UNIVERSITY
OF OULU

Cheat Prevention & Detection in online games

University of Oulu
Degree Programme in Information Processing Science
Bachelor’s Thesis
Riku Mäkynen
7.5.2018

Abstract

The purpose of this thesis is to describe ways in which bots (also known as robots) work in video games and other ways in which these bots can be differentiated from normal human players. Research question: Is it possible to deal with bot users in such a way that the game developers didn't have to stray away from their core skill sets? The most important finding of this thesis was the fact that by analysing user inputs with a neural network one could differentiate between a bot and a human with a good degree of certainty.

Keywords
Bot detection, MMORPG, online games, cheating, Robot, Aimbot, Silent AIM, FPS, Automation

Supervisor
Lecturer, Antti Juustila

Contents

1. Introduction

In this thesis the focus will be on bot detection and prevention in online games. In prevention one attempts to disrupt the behaviour of the bots while in detection the goal is to simply detect the bot while not affecting their operations. The methods described here contain both client-side and server-side solutions.

Why was the topic chosen? A major reason for why this topic was chosen was a legal one. Legal matters are most likely not the one of the main skill sets of a game developing company and trying to go into court with each bot developer could turn out to be a lengthy battle. Is it possible to deal with bots in such a way that the game developers wouldn't have to stray away from their core skill sets? This thesis aims to find an answer to this question. By using their efforts on bot developing algorithms for bot detection rather than getting into legal battles a game developing company could make a meaningful impact against bots while also staying within one of their core skills.

In Methods focuses on methods based on which bots operate. Also bot detection & prevention methods are described in chapter 2. In Results the results of some of the detection methods will be shown.

The most important finding was the fact that an automatic method analysing user inputs with the help of a cascade neural network could differentiate a bot from a human fast and reliably. Another finding was the fact that most methods aimed at preventing the bots functionality could often be counteracted by the bot developers with sufficient knowledge.

Chapter 4 is a discussion section where implications of the research is talked through. Chapter 5 contains conclusions of whether or not the research questions were answered sufficiently.

2. Methods

One must first understand how bots work if one wishes to build countermeasures against them. Descriptions on how bots works will be given. After this some examples on how one could prevent the bot from working. Lastly there will be discussion about ways in which bots could be detected.

2.1 How do bots work?

This will be a short introduction to methods that people use to build their bots. For bots to work their need to know information about their surroundings and then change the situation to a preferred state. In MMORPGs the preferred state could be to loot a monster and in FPS games the goal could be to aim the target.

2.1.1 How do bots get information about surroundings?

One of the simplest ways to get information about the game is to scan pixel values (Hoglund & McGraw, 2008, pg. 140). See Figure 1, here you simply use the visible User Interface and scan the color of a pixel in each location. From there you can estimate how much health your character has. This isn’t very fast since in order to get health value one has to scan several locations: A, B and C then compare their colour values.

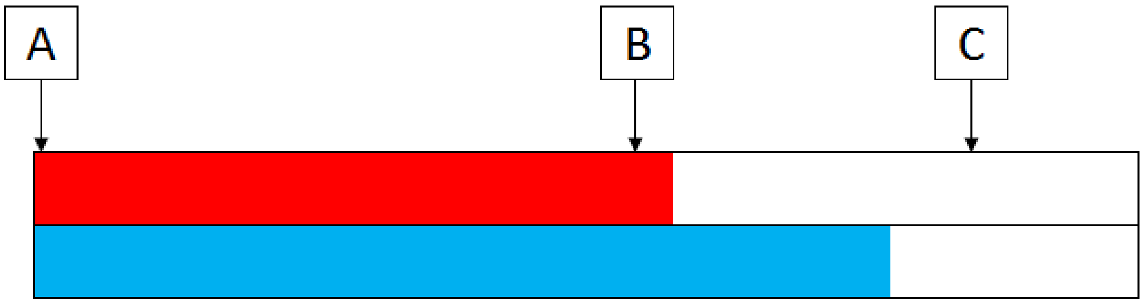

Figure 1 Getting characters health from the UI

Another way to get characters health would be to scan the memory of the running process. In order to get this information one has to read the memory from the correct location. This can be quite challenging since most programs contain a lot of memory that is not relevant for the bot to know. (Hoglund & McGraw, 2008, pg. 126). In order to find the correct memory location one can use debuggers such as Cheat Engine or OllyDbg.

The memory that assigned to an item in the game is not usually static, it moves around in the memory (Hoglund & McGraw, 2008, pg. 140). For example logging out and back into a game could easily move the health value to another address that is dynamically allocated there. There are however ways to overcome this. Using a debugger one can find the chain of pointers that leads to a desired value such as characters health. See Figure 2, here “Int 2” represents characters current health, which is allocated in a new place every time the program loads. If Struct A is located in a main loop of the program (or is a global variable) and get allocated to the same location every time the program starts, one could simply find the start address of "Struct A" and find the end of the struct which in this case is a pointer address to variable "Int 1". By reading the memory address after "Int 1" one can find the memory address of "Int 2", and finally by interpreting the value at that memory address as an Integer (or float or double, this depends on the game) you would get the health value of the character.

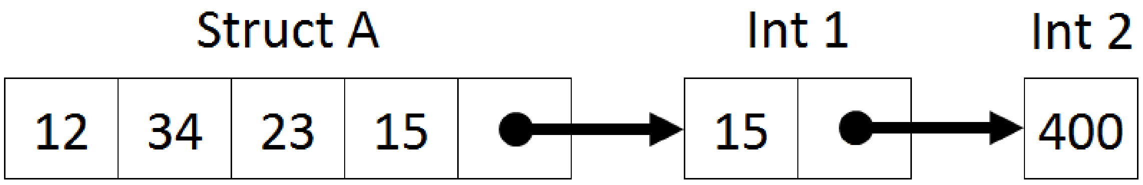

Figure 2 Example of a program's memory

Another way to get information is to rely on the traffic that is transferred between the server and the client. Often these packets are encrypted so one would need to decrypt the packets and after this determine the type of packet that is coming in. Lastly Hoglund & McGraw described a method that would rely on the fact the render information such as 3D objects need to somehow be communicated to the video card. By intercepting the communications between the game and the video Library of Microsoft Windows one could find the locations of the objects. (Hoglund & McGraw 2008, pg. 132).

