots are killed by those players who are not in their sight. This is mainly because cheaters are unskillful and more blatant. They usually rush



(a) CS-1.6. (b) CS:GO.

Fig. 9: Cumulative distribution of F_7.

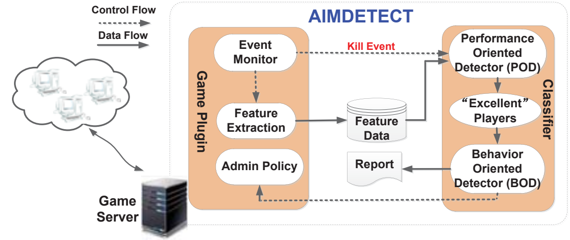


Fig. 10: Overview of AIMDETECT

recklessly and do not check their surroundings carefully. Second, we observe that aimbots are more likely to be killed when the fight lasts relatively longer, e.g., longer than two seconds. There are several possible reasons. Excellent players have more chance to survive in a long lasting fight because they are not only good at aiming but also protecting themselves. Also, most FPS games adopt a parameter, spread, which determines how possible and how far a shot deviates from the aimed point. The longer a player shoots continuously, the larger this parameter increases. Although aimbots can aim accurately, they cannot offset the effect of spread. In many cases, when an opponent cannot be killed quickly, inexperienced cheaters are likely to shoot continuously *by instinct*. Finally, since it is quite challenging to handle the dynamic and complicated situations in FPS games, aimbots can have glitches and malfunction in some scenarios.

Ratio of deaths in special cases (F_7). Based on the two findings, we thus compute the ratio of deaths in the above two situations to the total deaths. Figure 9 shows the results of the two games, indicating that more than 80% of excellent players and aimbots in both games are killed at least five times. Therefore, although death events are relatively infrequent for excellent players and aimbots, they can still be leveraged for aimbot detection.

D. Key Insight

In this section, we present seven features that can effectively differentiate aimbots from either excellent or average players. The key insight is that even though cheaters can resemble excellent players in shooting performance, both still manifest very different behaviors in skillfulness, such as cautiousness and self-defense, which cannot be mimicked by aimbots. We next show how to build an accurate and robust online aimbot detector by leveraging these features.

VI. AIMDETECT DESIGN

Aimbots seriously wreck the fairness in FPS games. The game operators are struggling to combat aimbots since aimbots resemble excellent honest players in many aspects, especially in those obvious metrics like impressive aiming accuracy. In practice, many excellent players are banned due to false detections [4]. Therefore, the *key challenge* in aimbot detection is to differentiate aimbots from excellent players. One of our main contributions in this work is to address this key challenge.

Meanwhile, if aimbots would like to mimic average players, they have to degrade the performance significantly. Note that most cheaters are unskillful, they will probably perform no better than an average player. Therefore, these gentle aimbots are tolerable from the perspective of game operators, because honest players are unlikely to complain an average player. Actually, in order to attract and retain new players, some commercial FPS games even incorporate a gentle aim-assist tool for novices [2].

With the above two observations in mind, we thus posit that a desired aimbot detector should be *accurate* in differentiating aimbots from excellent players and *robust* against evasion. The detector would be evaded only if one of two conditions below happens: (1) cheaters are as skillful and cautious as an excellent player; or (2) aimbots degrade their performance significantly and play like average players.

A. Design Overview

At a high level, AimDetect is motivated by our measurement results that aimbots usually manifest themselves in a series of suspicious behavioral patterns. In particular, we characterize the differences between aimbots and honest players from two aspects:

[A1] Anomalous performance resembling excellent players;

[A2] Blatant and unskillful nature of cheaters.

Figure 10 illustrates the architecture of AimDetect, which completely resides on game servers. The first component of AimDetect is an in-game plugin, which has three modules: event monitor, feature extraction, and admin policy. The first module monitors in-game events like aiming at the enemy, death and hit. These events drive the feature extraction module, which collects predefined in-game features. Administrators can set policies that are executed when aimbots are detected.

