BotScreen: Trust Everybody, but Cut the Aimbots Yourself

Minyeop Choi[1], Gihyuk Ko[2,3], and Sang Kil Cha[1,2]
[1]*KAIST* [2]*Cyber Security Research Center at KAIST* [3]*Carnegie Mellon University*
{okas832,gihyuk.ko,sangkilc}@kaist.ac.kr

Abstract

Aimbots, which assist players to kill opponents in First-Person Shooter (FPS) games, pose a significant threat to the game industry. Although there has been significant research effort to automatically detect aimbots, existing works suffer from either high server-side overhead or low detection accuracy. In this paper, we present a novel aimbot detection design and implementation that we refer to as BotScreen, which is a client-side aimbot detection solution for a popular FPS game, Counter-Strike: Global Offensive (CS:GO). BotScreen is the first in detecting aimbots in a distributed fashion, thereby minimizing the server-side overhead. It also leverages a novel deep learning model to precisely detect abnormal behaviors caused by using aimbots. We demonstrate the effectiveness of BotScreen in terms of both accuracy and performance on CS:GO. We make our tool as well as our dataset publicly available to support open science.

1 Introduction

Esports, which is a form of competitive video gaming, has been growing rapidly in recent years. In 2019, the global esports revenues reached $1 billion [18], and the number of esports viewers has become over 540 million as of 2023 [54].

Cheating in online games poses significant threats to the game industry, and it is indeed a major security threat to the esports industry. Cheaters can directly impact the revenue of the game publishers by annoying benign players and making them quit the game [58]. Cheating also undermines the fairness of games, and thus, damages the E-sports industry, which is now a billion-dollar market [26]. Although game publishers have been trying to prevent cheating, it is still difficult to detect cheating users in online games, causing significant damage to the game industry [34, 40, 44].

First-Person Shooter (FPS) games are no exception. In particular, "aimbot" is one of the most significant cheating threats in online FPS games, which helps a user to automatically aim towards an enemy. Aimbots enable less-skilled players to kill their opponents without having to carefully aim at them. Even experienced or professional game players can leverage aimbots to enhance their gaming capabilities [53, 56, 60].

Hence, there has been substantial interest in devising an anti-aimbot solution both from academia and industry. In particular, there have been many machine learning techniques that use statistical characteristics of aimbots [2, 19, 22, 36, 62–64]. For instance, Valve Corporation, the publisher of "Counter-Strike: Global Offensive" (CS:GO for short), has started to use deep learning to detect aimbot users [41].

However, current solutions suffer from one of the following challenges: (**C1**)–(**C4**).

(**C1**) Most existing solutions operate at a server side [2, 19, 36], causing concentrated workloads. As an example, Valve Corporation had to employ server machines with 3,456 cores to process 600K matches per day [41]. Unfortunately, such server-side detection mechanisms cannot scale with respect to a large number of players as the server is responsible for analyzing all the events.

(**C2**) Previous solutions are also troubled with low accuracy as they do not monitor sophisticated movements of players in order to lessen the burden on the servers. For example, Han *et al.* [22] focus on recorded statistics about game plays, such as playtime and winning rates. While the approach reduces sever-side overhead, such statistics are not directly relevant to the use of aimbots, thereby making the prediction less precise.

(**C3**) Existing client-side solutions [63, 64] suffer from memory tampering. That is, an attacker can manipulate the game state by directly accessing the memory and changing or dropping the network packets to the server as needed.

(**C4**) While it is imperative to have a high-quality game-play dataset to evaluate aimbot detection techniques, there is no publicly available large-scale dataset that contains both benign and malicious game-play data of real FPS players. Oftentimes, researchers populate aimbot data by playing a game by themselves [2, 19, 63, 64], but mimicking realistic behaviors of real cheaters remains challenging. Furthermore, gathering large-scale game data from high-profile FPS games is inherently difficult as it requires significant reverse engineering. Hence, previous works often rely on a proof-of-concept game [2] or an outdated game [36], where source code is available.

In this paper, we propose BotScreen, a novel aimbot detection system that tackles all the aforementioned problems. We demonstrate our idea on CS:GO, a mainstream FPS game, by a proof-of-concept aimbot detection system. BotScreen runs on the client side. Hence it does not incur any server-side overhead (**C1**). It also achieves high detection accuracy by leveraging a deep-learning model that analyzes players' aiming movements (**C2**). BotScreen manages game data within a Trusted Execution Environment (TEE), namely Intel SGX [12], so that it is difficult even for a strong attacker to tamper game data (**C3**). Finally, we hired game players, whose level of expertise ranges from average to professional, to collect realistic game-play datasets with or without using an aimbot (**C4**). When collecting cheating data, we asked each player to customize an aimbot setup to make it as human-like as possible based on their own domain knowledge.

The key innovation of BotScreen lies in how we achieve distributed detection of aimbots. Traditionally, client-side game hack detection has been limited because strong attackers can always manipulate the client-side game states. However, thanks to the recent advances in hardware-assisted TEE, one can secure a detection engine from memory tampering. This design choice allows each client to monitor complex movements of each player without having to worry about server-side overhead.

Moreover, our distributed design enables efficient detection of aimbots as each client only needs to monitor nearby players. In FPS games, players typically do not receive every game event in order to reduce the rendering cost. It is only when an opponent is nearby that a player can observe the opponent's game-play data. Therefore, BotScreen naturally ignores unnecessary game data while detecting an aimbot.

Our detection engine, inside a TEE, monitors every nearby opponent's aiming behaviors and decide whether its movements are natural (like human) or not. To do so, we devise an unsupervised deep-learning model to precisely detect aimbot cheaters. Our model learns normal aiming behaviors from normal players, and detects abnormal aiming movements from cheaters. Particularly, we use a Recurrent Neural Network (RNN) to distinguish between human- and bot-like behaviors. We note that RNN is specifically designed for handling sequential data, and game data are essentially a series of events in time order.

We design and implement BotScreen, which is the first publicly available aimbot detection system that runs on CS:GO, a mainstream FPS game. We evaluate BotScreen on a large dataset obtained from real game plays. Specially, we hired 14 players who are actively participated in a FPS game clan, and collected their game-play data, which include 7,817,380 frames and 93,044 player actions. Our evaluation shows that BotScreen can detect aimbots with 97.64% accuracy. When compared with previous detection methods, BotScreen's accuracy is 9% higher than the second best performing method. Moreover, we demonstrate that BotScreen incurs marginal overhead on modern machines. The contributions of this paper are as follows.

- We demonstrate BotScreen, a novel aimbot detection system, enabling distributed aimbot detection.
- We present a novel aimbot detection model with RNN.
- We evaluate our system on a real-world dataset.
- We make our system as well as our dataset publicly available (link).

2 Background

There are mainly three cheating mechanisms in FPS games: aimbot, Extra-Sensory Perception (ESP) hack, and wallhack. Aimbots help players shoot more accurately by automatically move the crosshair to an enemy in sight. ESP hacks display extra information about game objects including their health, name, and equipment. Wallhacks allow the user to see through a wall and even move or shoot through a wall. Our focus in this paper is on aimbot detection. With an aimbot, even a novice player can absolutely destroy a skilled player.

2.1 Aimbot Detection

Theoretically, it is easy for servers to detect the use of a game cheat because they can naturally control the exposure of sensitive data and see every game action of clients. However, server-based detection incurs significant costs and does not scale well with the increasing number of users.

Client-side aimbot detection, although it is more scalable, is also challenging. Modern commercial anti-cheat solutions [20, 24, 59] try to monitor and block cheating processes by making a blacklist of applications. But attackers can always bypass such a solution with the use of higher-level security ring or by changing their program signatures [29]. In this paper, we propose a new client-side solution that is simple yet effective.

2.2 Sophisticated Modern Aimbots

Notably, modern aimbots provide various advanced features that enable sophisticated aiming control. They often adaptively improve aiming and shooting performance [60], and provide user-configurable options to let the users fine control the level of assistance. For example, we summarize options of aimbots used by Osiris [33], the most popular open-sourced CS:GO aimbot, in Table 1. Other commercial aimbots, such as WinX Private [16], Project Infinity [23], and aimware [67], also employ similar options.

Highly motivated cheaters will set up those configuration parameters in such a way that their behaviors look as natural as possible. For example, aimbot users often enable **O1** because aiming an invisible player can raise suspicion. They also

Table 1: User-configurable options used in Osiris, one of the most popular CS:GO aimbots.

Option	Description
(**O1**) Visible Only	Aim only on visible players.
(**O2**) Scoped Only	Enable only when using a scope with a sniper rifle.
(**O3**) Ignore Flash	Aim a target even if it is invisible due to flashbang.
(**O4**) Ignore Smoke	Aim a target even if it is invisible due to smoke.
(**O5**) FoV	Field of View (FoV) in which aimbot is enabled.
(**O6**) Bone	Bones, e.g., head, stomach, pelvis, etc., to aim at.
(**O7**) Smooth	Adjust auto-aiming movements to make it more human-like.
(**O8**) Aim Tolerance	Aiming tolerance against big moves, e.g., jumps.

adjust **O7** to control how fast aiming changes to pretend to be human, and change **O8** to control aiming accuracy while a player jumps or runs.

Nevertheless, aiming actions with or without an aimbot will always differ. Hence, one should be able to distinguish the use of aimbots by elaborating a model. Our evaluation indeed includes detecting such a sophisticated aimbot user. Specifically, we ask a player to adjust an aimbot with their own expertise to make it like a human. We show that our model is robust against such a sophisticated adversary as we will discuss in §5.7.

2.3 Intel SGX

Intel Software Guard eXtension (SGX) [12] is an instruction set that allows creation of an enclave, which is a private region of memory that cannot be accessed by other processes. In this paper, we use Intel SGX to construct a Trusted Execution Environment (TEE) on each client machine to securely monitor players' actions. Note, we do not argue that SGX is foolproof as we discuss further in §6, but is a practical solution for our use case.

3 Overview

This section provides a high level overview of BotScreen. We start by defining our threat model. We then describe the overall architecture of BotScreen.

3.1 Threat Model

In this paper, we assume that an aimbot can change the input of a target game process without being detected. This means traditional client-side cheating detection, such as checking the integrity of game binaries, will not help.

However, our aimbot can manage the input only in a way that it does not break the game rule. For example, it *cannot* kill the opponents without shooting them. That is, it cannot directly alter the game state, because it is beyond the ability of aimbots. This is indeed a realistic assumption because such a ground-breaking change is readily visible by other peers.

We also assume that our aimbot can arbitrarily manipulate keyboard and mouse inputs according to the current game context. Thus, it helps users to aim and shoot the opponents while preventing a team kill. Sophisticated aimbots can also use the game context to imitate human behaviors by manipulating aiming movements. It is noteworthy that our deep learning model is not accessible to attackers, as it is only temporarily stored in an enclave by our design (§3.2).

