

GAN-Aimbots: Using Machine Learning for Cheating in First Person Shooters

Anssi Kanervisto, Tomi Kinnunen, *Member, IEEE*, and Ville Hautamäki, *Member, IEEE*

Abstract—Playing games with cheaters is not fun, and in a multi-billion-dollar video game industry with hundreds of millions of players, game developers aim to improve the security and, consequently, the user experience of their games by preventing cheating. Both traditional software-based methods and statistical systems have been successful in protecting against cheating, but recent advances in the automatic generation of content, such as images or speech, threaten the video game industry; they could be used to generate artificial gameplay indistinguishable from that of legitimate human players. To better understand this threat, we begin by reviewing the current state of multiplayer video game cheating, and then proceed to build a proof-of-concept method, *GAN-Aimbot*. By gathering data from various players in a first-person shooter game we show that the method improves players’ performance while remaining hidden from automatic and manual protection mechanisms. By sharing this work we hope to raise awareness on this issue and encourage further research into protecting the gaming communities.

Index Terms—video game, deep learning, cheating, security, generative adversarial networks

I. INTRODUCTION

VIDEO games attract millions of players, and the industry reports their revenue in billions of dollars. For instance, one of the biggest video game publishers, Activision Blizzard, reported more than 75 million players of their game *Call of Duty: Modern Warfare (2019)* and net revenue of over 3.7 billion US dollars in the first half of 2020 [1]. The gaming communities also contain e-sport tournaments with prize pools in millions, e.g. World Electronic Sports Games 2016 contained a prize pool of 1.5 million USD. With such popularity, cheating practices similar to doping in physical sports is commonplace in video games. For example, two public cheating communities have existed since 2000 and 2001, and have more than seven million members in total [2], [3], and this is only a fraction of such communities. In these communities, users can share and develop different ways to cheat in multiplayer games. Although cheating is more difficult in in-person tournaments, cheating in online games is still prevalent. For example, UnknownCheats [3] has 213 threads and more than a million views for the game *Call of Duty: Modern Warfare (2019)*. The presence of cheaters degrades the user experience of other players, as playing with cheaters is not fun. Game developers are therefore encouraged to prevent such cheating.

All authors are with School of Computing, University of Eastern Finland, Joensuu, Finland. V. Hautamäki is also with the Department of Electrical and Computer Engineering, National University of Singapore. E-mail: anssk@uef.fi, tkinnu@uef.fi, villeh@uef.fi

Manuscript received December 7th, 2021; Revised March 13th, 2022; Accepted May 3th, 2022.

A common way to cheat in online games is by using tools and software (“hacks”) used by “hackers” [4], dating back to 1990 with the first forms of hacking with Game Genie [5]. Hacking is prohibited by game publishers, and they monitor players with anti-cheat software to detect hacking. A standard approach is to check running processes on the player’s computer for known cheating software—similar to how antivirus scanners look for specific strings of binary on a machine. The publisher can also analyse players’ behaviour and performance to detect suspicious activity, such as a sudden increase in a player’s performance. While hacks can be hidden from the former detection approach, the latter cannot be bypassed, as the system runs on game servers. Analysis of player behaviour is thus an attractive option for publishers.

These data-based anti-cheats have also attracted the attention of the academic community, as well as from the game industry. Galli et al. (2011) [6] developed hacks that provide additional information to the player and move the mouse in the game *Unreal Tournament III*. The authors then used machine learning to distinguish these hackers from legitimate players. Hashen et al. (2013) [7] extended this work by evaluating the accuracy of different classifiers versus different hacks. Yeung and Lui (2008) [8] analysed players’ performance and then used Bayesian inference to analyse players’ accuracy and classify if a player was hacking.

All of these works have focused on **detecting** hackers with machine learning, but little attention had been given to **cheating with machine learning**. Yan and Randell [4] have mentioned the use of machine learning for cheating, but only in the context of board games like Chess and Go and not in video games. Recent advances in machine learning allow one to generate photorealistic imagery [9], speech to attack voice biometric systems [10] or computer agents that mimic the demonstrators [11], [12]. Similar methods could be applied to video games to generate human-like gameplay. We define “human-like gameplay” as artificially generated gameplay that is indistinguishable from genuine human gameplay, either by other humans or by automated detection systems. If used for cheating, these methods threaten the integrity of multiplayer games, as the previously mentioned anti-cheat systems could not distinguish these cheaters.

To develop an understanding of this threat, we design a machine learning method for controlling the computer mouse to augment the player’s gameplay and study how effective it is for cheating. This work and its contributions can be summarised as follows:

- We review different ways people hack in multiplayer video

games and methods for detecting such cheaters.

- We present a proof-of-concept machine learning method, GAN-Aimbot, for training a hack that mimics human behaviour using Generative Adversarial Networks (GANs) [13] and human gameplay data.
- We implement GAN-Aimbot along with heuristic hacks as baseline solutions. We then collect gameplay data from multiple human participants and compare how much these hacks improve players' performance.
- We implement an automatic cheat detection system using neural networks and use it to study how detectable the different hacks are.
- We record video clips of human participants using these hacks and then ask another set of participants to judge if the player is cheating, based on the recording.
- We discuss the ethical and broader impact of sharing this work.

By sharing this work and associated code and data publicly, we aim to encourage further research on maintaining trust in online gaming.

II. CHEATING AND ANTI-CHEATING: CLASSIC APPROACHES

We start with an overview of current hack and anti-cheat methods. We focus on multiplayer first-person shooter (FPS) games. While difficult for human players to master, these games are an easy target for hackers, as quick reflexes and accurate movement are more easily implemented on a computer program than grand, strategic planning.

A. Principles of video game hacks

Much like attacking any system with the intent of modifying it, hacking involves knowing how a game works and then using this knowledge to achieve what the hacker wants. We draw inspiration from biometric security [14], [15] and present different attack vectors in Figure 1. An attack vector is a point in information flow that the hacker can access to implement their hack. One must reverse engineer the game logic before a game can be attacked. For example, if the hacker can find the function that reduces their player's health when they take damage, they can modify this function to not subtract health, which renders them invulnerable. Similar modifications can be made to game files or network packets. Reverse engineering is arguably the hardest part of creating hacks and the most prominent topic in the game hacking communities like UnknownCheats [3].

While the above attack vectors mainly affect personal computers (PCs), for which the hacker has access to the whole computer system, closed systems like game consoles (e.g., Sony's PlayStation and Microsoft's Xbox) can also be modified to gain full access to the system ("jailbreaking"). Game streaming services, like Google Stadia, Nvidia GeForce Now and Sony PlayStation Now, further restrict this access by streaming the screen image and audio over a network, rendering system access impossible. However, two attack points remain; the user output (display and audio) and the user input devices (mouse and keyboard). While these are not trivial to attack, we believe these elements will be vulnerable to attacks in the future, which we discuss in Section III. For now, we focus on hacking done on PCs.

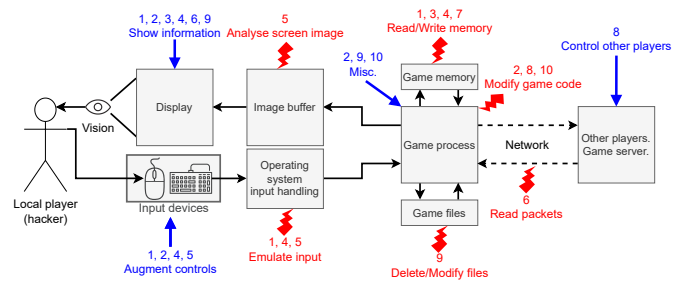


Fig. 1. Different attack vectors used by hacks which are listed in Table I (see the list under Table I). Red, lightning symbols indicate how hacks attack the information flow (how hacks work), while blue arrows show what hacks add to a hacker's gameplay (what hacks do).

B. Hacking in the wild

Table I includes a set of examples of publicly available hacks for popular PC games. They are often categorised by their "features" (advantages they provide to the player) and how they are implemented. Internal hacks modify the program code via injection of the hacker's code. This technique remains a classic and active form of hacking to this day but is easily detectable due to its intrusive nature. External hacks do not modify the game code. Rather, they rely on reading and writing game memory, inspecting network traffic or modifying game files to achieve desired effects. Computer vision (CV) hacks neither read nor write any game-related information. Instead, they capture the image shown to the player and extract necessary information via CV techniques. These include reading the colour of a specific pixel, shape detection or object detection [16].

The most common features are "wall hacks" and "radar hacks", which show the location of objects of interest, such as enemies. We group these types of hacks using a commonly used term, extrasensory perception (ESP). ESP encompasses hacks that provide information not normally available to players. Another common feature, albeit one specific to FPS games, is "aimbots". These help players aim at enemies, an act that normally requires accurate reflexes and is a core part of the competitive nature of shooter games (e.g., Quake, Unreal Tournament, Call of Duty, Battlefield). ESP and aimbots are the most common features across all FPS games, as they provide a strong advantage and can be implemented via most of the attacks listed above. For example, to implement an ESP feature, the hacker only needs information about the locations of the players and which of the players is the hacker's player. With this information alone, the hacker can draw a 2D map that indicates the locations of other players relative to theirs. Aimbots are similarly easy to implement, and only require the current aiming direction of the player. The hacker can then calculate how the aim should be adjusted towards a nearby enemy.