At runtime, a stand-alone classifier is driven by kill events. Every time a kill event is captured, the feature vectors of players are updated and the classifier is invoked. We build a cascade classifier that has two stages. First, a Performance Oriented Detector (POD) determines whether a player resembles excellent players in terms of the gaming performance. The identified "excellent" players are then examined by the Behavior Oriented Detector (BOD) in the second stage, which determines whether a player behaves like an excellent players, i.e., if its skillfulness is at the same level of excellent players.

B. Performance Oriented Detector (POD)

Feature Selection and Discussion. In the first stage, POD monitors a player's performance. POD ensures that aimbots cannot evade AimDetect without degrading their performance significantly. To this end, we select three features, F_2, F_4, and F_5, for POD.

One may question why several features are needed if one like local impact ranking can straightforwardly identify excellent players. On one hand, as more features that characterize players' performance from more aspects are used, it becomes more difficult for aimbots to evade. For instance, if only the time-to-kill (F_4) is used, aimbots can deliberately delay for one to two seconds and then execute a headshot. Cheaters can still have notable advantages over honest players. On the other hand, using more features can also reduce false positives. For example, assume all players in a game are novices. The local impact ranking may mistakenly identify one player as excellent. Since the player is actually a novice, our second stage detector may consider it to be inconsistent, generating a false positive.

Implementation. POD uses a one-class classifier that is trained on a dataset containing aimbots and excellent players. We choose a one-class classifier over a two-class because the two-class classifier requires input data from average players at the training phase, which is difficult to obtain. While it is reasonable to label the top 5% honest players as excellent, it is impossible to know whether the top 10% or 20% are also excellent or not.

Given a target class T represented by the training set $\mathcal{X}_{tr}$ and a new object z, the one-class classifier assigns z to T if the distance between z and T is smaller than a threshold ε:

$$f(z) = I(d(z, T) < \varepsilon), \tag{11}$$

where $I(\cdot)$ is an indicator function that returns 1 if the condition is met and 0 otherwise. $d(z, T)$ is a distance measure which we define as:

$$d(z, T) = \sum_i \frac{z_i - T_i}{\sigma_i}, \tag{12}$$

where σ_i denotes the standard deviation of the ith feature over n trials. We give higher weights to the features that have smaller variation by dividing σ_i. Also, conventionally the distance is summed over the absolute value of feature differences. We instead choose to keep the minus sign. Considering that each value of all three features negatively correlates with a player's performance, a large negative distance means a player performs even better than the samples in the training set. Such players should definitely further be checked by BOD. In our implementation, T is simply an average feature vector over $\mathcal{X}_{tr}$.

The threshold ε is a user defined parameter. A large value means a stricter detector, but may falsely report an honest player because more "excellent" players will be selected and checked by BOD. A small threshold reduces false positives, but allowing relatively gentle aimbots to slip through.

C. Behavior Oriented Detector (BOD)

Feature Selection and Discussion. Once a suspicious player is identified by POD, it is then checked by BOD to see if its performance is consistent with its behaviors (i.e., skillfulness) or not. BOD ensures that AimDetect cannot be evaded without extensive efforts from cheaters. Based on our measurement results in §V, we choose four features for BOD.

The first feature is the average of F_1 over a series of kill events. This feature basically characterizes the cautiousness of a player. In order to evade, aimbots may learn how excellent players behave using more advanced techniques. However, due to the highly dynamics nature of FPS games, it is very challenging and costly to reach this goal in real time under very different game scenarios.

BOD also uses F_3, which characterizes the blatant nature of cheaters. In general, it is difficult to require most cheaters to behave gently. Alternatively, aimbots can force cheaters to stand still during attacks. However, this can seriously affect user experience. For instance, in some cases, players need to rush and move for defending themselves.

The third feature BOD used is F_6. Considering that aimbots cannot help shoot at invisible targets, this feature effectively indicates the true expertise of a player. The last feature is F_7. Basically, even though aimbots can mimic how excellent players shoot, they cannot mimic how they get killed. Finally, we believe it is theoretically possible to build a perfect FA-bot that resembles excellent players in all aspects. However, as discussed in §II-B, few cheaters will use it in practice.