2.2 How do bots interact with the virtual world?

One common methods to control bots is to send keystrokes to the game window. This can happen while the windows is on the foreground or on the background. Say you are able to read the characters location and the angle the character is facing in by reading memory as described in section 2.1 above. You could then build an algorithms that navigates to the given coordinates simply by broadcasting the correct keyboard inputs to the client.

Another way to move your character in the world is to use build in movement functions such as click to move (Brown, 2014). In this type of bot the bot has usually injected something into the client (usually a dynamic link library) after the game has started (Hoglund & McGraw, 2008, pg. 166). From there it can then call the processes own functions since its now considered to be a part of the program. Now the bot writers work could get much easier since there is no need to write a complex algorithm to control character movement through broadcasted keyboard inputs.

A third way to control your character would be to rely on the network traffic. In this situation the bot wouldn't touch the client at all but could instead be sending the movement related packet to the server. Some bot developers have even gone so far as to reverse engineer the entire game based on the packets alone. This type of bot would not necessarily need the game client at all. This type of bot is known as an "Out Of Game" bot or OOG for short. Because OOG bots don't interfere with the client's memory at all, and this bot requires much less computing power as the game client is not necessarily required. Due to lowered CPU and RAM requirements, one could run dozens of these bot simultaneously from the same computer.

2.2.1 Aimbot

In section 2.1.1 there were discussion about reading memory and getting information. One could also write directly to program memory. In a simple aimbot ment for FPS games one could simply read the player and target locations, then calculate the angle the players where one needs to aim, and finally write the desired angle values directly into the programs memory. This would snap the players aiming direction to the enemy and would make aiming easy. Similar methods could be used in an MMO game in order to turn the character in the desired direction (Hoglund & McGraw, 2008, pg. 195).

2.2.2 Silent Aimbot

Pritchard (2000) described a way to hack without interacting with the client at all. This bot relies on the fact that the client has to send their actions to the server somehow. Essentially this type of aimbot works by intercepting the packet that the client sends to the server. The packet is modified in such a way that would lead the bullet to hit the target no matter where the player is actually pointing his gun.

To combat this type of bot one could encrypt outgoing packages (Kang, Woo, Park & Kim 2013; Wendel 2012; Baughman, Liberatore & Levine 2007; Pritchard 2000.), this hampers the bot developer, since now one would have to figure out the encryption scheme that is being used before proceeding into packet modification.

In a worst case scenario and experienced bot developer could break the encryption scheme. They could decrypt the packages then edit the packet and then encrypt it again before finally sending it to the servers. In this case it would be hard to detect the bot, especially if the bot (and/or client) is placed on a separated instance for example on a separated VMware instance or on a separate computer entirely. In this case even memory scanning software such as BattleEye would be less likely to work since the code that edits the clients outputs would not even exist on the same computer. (Pritchard 2000).

2.3 How to prevent the creation of bots?

One way to prevent bots form being created is to block their access to information. By encrypting the network packets that are being sent. On the other hand there are limits to how strong the encryption can be, it shouldn’t strain the computer which could result in a noticeable FPS (Frames per second) drop. The encryption should also not be done in such a way that it would increase the traffic requirements beyond a reasonable level and cause network lag. (Pritchard, 2000).

The developer should limit the information that is exposed to the clients. For instance in a first person shooter one could edit the walls to be transparent. This type of cheat would work only if the servers are sending all information to the clients about their locations. One way to prevent this cheat would be to limit the information that is sent to the clients. In this case the server would only send information to player about enemies that he is supposed to see from his current location. (Pritchard, 2000). In a 2D case one could do this check by using the Ray-casting algorithm ("Ray-casting algorithm," 2017). Another way to encrypt certain portions of the games memory to make tampering more difficult. For instance in a recent court case it was mentioned that "Blizzard encrypts the data pointers used by its games so that hackers cannot locate critical gameplay data such as enemy and ally positions and enemy health" ("Blizzard Entertainment Inc. v. Bossland GMBH et al.", 2016).

When a program receives a new update the memory addresses will change. This can often be a problem for bot developers since they need to update their memory addresses to match the new version of the client. This could lead to a lot of work for the bot developer especially if there are tens of variables present that need to be scanned form memory. In essence this means that patching frequently could potentially make it harder for the bot developer to keep their bot updated. (Hoglund & McGraw, 2008, pg. 141).

Another way to prevent bot creation would be to make it difficult (or impossible) for a debugger to work properly with the program. If this is successful the bot creator would not be able to get the bot working since one cannot get information from the game since the functionality of the debugger is denied. Hoglund & McGraw mentioned that runtime checks could be used to detect changes in the programs memory. A simple anti-debugging measure would to check the `BeingDebugged` from the programs process environment block (PEB). Hoglund & McGraw also describe a simple method in which that flag can easily overwritten to indicate that no debugging is taking place. Another way to prevent debugging would be to raise an exception in the program and bring it to the debugger which would halt the debugging event. However this type of exception can also be circumvented by using the `ContinueDebugEvent` function. (Hoglund & McGraw, 2008, pg. 123).

DLL injection was shortly discussed in section 2.2, where a foreign .dll file was introduced into the process. If a foreign dll is detected, then detected the game could then take appropriate actions such as inform the servers about the event or crash the client. In order to detect a foreign dll one could use the code found in Appendix A, which is a modified version of the code found on Microsoft website (Traversing the Module List, 2017). The code will print a list of modules that are attached to a given process.