While common, ESP and aimbots are not the only possible features. By exploiting game-specific mechanics, other hacks can remove game objects by removing files (e.g., "DayZ Wallhack"), update player location more frequently than the game does to allow flight (e.g., "DayZ Ghost") or shoot through walls (e.g., "BF4 Wallhack"). These are only limited examples

TABLE I
EXAMPLES OF HACKS “IN-THE-WILD”, MOST OF WHICH ARE PUBLICLY AND FREELY SHARED ON UNKNOWNCHEATS FORUMS. THESE EXAMPLES WERE PICKED TO DEMONSTRATE DIFFERENT ATTACK METHODS AND FEATURES, PRIORITIZING FOR POPULAR GAMES.

Game	Name	Attack vectors	Features	Attack methods
Call Of Duty: Modern Warfare 2	External ESP ¹	External, memory (read)	ESP, aimbot	Read memory for information. Draw on new window on top of game. Control mouse via OS function. Heuristic mouse movement.
	InvLabs M2 ²	Internal (inject)	ESP, aimbot, kill everyone	Read and modify game logic
ARMA 2 (DayZ mod)	DayZ Navigator ³	External, memory (read)	ESP	Read memory for information. Show enemy locations on browser-based map
	DayZ “Wallhack” ⁹	External, modify files	ESP, shoot through walls	Remove objects from game by removing files.
	DayZ Control Players ⁸	Internal (inject), exploit	Misc.	Take control of characters of other players using game’s functions.
DayZ	DayZ Ghost ⁷	External, memory (read/write)	Fly-hack	Write desired location of player into memory faster than game would modify it
Counter Strike: Global Offensive	Charlatano ⁴	External, memory (read)	ESP, aimbot	Read memory for information. Control mouse via OS function. Advanced heuristics for human-like behaviour (short movement, Bézier curves).
	Neural Network Aimbot ⁵	Displayed image	Aimbot	Use CV to detect enemies on the screen. Control mouse via OS function. Heuristic mouse movement.
PLAYERUNKNOWN’S BATTLEGROUNDS	PUBG-Radar ⁶	Network (read)	ESP	Read network packets for information. Draw info on a separate window.
Battlefield 4	BF4 Wallhack ¹⁰	Unknown, exploit	ESP, shoot through walls	Possibly exploiting the fact server does not check hits against walls.

¹ <https://www.unknowncheats.me/forum/call-duty-6-modern-warfare-2/64376-external-boxesp-v5-1-bigradar-1-2-208-a.html>

² https://www.unknowncheats.me/forum/call-of-duty-6-modern-warfare-2-a/91102-newtechnology_publicbot-v4-0-invllabs-m2.html

³ <https://www.unknowncheats.me/forum/arma-2-a/80397-dayz-navigator-v3-7-a.html>

⁴ <https://github.com/Jire/Charlatano>

⁵ <https://www.unknowncheats.me/forum/cs-go-releases/285847-hayha-neural-network-aimbot.html>

⁶ <https://www.unknowncheats.me/forum/pubg-releases/262741-radar-pubg.html>

⁷ <https://www.unknowncheats.me/forum/dayz-sa/228162-ghost-mode-free-camera-tutorial-source-video-compatible-online-offline.html>

⁸ <https://www.unknowncheats.me/forum/arma-2-a/114967-taking-control-players.html>

⁹ <https://www.unknowncheats.me/forum/arma-2-a/77733-dayz-wallhack-sorta.html>

¹⁰ <https://www.youtube.com/watch?v=yF7CyqbSS48>

of different hacks people have devised, but they generally are easily noticeable by other players or can be prevented by developers.

An open-source hack more similar to our core topic of human-like behaviour is “Charlatano”, which advertises itself as a “stream-proof” hack. This hack would not be detected by people observing the gameplay of the hacker and is achieved via a human-like aimbot, which adjusts aim in natural and small increments to avoid suspicion. These features are based on heuristics and on visual observations of what looks human-like to humans. Another hack, the “Neural Network Aimbot”, uses the deep neural network “YOLO” [16] to detect targets visible on the screen and then directs aim towards them by emulating mouse movement. The screen image is captured in a manner that mimics game-recording software, and the mouse is moved with operating-system functions, rendering this hack virtually invisible to game software. If combined with an external video-capture device and an independent micro-controller emulating a mouse, this hack would be independent of the machine that runs the game. The behaviour of the player, and specifically

the effect of aimbot on how a player moves their mouse, would be the only possible source for detecting this hack.

C. Countermeasures against hacking

To prevent hacking and cheating, passive methods include designing game flow such that hacking is not possible or obfuscating the compiled game binary such that reverse engineering becomes more difficult. An example of the former option is the game “World of Tanks”, in which the player client does not receive any information about enemy locations until they are within line of sight, rendering ESP hacks difficult to implement. The latter option can be a by-product of a digital rights management (DRM) system or an anti-tamper system like that of Denuvo [22], which encrypts the executable code and prevents the use of debugging tools for analysing game flow.

If the creation or use of hacks was difficult or required expensive hardware to execute, hacking would be limited, much like how long passwords are considered secure because randomly guessing them is virtually impossible. Unfortunately,

TABLE II

LIST OF COMMON ANTI-CHEAT SOFTWARE FOUND IN GAMES, ALONG WITH THE DESCRIPTION OF METHODS THEY USE FOR DETECTING HACKERS.

Name	Type	Features
BattleEye [17]	Traditional	Signature checks, screenshots
Punkbuster [18]	Traditional	Signature and driver checks, screenshots
FairFight [19]	Statistical	Various. Track player statistics, track if player aims through walls.
EasyAntiCheat [20]	Traditional + Statistical	Signature checks, low-level access to memory, track player statistics.
Valve Anti-Cheat [21]	Traditional + Statistical	Signature checking, deep learning-based aimbot detector based on mouse movement (VACNet)

as game logic and code are the same for all players a game, only one person needs to create and share a hack before everyone else can use it, much like the unauthorised sharing of game copies. As such, passive methods alone do not reliably prevent hacks unless they are guaranteed to prevent hacking. Active prevention methods are thus necessary.

Active prevention methods aim to detect players using hacks and then remove them from the game, temporarily or permanently (by banning the player). A simple method is to use human spectators who look for suspicious players, but this option does not scale with high numbers of players. Human spectators are also susceptible to false alarms; genuinely proficient players may be flagged as hackers.¹ CS:GO Overwatch [23] crowd-sources this by allowing the game community to judge recordings of other players.

A more common approach is to use anti-cheat software to identify these hackers. Table II lists common anti-cheats found in modern games. A traditional approach is to search for specific strings of bytes in computer memory obtained by analysing publicly available hacks. The finding of such a signature indicates that a known hack is running on the machine, and the player is flagged for hacking. This method alone is effective against publicly shared hacks but is not helpful against private hacks.

As with any attack-defend situation, with hackers being the attackers and anti-cheats the defenders, there is a constant battle between building better defences and avoiding these defences with better attacks. Signature checks can be avoided by obfuscating the binary code of hacks. In turn, anti-cheats can search for suspicious memory accesses to the game process. Hacks can hide this evidence by using lower levels in the security rings (“kernel-level”), which cannot be scanned by anti-cheats running on the higher levels. Naturally, anti-cheats can adapt by installing themselves on this lower level. However, as PCs are ultimately owned by their users, a hacker could modify their system to fool anti-cheats. Luckily for defenders, studying a user’s machine is not the only way to detect hackers.

D. Data-based approach to detecting hackers

Anti-cheats like FairFight [19] and Valve Anti-Cheat (VAC) [21] detect hackers by their performance and behaviour, in addition to traditional techniques. Hackers can be detected using heuristics (e.g., too many kills in a short time is suspicious) or with statistical models derived from data (e.g.,

¹For an example of such players, search for gameplay footage of professional FPS players. Here is an example of a professional player in Battlefield 4: <https://www.youtube.com/watch?v=6-Kt2duNKtY>.

VACNet [21] and related work, [6], [8]). As these types of anti-cheats are executed on the game servers and use the same data the player client must send to play the game in multiplayer games, they cannot be bypassed by hackers like traditional anti-cheats can be. Unfortunately, these methods do have their weakness: errors in classification [24], [25].

Identifying hackers is a binary-classification task: is the given player a hacker or *bona fide*?² With two possible cases and two possible decisions, there are four outcomes, two of which are desired (correct decisions) and two which are errors. In a false positive (FP) case, a *bona fide* player is predicted as a hacker, and in a false negative (FN) case a hacking player is predicted as a *bona fide* player. Since the two classes are unlikely to be perfectly distinguishable (i.e., mistakes will inevitably occur), system designers must balance between user convenience and security. A game developer likely wants to develop a system that exerts caution when flagging people hackers, as FPs harm *bona fide* players, which in turn harm the game’s reputation. However, if an anti-cheat only analyses player performance, then a hacker can adjust their hacks to mimic top-performing, *bona fide* players. These hackers are still better than most players and they avoid detection by an anti-cheat system that has been tuned not to flag top-performing players. This approach clearly has limitations, which brings us to the analysis of player behaviour.

