NTNU – Trondheim
Norwegian University of
Science and Technology

Cheating in Online Games

A Case Study of Bots and Bot-Detection in
Browser-Based Multiplayer Games

Erik Wendel

Master of Science in Communication Technology
Submission date: July 2012
Supervisor: Harald Øverby, ITEM

Norwegian University of Science and Technology
Department of Telematics

Problem Description

From being a mere curiosity in the 70's, computer gaming has become mainstream and grown into being one of the world's biggest digital entertainment industries. Its popularity has drawn the attention of hackers and exploiters, who quickly put game software security on the agenda. The games of the 90's and early 00's were easy to hack, and this effectively ruined online play in some games. Modern games employ various anti-cheat mechanisms, but different kind of cheats still exist and are being sold through cheater websites. This master thesis seeks to perform an analysis of what kind of cheats exists in the modern gaming market, and what countermeasures are employed.

Assignment given: January 16th 2012

Supervisor: Harald Øverby

Abstract

The video and computer gaming industry has seen a significant rise in popularity over the last decades and is now the worlds biggest digital entertainment industry [1]. Where games meant Space Invaders and Doom in the old days, gaming is now everything from ultra-realistic shooter games to farm management simulators integrated in the Facebook platform, to games on our smartphones and tablets. This popularity has brought with it the attention of hackers and exploiters, and game cheats flourish in the shady parts of the internet. This thesis has two parts. The first part presents a background study, an oveorview of today's situation in regard to cheating and anti-cheating in online games. The second part is a case study about bots and bot detection in Browser-Based Multiplayer Games (BBMG).

In the background study, the main finding is that there are over twenty websites selling cheats or cheat subscriptions, and that the type of cheats available for a game is heavily dependent on the game genre. Most or all first-person shooters (FPS) have available aimbots and wallhacks, and all major massively multiplayer online games (MMO) are exposed to bot programs. The cheats are being sold either alone or through monthly subscriptions, allowing free use of all cheats for all supported games by that vendor. There have been many anti-cheat actors over the past decade, but three known services being continually developed. The two biggest are Valve Anti-Cheat, used by most games managed through Valve's online gaming platform Steam, and PunkBuster, which protects amongst others the Battlefield series. The last big anti-cheat software is the proprietary Warden, used only in Blizzard's own games like the Diablo. Warcraft and Starcraft games. Many cheats are still able to bypass the security mechanisms provided by these, and it is a continuos arms race between cheat developers and anti-cheat developers, just like in the virus and anti-virus industry.

In the case study, two bots were written for a non-disclosed BBMG and their performance was tested quantitatively by playing several game accounts in parallel. It showed that bot use yields large performance gains both in the early game stages and for advanced players.

As a result of the case study research on bots in online games, a Bot-Detection System (BDS) is created and presented in the last chapter. The goal of the BDS is to detect the use of bots in BBMG by identifying a set of attributes and comparing these to average human behavior. A score system

ranging from 0 to 100 is introduced, where the average human behavior is defined as 0, while increasing non-human behavior is given a score >0, with max 100. The BDS is then employed on the two bots and returns scores of 64 and 52, while the six human play testers receive scores of 1 or 0.

Preface

This document serves as a masters thesis in Communication Technology at The Norwegian University of Science and Technology. It contains work done from January to June 2012.

First I would like to thank my supervisor Harald Øverby for very valuable discussions throughout the semester. The thesis has changed and evolved constantly and it has been a challenging and rewarding experience to conduct research in a self-chosen topic. I am very happy with the direction the thesis ended up taking.

I would also like to thank my girlfriend for helping me through this demanding time, and also my family for their support throughout my education here in Trondheim.

Lastly - thanks to all the wonderful people who contributed to making these five years an absolutely incredible experience - you know who you are.