A partial output of the code can be seen below. Here we can see what dlls are attached to the running executable called "a.exe".

```
MODULE NAME: a.exe

MODULE NAME: ntdll.dll

MODULE NAME: kernel32.dll

MODULE NAME: KERNELBASE.dll

MODULE NAME: libgcc_s_dw2-1.dll

MODULE NAME: msvcrt.dll

MODULE NAME: libwinpthread-1.dll

MODULE NAME: libstdc++-6.dll

MODULE NAME: USER32.dll

MODULE NAME: GDI32.dll

MODULE NAME: LPK.dll

MODULE NAME: USP10.dll
MODULE NAME: vmwsci.dll

MODULE NAME: PSAPI.DLL
```

This method works as long as the injected dlls are not hidden. There are of course ways to hide injected dlls from appearing in the list (Hoglund & McGraw, 2008, pg. 176).

2.4 How to detect bots?

In chapter 2.3 some detection methods were already described such as scanning for injected dlls. Now some other detection methods will be explored. It’s not always possible to prevent bots from roaming in games. For this reason detection is also important in addition to prevention. Next we discuss ways in which bots can be detected based on their behaviour in the virtual world.

2.4.1 Detection by other players

To combat these type of cheats one could rely on player reports. In order to check the player reports one would most likely need to hire staff to verify the accuracy of these reports, this of course costs money. An alternative to hiring staff would be to have trusted players monitor others. This type of method has been put into use in Counter Strike Global Offensive (Overwatch Faq, n.d.). To make this type of detection less effective the aimbot creators can also reduce the Field of View (FOV) in which the aimbot gets activated. From the spectators viewpoint this type of cheating seems more legitimate and will be harder to detect.

2.4.2 Detection based on movement patterns

In MMORPG games the bots usually move with the help of waypoints. First a human has to record the waypoint. Once a sufficient amount of waypoints has been recorded the bot can move using these waypoints. Bot movement typically follows the same waypoints in a set order, which leads to a lot of repetition on the exact same path. If the waypoints are stored in a two-dimensional map it could be rather trivial for a human to figure out which one was made by a bot and which one wasn’t. However for a large player base it could take a lot of work to figure out who is a bot and who isn’t. Ideally this type of bot could be detected automatically. (Mitterhofer, Kirda, Kruegel & Platzer, 2009)

The data collection for this can be done entirely on the server side, which is good when considering the privacy concerns of the players. It’s also good because bot writers won’t get a clue as to why the bot was banned since the bot write cannot see unusual happening on the clients side (such as game crashing, etc.).

Depending on the movement method a bot uses the server could receive the coordinates the player is currently moving (click-to-move) to, or their current location with a direction vector. In either case the player’s location is easily obtained on the server’s side. Player coordinates will serve as a basis for the bot detection. (Mitterhofer et al. 2009). To capture player locations one could simply write down player locations at short intervals such as 0.5-1 seconds. Each coordinate is linked with its previous coordinate. Once there are enough coordinates on could then use a line simplification algorithm such as Douglas-Peucker to reduce the amount of overlapping waypoints. (Mitterhofer et al. 2009)

Next one can extract the waypoints from the dataset. When the bot moves the same track repeatedly many of dots that accumulate from player locations will end up near each other forming clusters. The centre of these clusters will serve as the waypoint centre, and the waypoint itself has a fixed diameter. The waypoint size should be chosen in such a fashion that when the player moves through the waypoint their visit at the waypoint could be detected. Once the waypoint size has been determined one can then use a clustering algorithm to find all the waypoints, the cluster size should be limited to the waypoint size that was determined earlier. The end result could look something like the 2nd picture in "Figure 3 Waypoint extraction". (Mitterhofer et al. 2009)

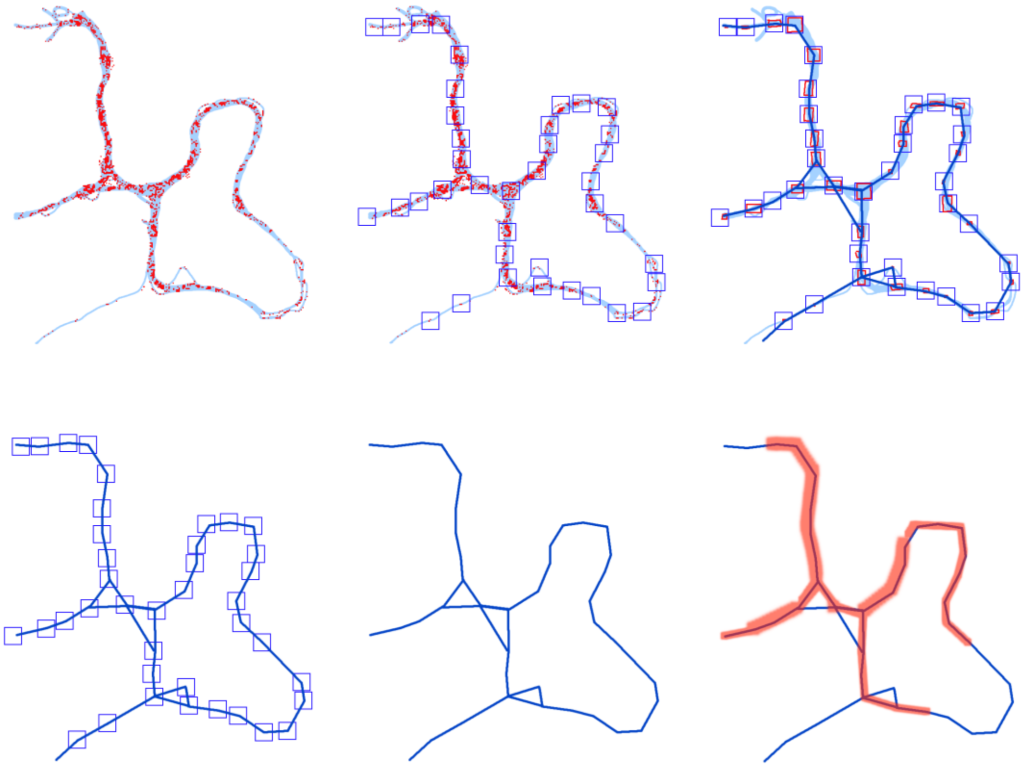

Figure 3 Waypoint extraction (recreated picture from Mitterhofer, Kirda, Kruegel & Platzer, 2009)