For example, FairFight gathers information on how a player moves and aims at targets through walls [26]. If a player often aims at enemies behind obstacles and walls, this behaviour signals that a hack is being used, and the player can be flagged. VACNet employs a similar approach: a deep learning model has been trained to classify between *bona fide* and hacking players according to their mouse movement, using a dataset of known hackers and *bona fide* players [21]. This kind of anti-cheat does not rely on high performance to detect a hacker, as it attacks the very core of hacking: altering the player’s behaviour to improve performance. A hack cannot improve a player’s performance if it does not alter their behaviour.

III. DATA-BASED APPROACHES TO HACKING

What if the behaviour of a hacker is indistinguishable from a *bona fide* human player? Consider a situation in which a hacker can perfectly clone the gameplay behaviour of a professional player using advanced methods. If the professional player is not flagged for cheating, then this hacker would not be flagged either, but clearly, the latter party is cheating.

²Genuine, legitimate, sincere. In our context “non-cheating”.

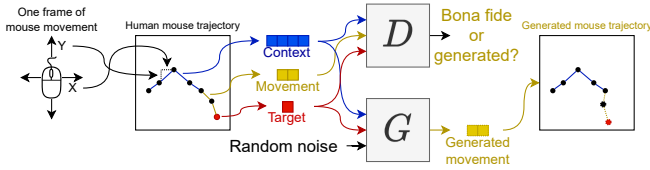


Fig. 2. An illustrative figure of the discriminator D and generator G networks and their inputs/outputs, with a context size of four and two movement steps. Note that “target” is an absolute location with respect to where the trajectory begins, while other values are one-step changes in location. This setup corresponds to training (Algorithm 1), where “target” is taken from human data.

For this paper, the scope of this cloning is limited to mouse movement. If this mimicry is combined with hardware that emulates mouse control and captures a screen image, and a neural network aimbot described earlier, this hack could not be detected by any of the anti-cheats methods described above.

While this example is extreme, recent deep learning advances enabled the generation of various realistic content that is indistinguishable from natural content, like images of faces [9] and speech [27]. Given these results, cloning human behaviour in a video game does not seem unrealistic. To assess the severity of this risk, we design a proof-of-concept method in the context of aimbots, dubbed GAN-Aimbot, and assess both its effectiveness as an aimbot and its undetectability. We limit the scope to aimbots to focus the scope of the experiments, and also because an undetectable aimbot would disrupt FPS games. To the best of our knowledge, we are the first to openly share such a method. Although our method could be used for malicious purposes, we believe it is better to openly share this information to spur discussion and future work to prevent this type of cheating. We further discuss the positive and negative impact of sharing this work in Section XI.

IV. GAN-AIMBOT: HUMAN-LIKE AIMBOT WITH GENERATIVE ADVERSARIAL NETWORKS

The task of an aimbot is to move the aim-point (where the weapon points at, illustrated by the green cross-hair in Figure 4) to the desired target (e.g., an enemy player) by providing mouse movement, which would normally be provided by the human player. In each game frame, the aimbot moves the mouse Δx units horizontally (to change the yaw of the aim direction) and Δy units vertically (to change pitch).

A. Generative adversarial networks for aimbots

GAN-Aimbot, as the name suggests, is built on GANs [13], which consists of two components. A generator G tries to create content that looks as realistic as possible (i.e. similar to the training data), while a discriminator D tries to distinguish between data from the training set and data from the generator. In GAN-Aimbot, the generator generates human-like mouse movement and a discriminator tries to distinguish this movement from the recorded, *bona fide* human data. After training, we can use the generator as an aimbot in the game. If the generator is able to fool the discriminator at the end of the training, we assume the same applies to other anti-cheat

Algorithm 1 Pseudo-code for training the GAN-Aimbot.

Input: Dataset containing human mouse movement $\mathcal{D}$, generator G and its parameters θ_G , discriminator D and its parameters θ_D , number of discriminator updates N_D , Adam optimizer parameters Adam_D and Adam_G and weight clip parameter $w_{\max}$.

while training do

for $n \in \{1 \dots N_D\}$ **do**

 //Sample human data

$x_D, y_D \sim \mathcal{D}$

 //Generate steps

$z \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$

$g = G(z|y_D)$

 //Update discriminator

$\mathcal{L}_D = D(x_D|y_D) - D(g|y_D)$

$\theta_D = \theta_D - \text{Adam}_D(\nabla_{\theta_D} \mathcal{L}_D)$

 //Clip weights

$\theta_D = \text{clip}(\theta_D, -w_{\max}, w_{\max})$

end for

 //Sample conditions

$y_G \sim \mathcal{D}$

 //Generate steps

$z \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$

$g = G(z|y_G)$

 //Create generator loss + target point loss (4)

$\mathcal{L}_G = D(g|y_g) + \text{dist}(g, t)$

 //Update generator

$\theta_G = \theta_G - \text{Adam}_G(\nabla_{\theta_G} \mathcal{L}_G)$

end while

systems, similar to how computer-generated imagery can fool human observers by appearing natural [28].

Training the hack to generate human-like mouse movement is only one part of this task. The hack should also redirect aim towards potential targets. We use a conditional GAN [29], where the desired target is given as a condition to both parts. We also provide mouse movement from previous frames in the condition (context), which allows the generator to adjust generation according to previous steps and the discriminator to spot any sudden changes.

During training, we use the aim-point of the *bona fide* data a few steps after the context as a target. The generator is then tasked with generating these steps (movement) such that they appear *bona fide* and such that the final aim-point is close to the target. We use several steps instead of just one to provide more generated data for the discriminator to classify, and we also allow the generator to generate different steps to reach the goal. See Figure 2 for an illustration.

This method is similar to imitation learning methods using adversarial training setup [11], [12], where the discriminator attempts to distinguish between the computer agent being trained and the demonstration data. The discriminator is then used to improve the agent. However, we have notable differences in our proposed setup:

- 1) We train the system much like the original GANs [13] on a fixed dataset (we do not need the game as a part of the training process) and use the generator directly as

a decision-making component.

- 2) The generator is used in conjunction with a human player and not as an autonomous agent.
- 3) We augment the discriminator training loss with an additional task (move aim point closer to the target).
- 4) We explicitly study both the behavioural performance (akin to imitation learning performance) and distinguishability (akin to GAN literature) from demonstration data of the generator, not just one.

The “Neural Network Aimbot” hack described in Section II-B also uses neural networks, but only to detect targets using image analysis; mouse movement is still rule-based. As such, GAN-Aimbot and Neural Network Aimbot are complementary approaches. In this work, we can obtain the location of targets directly from the game environment, and as such, we do not include Neural Network Aimbot as part of our experiments.

B. Training GAN-Aimbot

Let context size be c frames and the number of generated steps g leading to each data point sampled from the *bona fide* dataset be length $d := c + g$ frames. Each sample is a trajectory of mouse movements $(\Delta x_1, \dots, \Delta x_d)$ and $(\Delta y_1, \dots, \Delta y_d)$, where Δ is the amount of mouse movement in one frame. The last g frames are used to create the target $\mathbf{t} := (\sum_{i=c}^d \Delta x_i, \sum_{i=c}^d \Delta y_i)$, which represents where the aim-point aims g frames after the context. The condition then contains the context and target $\mathbf{i} := (\Delta x_1, \dots, \Delta x_c, \Delta y_1, \dots, \Delta y_c, t_1, t_2)$. The discriminator accepts this context as input as well as g steps $\mathbf{o} := (\Delta x_{c+1}, \Delta x_{c+2}, \dots, \Delta x_d, \Delta y_{c+1}, \Delta y_{c+2}, \dots, \Delta y_d)$, which represent mouse movement g frames after the context. These steps $\mathbf{o}$ can be from the *bona fide* dataset or generated by the generator. Finally, the generator accepts as input a Gaussian-noise vector $\mathbf{z} := \mathcal{N}(\mathbf{0}, \mathbf{I})$, $\mathbf{z} \in \mathbb{R}^k$ and condition $\mathbf{i}$ to generate steps $\mathbf{o}$. We use $k = 16$ which we found to be sufficient to generate good results. For a concrete example, the blue lines in Figure 2 correspond to context with $c = 4$, the yellow lines to generated steps with $g = 2$ and the red point is the target.