Contents

Nomenclature

BBMG	Browser-based Multiplayer Games
BDS	Bot Detection System
DLC	Downloadable Content, addon packs for games
FPS	First-person shooter
MMO	Massively-multiplayer online game
MMOG	Massively-multiplayer online game
MMORPG	Massively-multiplayer online role-playing game
RPG	Role-playing Games
RTS	Real-time strategy game
WoW	World of Warcraft

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

The concept of video gaming has changed drastically over the past years. Social gaming platforms like Steam and Battle.net have helped create a more connected and social game experience with your friends being only a click away. Innovation has spawned new game genres, and new gaming-enabled platforms such as Facebook or the smartphone arena has contributed to making games more available for the general public. The Entertainment Software Association (ESA) state that 72% of all American households play computer or video games[2], the average gamer is 37 years old and has been playing games for 12 years and 42% of all gamers are female. Video gaming is now the worlds largest digital entertainment industry [1].

This rise in popularity have brought with it cheaters and hackers. Cheat programs flourish in the shady parts of the internet, and seem to be available for all the major games. These can for example give a player superhuman aiming skills, the ability to see through walls or move incredibly fast. Cheat use is seemingly detrimental to the game companies and they try to stop it by implementing anti-cheat functionality in their games, or by including a third-party anti-cheat service such as Valve Anti-Cheat or PunkBuster in their games. Still, there are loads of cheating websites selling and distributing cheats for a variety of games today, many years after the first release of these anti-cheat efforts.

In the game genre of massively-multiplayer online games (MMOG) where thousands of players play together in a shared virtual world, automated programs called bots that perform the boring parts of the game for you is a big problem. The virtual economy related to selling in-game currency and virtual items is estimated to $3 billion in 2011, and it is thought that bots account for 50% of the items and

currency that generate this sum [3]. This leads to inflation in the games, creating an imbalance that penalizes the ordinary players.

Bots are also a threat to another type of games: browser-based multiplayer games (BBMG). These are games that are played in a web-browser, usually in a MMO setting with thousands of other players in the same game world. BBMG have gained huge popularity in recent years due to ubiquitous game access and appealing game mechanics. The simplicity of the game protocol between the game client and the game server makes it easy to make a bot that mimics the HTTP request pattern found in the games. These games also usually contain various game mechanisms that are well suited for automation and will yield large performance gains for the player.

There has been much research on the topic of game security and cheating in games, but mostly on isolated topics. [4] presents a cheating taxonomy, while [5, 6, 7, 8, 9] focus on bot detection in MMOG's based on character avatar movement patterns or patterns in user input. This thesis intends to perform a general background study on the use of cheating in modern online games, and conduct a case study on bots in BBMG.

1.2 Objectives

The thesis is consists of two parts: a background study and a case study. The background study shall investigate the current state of cheating in online games. This includes studying what kind of games are being cheated, what the cheats accomplish, the scope of cheating and how the cheats work. It shall also map out what is being done to stop the cheaters by the game developers and third party anti-cheating services, and present an overview of the available anti-cheat software and how they work. Finally, a case study on a particular browser-based multiplayer game is performed. This case study includes creating a bot for the game, measuring its performance compared to human play and propose a bot detection system that would prevent use of this bot. The case study is a significant part of this thesis.

The thesis goals are compressed down to four research question sets as shown below.

- **Q1:** What cheats exist for modern online games? Which games and game genres are being cheated? How are cheats distributed?
- **Q2:** What anti-cheat services exists today? What countermeasures are being used to stop cheating?
- **Q3:** How can one create a bot for use in browser-based multiplayer games? What benefits are achieved from using bots?

- **Q4:** How can bot use in browser-based multiplayer games be prevented?

1.3 Document structure

The rest of this document is outlined as follows:

Chapter 2 provides an introduction to video games.

Chapter 3 presents the first part of the background study. Section 3.1 contains an introduction about cheating in games. Section 3.2 contains an overview of the cheat types found in todays online games. It also describes each of these cheats in detail. Section 3.3 explains how cheats are made and distributed to cheaters through cheater websites. Section 3.4. presents the results of the background study on cheat availability and a discussion of these.