Implementation. BOD currently leverages the two-class Support Vector Machine (SVM) algorithm with hyperbolic tangent kernel [19]. We choose SVM because it has been proven to be effective and efficient in practice. Also, SVM works best when the dataset does not contain many outliers, which is the case in BOD. Again, BOD is trained over the dataset from excellent players and aimbots.

D. Decision Synchronization

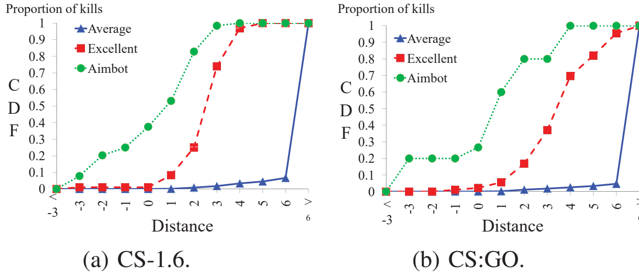
All features in POD depend on the kill events that occur frequently for aimbots and excellent players. Therefore, POD can make decisions fast. However, F_6 and F_7 in BOD depend on some relatively infrequent events like deaths for aimbots and excellent players. When POD has made a decision, BOD may still have insufficient data to make its detection. Therefore, in order to ensure that BOD can accumulate enough data, game operators can specify two configurable parameters, the maximum number of kills (p_k) and the minimum number of deaths (p_d). The detection does not start until one of the two conditions is satisfied.

E. Portability of AimDetect

We emphasize that AimDetect is generic and portable. All proposed features are based on a general model presented in §III. Although various contents could be supported in different games, characterizing all of them in AimDetect is unnecessary. Most of these elements are minor factors that can be ignored for simplicity and generality. For instance, some games may provide vehicles like cars and helicopters, which just affect the movement of players.

VII. AimDetect Evaluation

In this section, we evaluate the efficacy of AimDetect in terms of detection accuracy and system overhead.



(a) CS-1.6. (b) CS:GO.

Fig. 11: Cumulative distribution of the distance between a player and the trained model at each kill event.

A. Detection Performance

We use two metrics, true positive rate (TPR) and false positive rate (FPR), for detection accuracy. Given the notations in Table VII, TPR and FPR are defined as:

$$TPR = \frac{P_{tp}}{P_a}, \quad FPR = \frac{P_{fp}}{P_n} \tag{13}$$

Experiment Setup. We first split the dataset into three groups, D_a that contains aimbots, D_e that contains excellent players, and D_o that contains all the rest. We conduct our experiments using 10-fold cross validation. In each configuration, two in D_a and four in D_e are tested for CS-1.6. Since the dataset of CS:GO is relatively small, 10 random one-aimbot-one-excellent pairs are tested. During each test, we simulate the online detection by invoking the classifier at every kill event.

Setting p_k and p_d. As described in §VI-D, there are two configurable parameters for AimDetect. These parameters basically determine the detection speed of AimDetect. Large values improve the confidence of detection results, but slow down the detection speed. Small values trigger POD and BOD faster, but may cause more false detections. We empirically set p_k as 10 in our evaluation and p_d is set to five. However, if a player does not have sufficient kills or deaths, we still test them using all available data.

Results of POD. POD has one configurable parameter, the threshold ε. We first study the impact of the threshold on detection accuracy and speed. If a positive decision is made at a kill event, the remaining kills will be omitted. We find that the threshold does not affect the detection speed significantly. Once p_k or p_d is triggered, POD can almost instantly make a decision, which means ε basically affects detection accuracy only. Figure 11 shows the cumulative distribution of the distance between a player and the training set at each kill event in one of the tests. Obviously, average players significantly differ from aimbots and excellent players. In Table VIII, we present the number of identified players in each group for different values of ε.

Symbol	Definition
P_a	total # of aimbots
P_{tp}	# of aimbots *correctly* classified
P_n	total # of honest players
P_{fp}	# of honest players *falsely* classified as aimbots

TABLE VII: Notations of detection results.