The training procedure is presented in Algorithm 1. The procedure uses methods from Wasserstein GAN [30], which stabilise training over standard GAN. Contrary to the GAN literature, the discriminator output is maximized for the generated content. This is to stay consistent with the classification labels, where positive labels mean a positive match for a hacker. For a set of randomly sampled *bona fide* conditionals $\mathbf{i}_G, \mathbf{i}_D$ (which include targets $\mathbf{t}_G, \mathbf{t}_D$), corresponding *bona fide* mouse movement $\mathbf{o}_D$ and random Gaussian vectors $\mathbf{z}_G, \mathbf{z}_D$, we compute two loss values

$$\mathcal{L}_G := D(G(\mathbf{z}_G|\mathbf{i}_G)) + \text{dist}(G(\mathbf{z}_G|\mathbf{i}_G), \mathbf{t}_G) \quad (1)$$

$$\mathcal{L}_D := D(\mathbf{o}_D|\mathbf{i}_D) - D(G(\mathbf{z}_D|\mathbf{i}_D)|\mathbf{i}_D), \quad (2)$$

where $\text{dist}(\cdot, \cdot)$ calculates the Euclidean distance between the

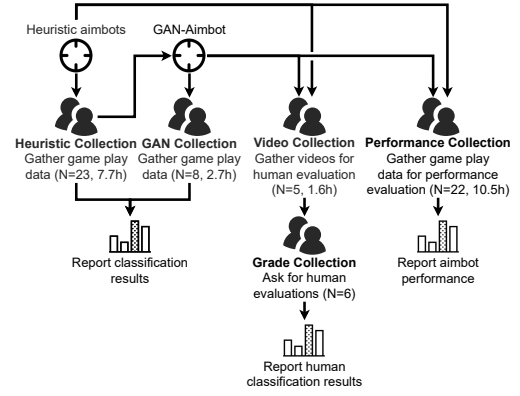


Fig. 3. An overview of the experiments and data collection. Data collection phases indicate the number of participants and the total amount of game time recorded.

final aim-point and the target

$$\mathbf{g} := G(\mathbf{z}_G|\mathbf{i}_G) = (\Delta x_1, \dots, \Delta x_g, \Delta y_1, \dots, \Delta y_g) \quad (3)$$

$$\text{dist}(\mathbf{g}, \mathbf{t}) := \sqrt{\left(\left[\sum_{i=1}^g g_i\right] - t_0\right)^2 + \left(\left[\sum_{i=g}^{2g} g_i\right] - t_1\right)^2} \quad (4)$$

Afterwards, networks are updated to minimise their corresponding losses using stochastic gradient descent and Adam optimisation [31]. Discriminators’ weights are clipped to preserve the smoothness of the output [30].

After training, we use the generator to create the proposed aimbot. We present the generator with a target point (the closest enemy to the aim-point), context and a fixed random vector; generate the next $\mathbf{o}$ mouse movements for g frames; and use the output for the first frame to move the player’s aim-point. This allows the aimbot to correct mouse movement in each game frame. We fix the random vector in one game to keep the generated behaviour consistent throughout the game’s duration.

V. OVERALL EXPERIMENTAL SETUP

For assessment we use two research questions (RQs):

- RQ1** Does the GAN-Aimbot improve players’ performance?
RQ2 Does the GAN-Aimbot remain undetectable by human observers and automatic anti-cheat systems?

We answer these questions via an empirical study where we create a game environment; collect gameplay of human participants playing with and without aimbots; and assess how distinguishable gameplay with aimbots is from *bona fide* human play, according to both an automated anti-cheat system and human observers. We compare the results to those of heuristic aimbots which use techniques outlined in Table I. The experiment process is outlined in Figure 3. The environment and aimbots are described in this section. The following sections describe individual experiments, including their setup and results.

The experiment code and collected data are available at <https://github.com/Miffyli/gan-aimbots>.



Fig. 4. ViZDoom environment used for the experiments. The player can also aim vertically in this implementation of Doom. Red and green boxes represent field-of-views of the aimbots, red for *strong* and GAN-Aimbot and green for *light* aimbot. Aimbots activate automatically when a target (white boxes) is inside the field-of-view. The white, green and red boxes are not visible to the player.

A. Game environment

We use Doom (1993) via ViZDoom library [32] as a game, as depicted in Figure 4. Doom is one of the first FPS games. While dated, Doom still includes core mechanics of FPS games, namely navigating the map to find enemies and killing them by aiming and firing weapons. We chose this game and library as ViZDoom allows granular control over game flow while remaining lightweight, which allows us to distribute the data collection programs to participants over the Internet.

The human players play games of “deathmatch” on a single map, where players have to score the highest number of kills to win. Everyone plays against each other. A single human player plays with six computer-controlled players of varying strength; some exhibit higher-than-normal mobility and speed, making them difficult targets to hit. Players can find weapons stronger than the starting weapon (a pistol), but most weapons require multiple hits before they kill a target, highlighting the need for accurate aiming. So-called “instant-hit” weapons are of the most interest to our work—the projectiles of these weapons hit the target location immediately. These weapons reward the accurate aiming of the player and benefit the most from the use of aimbots.

To collect data from participants, we provided them with an executable binary file that provided information about the study, asked for consent, provided instructions and finally launched the game with the desired settings. The software recorded the participants’ actions, mouse movement per frame (in degrees), damage done, weapons used, current ammunition, number of kills and deaths, and the bounding boxes of the enemies on the screen in each frame of the game (35 frames per second). Participants were recruited amongst lab members, authors’ colleagues and first-person shooter communities to include data from players with varying skill levels.

B. Heuristic aimbots

To implement the aimbots we use the ViZDoom library to read the necessary information, such as the location of the enemy players on the screen. In a real-world scenario, this

TABLE III
SETTINGS OF THE DIFFERENT AIMBOTS

Aimbot	Activation range	Movement per axis
Light	5 degrees	$0.4 \times \text{distance to the target}$
Strong	15 degrees	$0.6 \times \text{distance to the target}$
GAN	15 degrees	Neural network decides

information would be obtained by reading the game memory or by using an object detection network (see Section II-B). When the aimbot is activated, the code selects the closest enemy to the centre of the screen and provides the centre location of the enemy sprite (the white boxes in Figure 4) to the aimbot as a target. The aimbot then moves the aim-point closer to this target. The aimbot does not fire the weapon.

The heuristic aimbot has a limited field of view and uses “slow aim”. The aimbot activates once a target is close enough to the player and to the crosshair (green and red boxes in Figure 4). With the slow aim, the aimbot will not aim directly at the target in one frame; rather, it will move the mouse a portion of the distance to the target in each frame. Each movement by the aimbot includes a small amount of Gaussian noise to mimic the inaccuracy of human aiming, drawn from $\mathcal{N}(0, 0.2 \cdot d)$, where d is the amount mouse is moved on the corresponding axis (in degrees). The multiplier 0.2 was chosen with manual testing, where the value was increased until the aimbot’s performance deteriorated.

We use two aimbot variants in this setup: *strong* and *light* aimbots, where the stronger variant has a larger field of view and faster movement, and the lighter one more closely resembles existing hacks due to its smaller adjustments, which helps to avoid detection. The GAN-Aimbot uses the same field of view setting as the strong aimbot. These differences are detailed in Table III.

C. GAN-Aimbot setup

To create a discriminator in the GAN-Aimbot system, we use a neural network with two fully connected layers of 512 units for the discriminator and two fully connected layers of 64 units for the generator. The inputs and outputs of networks are described above in Section IV and Figure 2. Both use exponential-linear unit (ELU) activation functions [33]. We use a small network for the generator to speed up computation when the network is used as an aimbot. We found that ELU stabilised the training over rectified-linear units (ReLUs) and sigmoid function. We use hyper-parameters from [30], setting the RMSprop learning rate to $5 \cdot 10^{-5}$, clipping weights to 0.01 and updating the generator only every five training iterations. We use $c = 20$ steps for context size and $g = 5$ for the number of steps generated by the generator.

We train the system for 100 epochs using a dataset of 20 minutes of human gameplay, which was enough for losses to converge with a batch size of 64. As the data used to train this system may change the aimbot’s behaviour, we include two different systems in our experiments (Group 1 and Group 2), which use the same setup as above but with data from different players (See Section VI-C). The goal is not to mimic

an individual player but to mimic the dynamics of how humans control the mouse.

VI. COMPARING AUTOMATIC CLASSIFICATION SYSTEMS (ANTI-CHEATS)

We begin the experiments with two data collections and a comparison of different classifiers for automatic cheat detection. The goal is to find the strongest system, which we will then use for assessing aimbots' detection rates later.

A. Deep neural network classifier system

To automatically detect hacking players, we follow the VACNet setup [21], which uses a deep neural network (DNN) to detect players from mouse movement alone. A single feature vector is a tuple that contains mouse movement on both axes 0.5s before and 0.25s after a player has fired a weapon as well information on whether the shot hit the enemy. The final feature vector per shot is a tuple $(\Delta x_1, \dots, \Delta x_n, \Delta y_1, \dots, \Delta y_n, \text{is hit})$ with $n = 25$. We only include feature vectors when the player is holding an instant-hit weapon; otherwise, the "is hit" label does not coincide with mouse movement. All features apart from the "is hit" truth value are normalised by subtracting the mean and dividing by the standard deviation per dimension, computed over training data. Compared to other proposed machine-learning-based anti-cheat systems [6]–[8], these features focus on detecting aimbots but do not require game-specific modifications, and have been shown to be successful in practical applications [21].

The classifier network consists of two fully connected layers of size 512, each followed by a ReLU activation. The final layer outputs logits for both classes. We use an L2-regularization weight of 0.01 and train for 50 epochs with batches of 64 samples. These hyperparameters were obtained by manual trial-and-error to reach the same level of loss in training data and validation data (10% of samples from the training data). Network parameters are updated to minimise cross-entropy between true labels and predictions with gradient descent using an Adam optimiser [31] with a learning rate of 0.001. To avoid biased training, we weight losses by an inverse probability $1 - p$ of the class being represented in a training batch. We also experimented with using ELU units as in the GAN-Aimbot, but this reduced the performance.