To find repetitive paths one could simply see how many times a specific waypoint sequence is repeating itself. Mitterhofer et al. called this the Largest Common Prefix method (LCP). Another way to find repetition in the movement of a character is to use a path segments. A path segment consist of two waypoints A and B. If character traverses form Waypoint A directly to B we increase the value of this path segment by one. To get a meaningful unit of measure out of the path segments we can simply divide the total amount of segment passes with all the segments in the path. For a human player the number of segments will grow constantly while the bot repeats the same path over and over. This will lead to low number of average path segment passes for the humans and higher for bots.

2.4.3 Detection based on user inputs

A human operates the character in a different way than a bot. One could for instance observe how a player sends a keystroke, points their cursor, clicks their mouse or uses point-and-click and drag-and-drop. Bots can often move mouse in a much quicker fashion and send keystrokes faster than a human can. These differences can be analysed and a decision can be made whether a player is a bot or a human. (Gianvecchio, Wu, Xie and Wang, 2009)

Gianvecchio et al. collected input data from 30 participants with varying ages and gaming experience. A program called RUI (Kukreja, Stevenson and Ritter 2006) to capture keyboard and mouse inputs from the user. Data such as key press duration, their frequency, the speed of drag-and-drop and point & click were measured among others things. Players were encouraged to take part in farming activities such as finding treasures or killing monsters instead of other activities such as exploring or socializing. The game bot data was collected by WoW Glider bot in 7 locations. A total of 40 hours of gaming data was gathered from the bots. The graphs from the input data can be seen in the pictures below (Figure 4 Keystroke arrival times and keystroke duration (re-created picture from Gianvecchio et al. 2009)Figure 5 Mouse movement speed & duration). (Gianvecchio et al. 2009)

In Figure 4 Keystroke arrival times and keystroke duration one can see the keystroke duration distribution for bots and humans. For the bots the keypress duration is very short and the most common keypress duration was at the 0.1 second mark. For Humans most common keypress duration was 0.2 seconds and the durations were distributed on a wider area. In Figure 4 Keystroke arrival times and keystroke duration one can see the keystroke Inter-arrival time distribution for bots and human. One can see spikes in the inter-arrival time in the bot graph at the 0, 1 and at 5.5 seconds, which would indicate that the bot is using periodic timers in order to complete certain actions. (Gianvecchio et al. 2009)

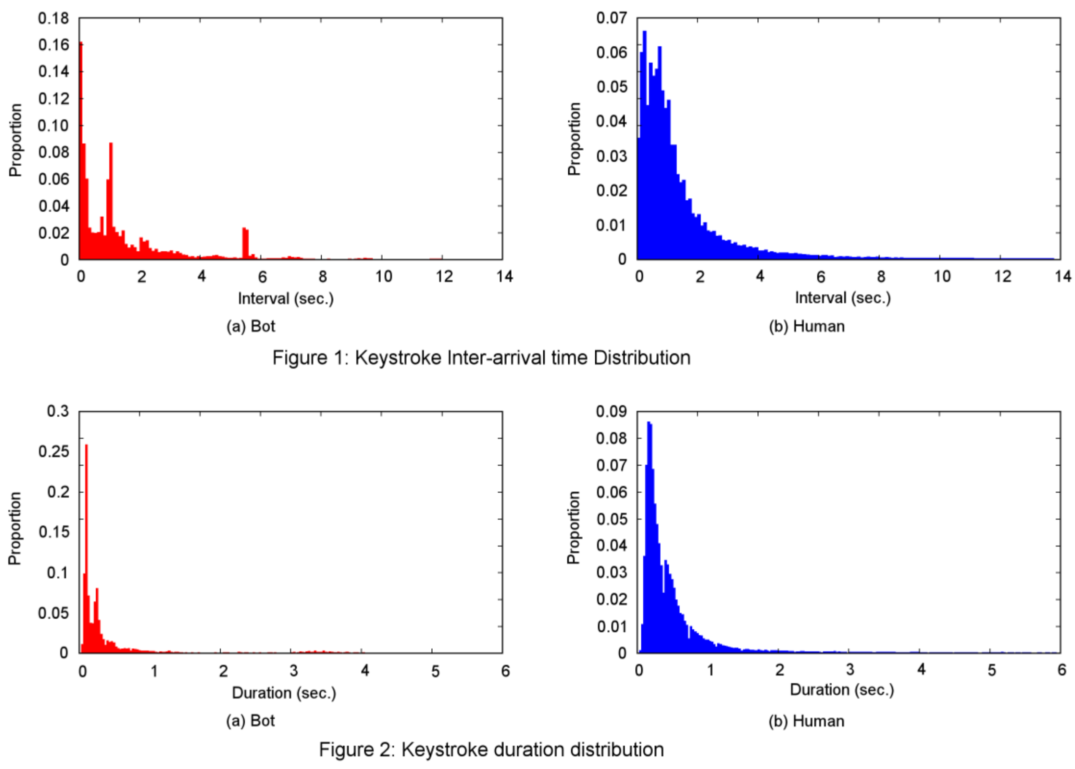

Figure 4 Keystroke arrival times and keystroke duration (re-created picture from Gianvecchio et al. 2009)