Chapter 4 contains the second part of the background study and is about anti-cheating. It begins with an introduction and some history about anti-cheating, before section 4.2 presents an overview of current anti-cheat software. Section 4.3 shows the game coverage of the three anti-cheats.

Chapter 5 presents the case study of a browser-game bot. Section 5.1 starts with describing browser-games in general, follows up with a brief description of the game in question and the methodology used for writing the bot. Sections 5.2 and 5.3 present the two bots written. The first bot is a resource optimization bot for advanced players, while the second bot is made to develop a player's base in the early stage of the game. Section 5.4 describes an experiment performed where an aggressive bot was written and run for several weeks with the intent of getting detected by the game company. Section 5.5 contains a discussion.

Chapter 6 presents a Bot-Detection System for Browser-Based Multiplayer Games. Section 6.1 contains a general description of the framework, while section 6.2 applies the framework on the game and the bots from chapter 5. Section 6.3 wraps up the chapter with a discussion of the results.

Chapter 7 discusses the main findings and their implications on a high level.

Chapter 8 answers the thesis objectives found in section 1.2.

1.4 Scope and limitations

Only multiplayer games studied

Only multiplayer games with online play will be included in the study. This is not the same as saying that singleplayer games are not interesting in a game security perspective, however, this would be a completely different thesis. A choice had to be made and it fell on multiplayer games. There are two main reasons for this. First of all, the author has vast experience with multiplayer games and is familiar with the online gaming scene. This increases the motivation for working with these games, as cheaters have always been present as an annoying problem. Second, some multiplayer games use a subscription model, where the players pay a monthly fee in order to be able to play. Most massively multiplayer online games (MMO's) use this model, and this means that they are dependent on a keeping a large user base over time , in contrast to other games where the game company only earns money through the initial game sale. This encourages them to continue improving the game and keeping the game interesting to play for the gamers after its release. A game filled with cheaters and bots is not that appealing, and the subscription model adds a large incentive for game companies to invest in anti-cheating measures.

Choice of games for background study

There are several hundreds of online multiplayers games on the market, if not thousands, and would be very time-consuming to do a full exhaustive study of these. Such a study would be very important, but it isn't the purpose of this thesis. In order to fulfill the thesis objectives, a selection of 25 games have been chosen for the background study. These have been chosen to form a wide range of modern online games, but still making sure to include all the big bestsellers. Some older games and games with smaller playerbases than the current blockbusters are also included to show the breadth of cheating in games.

Non-disclosure of the case study game

The identity of the browser-game used for the case study will not be disclosed in this thesis. This is to avoid revealing the weaknesses of the game to the general public and encourage cheating in the game. The game is therefore described in a general manner, not revealing the identity of the game while still revealing enough of the game mechanics for the reader to understand the concept of the game. Also, the source code for the bot will be slightly obfuscated, by refactoring some variable and game function names.

1.5 Contributions

This thesis contains five main contributions:

1. **Section 3.4:** Classification of cheat types and cheat availabilities for modern online games.

2. **Chapter 5:** Development and quantitative performance analysis of two bots for a BBMG.
3. **Section 6.1:** A Bot-Detection System for BBMG is proposed.
4. **Section 6.2:** Application of the proposed Bot-Detection System on the bots from chapter 5.
5. **Appendix C:** A scientific paper written about the Bot-Detection System, submitted to the CCSW 2012 conference.

1.6 Methodology

This section will describe briefly the methodology used in the thesis. Further descriptions exist in the respective parts where certain methods have been used.

In chapter 6, Wireshark was used to capture the data packets. A capture filter was used to capture only outgoing HTTP traffic by specifying a destination IP address for the packets to be captured. The Wireshark files were saved as .pcap-files as backup, and the contents were exported to comma-separated values (.csv-files) and then opened in Microsoft Excel. Excel was used for all data processing as well as making the graphs used in the thesis report. Data was also gathered from friends the author had made while playing. These were instructed to set up Wireshark and run it in the abovementioned manner, and run it for at least a day's worth of playing or until 1000 captured packets. The instruction letter sent to these participants can be seen in Appendix B.