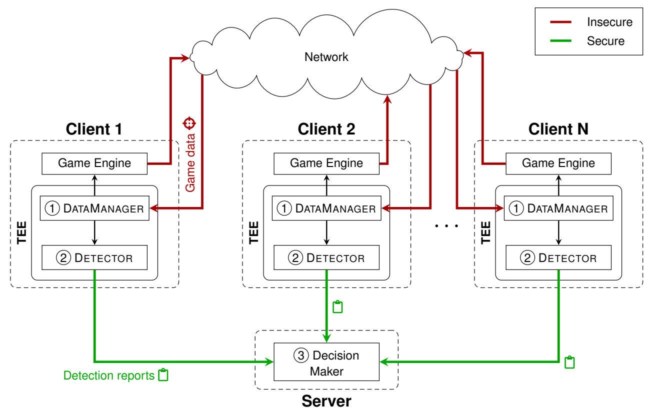

Figure 1: BotScreen architecture.

3.2 BotScreen Overview

Figure 1 illustrates the overall architecture of BotScreen. BotScreen achieves distributed aimbot detection by offloading detection tasks onto each client, i.e., every player polices each other, thereby resolving (**C1**). As each client can monitor sophisticated actions of nearby players, our design naturally addresses (**C2**). At a high level, BotScreen operates in the following three steps during a game play.

⓪ As a preprocessing step, each client performs two tasks before a game starts. First, each client establishes a TLS session with the server (inside its TEE) in order to securely deliver detection reports. Second, each client downloads our deep learning model from the server and temporarily stores the model in its enclave. This way, attackers cannot access our model.

① When a game starts, each game engine takes user inputs and shares its action information through the network with other clients. Specifically, DataManager (which is located inside a TEE of each client) takes in a stream of game events of all the nearby clients from the network and converts it into a stream of vectorized data. It also relays all the received game events to (1) the game engine to render the game; and (2) Detector to check who is cheating. Due to the nature of FPS games, major actions of other players are visible by each player, i.e., each client, as long as they are within the sight. We further detail the conversion process in §4.1.

② DETECTOR (in a TEE) then reads in the vectorized data and analyzes the shooting behaviors of other clients to decide whether they use an aimbot or not. The detection result is transferred to the server on the fly using the secure channel. Note that DETECTOR is essentially our deep-learning model, which can recognize abnormal shooting behaviors that deviate from normal human behaviors. We discuss the details of our model in §4.2.

③ Our server collects streams of detection reports from all the clients to detect cheater(s). If there is a cheating report, then the server can take a necessary action, such as disqualifying the cheater's account. Since DATAMANAGER and DETECTOR are within a TEE and all the reports are transferred through a secure channel, our server can trust reports from the clients, thereby addressing (**C3**). Although every report is trustworthy, our model can still produce a false alarm. Thus, our server will make a delayed decision after collecting a sequence of reports from each client. We detail this design in §4.3.

Consistency of Game Data. The design of BotScreen ensures that both game engine and DETECTOR view the same game data. This means our detection is performed only on the events that are used to render the game. While aimbots can change the players' actions, those changed behaviors are readily visible by the others.

On the Infeasibility of Adversarial Attacks. Our TEE-based client detection makes targetted adversarial attacks [7, 30, 43, 47, 57] extremely challenging under our threat model. First, any white-box adversarial attacks [30, 43, 47, 57], which require access to the pre-trained weights of the victim model, will fail as BotScreen stores pre-trained weights of RNN in a secure enclave.

A more diligent adversary may conduct black-box attacks, either by training a 'substitute' model to craft transferable adversarial inputs [4, 7, 66], or by estimating the victim models' decision boundary [8, 21]. However, both approaches require the attacker to query the victim model to collect multiple input-output pairs, which is not easily achievable under BotScreen—any outgoing detection report is sent via a secured channel between each client's TEE and the trusted server. There is still a possibility that the adversary can collect input-output pairs by potentially compromising SGX as we will discuss in §6.

Unless the adversary can break the TEE's security, it has to either blindly train a substitute model without any query access to the victim model, or inject random noises hoping to alter the model's result [48]. We note that adversary under this limited setting is referred to as 'no-box' attacker [8], and only a few studies exist [6, 35, 65]. To our knowledge, 'no-box' attacks on RNNs have not been extensively explored so far.

In addition to the difficulties in crafting adversarial inputs, we argue that it is impractical for the attacker to continuously inject adversarial inputs due to the real-time nature of FPS games. In order for the attacker to keep up with the game, it needs to craft adversarial inputs for *every frame*, meaning that each input needs to be created in under 17ms (assuming the game is run at 60 frames per second). While not directly comparable, a state-of-the-art white-box technique against RNN took at least 3.15 seconds to generate an input [38], for instance. Thus, we argue that adversarial attacks aside from injecting random noises are not practical.

4 Aimbot Detection

Our aimbot detection technique is inspired by a simple observation: aiming movements significantly vary with an aimbot. Specifically, dishonest players using an aimbot tend to have drastic and sudden angle changes until they obtain the target aim, but aiming movements become stable once the target aim is obtained. Although there have been various learning-based aimbot detection approaches exploiting the same intuition [14, 41, 63, 64], they all rely on a supervised learning approach, which requires both benign data and cheating (with an aimbot) data to train their models.

On the other hand, BotScreen uses *unsupervised* learning approach to train RNN. Hence, we simply need to gather benign data, which are far easier to obtain than cheating data—cheaters inherently do not want to reveal their identity, thus hiring them to collect real-world cheating data is not feasible. Moreover, our model detects aimbots by monitoring every event adjacent to a firing event, and hence, it allows a user to pinpoint exact cheating moment. With this idea, we design and implement a novel aimbot detector for CS:GO. Note that our detection model focuses on CS:GO. But our model design can be applied to other FPS games or can be tuned to detect aimbots in other games.

In this section, we first describe how we process CS:GO game events, to extract several key features that we use in our model (§4.1). We then explain how we use a RNN model to detect the use of aimbots (§4.2).

4.1 DATAMANAGER

BotScreen tracks every observable game event generated from all the observable players in order to decide whether each of them is using an aimbot or not. This is possible because every FPS game client constantly propagates their action events to others at real time through the network.

Note, however, that those game events give only partial information about the players' actions. For example, not every mouse movement, but only a subset of them, will be transferred through the network for efficiency. Plus, each user can

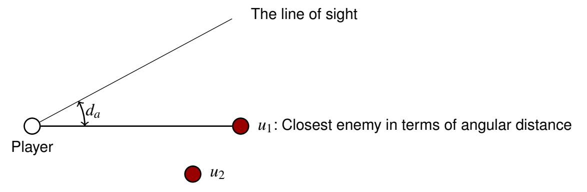

Figure 2: d_a is an angle between the line of sight and a potential target u_1.

only observe actions of nearby players because the game engine does not need to render invisible players at all. Such restrictions equally apply to our aimbot detection scheme.

This design provides several benefits. First, we do not need to modify the game engine for detecting aimbots because our detection scheme relies solely on data that are readily available for regular game clients. Second, each client can handle only a subset of game events, making our detection more focused and efficient.

To ease the explanation, let us denote observable events (i.e., players' actions) from a FPS game client as a series of tuples $(t, u_i, \vec{p}, \vec{a}_s, \texttt{e})$, where:

- t: time of observation.
- u_i: ID of the player being observed.
- $\vec{p}$: the player's position (in Cartesian coordinates).
- $\vec{a}_s$: the player's aiming angle (in spherical coordinates).
- `e`: in-game event (for the player u_i).

Here, an in-game event `e` corresponds to any game event generated by or sent to the player, such as when the player (or another player) *fires* its weapon, gets *hit*, or is *killed* by other player. We denote such events by `fire`, `hit`, and `dead`, respectively. In timestamps where no such events occur, we simply leave the field blank.

Note that there can be multiple events for the same t when there are more than one observable players at the same time. At a high level, DATAMANAGER takes in a stream of observable events as input from the network, and sends a stream of transformed feature vectors as output when in-game event that triggers detection occurs. It includes three major steps: (1) vector computation, (2) normalization, and (3) sanitization.

4.1.1 Vector Computation

DATAMANAGER in each client first transforms each observable event $(t, u_i, \vec{p}, \vec{a}_s, \texttt{e})$ into a feature vector $(t, u_i, \vec{p}, \vec{a}_c, \texttt{e}, \texttt{aim})$, where $\vec{a}_c$ is the aiming angle of the player u_i represented as a unit vector in a Cartesian coordinate, and `aim` is a Boolean value indicating whether the player u_i is currently aiming at any opponent. We convert a vector in a spherical coordinate ($\vec{a}_s$) to one in a Cartesian coordinate ($\vec{a}_c$) because differences in angles do not always entail different meanings—0° and 360° have two different values even though they represent the same state.

In order to obtain `aim`, each client first computes d_a, which represents the smallest angular distance between the player's current line of sight and any *observable* opponents. In Figure 2, enemy u_2 is closer to the player in terms of Euclidean distance, but u_1 is closer to the player in terms of angular distance. Thus, we consider u_1 as the player's current target, and calculate d_a based on it. When d_a is smaller than a threshold (30° in our current implementation), we set `aim` to `true`, otherwise `false`. Note that `aim` is an *estimated* value, as there are cases where an observer can have a wrong value. For example, when the player is aiming at an opponent that the current observer cannot see. This problem does not affect our detection results, because our model will make a decision only if both a player and its target are visible by a client. Furthermore, the value of `aim` does not affect the decision: only the aiming angle will eventually affect the decision.

4.1.2 Normalization

In practice, each client collects an event *at the exact time* it occurs. As a result, collected events have irregular timestamps, producing a time series with uneven time intervals. Since RNNs generally expect to handle regular sequences, collected logs need to be *normalized* to a fixed time interval.

In our current implementation, we conduct *linear interpolation* to fill in missing values by following the linear trend among existing observations. Since most modern FPS games run on 60 frames per second, we normalize the collected events according to time interval of 16ms. Suppose a player was at $(1.0, 1.0, 1.0)$ when $t = 0$, at $(1.1, 0.9, 0.1)$ when $t = 8$, and at $(1.5, 0.6, 0.8)$ when $t = 34$. After the normalization step, we get three events at $t = 0$, 16, and 32. For example, we have an interpolated event $(1.235, 0.811, 0.376)$ at $t = 16$. As the time interval (i.e., 16ms) is too small to have sudden changes, we believe that linear interpolation is sufficient for normalizing collected game-play logs.

4.1.3 Sanitization

The final step is to filter out unnecessary events before feeding the transformed events into DETECTOR. In particular, we often observe a sudden change in position and aiming direction when a player dies and respawns (i.e., when `e = dead`), in which case our deep-learning model can incorrectly flag the player as suspicious. This happens in a deathmatch where the goal is to kill other players as many times as possible. In our current implementation, we always remember the previous event, and compare the current event with the previous one to detect abrupt changes in players' locations and aiming directions. When such a case is identified, DATAMANAGER will simply not send the event to DETECTOR.