B. Baseline classifier systems

Instead of neural networks, previous work has explored using different methods, including support vector machines (SVM) [7] and Bayesian networks [8]. Motivated by this, we compare the DNN method to other binary classifiers using the auto-sklearn toolkit [34] with the scikit-learn library [35]. This toolkit automatically tunes the hyperparameters of algorithms based on a withheld dataset, randomly sampled from the training data. We run experiments with a naive Bayes classifier, decision trees, random forests, linear classifiers, support vector machines (SVM) and linear discriminant analysis (LDA). Each algorithm uses one-third of the training data for validation as recommended by the library documentation, and is then tuned

TABLE IV
BALANCED ACCURACIES (%) OF DIFFERENT METHODS OF DETECTING HACKERS.

Algorithm	Train	Test
Naive Bayes	69.34	69.06
Decision Tree	80.47	72.09
LDA	62.91	60.23
SVM	89.40	84.34
Random Forest	94.54	77.33
Linear Classifier	62.47	60.73
Neural Network	87.41	86.41

for five hours on four 4.4Ghz CPU cores, which we found to be sufficient for generating stable results given the relatively small size of our datasets.

C. Data collection

The first data collection (Heuristic Collection) setup asked participants to play four five-minute games, where in the first two games no aimbot was enabled, the third game had a *light* aimbot enabled and the fourth game had a *strong* aimbot enabled. Two *bona fide* games were used to balance the amount of *bona fide* and hacking data during classification. We collected data from 23 players; 18 for the training set of the classifier and 5 for the testing set. In total there were 8748 negative (*bona fide*) data points and 10732 positive (cheating) points for training, and 2973 negative and 3977 positive points for testing.

We then took four players from the testing set and split them into two groups to train the Group 1 and 2 GAN-Aimbots. We used four players to train both GANs on a similar amount of data. After training, we repeated the above data collection process (GAN Collection), where instead of using *light* and *strong* aimbots, players played with Group 1 and Group 2 GAN-Aimbots. We recruited the participants using the same channels but this time only included data from new participants to avoid mixing data of the same player in both training and testing sets. This resulted in fewer participants than in the first data collection by design, as otherwise there would have not been enough data to train the two GAN-Aimbots after train/test split. We collected data from 8 players; 4 for the training set and 4 for the testing set. The training set had 2007 negative and 2266 positive points, and the testing set had 2074 negative and 2126 positive points. Note that this data is not used in this experiment, but it was recorded immediately after the first collection and will be used in Section VIII.

D. Results for classifier comparison

We compare classifiers with balanced accuracy, which is defined as the mean of individual accuracies $\frac{1}{2}(\text{acc}_{\text{pos}} + \text{acc}_{\text{neg}})$, with acc_{pos} being the accuracy of positive samples and acc_{neg} accuracy for negative samples.

Table IV presents the balanced accuracies in the train and test sets of the Heuristic Collection. SVM, random forest and neural network shared similarly high performance in the testing set, which concurs with previous results [7], [21]. We use the term "high" to describe the performance as this accuracy of

TABLE V
AVERAGE ($\pm$ STANDARD DEVIATION) PLAYER ACCURACY AND KILLS. ASTERISKS INDICATE A SIGNIFICANT DIFFERENCE TO “NONE” ACCORDING TO TWO-TAILED WELCH’S T-TEST ($N = 22$, $*p < 0.05$, $**p < 0.01$, $***p < 0.001$).

Aimbot	Accuracy	Kills
None	31.1 $\pm$ 13.8	12.1 $\pm$ 5.1
Light	42.3 $\pm$ 8.2**	15.7 $\pm$ 6.0*
Strong	56.1 $\pm$ 8.7***	22.0 $\pm$ 7.5***
GAN	42.7 $\pm$ 10.3**	15.5 $\pm$ 5.6*

86% is based on a small segment of mouse movement. Based on these results, we used the DNN system for the following experiments.

VII. EVALUATING AIMBOT PERFORMANCE

Before assessing detection rates of aimbots, we study how much the GAN-Aimbot improves players’ performance.

A. Aimbot performance metrics

To measure the advantage aimbots give to the players, we track the accuracy and number of kills. Accuracy is the ratio of shots that hit an enemy. To measure accuracy, we only include shots fired with instant-hit weapons, as aimbots directly target the enemies and thus only help with instant-hit weapons.

B. Data collection for evaluating aimbot performance

In a real-world scenario, a cheater has to learn to play with their hack, as it may interfere with their behaviour. In our case, the aimbot takes control of the mouse abruptly when a target is close. For this reason, we created a more elaborate setup for aimbot performance evaluation (Performance Collection).

Players first played 10 minutes without an aimbot, then for 10 minutes with a *light* aimbot and finally played four games of 2.5 minutes with the four aimbots (none, *light*, *strong* and GAN Group 1), in random order. The first two games are to ensure the player has sufficient time to learn game mechanics with and without an aimbot. We randomised the order of these last four games to reduce any bias from further learning during the final 10 minutes. We only report the results of these final four games. Both of the trained GAN-Aimbots behaved in a similar way, hence we only included one of them.

C. Results for aimbot performance evaluation

Table V presents the average player performance metrics with and without different aimbots. All aimbots increased both metrics by a statistically significant amount. Since aimbots force players to aim at the enemy, slow-projectile weapons like rocket launchers and plasma guns become impractical to use (the player can only aim directly at the enemies, not in front of them), which explains the smaller increase in the number of kills, as these weapons are amongst the strongest. However, increased accuracy with the GAN-Aimbot indicates that our hack indeed works as an aimbot, making it a viable hack to use for cheating.

VIII. DETECTION RATES OF THE AIMBOTS

In this section, we perform a detailed study of how easily the different aimbots can be detected. We use the data collected in the Heuristic Collection and the GAN Collection (Section VI-C).

A. Detection rate metrics

To evaluate the DNNs performance in distinguishing different aimbots from humans, we use detection error tradeoff (DET) [36] curves, which indicate the trade-off between false-positive (FPR) and false-negative rates (FNR) when the threshold is changed; a positive decision indicates that a player is flagged as a hacker.

We also include equal error rates (EERs) and a detection cost function (DCF) [37]. The EER is the error rate at the threshold where FPR equals FNR, which summarizes the DET curve in a single scalar value. The DCF incorporates two new parameters: the costs C_{FP} and C_{FN} for both error cases as well as a prior $p_{\text{hacker}} \in (0, 1)$, which reflects the assumed probability of encountering a hacking player. The DCF is then defined as [37]

$$\text{DCF} = p_{\text{hacker}} C_{FN} P_{FN} + (1 - p_{\text{hacker}}) C_{FP} P_{FP}, \quad (5)$$

where P_{FN} and P_{FP} are FNR and FPR at a fixed threshold. The meaning of a positive/negative outcome is opposite to that of previous DCF literature, where a positive outcome is often a “good” thing (e.g., indicates a correct match, not a malicious actor). Our interpretation of the outcome is consistent with the previous cheat-detection literature, where a positive match indicates a cheating player [6], [8].

For a single-scalar comparison, we report $\text{DCF}_{\min}$, which corresponds to the minimal DCF obtained over all possible thresholds. This version of $\text{DCF}_{\min}$ does not have an interpretable meaning, other than “lower is better”, but dividing this value by $\min\{p_{\text{hacker}} C_{FN}, (1 - p_{\text{hacker}}) C_{FP}\}$ gives it an upper bound of one. If $\text{DCF}_{\min} = 1.0$, the classifier is no better than always labelling samples as a hacker or *bona fide*, depending on which has a smaller cost. All reported DCF values are normalised, minimum DCFs.

We set costs to unit costs $C_{FP} = C_{FN} = 1$ but vary the prior of hacking player and report the results at different levels $p_{\text{hacker}} \in \{0.5, 0.25, 0.1, 0.01\}$. We selected these values to cover a purely hypothetical situation in which half of the players are hackers but also more realistic scenarios where 1% of players are hackers.³ We use unit-costs as we argue both error situations are equally bad: an undetected hacker can ruin the game of several players for a moment, but banning an innocent player can ruin the game for that individual until their appeal to lift the ban has been processed.

If these systems were applied to the real world, their performance would be lower than indicated by the $\text{DCF}_{\min}$. This is because this metric provides an optimal threshold, but in practice, finding the correct threshold may be difficult and the threshold likely changes over time [25].

³To provide a rough scale, the UnknownCheats forum reported 3.4M registered users in 2020 (user status is required to download hacks), while the Steam platform reported an average of 120M active monthly users in 2020 [38].

B. Detection rate evaluation scenarios

In a real-world scenario, game developers may or may not be aware of the hacks players are using. To reflect this, we design four different evaluation scenarios. Recall that there are two separate GAN-Aimbot systems trained using data from different players (Groups 1 and 2, Section VI-C). The training and test sets are split player-wise, as further discussed in Section VI-C.

Worst-case scenario: The hack is not known to the defender to any degree. In our case, this means that the GAN-Aimbot is a new type of aimbot and the anti-cheat is unaware of it. We train the classifiers only on data from heuristic aimbots and then evaluate using data from the two GAN-Aimbots. There is no worst-case scenario for *light* and *strong* aimbots, as these aimbots are commonly known at the time of writing.