Ideally, there would be more participants in the study and these would play in a controlled environment at the university to avoid possible sources of error and to ensure correct data results. As the players agreed to participate by their own choice, it was not possible to select an ideal group in terms of age, gender and game experience-distribution. Creating a more accurate human average from a more statistically correct test panel in a controlled environment could be done as further work.

Chapter 2

Background

2.1 History of video games

The history of video games started in 1947, when two Americans filed for a patent on what they called a "cathode ray tube amusement device". This was a simple, analog device where the user controlled a vector-drawn dot on the screen, simulating a missile being fired. Games developed slowly over the next twenty years, where most of them were being run on mainframe computers located in universities in the Unites States. One of the first notable games were "Spacewar!", developed in 1961 by a group of students from MIT. Later being distributed through what was later to become the internet, this can be seen as one of the first digitally-distributed games. The 1970s was the start of what can be called the golden age of arcade video gaming, where single-purpose gaming machines were set up in restaurants, bars and amusement parks.

Gaming consoles was the next big thing, starting with the Magnavox Odyssey in 1972 which could play 28 different games on 10 different cartridges. Game consoles are dedicated gaming machines that connect to your TV and use it as the gaming screen, using custom-made controllers for user input. Nintendo hit a home run with their Nintendo Entertainment System in 1985, and their hugely successful game Super Mario Bros attracted lots of new people to gaming. Arcade games and game consoles along with a few blockbuster titles helped to popularize gaming and attracted more and more people throughout the next decades. Sony's Playstation released in 1995 and Nintendo's alternative Nintendo 64 from 1996 was the start of todays popular consoles, consisting of Playstation 3, Nintendo Wii and Microsoft's alternative XBOX360. These three console alternatives together with gaming on the PC platform constitute the basis of today's gaming.

Figure 2.1: The Nintendo Entertainment System (NES)

Facebook and smart phones like the iPhone has brought with it a new gaming-niche the later years. Games like "Angry Birds" for mobiles or "FarmVille" for Facebook have had great success, and must also be taken into account when defining video games. These new additions, along with new game genres like "The Sims" where you control a virtual person where you can create your own house, take jobs and advance your career, start relationships etc have widened gaming to not just be all about killing monsters and playing war.

An important aspect of modern gaming is the social one. Most games come with a multiplayer mode, while some games revolve exclusively around multiplayer and does not have a single-player mode. Single-player games typically have one campaign which is usually linear it its progression, and when completed the replay value will most often be minimal. This is usually not the case with multiplayer games, as they tend to give games a prolonged lifetime, because every game match can potentially be very different depending on who you play with, the number of persons playing, the game map and other factors. Just like watching or playing football never gets old because every match is different, watching the same movie over and over will in the end become boring (depending on the movie's quality and complexity).

The social experience of gaming is taken even further with new gaming platforms such as Xbox Live and Steam. These cross the borders of specific games and connect all the gamers on the platform. Xbox LIVE is used for Microsoft's XBOX360, where players can chat, keep friend lists, initiate games with friends, earn achievements from in-game accomplishments and much more. Playstation 3 has the

Playstation Network, while there is single platform on the PC being used by all gamers. The digital game store Steam has a popular solution, but not all modern games use this. Some game developers, like the critically acclaimed Blizzard Entertainment, use their own proprietary platform *Battle.net*. Blizzard has made blockbuster games such as the *Diablo* series, the *Starcraft* games and the *Warcraft* series, including the famous *World of Warcraft* (WoW). WoW is the by far world's most popular online role-playing game, while Starcraft II is considered to be the leading strategy game. In common for all these platforms is that they enrich the gaming experience through social functions, friend lists, matchmaking, chat channels, providing extra game content through game content packs known as DLC's (DownLoadable Content) and much more.