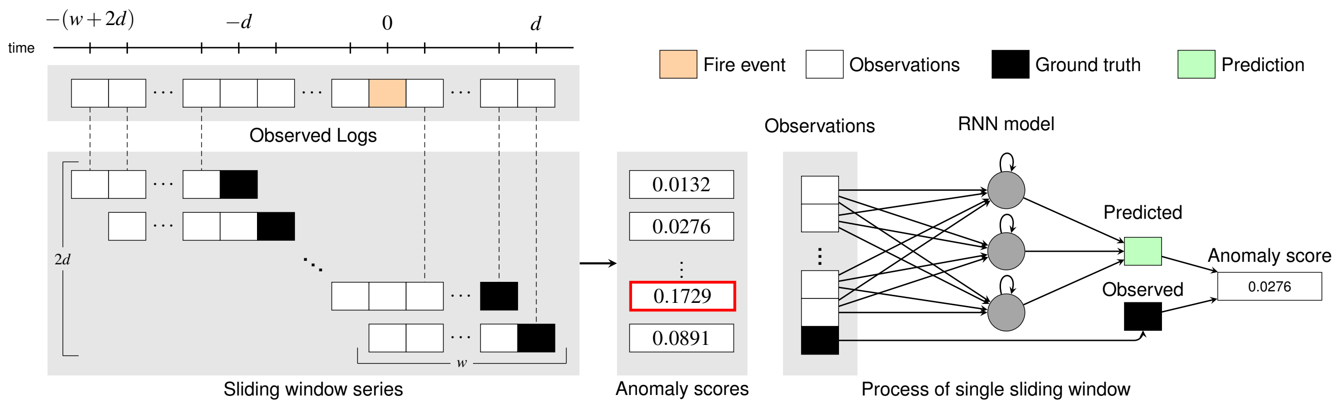

Figure 3: Anomaly scoring based on RNN used in DETECTOR.

4.2 DETECTOR

In BotScreen, each client detects aimbots, in a distributed manner, by checking 'abnormal' aiming movements at firing events using a model built from observing normal aiming movements. To model normal aiming movements, we use a Recurrent Neural Network (RNN) trained *only on* the observation logs of benign players. Specifically, DETECTOR takes a stream of pre-processed events returned by DATAMANAGER, and analyzes the events in the following three steps: (1) event filtering, (2) anomaly scoring, and (3) reporting.

4.2.1 Event Filtering

In an attempt to bypass easy detection, modern aimbots default to the 'between shots' mode, where aimbot locks on to a target *only when* the user has triggered (fired) the weapon. When this mode is on, an aimbot user and a benign user are essentially indistinguishable unless a weapon is fired. Therefore, we do not track every event during the game play. Instead, we only focus on each firing event (i.e., `e` = `fire`) as well as relevant events adjacent to (before and after) the event. We also filter out all targetless `fire` events (i.e., `aim` = `false`), as aimbots may not be activated without a proper target.

In practice, we accumulate more events before than after the exact event the weapon is fired as RNNs generally require a set of previously observed data points to operate. For a given RNN's window size w and a predetermined duration d, we accumulate a total of $w+2d$ *relevant events* for each `fire` event, which consist of $w+d$ events before a fire and the rest d events including the `fire` action.

We empirically select the parameters to be $w = 20$ and $d = 10$ in our current implementation to optimize BotScreen's detection performance, as setting w and d too small or too large can provide insufficient or excessive information to each set of relevant events, respectively. §5.3 presents our experimental results where we measure the detection performance under different model parameters.

4.2.2 Anomaly Scoring

To effectively detect aimbots, we distribute a pre-trained RNN, which encodes normal players' aiming movements, to each client. Using the RNN model, each client calculates an *anomaly score* for each set of observed aiming movements (before and after a `fire` event), where a higher score indicates the likelihood of being an aimbot user.

Specifically, our RNN model takes in $w-1$ consecutive observations of aiming angles ($\vec{a}_c$) as input, and predicts the aiming direction for the wth event. The anomaly score is then a prediction error, calculated by comparing the L1 distance between the predicted value and the actual value for the wth event. Since our RNN is trained *only* on the aiming movements of benign players, aiming movements of an aimbot will result in notably worse RNN predictions compared to those of benign players. Consequently, we will observe higher anomaly scores for aimbot users.

Figure 3 illustrates how our RNN-based anomaly scoring works. By dividing firing-relevant events into multiple sliding windows, we obtain $2d$ distinct RNN input sequences of length w, hence resulting in a total of $2d$ anomaly scores for each `fire` event[1]. Among all the anomaly scores, we select the maximum as the resulting anomaly score of the event.

We note that our event-wise inspection approach has two main advantages over previously suggested methods that use aggregated statistics, such as mean changes in aiming acceleration [64]. First, it is possible to identify exact moment of cheating. Second, it is significantly more difficult to develop an aimbot that can avoid our detection mechanism, as it will need to monitor and control event-level movements.

[1]While RNNs do not necessarily require a fixed input sequence length, we fix the length of input sequences in an attempt to normalize resulting prediction errors and enable more effective anomaly detection.

4.2.3 Reporting Results

After computing anomaly scores for each `fire` event, DETECTOR compares those scores with a threshold in order to determine the final label. The threshold is determined with a small validation dataset.

In BotScreen, we propose and evaluate on two basic criteria in determining thresholds: (1) *best accuracy*, and (2) *best precision*. As the names suggest, *best accuracy* chooses a threshold such that aimbot prediction accuracy is maximized in a given validation data. Likewise, *best precision* chooses a threshold such that no false positives exist in a given validation data. As a result, the former achieves the best prediction accuracy by balancing FPs and FNs, while the latter prevents any benign user to be falsely accused of cheating. We note that the latter approach is more practical, as falsely punishing benign users will harm the game's usability.

Once the decision is made, each client sends the cheating report to the server using a secure channel. Each cheating report is simply a tuple $(\texttt{aimbot}, u_i, t)$, where `aimbot` is a Boolean variable denoting the detection result (`true` if an aimbot is used by the player u_i), u_i is the cheater being observed, and t is the timestamp of the corresponding `fire` event. Note, in our scheme, each client sends a report for every observed `fire` event instead of sending a single aggregated result for the entire duration of the game. The purpose of this design is twofold: (1) to prevent attackers from guessing the decisions made by each client, and (2) to provide more flexibility to the server when aggregating the results.

4.3 Report Aggregation Policy

It is the server, or a game provider, who is responsible for making a final decision based on the reports obtained from the clients. One straightforward policy is to strictly disqualify a player if there is at least one cheating report. This policy, however, suffers from potential false alarms caused by the inaccuracy of our model, although each report is trustworthy.

We address this challenge by delaying our decision. Since cheating reports are submitted multiple times (11.83 times on average in our experiments) during a single game, our server can accumulate those reports, and make a more informed decision later. Specifically, we can disqualify a player when the number of consecutive cheating reports during a game play is larger than a threshold θ. For example, suppose our model has a false positive rate of 5%. Then, the probability of falsely accusing a player as a cheater is only 0.00003% when $\theta = 5$. This way, game providers can easily offset the inaccuracy of the model.

5 Evaluation

In this section, we evaluate the effectiveness of BotScreen on real-world game-play data of CS:GO.

5.1 Experimental Setup

5.1.1 Ethical Assessment

We discuss the ethics of this research based on the guideline of the Menlo Report [31]. This research has received IRB approval as human subject research. We considered the participants in terms of rights, risk, and equality while planning and executing the experiment.

We also carefully followed the Steam Subscriber Agreement [10]. In this agreement, we are allowed to deploy a dedicated server with supported options. We only used a dedicated (private) server with VAC disabled option, which prevents the harm to official or third-party servers. Also, CS:GO provides "Insecure Mode" which allows to inject any code into CS:GO client [11]. We ordered participants to run CS:GO with insecure mode and inject the Osiris with a DLL injection method which is the suggested method by Valve.

Finally, we did not create Osiris nor support cheat developers. We only used it for our research purpose and made participants to remove the provided materials to prevent subsequent harm to other players.

5.1.2 Data Collection

Hired Players. Recall that there is no high-quality public dataset for evaluating aimbot detectors (**C4**). To address this challenge, we collected real-world CS:GO game-play data by recruiting players from a FPS game clan. We hired 14 players in total, whose skill levels are summarized in Table 8 (Appendix B). We note that the hired players were experienced individuals of various skill levels from average to expert, but not novices. This is primarily in effort to better reflect the real-world FPS game scenarios. Aimbots are known to "resemble excellent honest players" [36], so that they share several traits such as high aiming accuracies and fast reaction times. This naturally means that distinguishing aimbots from experts is more challenging than from novices.

Game Setup. In order to collect real-world game-play data, we set up a private CS:GO server, and collected data from a total of 28 classic deathmatch games, where each game ran for 10 minutes as summarized in Table 9 (Appendix B). For each game, we randomly chose a map among the following six: `de_dust2`, `de_inferno`, `de_nuke`, `de_mirage`, `de_vertigo`, and `de_train`. We also designated a varying subset of the participants as 'cheaters' while ensuring each player evenly takes the role of benign player and cheater in order to represent a variety of aimbot usages. Specifically, we collected game-play logs under three scenarios: no-cheater, one-cheater-per-team, and all-cheaters game. A detailed benign-cheater player assignments as well as aggregated statistics from each game can be found in Table 9.

We asked each player in the 'cheater group' to fine-tune the aimbot configurations (Table 1) based on their own expertise

Table 2: Architecture of SGRU used for anomaly scoring. L is the length of input sequence.

Layer	Type	Output Shape	# of Parameters
1	Input	(3,L)	0
2	GRU (bidirectional)	(128,L)	26,496
3	Dropout 10%	(128,L)	0
4	GRU (bidirectional)	(128,L)	74,496
5	Dropout 10%	(128,L)	0
6	GRU (bidirectional)	(128,1)	74,496
7	Fully connected	(3,1)	387

to make their movements as natural as possible. This is due to two reasons: (1) to simulate advanced cheaters who attempt to avoid being detected by mimicking real players, and (2) to collect aimbot data under a variety of configurations.

Event Logging. During a game, each player used our event logger implemented on top of Osiris [33] to collect every game event. Note that our event logger records every observable players' positions as well as their aiming angles in each frame, recorded *in perspective of the player*. It also monitors when each player triggers a weapon (`fire`), gets hit by another player (`hit`), and gets killed (`dead`).

To support open science, we publicize this dataset at https://zenodo.org/record/8003842. To our knowledge, this is the first public dataset for aimbot detection obtained from a high-end FPS game.