In Figure 5 Mouse movement speed & duration one can see the average speed compared to the displacement for point-and-click actions. For bots the speed seems to increase a lot when the displacement distance increases. For humans the cursor movement speed increases only slightly when the displacement distance increases. Figure 5 Mouse movement speed & duration also contains the drag-and-drop duration distribution graph in which the bots mostly complete their actions very fast and have their duration distributed in a narrow area. For humans the drag-and-drop duration is more varied, extending well above the one second mark. (Gianvecchio et al. 2009)

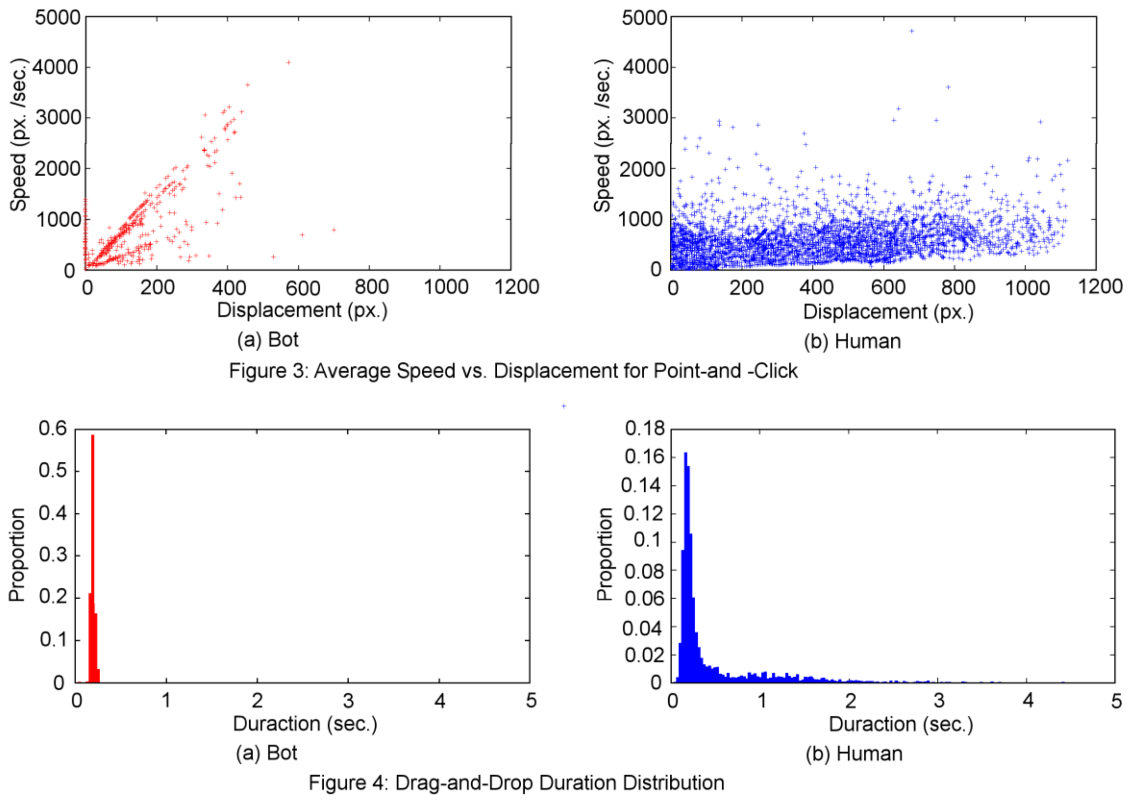

Figure 5 Mouse movement speed & duration (re-created picture from Gianvecchio et al. 2009)

2.4.4 Other detection methods

Thawonmas, Kashifuji & Chen described a method that would check how often a player would do certain activities. Bots are known to repeat similar behaviours over and over and this could be used as a basis for detection. (Thawonmas, Kashifuji & Chen 2008). Another way to detect bots would be to track the trajectory of a mouse. The trajectory would then be analysed and a result between a bot and a human could be identified. (Pao, Fadlil, Lin & Chen 2012) (Pao, Chen, & Chang 2010)

For online card games there was method in which the cards that would be shown to players would have the card icons change. For instance the "King of Hearths" card could occasionally have a text on it that says "King of Hearths". This can easily throw off a bot that is working based on image recognition. (Yampolskiy & Govindaraju 2008)

Golle & Ducheneaut described a method in which all input actions should be verified with special input devices. Any way to bypass these inputs with non-physical inputs would lead the player to be disconnected. The downside with this method is that it would require keyboards, joysticks and other devices to be specifically developed for this purpose. (Golle & Ducheneaut 2005). At this point and time not many (if any) people own these types of devices and for a game developer this isn't a very feasible approach since they would need the person to own the specific hardware.

Chen, Jiang, Huang, et al. described a way to detect bots that would include analysing the arrival times of the packets. Bots often use timers to do certain actions which could lead to highly regular patterns that could be detected. (Chen, Jiang, Huang, et al. 2008). On a related note one could also identify if the packet has an irregular form. For example in 2.2 there was a mention of an OOG bot that would function only based on network packets. This kind of bot could easily be detected simply by updating some basic function by adding one additional bit of data at the end that the normal client sends. Once legitimate user updates their client their client will now send one extra bit of information per certain packet (such as movement packet). The bots would still be sending the old movement packet with one bit missing and that could an easy way to detect this type of bot. From there on it would be easy to spot that the user is not playing the game with the regular client.

Other approaches include analysing the trading networks of the user (Keegan, Ahmed, Williams et al. 2010) or observing the times of the day (or week) that the user is active (Chu, Gianvecchio, Wang & Jajodia. 2010).