Known-attack: The defender is familiar with the mechanisms of the aimbots and is trained to detect them. However, the defender does not know the precise parameters of the aimbot—only its general technique. This situation corresponds to the hack being public knowledge, but each hacker sets up their hack with different settings. In this scenario, we train a single classifier per aimbot type (four classifiers) and then evaluate the classifier using a similar but novel aimbot: the classifier trained using the Group 1 GAN is evaluated using Group 2 GAN data and vice versa, and the classifier trained using *strong* aimbot data is evaluated using *light* aimbot data and vice versa.

Oracle: The defender knows the exact GAN-Aimbot used by the hackers, including the network parameters. This corresponds to a hack being shared publicly on a hacking community; the defender can gather data on the hack and include it in the training set. To reflect this scenario, we train four classifiers as above but evaluate the classifiers using a test set of the same aimbot data.

Train-on-test: A scenario designed to test if the aimbot data is separable from *bona fide* human data by including testing data in training. This scenario maximises accuracy but does not reflect a practical scenario. We train two classifiers, one for heuristic aimbots and one for GAN-Aimbots, and evaluate them using corresponding aimbots to ensure that there is enough data for training. The testing set is the same as in the previous scenarios. Note that this is a separate scenario from overfitting: we aim to determine how separable *bona fide* data is from the aimbots' data. We observe that the training and validation losses stay roughly equal during training, which indicates that the network did not overfit.

C. Results for detection rates of aimbots

The DET curves in Figure 5 highlight the major differences between the heuristic and the GAN aimbot in detectability: even when trained on the testing data, the FPR began to increase instantly when FNR decreased (left-most side in the figures). This result indicates that a large portion of samples from heuristic aimbots are separable from human data, while GAN-Aimbot samples are confused with *bona fide* data. In more realistic scenarios, the error rates increased notably (note the normal-deviate scale of axis).

Table VI presents the EER and DCF_{min} values of these same scenarios. Although a GAN-Aimbot improved players'

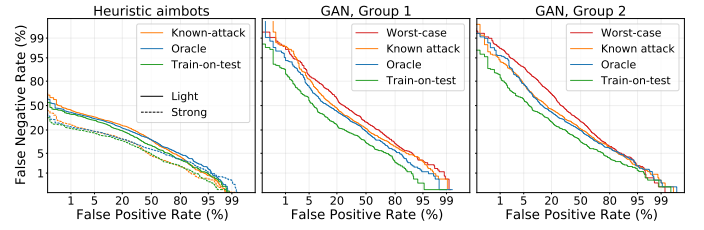


Fig. 5. DET curves of detecting different aimbots under different scenarios (Section VIII-B) using DNN classifier (Section VI-A). Curves towards the lower-left corner are better for security (easier to detect), while the worst possible case is a diagonal, descending line in the middle.

TABLE VI
EVALUATION RESULTS OF DETECTING AIMBOTS, WITH LOWER VALUES MEANING BETTER FOR THE ANTI-CHEAT. BOLDDED DCF_{min} VALUES ARE ABOVE 0.95, MEANING THE CLASSIFIER CAUSED ALMOST AS MUCH HARM AS NOT USING IT. EERS ARE IN PERCENTAGES (%).

Aimbot	Scenario	EER	$DCF_{min}, P_{hacker} = \dots$			
			0.5	0.25	0.1	0.01
Light	Worst-case	-	-	-	-	-
	Known-attack	21.06	.3713	.4099	.4494	.5972
	Oracle	21.91	.3846	.4344	.4857	.6761
	Train-on-test	19.02	.3365	.3957	.4549	.5386
Strong	Worst-case	-	-	-	-	-
	Known-attack	14.65	.2300	.2543	.2974	.4635
	Oracle	13.27	.2226	.2507	.2801	.3985
	Train-on-test	12.55	.2052	.2376	.2793	.3379
GAN1	Worst-case	41.92	.8380	.9990	1.0000	1.0000
	Known-attack	30.03	.5779	.8016	.9236	1.0000
	Oracle	28.27	.5388	.7499	.9393	.9869
	Train-on-test	19.40	.3795	.6114	.8379	.9785
GAN2	Worst-case	38.31	.7651	.9978	.9989	.9991
	Known-attack	33.21	.6200	.7901	.9168	.9735
	Oracle	25.90	.4955	.6839	.8199	.9672
	Train-on-test	19.55	.3805	.5788	.7578	.9276

accuracy by an amount similar to a *light* aimbot, the equal-error rates are notably higher than with the *light* aimbot (i.e., GAN-Aimbot is harder to detect). When we consider the ratio of hackers to *bona fide* players with DCF_{min} , we observe that the neural network classifier can be worse than not using it at all, even when testing data were contained in the training data. This is because the number of *bona fide* players being flagged as hackers negated the benefit of occasionally detecting a hacker.

D. Impact of player skill on detection rate

To study if the player's skill in the game affects the detection rate of the aimbot, we computed the correlation between players' *bona fide* performance (kills and accuracy) and the average detection scores, using data from the Performance Collection in the oracle scenario. We found virtually no correlation when GAN-Aimbot was used, neither with kills nor accuracy (-0.087 and -0.037 , respectively). Note that, because of how the collection was performed, data of the same player may have been used to train the classifier and in this evaluation, thus skewing the results. As such, these results are preliminary.

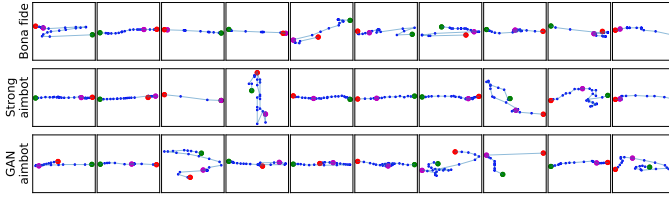


Fig. 6. Randomly selected examples of mouse trajectories. Green marks the beginning of the trajectory, purple is the point where the player fires the gun and red is where the trajectory ends. Axis ranges are not shared between figures but are equal per figure (aspect ratio is one).

TABLE VII

STATISTICS OF THE MOUSE MOVEMENT WITH DIFFERENT AIMBOTS: AVERAGE MOUSE MOVEMENT PER AXIS, PEARSON CORRELATION BETWEEN MOVEMENT PER AXIS AND CORRELATION BETWEEN SUCCESSIVE STEPS PER AXIS. THE FIRST THREE ROWS ARE COMPUTED ON ABSOLUTE UNITS.

Measure	None	Heuristic	GAN
Avg. yaw	1.47 ± 3.07	1.70 ± 3.12	1.58 ± 3.08
Avg. pitch	0.13 ± 0.27	0.25 ± 0.58	0.19 ± 0.29
Axis corr.	0.392	0.255	0.342
Step corr. (yaw)	0.742	0.749	0.728
Step corr. (pitch)	0.658	0.042	0.648

E. Analysis of mouse movement

Since the classifier was trained to distinguish *bona fide* players from hacking players, one could assume that an aimbot that evades detection behaves like a *bona fide* player. This is not necessarily true, however, as the machine learning aimbot could have simply found a weakness in the classification setup akin to adversarial attacks [39].

Figure 6 outlines how the GAN-Aimbot moves the mouse during gameplay; the network has learned to move in small steps. If one views videos of the gameplay, GAN-Aimbot also occasionally “overshoots” the target point and intentionally misses the target, unlike heuristic aimbots that stick to the target.

Looking at the average, absolute per-step mouse movement in Table VII (first and second rows), we see that the heuristic aimbots had the most movement on the vertical axis (pitch), followed by GAN-Aimbots and finally by *bona fide* players. This result is expected, as moving the PC mouse vertically is less convenient than moving it horizontally (give it a try). Heuristic aimbots may not take this into account, but the GAN-Aimbot learned this restriction from human data. The same can be seen in the correlation of the two variables (third row, significantly different from zero with $p < 0.001$), where no aimbot and GAN aimbots move the mouse in both axes at the same time more than the heuristic aimbot does.

Finally, the last two rows show the correlation between the successive mouse movements, measured as $\text{Corr}(\Delta y_{t+1}, \Delta y_t)$, averaged over all data points. Heuristic aimbots also have a near-zero correlation on the vertical axis. If the target moves closer or further away from the player, the target point moves vertically on the screen, and heuristic aimbots adjust accordingly. GAN-Aimbot has learned to avoid this.

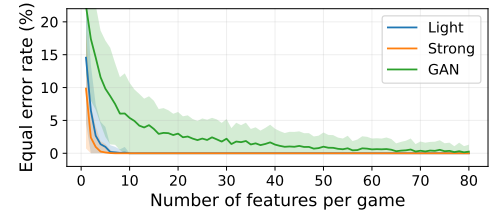


Fig. 7. EERs of classifying whole games, using a varying number of randomly sampled feature vectors (x-axis) and averaging the score over all these vectors. The line is the average EER over 200 repetitions, and the shaded region is plus/minus one standard deviation.

F. Classifying whole games with multiple feature vectors

While classifying individual feature vectors simplifies the experiments, practical applications may aggregate data over multiple events. To study how this affects classification results, we use the classifiers trained in the Oracle scenario (Section VIII-B) to score individual feature vectors and then average the scores per game. We then compute the EER (*bona fide* vs. cheating) of classifying whole games based on these averaged scores. We repeat this for a different number of feature vectors, randomly sampling the vectors per game, and repeat the process 200 times to stabilize the noise from sampling.