5.1.3 RNN Model and Training

We selected Stacked Gated Recurrent Unit (SGRU for short) as our choice of RNN for anomaly detection. GRU is a well-studied variant of RNN, known to be efficient in training while showing similar performance to LSTM [9]. We construct SGRU by stacking 3 layers of bidirectional GRU with 64 hidden states, followed by a fully connected layer (Table 2).

To construct a training dataset, we randomly selected a portion of games among the collected dataset. Under an ideal scenario, game-play logs of a player observed by two different observers should be identical. However, in practice, this may not be the case because of the network delays and random packet losses. Therefore, we only use *self-observed* logs for training our model. Since our scheme is based on *unsupervised* learning, logs from the benign users are sufficient to train the model. In particular, we trained a SGRU model using AdamW optimizer [37] with standard parameters (0.001 learning rate and 0.01 weight decay), minimizing the MSE loss, for 64 epochs on a batched dataset of size 64. We present the training losses of our model in Appendix A.

5.1.4 Environment

To train our model, we used a server machine running Ubuntu 20.04 LTS equipped with four Intel Xeon Silver 4214 CPUs (2.2GHz/12 cores) and four NVIDIA RTX 2080 Ti GPU cards. We used Python 3 and PyTorch [50] to implement and train SGRU. Roughly, it took less than 10 hours to train each RNN. To evaluate the performance overhead of our system, we ran CS:GO on the following three different configurations.

- **High-performance:** Windows 11 (Build 22000.613) with Intel i7-10700K (8 cores), 32GB of RAM, and NVIDIA GeForce RTX 3080.
- **Mid-performance:** Windows 10 (Build 19044.1706) with Intel i7-7700 (4 cores), 16GB of RAM, and NVIDIA Geforce GTX 1050.
- **Low-performance:** Windows 10 (Build 19044.1706) with Intel i7-7700 (1 core only, limited the clock speed to 2.4GHz), 4GB of RAM, and CPU-integrated graphics to imitate the minimal requirement of CS:GO.

5.2 Aimbot Detection Performance

Can our SGRU model accurately detect the use of aimbots? To answer this question, we simulated each game using our dataset and applied BotScreen to detect the use of aimbots. While our experiments are performed by simulating games, this does not mean that BotScreen can only run in a simulated environment. We have implemented a system that can detect aimbots at runtime, which incurs only marginal performance overhead as shown in the later part of this section. Simulation is necessary for fair comparison to evaluate and compare different detection algorithms on the exactly the same dataset. We evaluated our SGRU model by dividing our dataset based on two different criteria. One is by dividing the dataset by game (§5.2.1), and the other is by dividing it by player (§5.2.2).

5.2.1 Game-based Split

To test the effectiveness of our model on logs collected at a game level, we first randomly divided the collected game logs into k different splits, where each split contains one or more per-game logs. We then performed k-fold cross-validation on the splits; we used each split to validate the model trained on the remaining $k-1$ splits. We chose $k = 7$ as it allows each split to have identical number of games while following the usual practice of setting k between 5 and 10.

Table 3 presents the detection accuracy of BotScreen measured on each split. Recall from §4.2 that our DETECTOR determines labels by thresholding on anomaly scores, where the thresholds can be decided based on two baseline policies: (1) *best accuracy*, or (2) *best precision*. We denoted the accuracies measured by each threshold as `best_acc` and `best_prec`, respectively in Table 3. Note that we also recorded the number of players in each split as games in our dataset can have different numbers of participants. We calculated a weighted average of the accuracies using the number of players (i.e., the number of predictions) as a weight of each split.

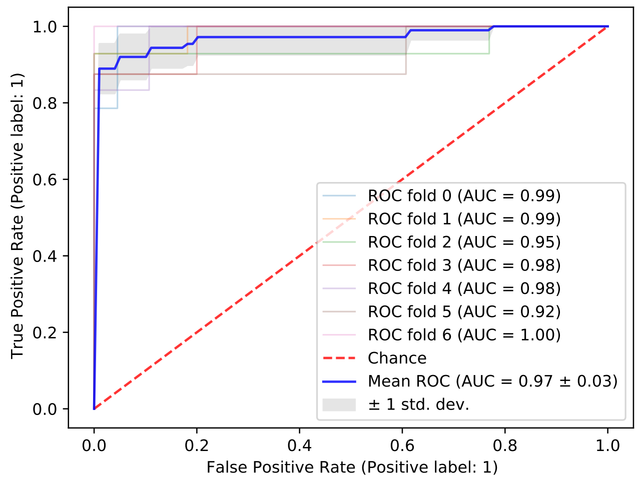

Figure 4: ROC curves for aimbot detection. ROC curves are drawn for each split randomly divided for cross-validation ($k = 7$), represented in thin colored lines. Weighted average of all splits are represented as a thick blue line (Mean ROC).

While the prediction accuracy provides a good first-hand indication on the predictor's performance, it can be misleading when datasets have unbalanced labels. In order to assess how well the anomalies (aimbot users) can be separated from the normal instances (benign users), we plot Receiver Operating Characteristic (ROC) curves for each split in Figure 4 (AUC-ROC values are shown in Table 3).

From both Table 3 and Figure 4, BotScreen consistently shows high prediction accuracies across different splits (min 97.06%, max 100%). Even in the case of `best_prec`, BotScreen maintains a close-to-optimal prediction accuracies. These results clearly indicate that our methodology can precisely detect aimbots in real-world FPS games.

In the repeated experiments on differently sliced datasets, we observed that our method consistently shows high performance. Specifically, when we repeated our evaluations 5 times on each differently sliced dataset, our method accurately predicted aimbots on average of 96.14% and at worst 95.67%, under `best_prec` policy.

5.2.2 Player-based Split

While game-based training and testing (as performed in §5.2.1) is natural for detecting aimbots, game-based splits may contain the same players' data in both training and testing splits. To test if BotScreen is able to correctly classify previously unseen players, we train-test-split the collected logs in player-wise manner, where the training split contains logs from six-seventh of all players (i.e., 12 players), and the test split has logs of the remaining two players. We note that even though the test split only contains the logs of two players, it is obtained from multiple different games where each player may have acted as either a benign player or a cheater. In our evaluation, we considered all 91 (= 14 choose 2) possible train-test combinations to measure the average aimbot detection accuracy.

Table 3: Aimbot detection performance of BotScreen, where each split represents a subset of game-based logs.

Split	# Players	`best_acc`	`best_prec`	AUC-ROC
0	36	0.9722	0.9167	0.9903
1	36	0.9722	0.9722	0.9870
2	40	0.9750	0.9750	0.9451
3	38	0.9737	0.9737	0.9750
4	34	0.9706	0.9706	0.9821
5	36	0.9722	0.9722	0.9241
6	34	1.0000	1.0000	1.0000
Weighted Avg.		**0.9764**	**0.9685**	**0.9712**

Table 4: Aimbot detection performance of BotScreen evaluated on a player basis.

	`best_acc`	`best_prec`	AUC-ROC
Top 3 players	0.9790	0.9790	0.9782
Bottom 3 players	0.9856	0.9856	0.9834
Best accuracy	1.000	1.000	1.000
Worst accuracy	0.9567	0.9567	0.9694
Average	**0.9823**	**0.9800**	**0.9792**

Table 4 shows the results. Similar to Table 3, we measure the prediction accuracies based on `best_acc` and `best_prec` policies, alongside with AUC-ROC. Additionally, we recorded the accuracies at the best and the worst performing combinations, as well as the average accuracies in predicting three best skilled players (A, B, C in Table 8) and three least skilled players (L, M, N).

The results clearly indicate that BotScreen is capable of accurately detecting aimbots, even if the model has not previously seen the player. BotScreen showed above 98% (min 95.67%, max 100%) detection accuracy on average, which is impressive considering that this was an average from all 91 combinations. We also note that BotScreen shows a consistent performance regardless of the players' skill levels—the average difference in detection accuracies was less than 0.6% between groups of three most skilled and three least skilled players, respectively.

5.3 Parameter Selection

Are the parameters selected in BotScreen optimal for the aimbot detection? To justify the choice, we evaluated the performance of BotScreen using different parameter values.

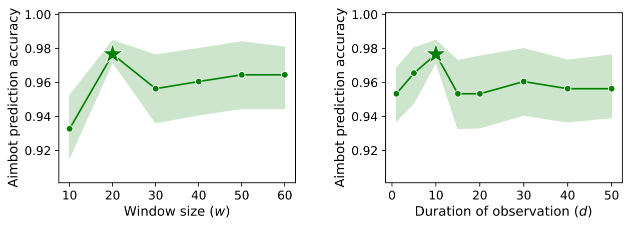

Figure 5: Aimbot detection accuracies of BotScreen with varying w and d. The star ($\star$) marks indicate the parameters used by BotScreen.

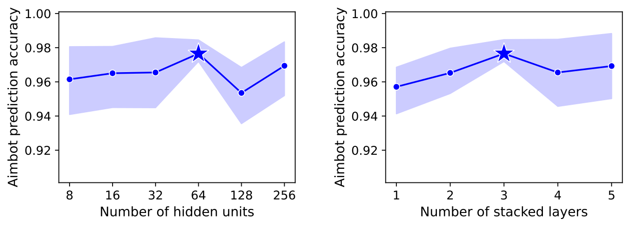

Figure 6: Aimbot detection accuracies of BotScreen with varying hidden units and stacked layers. The star ($\star$) marks indicate the parameters used by BotScreen.

5.3.1 Different Event Sizes

Recall from §4.2 that BotScreen collects a few adjacent *relevant events* for each observed `fire` event based on the two parameters w and d. Here, we show the impact of using different w and d, and empirically justify our choice.

Figure 5 shows the effect of varying w and d on aimbot detection performance. The differences observed from the plots are not so significant, but we can clearly see that the values marked with a star ($\star$) show the best performance. Setting a too small w value, for instance, negatively impacts the accuracy due to limited amount of information gathered. Therefore, we selected $w = 20$ and $d = 10$ for our evaluation as noted in §4.2.

5.3.2 Different RNN Hyperparameters

Next, we test how different RNN model hyperparameters impact aimbot prediction accuracies. Specifically, we recorded accuracies with varying numbers of hidden units in each GRU unit, and with varying numbers of stacked layers in SGRU.

Figure 6 shows the effect of hyperparameter changes for our SGRU model, evaluated on each of the 7-fold validation splits. While all variants show high performance ($> 95\%$ on average), parameters used in BotScreen (marked with $\star$) show the best prediction performance both in terms of accuracy, and their stability.

Table 5: Performance comparison of aimbot detection methods. Each recorded value is a weighted average from 7-fold cross validation. The best results are highlighted in boldface.