3. Results

In this chapter one can find some of the test results from certain bot detection mechanics. The methods that were used here have been described in Chapter 2.

3.1 Bot detection Based on movement patterns

In Chapter 2.4.2 there were discussion about bot detection based on their movement patterns. In this chapter contains discussion about the effectiveness of the methods. In order to test the validity of the LCP method and the “average segment paths crossed” method ten human players player World of Warcraft on a custom server. The players were instructed to concentrate on resource gathering, in order to make their gaming resemble that of the bots. For the bots two different paths were created. Each of the bot paths were traversed with both Glider bot and the ZoloFighter bot, so 4 bot traces in total. For the second trade for the Zolofighter a human take over at around packet 1200 which can be seen as a drop in both the LCP value and in the “Average Line Segments Passed” values. (Mitterhofer et al. 2009)

When looking at “Figure 6 Average Line Segments passed” one can see that the Line segments passed for humans remain at or below 2 for most of the time. For bots the value keeps rising steadily. In “Figure 7 Average LCP values” one can see that humans have a very low LCP value compared to the bots. In the study all the bots could be detected within 12-60 minutes while there being no false positives from humans.

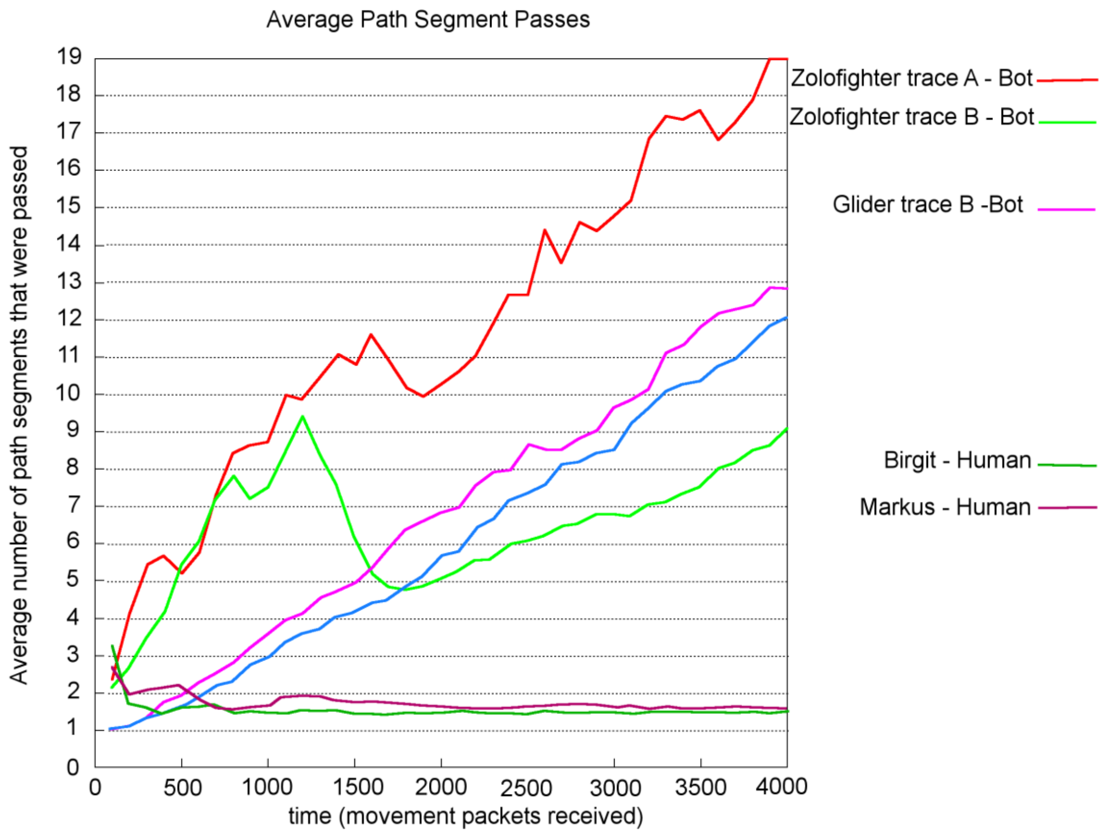

Figure 6 Average Line Segments passed (re-created picture from Mitterhofer et al. 2009)