Gamer demography

So who plays games? According to The Entertainment Software Association, an *"association exclusively dedicated to serving the business and public affairs needs of companies that publish computer and video games"*, seventy-two percent of American households play computer or video games [2]. This may seem high, but taking into account that not this is not just PCs and console games (Battlefield, World of Warcraft, FIFA), but also every little Facebook game, game on your smart phone (Wordfeud, Angry Birds) and web browser games, its more like its on the low side. Fifty-five percent of the gamers play games on their mobile phones, and forty-two percent of all gamers are women. The average gamer is 37 years old, has been playing games for 12 years and the most frequent game purchaser is 41 years old. These figures will probably help break a few myths or misconceptions for many people, and they show that gaming as entertainment has reached the large masses.

The economical impact of gaming

The gaming industry has in fact become the biggest digital entertainment industry in the world [10], and the big game releases flex muscles alongside the big movie releases, such as *Avatar* in 2010 . *Avatar* set a new record when it surpassed $1 billion in sales in just 17 days, but it took only little over a year for the game *Call of Duty: Modern Warfare 3* to beat the record by a day [11]. When its predecessor *Call of Duty: Modern Warfare 2* was released in 2009, it cashed in more money during the first week of sales than the hugely popular movies *The Dark Knight* and *Harry Potter and the Half-Blood Prince* [11]. This says a lot of the magnitude of gaming and how it has become a major player in the digital entertainment industry. Also subscription-based games with monthly fees to play like World of Warcraft are generating huge revenues. World of Warcraft peaked at 12 million paying subscribers in 2011, each paying $17 monthly to play. Even with their current subscriber base of around 10 million, this results in around $2 billion in yearly revenue. And that's recurring revenue. Table 2.1 shows the market sizes as of 2010 for the big four entertainment industries games, movies,

(a) Wolfenstein 3D (1992) (b) Half-Life (1998) (c) Call of Duty: Modern Warfare 3 (2011)

Figure 2.2: Screenshots of popular FPS games what were revolutionary in its time

music and DVD's.

Game industry	$66 bn [12]
Movie industry	$31 bn [13]
Music industry	$16 bn [14]
DVD industry	$ 4.5 bn [15]

Table 2.1: Market size comparison of various entertainment industries, 2010

2.2 Game genres

This section will present a brief overview of the most popular multiplayer game genres. These are first-person shooters (FPS), real-time strategies (RTS) and massively multiplayer online games (MMO).

2.2.1 First-person shooters

These games are named pretty descriptively from the fact that they are war games usually with projectile-based weapons, and the game world is seen from a first-person perspective. The first-person perspective aspect of FPS' is not unique to FPS's, as many other games also use this, such as the Oblivion series which are role-playing games (RPG). The genre was around already from the '70s, but the popular game Wolfenstein 3D by id Software released in 1992 has been credited with really inventing FPS games. Later classics such as *Quake*, *Doom* and *Half-Life* made further contributions to establishing the genre as one of the most popular in video gaming. Today, some of the most-popular and most-selling video games are FPS's. *Call of Duty:Modern Warfare 3* released in 2011 sold a whopping 25.3 million games, becoming one of the biggest blockbusters of video gaming ever.

Early games were pretty simple with focus on explicit gameplay and dark moods, but this changed with *Half-Life*. It contained a very strong narrative and a great plot, which made the game much like playing the lead character in a well-written movie. When it also came with a stable online gaming platform and a modifiable game engine, this catalyzed online FPS gaming. *Team Fortress Classic* and *Counter-Strike* along with dozens of other modifications or mods for *Half-Life* was released, immediately catching on and sparking online gaming. Over the next ten years, more and more FPS's with focus on multiplayer action was released, some of them being the *Battlefield* and *Call of Duty* series, which together make up the most popular FPS games of 2012. In these games, as many as 32 vs 32 players face off on huge map, also using tanks, aircraft and helicopters to combat each other. Many new game modes have also come into play, like team deathmatch, capture the flag and point control.