Figure 7 show the results. *Light* and *strong* aimbots are easily detected with less than ten feature vectors (roughly 7 seconds of game time). GAN-Aimbot becomes increasingly detectable with more data, approaching zero EER but not reaching it reliably. This indicates that the correct aggregation of detection scores over individual events will likely work as a good detection mechanism, assuming the GAN-Aimbot cheat is available to the anti-cheat developers.

IX. EVALUATING HUMAN DETECTION RATES

In addition to automatic detection systems, we are interested in whether humans are able to distinguish users of aimbot from *bona fide* players by watching video replays of the players or not.

A. Data collection for video recordings

To gather video recordings of gameplay (Video Collection), we organised a data collection procedure similar to the Heuristic Collection and GAN Collection with different aimbots (none, *light*, *strong* and GAN Group 1), in a randomised order to avoid constant bias from learning the game mechanics. We asked five participants from the previous data collection process (GAN Collection) to participate as they were already familiar with the setup, and had these participants record videos of their gameplay.

We then obtained three non-overlapping clips 30s in length at random from each recording, where each clip contained at least one kill. In total, we collected 60 video clips, 15 per aimbot including the no aimbot condition.

B. Setup for collecting human grading

We let human judges grade the recordings (Grade Collection) with one of the following grades and instructions:

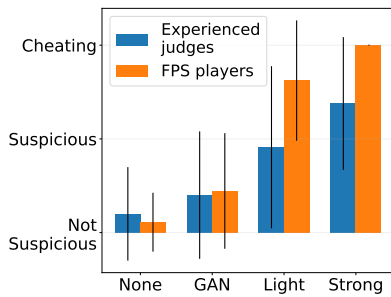


Fig. 8. Average grading (y-axis) given by human graders for different aimbots (x-axis), with a black line representing plus-minus one standard deviation, computed over 90 samples. For *strong* aimbot, all FPS player judges voted cheating, hence the standard deviation is zero.

TABLE VIII

RATIO OF GRADES PER AIMBOT PER GRADER GROUP, IN PERCENTAGES (%).

Experienced judges	None	GAN	Light	Strong
Not suspicious	84.4	71.1	42.2	13.3
Suspicious	11.1	17.8	24.4	35.6
Cheating	4.4	11.1	33.3	51.1
FPS players				
Not suspicious	88.9	62.2	8.9	0.0
Suspicious	11.1	31.1	20.0	0.0
Cheating	0.0	6.7	71.1	100.0

- 1) Not suspicious. I would not call this player a cheater.
- 2) Suspicious. I would ask for another opinion and/or monitor this player for a longer period of time to determine if they were truly cheating.
- 3) Definitely cheating. I would flag this player for cheating and use the given video clip as evidence.

The above grading system was inspired by situations in the real world. In the most obvious cases, a human judge can flag the player for cheating (Grade 3). However, if the player seems suspicious (e.g., fast and accurate mouse movements but not otherwise suspicious), they can ask for a second opinion or study other clips from the player (Grade 2).

We include both experienced judges who have experience in studying gameplay footage in a similar setup and FPS players with more than a hundred hours of FPS gameplay performance. We ensure that all participants are familiar with the concept of aimbots and how to detect them. We provide judges with a directory of files and a text file to fill and submit back, without time constraints. We obtained the grades from three experienced judges and three FPS players.

C. Results of human evaluation

The results of the human evaluation are presented in Figure 8 and Table VIII. Much like with automatic cheat detection, the GAN-Aimbot was less likely to be classified as suspicious or definitely cheating, while the *strong* aimbot was always correctly detected by FPS players with some hesitation from the experienced judges. The latter group commented that cheating had to be exceptionally evident from the video clip to mark the player a cheater, as these judges were aware of *bona fide* human players who may appear inhumanly skilled.

X. DISCUSSION

The results provide direct answers for our RQs: the data-based aimbot does improve players' performance (RQ1) while remaining undetectable (RQ2). This indicates that the risk of data-based aimbots to the integrity of multiplayer video games exists, especially considering that the evaluated method only required less than an hour's worth of gameplay data to train.

Beyond the field of gaming, the same human-like mouse control could be used to attack automated Turing tests, or CAPTCHAs [40], which study a user's inputs to determine whether a user is a bot. Previous work has already demonstrated effective attacks against image-based CAPTCHAs [41] and systems that analyse the mouse movement of the user [42]. The latter work simulated the CAPTCHA task and trained a computer agent to behave in a human-like way, whereas our system only required human data to be trained.

However, our results have some positive outcomes. As the trained aimbot was harder to distinguish from human players, it is reasonable to assume its behaviour was closer to human players. Such a feature is desired in so-called aim-assist features of video games, where the game itself helps player's aim by gently guiding them towards potential targets [43]. Our method could be used for such purposes. As it does not set restrictions on types of input information, it could also be a basis for a more generic game-assist tool that game developers could include in their games to make learning the game easier. Alternatively, this methodology could be used to create human-like opponents to play against, similar to the 2K BotPrize competition [44], where participants were tasked to create computer players that were indistinguishable from human players in *Unreal Tournament*. Such artificial opponents could be more fun to play against as they could exhibit human flaws and limitations.

A. Possible future directions to detect GAN-Aimbots

Let us assume that hacks similar to GAN-Aimbot become popular and hacking communities share and improve them openly. How could these hacks be prevented by game developers? Results of Section VIII indicate that GAN-Aimbots evade detection, so we need new approaches to detect hackers. The following are broad suggestions that apply to GAN-Aimbots but could be also applied to detect other forms of cheating.

An expensive but ideal approach is to restrict hackers' access to the game's input and output in the first place (e.g., game consoles that prevent the user from controlling the game character by means other than the gamepad). An enthusiastic hacker could still build a device to mimic that gamepad and send input signals from program code—or even construct a robot to manipulate the original gamepad—but we argue that such practices are not currently a major threat, as many players would not want to exert so much effort just to cheat in a video game.

Another approach would be to model players' behaviour over time. Sudden changes in player behaviour could be taken as a sign of potential hacking and the player could be subjected to more careful analysis. The FairFight anti-cheat advertises similar features [19].

Extending the idea above, anti-cheat players could create “fingerprints” of different hacks, similar to byte-code signatures already used to detect specific pieces of software running on a machine. These fingerprints would model the behaviour of the hack, not that of the players. However, this process would require access to hacks to build such a fingerprint.

Finally, anti-cheats could extend beyond just modelling the gameplay behaviour of the player in terms of device inputs and instead include behaviour in social aspects of the game. Recent work has indicated that the social behaviour of hackers (e.g., chatting with other players) differs notably from that of *bona fide* players [45], and combined with other factors, this information could be used to develop an anti-cheat that could reliably detect hackers with and provide an explanation of why this person player was flagged [46].

B. Limitations of this work

We note that all experiments were run using one type of classifier and a single set of features, both of which could be further engineered for better classification accuracy. However, the GAN-Aimbot could also be further engineered, and the discriminator element could be updated to match the classifier. Theoretically, the adversarial-style training should scale up to any type of discriminator, until the generator learns to generate indistinguishable mouse movement from *bona fide* players’ data. We also used a fixed target inside an enemy player sprite, but a human player is unlikely to always aim at one very specific spot. This target point could also be determined via machine learning to appear more human-like.

The experiments were conducted using a single game environment and focused solely on mouse movement. While the results might apply to other FPS games, they may not necessarily generalise well to other games and other input mechanisms. Players of a real-time strategy game (e.g. Starcraft II) could theoretically design a system to automatically control the units, for example.

XI. BROADER IMPACT

We detail and present a method that, according to our results, achieves what was deemed negative (a cheat that works and is undetectable). A malicious actor could use the knowledge of this work to implement a similar hack in a commercial game and use it. For this reason, we have to assess the broader impact of publishing this work [47], for which use the framework of Hagendorff (2021) [48]. This decision framework, based on similar decision frameworks in biological and chemical research, is designed to address the question of whether the technology could be used in a harmful way by defining multiple characteristics of the harm. We answer each of these with a “low” or “high” grading, where “high” indicates a high chance of harm or a high amount of harm.

a) Magnitude of harm.: Low. The focus is on the individuals. This work could lead to players disliking games as new hackers emerge, to erosion of trust in the video game communities and to monetary costs to companies. However, no directed harm is done to individuals (e.g., no bodily harm, no oppression), and the nuisance is limited to their time in

video games. Cheating is already prevalent (Section II-B), and while this knowledge could make it more widespread, it would not be a sudden, rapid change in the video game scene.

b) Imminence of harm.: High. With enough resources and IT skills, a malicious actor can use the knowledge of our study for harm.

c) Ease of access.: Low chance of harm. The method does not require specialized hardware but requires setting up a custom data collection. There are also no guarantees that the described method would work as is in other games.

d) Skills required to use this knowledge: Low chance of harm. Using this knowledge for malicious purposes requires knowledge in machine learning and programming. The included source code only works in the research environment used here and can not be directly applied to other games. With all the likelihood, this will also require training and potentially designing new neural network architectures with associated practical training recipes.

e) Public awareness of the harm.: High (hidden) impact of the harm. Given the relative simplicity of this method and the number of active hacking communities, it is possible that a private group of hackers have already used a similar method to conceal their cheating. This would make this work more important, as now the defenders (anti-cheat developers, researchers) are also aware of the issue and can work towards more secure systems.