We can see that in all cases, all aimbots are successfully identified by POD and hence the threshold does not affect TPR. Meanwhile, FPR decreases along with the threshold. This is because when the threshold decreases, fewer honest players are selected for further checking by BOD, which lowers the risk of generating false positives. Moreover, a relatively large portion of players in D_o are also identified by POD. This demonstrates the difficulty to select the set of average players for training, and it is more reasonable to use the one-class classification in POD.

Results of BOD. For all the players identified in POD, BOD next checks if their playing performance is consistent with their behaviors. The results are presented in Table VIII. In almost all cases, BOD demonstrates promising accuracy. BOD successfully identifies 32 out of 35 aimbots for CS-1.6 and 7 out of 8 for CS:GO. We manually investigate the missed cases and find that those cheaters are more skillful and cautious. For instance, most of their hits are achieved when they stand still and they do not differ significantly with excellent players in F_1.

In all but one cases, BOD raises some false positives. In general, we believe it is impossible to eliminate false positives. An excellent player can resemble cheaters sometimes, especially when the opponents play much worse than the excellent players. However, we believe this will not prevent AimDetect from being deployed on commercial games. In particular, existing games mostly rely on manual inspection to identify aimbots, which can suffer from much higher false positive rate.

B. Comparison with VAC

We also compare the detection performance of AimDetect with that of the Valve Anti-Cheat (VAC) system adopted by Steam. VAC represents a state-of-the-art commercial detector. We recruit four participants to play with aimbots in a private server we set up on Steam. Each participant plays with aimbots for about 40 minutes in turns. No cheating attempt is detected by VAC. In contrast, AimDetect successfully identifies all four cheaters. Moreover, we find many players report in mpgh.net that they go undetected using aimbots [6].

C. System Overhead

AimDetect is deployed on the server side and must monitor hundreds of online players simultaneously. Thus, it needs to be efficient in terms of CPU and memory consumption. We measure the CPU overhead of AimDetect on our public server. During the entire field deployment, AimDetect consumes 0.1% extra CPU utilization on average. Moreover, our measurement shows that there is no decrease on the number of frames per second (usually ∼100) caused by our in-game plugin. Thus, AimDetect is CPU-efficient.

The primary memory consumption of AimDetect is caused by accommodating each player's behavioral data. Each player consumes about 100 bytes to record data, including IP, time, features, and other information. If a game server is serving

Game	Classifier	Group	Threshold ε			
			4	**3**	**2**	**1**
CS-1.6	**POD**	**Average**	153/2430	147/2430	116/2430	48/2430
		Excellent	20/20	20/20	20/20	17/20
		Aimbots	40/40	40/40	40/40	40/40
	BOD†	**Average**	39/153	34/147	21/116	8/48
		Excellent	2/20	2/20	2/20	1/17
		Aimbots	37/40	37/40	37/40	37/40
	TPR		92.5%	92.5%	92.5%	92.5%
	FPR		1.6%	1.4 %	0.9%	0.3%
CS:GO	**POD**	**Average**	337/1300	174/1300	127/1300	71/1300
		Excellent	10/10	10/10	10/10	10/10
		Aimbots	10/10	10/10	10/10	10/10
	BOD†	**Average**	71/337	24/174	18/127	7/71
		Excellent	2/10	2/10	2/10	2/10
		Aimbots	9/10	9/10	9/10	9/10
	TPR		90%	90%	90%	90%
	FPR		5.6%	2.0%	1.5%	0.7%

TABLE VIII: Detection results of AimDetect for CS-1.6 and CS:GO. † The results denote the number of players identified as aimbots and thus they are false positives (i.e, P_{fp}) for excellent and average players.

30,000 concurrent players, the induced memory consumption would be only $\sim$2.9MB in total. This overhead is negligible for modern game servers. Finally, since AimDetect is deployed purely on the server side, it incurs no extra network overhead.