Method	Accuracy	FPR	FNR
`th_VarA`	0.7323 (186/254)	0.0000 (0/186)	1.0000 (68/68)
`th_AccA`	0.7323 (186/254)	0.0000 (0/186)	1.0000 (68/68)
`th_Kill`	0.8858 (225/254)	0.0054 (1/186)	0.4118 (28/68)
`ks_AccA` [64]	0.4803 (122/254)	0.5430 (101/186)	0.4559 (31/68)
`os_CAC` [15]	0.6969 (177/254)	0.0484 (9/186)	1.000 (68/68)
`os_LAC` [27]	0.7480 (190/254)	0.2097 (39/186)	0.3676 (25/68)
`os_SMAC` [55]	0.7244 (184/254)	0.2634 (49/186)	0.3088 (21/68)
BotScreen	**0.9764 (248/254)**	**0.0054 (1/186)**	**0.0735 (5/68)**

5.4 Comparative Study

We now compare BotScreen against seven different detection schemes including four implemented methods and three existing open-source tools. We consider two different scenarios: (1) per-game detection, where a decision is made for each game, and (2) history-based detection, where a decision is made for each player based on the player's behavior over several matches.

Comparison Targets We design and implement four aimbot detection methods based on simple statistical features used in prior literature as there is a lack of available solutions. They are similar to our anomaly-based scoring approach, but use different ways to decide labels. The following list summarizes each method:

- `th_VarA`: blames a player when the variation of aiming exceeds a threshold [2, 19].
- `th_AccA`: blames a player when the acceleration of aiming exceeds a threshold [19].
- `th_Kill`: blames a player when the overall kill count exceeds a threshold [36].
- `ks_AccA`: blames a player when the acceleration of aiming is dissimilar from ones observed from benign users based on a statistical test [64].

We picked the thresholds for the first three methods by selecting the one that maximizes the accuracy of each method in a validation dataset (one out of 7 splits). The fourth method (`ks_AccA`) is a reimplementation of [64], where it uses the two-sample Kolmogorov-Smirnov test [39] in order to recognize the dissimilarities between observed statistics from a benign user and an aimbot user. In addition to the reimplemented methods, we evaluated BotScreen against three existing open-source tools listed below. Note, we ran all these tools in our simulation environment for fair comparison.

- COW Anti-Cheat [15] denoted as `os_CAC`.
- Little Anti-Cheat [27] denoted as `os_LAC`.
- SMAC [55] denoted as `os_SMAC`.

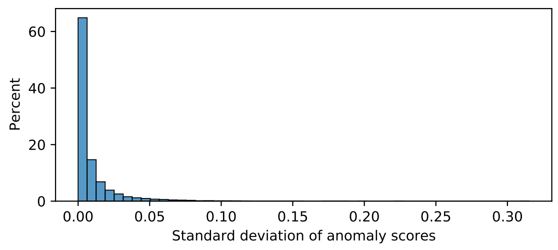

Figure 7: Histogram of standard deviations in anomaly scores.

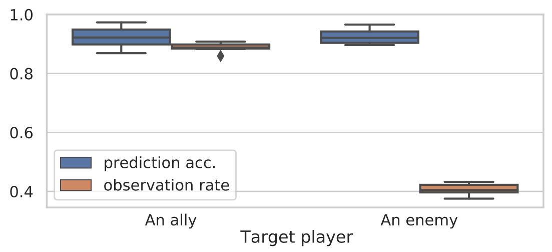

Figure 8: Prediction accuracies and observation rates for the allies and the enemies.

Per-Game Detection Results. We first compared the accuracies of each detector in a scenario where a decision is made per each game. Table 5 summarizes the results. The numbers in the parentheses represent success/failure counts. For BotScreen, we measured the weighted average from 7-fold cross validation as in §5.2. For the reimplemented methods, we recorded prediction accuracy evaluated on the threshold selected by *best accuracy* policy, indicating the maximum accuracy obtainable using the detection method. For the open-source tools, we used their default settings.

The overall results show that BotScreen significantly outperforms all the other methods. We observed more than 9% improvement over `th_Kill`, which is the second best method. We note that `ks_AccA` shows poor performance compared to other methods. This result indeed coincides with the results reported from their paper [64], which shows a poor recall rate (46.83%, to be exact). We also note that `th_Kill` is effective in terms of detecting aimbots. However, simply using kill counts as an evidence of cheating would be discouraging for benign players who rightfully achieved a high skill level.

History-based Detection Results. We also compared the detection accuracies in a scenario where a decision is made per each player based on accumulated logs over a number of games. Specifically, we say a player is a cheater if the player is detected as an aimbot in at least one of the games in the history. To this end, we first extracted the game logs for each player based on their participation in the games, e.g., if a player participated 10 games, then we extracted logs from only those 10 games. We then simulated the detection process in chronological order using the extracted game logs for each player, and analyzed when the use of an aimbot was detected.

Table 10 in Appendix C summarizes the results. There are two notable points. First, BotScreen consistently outperforms the other methods in terms of detection accuracy. Although there are two cases where BotScreen suffered, those were the cases where there are only a few data points with only a single false negative as described in Appendix C. Second, open-source tools show relatively better performance compared to the game-based detection scenario. This is mainly because existing tools are mainly concerned with reducing false positives, and history-based accumulation will offset their imprecision. Overall, BotScreen showed the best performance for detecting a physical aimbot user.

5.5 Effect of Client-side Detection

Since BotScreen is a client-side aimbot detector, its detection performance can be impacted by differences in observations made by each client. Thus, we conduct a number of experiments in order to evaluate how client-side detection affects the performance of our system.

5.5.1 Differences in Anomaly Scores

In our detection scheme, each client uses an RNN to assign anomaly scores for each observed `fire` event. Under ideal conditions, anomaly scores for a given event will be identical (or similar) across different observers. In practice, we can quantify the amount of inconsistency between observers by measuring standard deviation of anomaly scores obtained from different observers for each event. It is, thus, desirable to observe small standard deviations.

Figure 7 shows a histogram of standard deviations measured from different anomaly scores of each `fire` event, which shows significantly small differences: more than 96% are below 0.05. Furthermore, when our model makes final predictions on the same data, only 1.26% of the observations resulted in different prediction results.

5.5.2 Differences in Observation Rates

Next, we study how the observation rate impacts the prediction accuracy. Due to the competitive nature of FPS games, chances of *observing*—being nearby at the moment of—other players firing a weapon depend heavily on whether or not the player is in the same team as the observer. That is, a player is more likely to observe events from allies than from enemies, as more time is spent with them.

Intuitively, less observation rates can lead to worse aimbot detection predictions, as chances in observing any suspicious `fire` events will also decrease. In such a case, our scheme will result in strong disagreements in predictions between allies and enemies, leading to a potential dispute between

Table 6: Frame time with or without BotScreen (averaged over 10 repeated experiments).

	Time (s)	#Frames	Frame Time (ms) Avg.	Min	Max	Low 1%	Low 0.1%
Baseline[1]	6,000	1,454,686	4.125	0.786	172.772	9.499	13.832
BotScreen [1]	6,000	1,428,974	4.199	0.899	156.411	9.830	14.804
Baseline[2]	6,000	1,040,482	5.758	1.262	189.083	12.446	16.973
BotScreen [2]	6,000	913,818	6.566	1.380	136.746	14.957	20.441
Baseline[3]	6,000	185,285	32.379	4.717	1,167.064	91.055	285.298
BotScreen [3]	6,000	145,530	41.224	7.127	1,508.617	112.963	325.929

[1] Evaluated on the high-performance setup (see §5.1.4).
[2] Evaluated on the mid-performance setup (see §5.1.4).
[3] Evaluated on the low-performance setup (see §5.1.4).

teams. Thus, a consistent prediction performance even under low observation rate is desirable.

Figure 8 is a box plot of prediction accuracies and observation rates for the allies and the enemies. Note that while the chances of observing allies are noticeably higher than observing enemies, the prediction accuracies are similar. In fact, the Pearson Correlation Coefficient between observation rate and prediction accuracy is $R = -0.035$, meaning almost no correlation. Thus, we conclude that BotScreen can enable accurate aimbot detection even under low observation rates.

5.6 Performance Overhead

We now measure the performance overhead imposed by BotScreen. Our implementation is based on Osiris [33], and is written in 850 lines of C code. Note that in our current implementation of BotScreen, DATAMANAGER is outside of a TEE because we cannot change the data transferring module of CS:GO (see §6). However, it is straightforward for game providers to modify a game engine to completely realize the BotScreen architecture. Furthermore, such a small difference would only incur a negligible performance difference. To measure the performance overhead, we played a classic deathmatch for 10 minutes with MSI Afterburner [45], a GPU benchmark program, on three different setups shown in §5.1.4. We repeated the experiment 10 times for each setup and reported the averages.

Table 6 summarizes the results with or without BotScreen on the three setups; see Appendix D for detailed results for each experiment. In each setup, BotScreen increased the average frame time by 0.074 ms (1.7%), 0.807 ms (14%), and 8.844 ms (27%), respectively.

Note, however, that the overhead caused by BotScreen is indistinguishable from the frame time errors of CS:GO observed over repeated runs for the high-performance and the mid-performance setup (see Appendix D). Although we can observe noticeable overhead on the low-performance setup, it is not surprising because CS:GO already fully utilizes the resources of the machine, hence it does not have any room to absorb the overhead caused by BotScreen. Therefore we conclude that BotScreen incurs negligible performance overhead on modern machines equipped with SGX.

Table 7: Real-world demonstration of BotScreen on CS:GO, under various aimbot configurations.

	FoV	Smooth	# Detected	# Games
No aimbot	-	-	0	3
Default setting	9.0	10.0	3	3
FoV control	255.0	10.0	3	3
Smooth control	9.0	1.0	3	3

5.7 Real-World Demonstration

Can BotScreen detect aimbots in a real game play? We answer this question by running our implementation of BotScreen during a live game play. Specifically, we prepared two CS:GO clients: (1) player A who uses Osiris, i.e., an aimbot, and (2) player B who uses BotScreen with the model showing best performance from above to detect the use of an aimbot. Player A uses Osiris with varying **O5** (FoV) and **O7** (Smooth). We set the other options to use a default value for simplicity.

We played 5 versus 5 deathmatch games, where 8 of the players are controlled by a computer, and two other players (A and B) are controlled by human players. With four different aimbot configurations, we ran three 10-minute games per each configuration, and reported the detection results in Table 7. As a result, BotScreen was able to successfully detect all aimbots without any false alarm. This evaluation confirms the effectiveness of BotScreen in a live game scenario.

6 Discussion

Limitation of SGX. While Intel SGX provides a TEE, it has several limitations that make it vulnerable to side-channel attacks [13, 17]. This means a highly motivated attacker can potentially extract the model parameters of BotScreen located in the enclave or sniff the detection reports.