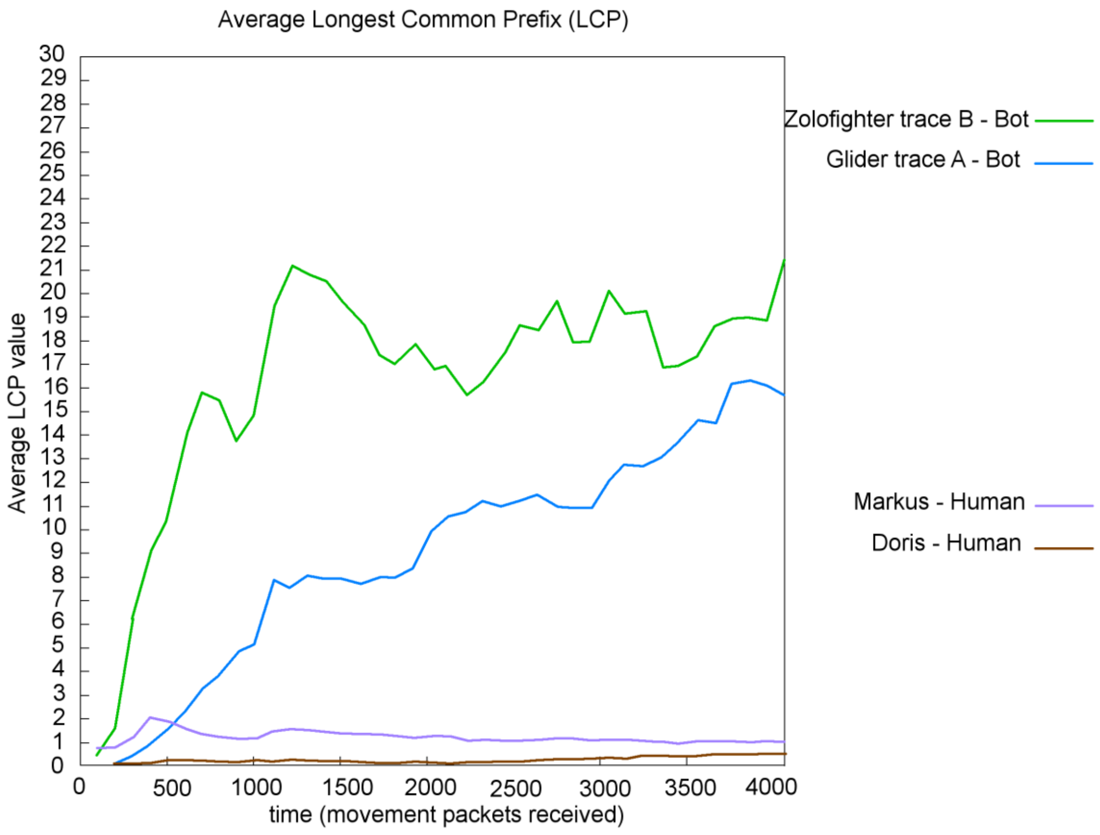

Figure 7 Average LCP values (re-created picture from Mitterhofer et al. 2009)

3.2 Bot detection Based on user inputs

11 discussed a method for detecting bots based on user inputs. Best results were obtained from a cascade neural network with 40 nodes in total, 4 of which were inputs. The resulting network could differentiate bots from humans with 99% accuracy within the first 60 seconds. True positive rate of 0.998 was achieved in World of Warcraft. The same network had its threshold values adjusted with new traces from Diablo2. This network reached a true positive rate of 0.864. This shows that even though the network was not optimized for Diablo 2 it could still effectively differentiate between a human player and a bot. (Gianvecchio et al. 2009)

4. Discussion

Bot detection can be viewed as an arms race between the bot developers and the game developers, each on is trying to one-up the defences of the opponent and launch a counter-attack to detect the vulnerabilities of the opponent. In the cases that were presented the bots could be detected with a fair degree of certainty, and the game developing company would then be free to take necessary actions. For a game developer it might be beneficial to approach the subjects on many fronts: trying to prevent bots from functioning and also by building algorithms that will detect the bots based on their behaviour. By doing so one can assure that bot developers would have a non-trivial or labour intensive task ahead of them if they had to for instance, mimic the keyboard outputs of a human player, or trying to get around a protection mechanism that encrypts certain parts of a programs memory.

5. Conclusions

This thesis described methods that can be used to detect bots. One way to prevent bots would be to prevent the bot software from functioning. The second way would be to detect a functioning bot based on their actions. Two detection methods: Input analysis, trajectory analysis were used in determining whether or not a player is a bot. The result in both bases were that bots could be detected with a high degree of accuracy. The method that was based on input analysis could differentiate users within one minute. The method related to trajectory analysis could differentiate a bot from a human within 12 minutes. To conclude it's possible for game developers to stay within their core skill sets (coding) and deal with bots with other than legal means.

6. References

Baughman, N. E., Liberatore, M., & Levine, B. N. 2007. Cheat-Proof Playout for Centralized and Peer-to-Peer Gaming. in IEEE/ACM Transactions on Networking, vol. 15, no. 1, pp. 1-13, Feb. 2007. Available: https://dl.acm.org/citation.cfm?id=1241833

Blizzard Entertainment Inc. v. Bossland GMBH et al. 2016. Central District of California July 1 2016. Available: https://www.unitedstatescourts.org/federal/cacd/652412/

Brown Andrew. 2014. "As a game developer, how can I tell if a player is botting?". Cited on 17.10.2017. Available: https://www.quora.com/As-a-game-developer-how-can-I-tell-if-a-player-is-botting

Chen, KT., Jiang, JW., Huang, P. et al. 2008. Identifying MMORPG Bots: A Traffic Analysis Approach in EURASIP J. Adv. Signal Process. (2008) 2009: 797159. Available: https://doi.org/10.1155/2009/797159

Chu, Z., Gianvecchio, S. Wang,H., Jajodia, S., 2010. Who is tweeting on Twitter: human, bot, or cyborg?. In Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC '10). ACM, New York, NY, USA, 21-30. Available: https://dl.acm.org/citation.cfm?id=1920265

Gianvecchio, S., Wu, Z., Xie, M. and Wang, H. 2009. Battle of Botcraft: Fighting Bots in Online Games with Human Observational Proofs. CCS '09, Proceedings of the 16th ACM conference on Computer and communications security, pp. 256-268. Available: https://dl.acm.org/citation.cfm?id=1653694

Golle, P. and Ducheneaut, N. 2005. Preventing Bots from Playing Online Games. ACM Computers in Entertainment, Vol. 3, No. 3, July 2005. Available: http://dl.acm.org/citation.cfm?doid=1077246.1077255

Hoglund Grew, McGraw Gary 2008. Exploiting online games : cheating massively distributed systems. Pearson, 340 pages. ISBN 978-0-13-227-191-2

Kang, A.R., Woo, J., Park, J. & Kim, H. K. 2013. Online game bot detection based on party-play log analysis. In Computers & Mathematics with Applications, Volume 65, Issue 9, 2013, Pages 1384-1395, ISSN 0898-1221. Available: http://www.sciencedirect.com/science/article/pii/S0898122112000442