VIII. Conclusion

In this paper, we conduct the first field measurement to understand how aimbots play in the wild, especially how they kill opponents and how they get killed. Our major observation is that cheaters who use aimbots cannot maintain the same level of consistency as excellent honest players between performance and skillfulness in all aspects. Based on the measurement results, we identify a set of features, which are very hard to mimic by aimbots, to characterize the difference between cheaters and excellent honest players. Leveraging these features, we develop an accurate and robust server-side aimbot detector called AimDetect. The cascaded classifier of AimDetect is able to detect the inconsistency between performance and skillfulness of cheaters. We implement a prototype of AimDetect and evaluate its effectiveness using the real traces of the two most popular FPS games, CS-1.6 and CS:GO. Our evaluation results show that AimDetect can capture almost all of the aimbots with very few false positives and minor overhead. In the future work, we plan to implement and evaluate bait-target and AimDetect across more FPS games. Moreover, since our current measurement study lasts no more than one month, we will deploy more public game servers and attempt to collect all kinds of human playing behaviors over an even longer time period.

References

[1] AMX Mod X. http://www.amxmodx.org/.

[2] Cheating in online games. http://en.wikipedia.org/wiki/Cheating_in_online_games.

[3] Crossfire. http://en.wikipedia.org/wiki/CrossFire_(video_game).

[4] Crossfire official forum. http://bbs.cf.qq.com/forum.php.

[5] Crossfire: Tencent's top earning free-to-play game you've never heard of. http://www.forbes.com/sites/insertcoin/2014/01/20/crossfire-tencents-top-earning-free-to-play-game-youve-never-heard-of/.

[6] Cs:go aimbot. http://www.mpgh.net/forum/showthread.php?t=1005526.

[7] First-person shooter. https://en.wikipedia.org/wiki/First-person_shooter.

[8] Geolite2-city. http://dev.maxmind.com/geoip/geoip2/geolite2/.

[9] Global games market forecast by 2019. http://venturebeat.com/2016/03/14/global-games-market-forecast-to-hit-100b-by-2019/.

[10] Metamod. http://metamod.org/.

[11] Official forum of crossfire. http://bbs.cf.qq.com/forum.php?mod=forumdisplay&fid=30829.

[12] Sourcemod. https://www.sourcemod.net/.

[13] Steam stats. http://store.steampowered.com/stats.

[14] sXe. http://www.sxe-injected.com/.

[15] H. Alayed, F. Frangoudes, and C. Neuman. Behavioral-based cheating detection in online first person shooters using machine learning techniques. In *Computational Intelligence in Games (CIG)*, 2013.

[16] N.E. Baughman and B.N. Levine. Cheat-proof playout for centralized and distributed online games. In *IEEE INFOCOM*, 2001.

[17] D. Bethea, R. Cochran, and M. Reiter. Server-side verification of client behavior in online games. In *NDSS*, 2010.

[18] E. Bursztein, M. Hamburg, J. Lagarenne, and D. Boneh. Openconflict: Preventing real time map hacks in online games. In *IEEE Symposium on Security and Privacy (S&P)*, 2011.

[19] C. Chang and C. Lin. Libsvm: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2011.

[20] R. Cochran and M. Reiter. Toward online verfication of client behavior in distributed applications. In *NDSS*, 2013.

[21] L. Galli, D. Loiacono, L. Cardamone, and P. L. Lanzi. A cheating detection framework for unreal tournament III: A machine learning approach. In *Computational Intelligence and Games (CIG)*, 2011.

[22] S. Gianvecchio, Z. Wu, M. Xie, and H. Wang. Battle of botcraft: Fighting bots in online games with human observational proofs. In *ACM CCS*, 2009.

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

[24] E. Kaiser, W. Feng, and T. Schluessler. Fides: Remote anomaly-based cheat detection using client emulation. In *ACM CCS*, 2009.

[25] E. Lee, J. Woo, H. Kim, A. Mohaisen, and H. Kim. You are a game bot!: Uncovering game bots in mmorpgs via self-similarity in the wild. In *NDSS*, 2016.

[26] S. Mitterhofer, C. Kruegel, E. Kirda, and C. Platzer. Server-side bot detection in massively multiplayer online games. *IEEE Security and Privacy*, 7(3), May 2009.

[27] S. Yu, N. Hammerla, J. Yan, and P. 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 (ICONIP)*, 2012.

[28] S. Yu, N. Hammerla, J. Yan, and P. Andras. A statistical aimbot detection method for online fps games. In *The 2012 International Joint Conference on Neural Networks (IJCNN)*, 2012.