We also notice that Intel SGX had been deprecated in recent CPUs [52]. However, other TEEs such as ARM TrustZone [51] and AMD SEV [1] are still available, and it is straightforward to port our scheme to them. Furthermore, Intel announced a new trusted computing technology, named TDX (Trusted Domain eXtension), which will replace SGX [25]. Therefore, the core idea of BotScreen is still valid and will be effective as long as CPU vendors continue to provide trusted computing features. We leave it as future work to extend BotScreen to work with other TEEs, such as TDX and

TrustZone.

Applicability to other FPS games. Although BotScreen is primarily implemented and evaluated on CS:GO, we believe our distributed detection scheme is applicable to *most* modern FPS games for two reasons. First, our detection scheme only uses general features that are essential to any FPS game—player position, aiming direction, and in-game events such as `fire`, `hit`, or `dead`. By definition, any FPS game will require the client to send and receive such events in order to properly run the game. Second, our unsupervised training of the RNN model enables re-calibration for other FPS games. Unlike any other supervised learning based approaches, our scheme does not require data from functioning aimbots in order to be trained. As a result, our method can even be applied to games without publicly known aimbots.

DATAMANAGER implementation. Our current implementation does not put DATAMANAGER inside SGX as it requires patching the game engine. There are mainly two problems. First, we need to reverse-engineer the game binary, which is not a trivial task [32,42]. More importantly, patching the existing game binary is against the EULA of CS:GO. Nonetheless, we believe our design can be easily adopted by game companies as they can patch the game engine directly.

7 Related Work

Statistical Aimbot Detection. Han *et al.* [22] detect aimbots by measuring statistical differences of various features, such as game playtime, experience points earned, and winning rates. Those features are not directly relevant to the use of an aimbot, hence making its prediction fundamentally less precise. Yu *et al.* [63, 64] detect aimbots with two statistical features: cursor acceleration and duration of locking aim on a target. Specifically they compare statistical differences of those features obtained from benign players and dishonest players with the KS test [39]. However, cursor acceleration is not visible from other clients or from the server. Therefore, this approach only works at a client side, which is fundamentally limited by memory tampering. Yeung *et al.* [62] employ five simplified features based on play patterns, such as Boolean variables representing whether a player moves or not, and whether a player changes its aim or not, etc. Unfortunately, those simplified features cannot accurately capture sophisticated usage patterns of aimbots.

Performance-skllfulness Inconsistency-based Detection. AimDetect [36] exploits the fact that aimbot users often show less-skillful behaviors as opposed to their performance, e.g., the number of kills. However, the proposed approach does not work well for highly motivated experts, such as a professional gamer, who wants to win a competition. On the other hand, our RNN-based model can detect the use of aimbots from skillful users as our evaluation shows.

ML-based Detection. Galli *et al.* [19] and Alayed *et al.* [2] propose supervised learning methods. They extract features including aiming angles as in BotScreen. However, they rely on many complex features unlike BotScreen. Plus, their technique requires a labeled dataset obtained from both benign and malicious players, which is inherently difficult to obtain. On the other hand, BotScreen relies solely on a benign dataset, which is far easier to obtain. Meanwhile, if one can manage to make such detection model for their games, they can apply the BotScreen's architecture with their detection model for target game.

Open-Source Anti-Cheats. We also survey three open-source anti-cheat solutions, specifically designed to detect aimbots in CS:GO. We note that each solution uses a predefined heuristic to detect suspicious aim changes. COW Anti-Cheat [15] checks if the difference between aiming angles before and after hitting a target is greater than 15 degrees. When there are more than five suspicious such movements, it flags the player as a cheater. Little Anti-Cheat [27] accuses a player as a cheater if its aiming angle is changed by more than 80% in 0.5 seconds before killing a player. Similarly, SMAC [55] marks a player as a cheater if its aiming direction was changed by more than 45 degrees in 0.5 seconds before killing a player. While BotScreen and open-source anti-cheats operate under the same intuition that aimbots tend to generate sudden aiming movements, open-source solutions are easier to evade and lack flexibility as they operate on a set of codified rules.

Cheating Prevention. There are many general cheating prevention techniques, although they are not directly applicable to aimbot detection. For example, Watchmen [61] detects cheating attempts by employing a network proxy, which monitors network packets from clients. Although this approach can lessen the burden of the server, it still does not scale well with popular games with substantially busy network traffic. Kalra *et al.* [28] propose a blockchain-based cheating detection technique, where every game states are shared through a smart contract. However, it also suffers from the scalability issue. Bethea *et al.* [3] leverage symbolic execution to validate client states, but it also bounds to the server-side detection problems. OpenConflict [5] designs the Oblivious Set Intersection Protocol that can prevent the use of maphacks. BlackMirror [49] utilizes Intel SGX to prevent the use of wallhacks by storing the sensitive information inside an SGX enclave. However, they assume that the server includes a game engine, and every action is computed from the server, which is not applicable in large-scale FPS games, such as CS:GO.

8 Conclusion

Cheating in online games, particularly aimbots, pose a significant threat to the game industry, and existing aimbot detection solutions suffer from either high server-side overhead or poor detection performances. This paper presented BotScreen, a novel client-side distributed aimbot detection scheme, which enables distributed aimbot detection. Our design utilizes an RNN-based model to enable high detection accuracy while only using features available to distributed clients. Our implementation and evaluation with CS:GO shows that BotScreen can detect aimbots with high precision, and its runtime overhead is marginal on a modern gaming PC.

Acknowledgement

We thank anonymous reviewers and our shepherd for their constructive feedback. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2021-0-01332, Developing Next-Generation Binary Decompiler).

References

[1] Advanced Micro Devices. AMD SEV-SNP: Strengthening vm isolation with integrity protection and more. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf, 2020.

[2] Hashem Alayed, Fotos Frangoudes, and Clifford Neuman. Behavioral-based cheating detection in online first person shooters using machine learning techniques. In *Proceedings of the IEEE Conference on Computational Inteligence in Games*, pages 1–8, 2013.

[3] Darrell Bethea, Robert A. Cochran, and Michael K. Reiter. Server-side verification of client behavior in online games. In *Proceedings of the Network and Distributed System Security Symposium*, 2010.

[4] Wieland Brendel, Jonas Rauber, and Matthias Bethge. Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. In *Proceedings of the International Conference on Learning Representations*, 2018.

[5] Elie Bursztein, Mike Hamburg, Jocelyn Lagarenne, and Dan Boneh. OpenConflict: Preventing real time map hacks in online games. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 506–520, 2011.

[6] Zikui Cai, Shantanu Rane, Alejandro E. Brito, Chengyu Song, Srikanth V. Krishnamurthy, Amit K. Roy-Chowdhury, and M. Salman Asif. Zero-query transfer attacks on context-aware object detectors. In *Proceedings of IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 15004–15014, 2022.

[7] Nicholas Carlini and David A. Wagner. Towards evaluating the robustness of neural networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 39–57, 2017.

[8] Pin-Yu Chen, Huan Zhang, Yash Sharma, Jinfeng Yi, and Cho-Jui Hsieh. ZOO: zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In *Proceedings of the ACM Workshop on Artificial Intelligence and Security*, pages 15–26, 2017.

[9] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. In *Proceedings of the Workshop on Syntax, Semantics and Structure in Statistical Translation*, pages 103–111, 2014.

[10] Valve Corporation. Steam subscriber agreement. https://store.steampowered.com/subscriber_agreement/, 2022.

[11] Valve Corporation. CS:GO - trusted mode. https://help.steampowered.com/en/faqs/view/09A0-4879-4353-EF95, 2023.

[12] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, 2016:86, 2016.

[13] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

[14] Harry Dunham. Cheat detection using machine learning within counter-strike: Global offensive. https://openworks.wooster.edu/cgi/viewcontent.cgi?article=11803&context=independentstudy, 2020.

[15] eedson. Cow anti-cheat. https://github.com/eedson/Cow-Anti-Cheat, 2018.

[16] ezcheats. Winx private. https://ezcheats.com/csgo/winx-private-csgo.html, 2022.

[17] Shufan Fei, Zheng Yan, Wenxiu Ding, and Haomeng Xie. Security vulnerabilities of SGX and countermeasures: A survey. *ACM Computing Surveys (CSUR)*, 54(6):1–36, 2021.

[18] Mayra Rosario Fuentes and Fernando Mercês. Cheats, hacks, and cyberattacks: Threats to the esports industry in 2019 and beyond. *Trend Micro Research*, 8, 2019.

[19] Luca Galli, Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. A cheating detection framework for unreal tournament iii: A machine learning approach. In *Proceedings of the IEEE Conference on Computational Inteligence in Games*, pages 266–272, 2011.

[20] Epic Games. Easyanticheat. https://www.easy.ac/, 2004.

[21] Chuan Guo, Jacob R. Gardner, Yurong You, Andrew Gordon Wilson, and Kilian Q. Weinberger. Simple black-box adversarial attacks. In *ICML*, volume 97 of *Proceedings of Machine Learning Research*, pages 2484–2493. PMLR, 2019.

[22] Mee Lan Han, Jung Kyu Park, and Huy Kang Kim. Online game bot detection in FPS game. In *Proceedings of the Asia Pacific Symposium on Intelligent and Evolutionary Systems*, pages 479–491, 2015.

[23] Hype. Project infinity. https://project-infinity.cloud/, 2022.

[24] BattlEye Innovations. Battleye. https://www.battleye.com/, 2004.

[25] Intel. Intel trust domain extensions. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html, 2022.

[26] Insider Intelligence. Esports ecosystem in 2022: Key industry companies, viewership growth trends, and market revenue stats. https://www.insiderintelligence.com/insights/esports-ecosystem-market-report/, 2022.

[27] J-Tanzanite. Little anti-cheat. https://github.com/J-Tanzanite/Little-Anti-Cheat, 2021.

[28] Sukrit Kalra, Rishabh Sanghi, and Mohan Dhawan. Blockchain-based real-time cheat prevention and robustness for multi-player online games. In *Proceedings of the 14th International Conference on Emerging Networking Experiments and Technologies*, pages 178–190, 2018.

[29] Anssi Kanervisto, Tomi Kinnunen, and Ville Hautamaki. Gan-aimbots: Using machine learning for cheating in first person shooters. *IEEE Transactions on Games*, pages 1–1, 2022.

[30] Fazle Karim, Somshubra Majumdar, and Houshang Darabi. Adversarial attacks on time series. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(10):3309–3320, 2021.

[31] Erin Kenneally and David Dittrich. The menlo report: Ethical principles guiding information and communication technology research. *Available at SSRN 2445102*, 2012.

[32] Hyungseok Kim, Junoh Lee, Soomin Kim, Seungil Jung, and Sang Kil Cha. How'd security benefit reverse engineers? the implication of intel cet on function identification. In *Proceedings of the International Conference on Dependable Systems Networks*, pages 559–566, 2022.

[33] Daniel Krupiński. Osiris. https://github.com/danielkrupinski/Osiris, 2018.