Keegan, B., Ahmed, M.A., Williams, D., Srivastava, J., Contractor, N. "Dark Gold: Statistical Properties of Clandestine Networks in Massively Multiplayer Online Games," in 2010 IEEE Second International Conference on Social Computing, Minneapolis, MN, 2010, pp. 201-208. Available: http://ieeexplore.ieee.org/document/5590455/

Kukreja, U., Stevenson, W. E. and Ritter, F. E. RUI: Recording user input from interfaces under Windows and Mac OS X. Behavior Research Methods (2006) 38: 656. Available: https://doi.org/10.3758/BF03193898

Mitterhofer, S., Kirda, E., Kruegel, C. & Platzer, C. 2009. Server-Side Bot Detection in Massively Multiplayer Online Games. IEEE Security & Privacy, Issue No. 03 - May/June (2009 vol. 7), ISSN: 1540-7993,pp: 29-36. Available: http://doi.ieeecomputersociety.org/10.1109/MSP.2009.78

Overwatch Faq. n.d. n.a. Cited on 9.10.2017. Available: http://blog.counter-strike.net/index.php/overwatch/

Pao, H.K., Chen, K.T., Chang, H.C. "Game Bot Detection via Avatar Trajectory Analysis" in IEEE Transactions on Computational Intelligence and AI in Games, vol. 2, no. 3, pp. 162-175, Sept. 2010. Available: http://ieeexplore.ieee.org/document/5560779/

Pao, H.K., Fadlil, J., Lin, H.Y., Chen, K.T, Trajectory analysis for user verification and recognition, In Knowledge-Based Systems, Volume 34, 2012, Pages 81-90, ISSN 0950-7051. Available: https://www.sciencedirect.com/science/article/pii/S0950705112000767

Pritchard, M. 2000. How to Hurt the Hackers: The Scoop on Internet Cheating and How You Can Combat It. Cited on 3.10.2017. Available: https://www.gamasutra.com/view/feature/131557/how_to_hurt_the_hackers_the_scoop_.php

Ray-casting algorithm. n.d. N.A. Cited on 23.10.2017. Available: https://rosettacode.org/wiki/Ray-casting_algorithm

Thawonmas, R., Kashifuji, Y., Chen, K.T. 2008. Detection of MMORPG bots based on behavior analysis. ACE '08 Proceedings of the 2008 International Conference on Advances in Computer Entertainment Technology Pages 91-94. Available: http://dl.acm.org/citation.cfm?id=1501770

Traversing the Module List. n.d. n.a. Cited on 23.10.2017. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms686849(v=vs.85).aspx

Wendel, E. (2012). Cheating in Online Games: A Case Study of Bots and Bot-Detection in Browser-Based Multiplayer Games. Master's thesis, Norwegian University of Science and Technology, Trondheim, Norway. Available: http://hdl.handle.net/11250/262729

Wu, C., Zhu, G.h., Wu, Y.h. and Xiang, R. 2009. The Study of Bot Technology for Online Games. Third International Conference on Genetic and Evolutionary Computing, Guilin, 2009, pp. 654-656. Available: http://ieeexplore.ieee.org/document/5402750/

Yampolskiy, R.V., Govindaraju, V.. 2008. Embedded noninteractive continuous bot detection. Comput. Entertain. 5, 4, Article 7 (March 2008), 11 pages. Available: https://dl.acm.org/citation.cfm?id=1324205

Appendix A. Detect active dlls in programs memory.

```
#include <windows.h>

#include <tlhelp32.h>

#include <tchar.h>

#include <iostream>


//  Forward declarations:

BOOL ListProcessModules( DWORD dwPID );

void printError( TCHAR* msg );


int main( void )

{

  ListProcessModules(GetCurrentProcessId() );

  return 0;

}


BOOL ListProcessModules( DWORD dwPID )

{

  HANDLE hModuleSnap = INVALID_HANDLE_VALUE;

  MODULEENTRY32 me32;


//  Take a snapshot of all modules in the specified process.

  hModuleSnap = CreateToolhelp32Snapshot( TH32CS_SNAPMODULE, dwPID );

  if( hModuleSnap == INVALID_HANDLE_VALUE )

  {

    //printError( TEXT("CreateToolhelp32Snapshot (of modules)") );

    return( FALSE );

  }
//  Set the size of the structure before using it.
  me32.dwSize = sizeof( MODULEENTRY32 );


//  Retrieve information about the first module,
//  and exit if unsuccessful
  if( !Module32First( hModuleSnap, &me32 ) )
  {
    //printError( TEXT("Module32First") );  // Show cause of failure
    CloseHandle( hModuleSnap );        // Must clean up the snapshot
object!
    return( FALSE );
  }


//  Now walk the module list of the process,
//  and display information about each module
  do
  {
        std::cout<<"\n     MODULE NAME:     "  <<me32.szModule;
        std::cout<<"\n     process ID       " << me32.th32ProcessID;
  } while( Module32Next( hModuleSnap, &me32 ) );


  CloseHandle( hModuleSnap );
  return( TRUE );
}
```

```
void printError( TCHAR* msg )

{

  DWORD eNum;

  TCHAR sysMsg[256];

  TCHAR* p;



  eNum = GetLastError( );

  FormatMessage(              FORMAT_MESSAGE_FROM_SYSTEM              |
FORMAT_MESSAGE_IGNORE_INSERTS,

         NULL, eNum,

         MAKELANGID(LANG_NEUTRAL,    SUBLANG_DEFAULT),    //    Default
language

         sysMsg, 256, NULL );



  // Trim the end of the line and terminate it with a null

  p = sysMsg;

  while( ( *p > 31 ) || ( *p == 9 ) )

    ++p;

  do { *p-- = 0; } while( ( p >= sysMsg ) &&

                          ( ( *p == '.' ) || ( *p < 33 ) ) );

}
```