While in the short term sharing this work may support cheating communities (unless it has already been used secretly), we believe it will be a net-positive gain as researchers and game developer companies improve their anti-cheat systems to combat this threat. If new threats are gone unnoticed, the defence systems may be ill-designed to combat them; our system can detect heuristic aimbots reliably with less than ten feature vectors (seven seconds of game time), but GAN-Aimbot requires more than 70 (one minute), as illustrated by results in Section VIII-F. As security is an inherently adversarial game whereby novel attacks are being counteracted by new defences (and vice versa), we hope our study supports the development of new defences by providing information on emerging attacks.

In the short-term future, video game developers could employ stricter traditional methods (Section II-C) for sensitive games, such as requiring a phone number for identification. For data-based solutions, results in Section VIII-F indicate that aggregating results over multiple events can still work as a detection measure, though detection of GAN-Aimbots was found to require more data for reliable detection. For more sustainable solutions, however, options described in Section X should be explored.

XII. CONCLUSION

In this work, we covered the current state of cheating in multiplayer video games and common countermeasures used by game developers to prevent cheating. Some of these countermeasures use machine learning to study player behaviour to spot cheaters, but we argue that cheaters could do the same and use machine learning to devise cheats that behave like humans and thus avoid detection. To understand this threat, we designed

a proof-of-concept method of such an approach and confirmed that the proposed method is both a functional cheat and also evades detection by both an automatic system and human judges. We share these results, as well as the experimental code and data (at <https://github.com/Miffyli/gan-aimbots>), to support and motivate further work in this field to keep video games trustworthy and fun to play for everyone.

REFERENCES

- [1] "Activision blizzard's second quarterly results 2020," <https://investor.activision.com/static-files/926e411c-7d05-49f8-a855-6dc7646e84c4>, accessed February 2022.
- [2] "Multiplayer game hacking," <https://mpgh.net>, accessed February 2022.
- [3] "Unknowncheats," <https://unknowncheats.me>, accessed February 2022.
- [4] J. Yan and B. Randell, "A systematic classification of cheating in online games," in *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*. ACM, 2005, pp. 1–9.
- [5] M. Nielsen, "Game genie - the video game enhancer," <http://www.nesworld.com/gamegenie.php>, accessed February 2022, 2000.
- [6] L. Galli, D. Loiacono, L. Cardamone, and P. L. Lanzi, "A cheating detection framework for unreal tournament iii: A machine learning approach," in *Conference on Computational Intelligence and Games*. IEEE, 2011, pp. 266–272.
- [7] H. Alayed, F. Frangoudes, and C. Neuman, "Behavioral-based cheating detection in online first person shooters using machine learning techniques," in *Conference on Computational Intelligence and Games*. IEEE, 2013, pp. 1–8.
- [8] S. F. Yeung and J. C. Lui, "Dynamic bayesian approach for detecting cheats in multi-player online games," *Multimedia Systems*, vol. 14, no. 4, pp. 221–236, 2008.
- [9] T. Karras, S. Laine, and T. Aila, "A style-based generator architecture for generative adversarial networks, 2019 ieee," in *Computer Vision and Pattern Recognition*, 2018, pp. 4396–4405.
- [10] A. Sizov, E. Khoury, T. Kinnunen, Z. Wu, and S. Marcel, "Joint speaker verification and antispoofing in the *i*-vector space," *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 4, pp. 821–832, 2015.
- [11] J. Ho and S. Ermon, "Generative adversarial imitation learning," in *Advances in neural information processing systems*, 2016, pp. 4565–4573.
- [12] J. Fu, K. Luo, and S. Levine, "Learning robust rewards with adversarial inverse reinforcement learning," in *International Conference on Learning Representations*, 2018.
- [13] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, pp. 2672–2680.
- [14] N. K. Ratha, J. H. Connell, and R. M. Bolle, "Enhancing security and privacy in biometrics-based authentication systems," *IBM systems Journal*, vol. 40, no. 3, pp. 614–634, 2001.
- [15] C. Roberts, "Biometric attack vectors and defences," *Computers & Security*, vol. 26, no. 1, pp. 14–25, 2007.
- [16] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Computer Vision and Pattern Recognition*, 2016, pp. 779–788.
- [17] B. Innovations, "Battleye," <https://battleye.com/>, accessed February 2022.
- [18] Even-Balance, "Punkbuster," <https://evenbalance.com>, accessed February 2022.
- [19] GameBlocks, "FairFight," <https://www.i3d.net/products/hosting/anti-cheat-software>, accessed February 2022.
- [20] Kamu, "Easy Anti-Cheat," <https://easy.ac>, accessed February 2022.
- [21] J. McDonald, "Using deep learning to combat cheating in CSGO," GDC 2018, <https://www.youtube.com/watch?v=ObhK8IUfllc>, 2018.
- [22] Irdeto, "Denuvo," <https://irdeto.com/denuvo/>, accessed February 2022.
- [23] "Counter strike: Global offensive. overwatch faq," <https://blog.counter-strike.net/index.php/overwatch/>, accessed February 2022.
- [24] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern classification*. John Wiley & Sons, 2012.
- [25] N. Brummer, "Measuring, refining and calibrating speaker and language information extracted from speech," Ph.D. dissertation, Stellenbosch: University of Stellenbosch, 2010.
- [26] "Unknowncheats—avoid fairfight bans," <https://unknowncheats.me/forum/infestation-survivor-stories/101840-iss-avoid-fairfight-bans-more.html>, accessed February 2022.
- [27] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," in *9th ISCA Speech Synthesis Workshop*, 2016, pp. 125–125.
- [28] A. Rossler, D. Cozzolino, L. Verdoliva, C. Riess, J. Thies, and M. Nießner, "Faceforensics++: Learning to detect manipulated facial images," in *Computer Vision and Pattern Recognition*, 2019, pp. 1–11.
- [29] M. Mirza and S. Osindero, "Conditional generative adversarial nets," *arXiv preprint arXiv:1411.1784*, 2014.
- [30] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein generative adversarial networks," in *International Conference on Machine Learning*, 2017, pp. 214–223.
- [31] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference on Learning Representations*, 2015.
- [32] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaśkowski, "Vizdoom: A doom-based ai research platform for visual reinforcement learning," in *Conference on Computational Intelligence and Games*, 2016, pp. 1–8.
- [33] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units," in *International Conference on Learning Representations*, 2016.
- [34] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *Advances in Neural Information Processing Systems* 28, 2015, pp. 2962–2970.
- [35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
- [36] A. Martin, G. Doddington, T. Kamm, M. Ordowski, and M. Przybocki, "The DET curve in assessment of detection task performance," National Inst of Standards and Technology Gaithersburg MD, Tech. Rep., 1997.
- [37] G. R. Doddington, M. A. Przybocki, A. F. Martin, and D. A. Reynolds, "The nist speaker recognition evaluation—overview, methodology, systems, results, perspective," *Speech communication*, vol. 31, no. 2-3, pp. 225–254, 2000.
- [38] "Steam—2020 year in review," <https://steamcommunity.com/groups/steamworks/announcements/detail/2961646623386540827>, accessed February 2022.
- [39] I. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *International Conference on Learning Representations*, 2015.
- [40] L. Von Ahn, M. Blum, and J. Langford, "Telling humans and computers apart automatically," *Communications of the ACM*, vol. 47, no. 2, pp. 56–60, 2004.
- [41] Y. Zi, H. Gao, Z. Cheng, and Y. Liu, "An end-to-end attack on text captchas," *Transactions on Information Forensics and Security*, vol. 15, pp. 753–766, 2019.
- [42] I. Akrou, A. Feriani, and M. Akrou, "Hacking Google reCAPTCHA v3 using reinforcement learning," in *Conference on Reinforcement Learning and Decision Making*, 2019.
- [43] R. Vicencio-Moreira, R. L. Mandryk, C. Gutwin, and S. Bateman, "The effectiveness (or lack thereof) of aim-assist techniques in first-person shooter games," in *Conference on Human Factors in Computing Systems*, 2014, pp. 937–946.
- [44] P. Hingston, "The 2k botprize," in *IEEE Symposium on Computational Intelligence and Games*, 2009.
- [45] M. L. Bernardi, M. Cimitile, F. Martinelli, F. Mercaldo, J. Cardoso, L. Maciaszek, M. van Sinderen, and E. Cabello, "Game bot detection in online role player game through behavioural features," in *ICSOFT*, 2017, pp. 50–60.
- [46] T. Jianrong, X. Yu, Z. Shiwei, X. Yuhong, L. Jianshi, W. Runze, and F. Changjie, "XAI-driven explainable multi-view game cheating detection," in *Conference on Games*, 2020.
- [47] M. Cook, "The social responsibility of game ai," in *Conference on Games*, 2021.
- [48] T. Hagendorff, "Forbidden knowledge in machine learning reflections on the limits of research and publication," *AI & SOCIETY*, vol. 36, no. 3, pp. 767–781, 2021.