[34] Ryan Lemay. Modern warfare 2 cheat makers charged $3 million in damages to activision. https://www.dexerto.com/call-of-duty/modern-warfare-2-cheat-makers-charged-3-million-in-damages-to-activision-2068853/, 2023.

[35] Qizhang Li, Yiwen Guo, and Hao Chen. Practical no-box adversarial attacks against dnns. In *NeurIPS*, 2020.

[36] Daiping Liu, Xing Gao, Mingwei Zhang, Haining Wang, and Angelos Stavrou. Detecting passive cheats in online games via performance-skillfulness inconsistency. In *Proceedings of the International Conference on Dependable Systems Networks*, pages 615–626, 2017.

[37] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *Proceedings of the International Conference on Learning Representations*, 2019.

[38] Mingjun Ma, Dehui Du, Yuanhao Liu, Yanyun Wang, and Yiyang Li. Efficient adversarial sequence generation for RNN with symbolic weighted finite automata. In *Proceedings of the Workshop on Artificial Intelligence Safety*, volume 3087 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2022.

[39] Frank J Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78, 1951.

[40] Andy Maxwell. Bungie & destiny 2 cheat creator agree $13.5m copyright damages judgment. https://torrentfreak.com/bungie-destiny-2-cheat-creator-agree-13-5m-damages-judgment-220610/, 2022.

[41] John McDonald. Robocalypse now: Using deep learning to combat cheating in 'counter-strike: Global offensive'. https://www.gdcvault.com/play/1024994/Robocalypse-Now-Using-Deep-Learning, 2018.

[42] Xiaozhu Meng and Barton P. Miller. Binary code is not easy. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 24–35, 2016.

[43] Gautam Raj Mode and Khaza Anuarul Hoque. Adversarial examples in deep learning for multivariate time series regression. In *Proceedings of the Applied Imagery Pattern Recognition Workshop*, pages 1–10. IEEE, 2020.

[44] Jared Moore. Pubg mobile cheat makers ordered to pay $10 million in damages. https://www.ign.com/articles/pubg-mobile-cheat-makers-pay-10-million-damages, 2022.

[45] MSI. Msi aterburner. https://www.msi.com/Landing/afterburner/graphics-cards, 2021.

[46] Tracker Network. Tracker network. https://tracker.gg/, 2023.

[47] Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *Proceedings of the IEEE European Symposium on Security and Privacy*, pages 372–387, 2016.

[48] Sang-Ok Park, Ohmin Kwon, Yonggon Kim, Sang Kil Cha, and Hyunsoo Yoon. Mind control attack: Undermining deep learning with GPU memory exploitation. *Computers & Security*, 102:102115, 2021.

[49] Seonghyun Park, Adil Ahmad, and Byoungyoung Lee. BlackMirror: Preventing wallhacks in 3D online FPS games. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 987–1000, 2020.

[50] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Proceedings of the International Conference on Neural Information Processing Systems*, pages 8024–8035, 2019.

[51] Sandro Pinto and Nuno Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys*, 51(6), jan 2019.

[52] Anil Rao. Rising to the challenge - data security with intel confidential computing. https://community.intel.com/t5/Blogs/Products-and-Solutions/Security/Rising-to-the-Challenge-Data-Security-with-Intel-Confidential/post/1353141, 2022.

[53] Reuters. Optic india disqualified after forsaken found cheating. https://www.espn.com/esports/story/_/id/25026623/optic-india-disqualified-forsaken-found-cheating, 2018.

[54] Daniel Ruby. 44+ esports statistics for 2023 (trends, facts & insights). https://www.demandsage.com/esports-statistics/, 2023.

[55] Silenci0. Smac. https://github.com/Silenci0/SMAC, 2019.

[56] Mike Stubbs. 'valorant' player caught cheating in $2,000 tournament. https://www.forbes.com/sites/mikestubbs/2020/08/24/valorant-player-caught-cheating-in-2000-tournament/, 2020.

[57] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *Proceedings of the International Conference on Learning Representations*, 2014.

[58] Paul Tassi. Anger over 'destiny 2' pc cheaters has reached a tipping point. https://www.forbes.com/sites/paultassi/2020/04/21/anger-over-destiny-2-pc-cheaters-has-reached-a-tipping-point/, 2020.

[59] Byfron Technologies. hyperion. https://byfron.com/, 2021.

[60] Tim Witschel and Christian Wressnegger. Aim low, shoot high: Evading aimbot detectors by mimicking user behavior. In *Proceedings of the European Workshop on Systems Security*, pages 19–24, 2020.

[61] Amir Yahyavi, Kevin Huguenin, Julien Gascon-Samson, Jorg Kienzle, and Bettina Kemme. Watchmen: Scalable cheat-resistant support for distributed multi-player online games. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, pages 134–144, 2013.

[62] S. F. Yeung, John C. S. Lui, Jiangchuan Liu, and Jeff Yan. Detecting cheaters for multiplayer games: Theory, design and implementation. In *Proceedings of the IEEE Consumer Communications and Networking Conference*, pages 1178–1182, 2006.

[63] 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 International Conference on Neural Information Processing*, pages 654–661, 2012.

[64] Su-Yang Yu, Nils Hammerla, Jeff Yan, and Peter Andras. A statistical aimbot detection method for online FPS games. In *Proceedings of the International Joint Conference on Neural Networks*, pages 1–8, 2012.

[65] Chaoning Zhang, Philipp Benz, Adil Karjauv, and In So Kweon. Data-free universal adversarial perturbation and black-box attack. In *ICCV*, pages 7848–7857. IEEE, 2021.

[66] Mingyi Zhou, Jing Wu, Yipeng Liu, Shuaicheng Liu, and Ce Zhu. DaST: Data-free substitute training for adversarial attacks. In *Proceedings of IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 231–240, 2020.

[67] Zim. Aimware. https://aimware.net/, 2014.

A Training Loss for RNN Model

We present the training loss of the proposed SGRU model as a function of training epochs in Figure 9. The solid line represents the average loss, and the shaded areas represent confidence intervals. It can be easily seen that the training is effectively done, where the loss converges to close zero.

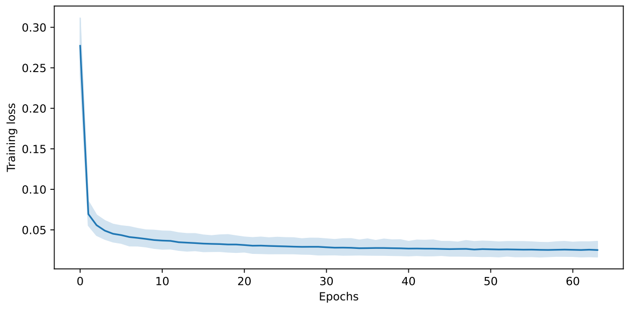

Figure 9: Training loss of SGRU models over epochs.

Table 8: Summary on the skill levels of the players participated in our study. All numbers are retrieved from Tracker Network [46], or official matchmaking records of each game.

ID	Playtime	Skill summary
A	> 6,200 hrs	Apex Legend - BR Diamond 3 (Top 2%) CS:GO - Global Elite-level (Top 1.4%) Overwatch - Grand Master (Top 1%)
B	> 5,700 hrs	Apex Legend - BR Diamond 2 (Top 1%) CS:GO - Distinguished Master Guardian (Top 12%) Destiny 2 - 1.19 K/D (Top 16%)
C	> 1,300 hrs	Apex Legend - BR Diamond 4 (Top 7%) Rainbow Six Seige - Platinum 3 (Top 32%)
D	> 4,000 hrs	Destiny 2 - 1.54 K/D (Top 8%) Overwatch - Diamond (Top 20%) PUBG - Platinum IV (Top 14%)
E	> 3,000 hrs	Destiny 2 - 1.17 K/D (Top 18%)
F	> 3,500 hrs	Sudden Attack - 1.5 K/D
G	> 920 hrs	Destiny 2 - 1.14 K/D (Top 20%) Sudden Attack - 1.4 K/D
H	> 4,000 hrs	Overwatch - Diamond (Top 20%) Rainbow Six Seige - Platinum 3 (Top 32%)
I	> 1,200 hrs	Rainbow Six Seige - Platinum 3 (Top 32%)
J	> 1,450 hrs	Destiny 2 - 1.03 K/D (Top 33%)
K	> 7,500 hrs	Destiny 2 - 1.02 K/D (Top 34%),
L	> 3,500 hrs	CS:GO - Master Guardian 1 (Top 35%)
M	> 1,350 hrs	Rainbow Six Seige - Gold 1 (Top 43%)
N	> 2,600 hrs	Destiny 2 - 0.71 K/D (Bottom 23%) Fortnite - 1.11 K/D (Top 29%)

Table 9: Configurations and aggregated statistics of each game. Duration includes the preparation time before beginning of each game.

Game Configuration				# of Events		
Day	Format (Players)	# of Cheaters (Cheaters)	Duration (s)	`fire`	`hit`	`dead`
1	4 vs 4 (A, B, D, E, I, K, L, N)	0 (∅)	616.048	3,657	850	186
		2 (I, L)	615.345	2,532	534	182
		2 (A, B)	627.684	3,308	746	159
		2 (K, N)	614.707	3,516	702	214
		2 (D, E)	624.560	2,595	600	203
		8 (all)	469.043*	2,600	499	190
		2 (A, E)	619.000	2,981	626	200
2	5 vs 5 (A, C, D, E, F, G, H, I, J, N)	0 (∅)	626.224	4,999	986	228
		2 (D, J)	623.171	4,327	983	249
		2 (C, H)	636.606	3,566	981	260
		2 (G, I)	623.332	4,473	1,100	287
		2 (F, N)	635.831	4,322	933	290
		2 (A, E)	636.056	3,719	888	278
		10 (all)	618.554	4,495	787	271
		2 (D, J)	628.690	3,841	785	231
		2 (E, G)	619.498	3,484	743	245
		2 (A, H)	627.427	2,813	1,281	258
3	4 vs 4 (A, D, E, F, G, I, M, N)	0 (∅)	629.466	2,621	1,031	189
		2 (D, I)	609.438	2,775	691	203
		2 (F, N)	624.307	2,579	522	188
		2 (A, G)	621.398	2,414	555	176
		2 (E, M)	627.289	2,524	556	178
		8 (all)	614.902	7,103	851	211
4	5 vs 5 (A, D, E, F, G, H, I, L, M, N)	0 (∅)	633.453	3,916	1,008	240
		2 (M, N)	631.405	3,718	940	296
		2 (G, I)	627.982	3,234	860	277
		2 (E, L)	617.924	3,229	822	230
		2 (A, F)	611.621	3,448	832	239

* Due to a server glitch, we were able to record a part of the game only.

B Collected Game Information

We present the detailed setups and statistics for the dataset described in §5.1.2. We hired 14 participants from a FPS game clan, and summarized their in-game ranks as well as their accumulated playtime in Table 8. The table lists all the participants in the order of their skill levels and playtime. The least skilled players, Player M and N, can be considered as an average-level player given their standings.

Table 9 shows the detailed configuration of each game we ran for four days. Since not all the players were able to participate in all the games, we have different sets of participants in each day. Each game was set to run for 10 minutes (= 600 seconds). However, the "Duration" column in the table shows slightly more than 10 minutes as our recording includes preparation session before each game starts. Every participant played either as a cheater or a non-cheater in each game and we distributed the roles uniformly among the participants.

Table 10: History-based detection accuracies for each detection technique, measured over 28 games played for four days. Numbers in parentheses are respectively the number of falsely accused players (FP), the number of undetected cheaters (FN), and the total number of players considered. A player is considered to be a cheater there has been at least one detection report.

	Reimplemented Methods				Open Source			
Player	`th_VarA`	`th_AccA`	`th_Kill`	`ks_AccA`	`os_CAC`	`os_LAC`	`os_SMAC`	**BotScreen**
A	0.071 (0/26/28)	0.071 (0/26/28)	**1.000 (0/0/28)**	0.964 (0/1/28)	0.071 (0/26/28)	**1.000 (0/0/28)**	**1.000 (0/0/28)**	**1.000 (0/0/28)**
B	0.286 (0/5/7)	0.286 (0/5/7)	**1.000 (0/0/7)**	0.714 (2/0/7)	0.286 (0/5/7)	0.571 (0/3/7)	0.571 (0/3/7)	0.571 (0/3/7)
C	0.200 (0/8/10)	0.200 (0/8/10)	0.900 (0/1/10)	0.900 (1/0/10)	0.200 (0/8/10)	**1.000 (0/0/10)**	0.900 (1/0/10)	0.600 (0/4/10)
D	0.143 (0/24/28)	0.143 (0/24/28)	**1.000 (0/0/28)**	0.964 (1/0/28)	0.321 (0/19/28)	**1.000 (0/0/28)**	0.964 (1/0/28)	**1.000 (0/0/28)**
E	0.143 (0/24/28)	0.143 (0/24/28)	**1.000 (0/0/28)**	0.857 (4/0/28)	0.143 (0/24/28)	**1.000 (0/0/28)**	0.964 (1/0/28)	**1.000 (0/0/28)**
F	0.190 (0/17/21)	0.190 (0/17/21)	0.952 (0/1/21)	0.857 (3/0/21)	0.190 (0/17/21)	**1.000 (0/0/21)**	**1.000 (0/0/21)**	**1.000 (0/0/21)**
G	0.143 (0/18/21)	0.143 (0/18/21)	0.952 (0/1/21)	0.905 (2/0/21)	0.381 (0/13/21)	0.857 (3/0/21)	**1.000 (0/0/21)**	**1.000 (0/0/21)**
H	0.133 (0/13/15)	0.133 (0/13/15)	0.933 (1/0/15)	0.933 (1/0/15)	0.400 (0/9/15)	**1.000 (0/0/15)**	**1.000 (0/0/15)**	**1.000 (0/0/15)**
I	0.036 (0/27/28)	0.036 (0/27/28)	**1.000 (0/0/28)**	0.857 (0/4/28)	0.179 (0/23/28)	0.857 (0/4/28)	0.964 (1/0/28)	**1.000 (0/0/28)**
J	0.100 (0/9/10)	0.100 (0/9/10)	0.100 (0/9/10)	0.800 (2/0/10)	0.100 (0/9/10)	0.100 (0/9/10)	0.100 (0/9/10)	**1.000 (0/0/10)**
K	0.429 (0/4/7)	0.429 (0/4/7)	**1.000 (0/0/7)**	**1.000 (0/0/7)**	0.429 (0/4/7)	0.714 (0/2/7)	0.571 (3/0/7)	**1.000 (0/0/7)**
L	0.083 (0/11/12)	0.083 (0/11/12)	0.250 (0/9/12)	**1.000 (0/0/12)**	0.500 (0/6/12)	**1.000 (0/0/12)**	**1.000 (0/0/12)**	**1.000 (0/0/12)**
M	0.364 (0/7/11)	0.364 (0/7/11)	0.727 (0/3/11)	0.636 (4/0/11)	0.364 (0/7/11)	0.636 (4/0/11)	0.636 (4/0/11)	**1.000 (0/0/11)**
N	0.107 (0/25/28)	0.107 (0/25/28)	**1.000 (0/0/28)**	0.929 (0/2/28)	0.107 (0/25/28)	0.929 (0/2/28)	0.929 (0/2/28)	**1.000 (0/0/28)**

C History-based Detection Comparison

Recall from §5.4 that we compared BotScreen against seven previous detection methods in a history-based detection setting, where a player is considered as a cheater if there has been at least one detection report in the history. From the game data we have in Table 9, we extracted games that each player participated, and simulated the detection process for each player.

Table 10 summarizes the results. Each cell in the table represents a detection accuracy of a detection method in a player basis, and numbers in parentheses represent the number of false reports (FP), the number of missed reports (FN), and the total number of reports. It is clear that BotScreen significantly outperforms all the other methods in terms of detection accuracy. There are two notable cases where BotScreen suffered (Player B and C). Those are the players who participated only for a small number of games. Even if there was a false negative case for a single game, BotScreen could not detect the player as an aimbot user for the rest of the games because there is no further evidence to accuse the player. Nonetheless, the other detection methods also suffered from the same problem for these players, and BotScreen showed consistently better performance compared to the others.

D Detailed Performance Benchmark Results

Recall from §5.6, we measured performance overhead of BotScreen on three different setups: high-performance, mid-performance, and low-performance machines (§5.1.4). For each setup, we ran MSI Afterburner for 10 minutes and repeated the experiment 10 times. The experimental results for the high-performance, mid-performance, and low-performance setup are tabulated in Table 11, Table 12 and Table 13, respectively.

Table 11: Measured frame times on high-performance setup.

			Frame Time (ms)				
		#Frames	**Avg.**	**Min**	**Max**	**Low 1%**	**Low 0.1%**
Baseline	1	153,371	3.912	0.901	82.946	8.764	12.612
	2	147,699	4.062	0.973	92.483	9.322	13.370
	3	141,295	4.246	0.786	98.557	9.512	14.143
	4	143,211	4.190	0.909	35.039	8.963	12.162
	5	146,382	4.099	0.893	113.429	8.622	13.819
	6	150,862	3.977	0.901	87.403	9.056	14.287
	7	126,971	4.725	0.882	75.936	11.017	16.820
	8	146,598	4.093	0.832	172.772	9.426	14.314
	9	151,556	3.959	0.853	35.786	9.243	12.342
	10	146,761	4.088	0.922	54.022	9.046	12.839
BotScreen	1	146,659	4.091	1.039	69.957	9.525	14.358
	2	132,267	4.536	0.988	99.646	10.047	14.701
	3	137,735	4.356	1.057	153.207	10.016	15.156
	4	149,137	4.023	1.093	103.817	9.723	15.525
	5	131,332	4.569	1.101	156.411	10.357	14.880
	6	166,246	3.609	0.904	93.441	8.657	14.233
	7	144,860	4.142	0.899	146.000	9.645	17.715
	8	136,527	4.395	1.040	35.293	10.015	14.113
	9	130,683	4.591	1.005	28.304	10.114	13.932
	10	153,548	3.908	0.923	39.580	9.055	12.634

Each table presents the performance benchmark results with and without BotScreen. Note that the frame time significantly varies from 3.9ms to 4.7ms in the high-performance setup, and the fluctuation is much higher in the other setups. Given that the performance overhead of BotScreen is much smaller than the frame time variation in the high-performance and the mid-performance setup, we can conclude that BotScreen does not affect the performance significantly for modern machines.

Table 12: Measured frame times on mid-performance setup.

			Frame Time (ms)				
		#Frames	**Avg.**	**Min**	**Max**	**Low 1%**	**Low 0.1%**
Baseline	1	85,171	7.045	1.457	189.083	13.057	22.001
	3	123,310	4.866	1.325	64.152	10.538	14.224
	4	91,682	6.544	1.366	34.691	12.502	16.400
	5	99,918	6.005	1.368	44.541	12.428	16.842
	2	127,766	4.696	1.297	39.900	10.266	13.763
	6	96,033	6.248	1.373	92.231	12.660	19.106
	7	89,820	6.680	1.407	76.699	12.310	17.497
	8	126,398	4.747	1.262	44.809	10.881	14.206
	9	117,066	5.125	1.310	71.483	11.158	15.217
	10	84,770	7.078	1.404	46.267	14.326	17.872
BotScreen	1	86,000	6.977	1.408	136.746	15.688	24.188
	2	81,591	7.354	1.392	103.724	16.688	24.821
	3	84,267	7.120	1.525	50.944	15.894	20.556
	4	85,654	7.005	1.439	95.644	15.100	22.190
	5	87,116	6.887	1.475	59.744	14.749	19.433
	6	110,912	5.410	1.380	41.637	12.982	16.620
	7	98,251	6.107	1.490	44.563	13.688	17.472
	8	84,145	7.130	1.537	47.840	15.567	20.370
	9	103,330	5.807	1.421	56.707	12.793	17.479
	10	92,572	6.481	1.481	45.653	13.601	17.974

Table 13: Measured frame times on low-performance setup.

			Frame Time (ms)				
		#Frames	**Avg.**	**Min**	**Max**	**Low 1%**	**Low 0.1%**
Baseline	1	20,960	28.626	4.717	1,167.064	119.403	415.093
	2	19,447	30.853	4.810	919.616	176.530	486.293
	3	18,828	31.868	4.866	960.538	115.277	361.149
	4	18,852	31.827	5.591	404.953	76.804	188.311
	5	22,955	26.138	4.947	84.492	47.026	62.277
	6	15,651	38.338	8.275	644.159	76.613	194.331
	7	18,666	32.144	8.500	450.727	59.374	134.551
	8	18,264	32.851	22.680	156.349	50.704	72.187
	9	15,835	37.891	8.554	551.610	70.554	152.392
	10	15,847	37.864	7.831	556.640	69.940	125.269
BotScreen	1	14,754	40.667	7.188	1,508.617	151.420	633.001
	2	21,968	27.313	18.010	339.299	51.622	91.830
	3	19,334	31.033	19.674	738.953	72.496	192.380
	4	15,091	39.759	7.127	232.859	84.852	150.191
	5	11,332	52.951	13.996	672.676	117.582	313.267
	6	12,694	47.269	13.227	1,045.163	162.825	574.082
	7	13,102	45.797	13.813	886.827	128.910	404.839
	8	11,719	51.202	14.186	878.528	116.041	280.454
	9	13,333	45.003	13.643	1,021.265	108.492	262.184
	10	12,223	49.089	12.248	444.153	112.197	234.817