2.2.2 Real-time strategies

Real-time strategies are games where a player controls units and structures on a battlefield versus one or more opponents, managing their income and moving around the map in order to destroy the assets of the opponents. In most games, a player starts with just a few buildings and no units, but with the means to gather resources and build new structures which in turn provides new technologies, structure and units. These resource location are key points of a map and are fought over by all players. The nature of the resources, be it oil, dollars or crystal blue minerals varies from game to game but the notion of accumulating resources to spend on buildings and units is shared by all strategy games. To sum up, the RTS genre including base production, resource gathering, technology research, structure building and control of units.

The genre-defining moment of RTS's came with the release of *Dune II: The Building of a Dynasty* in 1992. It was the first RTS to include the use of a keyboard and mouse, which drastically improved user control precision and allowed for e.g control of single units instead of entire armies. Blizzard stood for the next chapter of the genre development as their first game in the Warcraft series was released in 1994 with *Warcraft: Orcs and Humans*. It took the structure concept a bit further, including buildings such as farms which was needed to provide food for your army, giving the notion of complete society building, not just the military aspect. When Westwood Studios' big hit *Command&Conquer* was released in 1995 including competitive multiplayer play, the genre was very similar to its 2012 version. The most notable improvements over the later years have been better in-game videos, more voice-acting, larger single-player campaigns and perhaps most importantly the game graphics.

The multiplayer aspect of a RTS is very important. Many games live long and prosperous after their release due to a thriving online gaming community. *Starcraft*, released in 1996, is still played online by people from all over the world. The reason for this is the game's perfect balance - no strategy is superior to all others. Some early RTS's were subject to bad balance, and one of them was *Command&Conquer: Red Alert*. There was one type of unit which were superior to all others in terms of fighting power

and cost, which means that this was by far the best approach to winning a game. This makes a game dull and predictable, and reduces the game's complexity. In *Starcraft*, there were three races all with distinct units and structures, and every possible game strategy had a counter. The balance of all these strategies and units were so good that no matter what a strategy a player was attempting to do, there was always a counter strategy. This made the game much more enjoyable and most importantly, replayable. *Starcraft*'s successor *Starcraft* II was released in 2010, and is perhaps the genre-leading RTS game today.

In real-time strategy, one often talks about micro- and macromanagement. Micromanagement is the precision control of units in a battle, in order to target specific enemy weak units or powerful units to kill them first, or simply maneuvering your army in a good fighting position. The best players are especially good at this, and can manage over 300 actions per minute with their keyboard and mouse in the most tense battle situations. At the highest levels of play this can be game-defining in the critical fights of a game. Macromanagement on the other hand, is the management of resources, staying on top of resource gathering and building structures. In the end, RTS's rely heavily on a player's ability to think fast, and lightning reflexes when it comes to handling a mouse and keyboard.

2.2.3 Massively multiplayer online games

Massively multiplayer online games, often short-termed MMO or MMOG, are as the name implies where a large number of players play in a shared world. While players typically play in game instances of a few to a couple of dozen players in FPS's, there can be several tens of thousands of players in a MMO server. Some games have a single, shared server for all players (*Second Life*), while other games distribute the player base over several realms, based on geographical positions and native language. For instance, *World of Warcraft* has around 241 North American and 109 European servers (realms in WoW lingo) where each realm hosts from a few thousand to tens of thousands of players.

The most popular MMO's as of 2012 are shown in table 2.2. Note that genuine subscriber numbers are hard to come by, so these numbers should be considered crude approximates. As this is crucial business information, the game companies are very reluctant to publish this information and only do so very rarely.

MMO's are also sometimes referred to as MMORPG's, or massively multiplayer online role-playing games. This has its roots from the many of the big and trend-setting MMO's such as *World of Warcraft*, *Ultima Online* and *Dark Age of Camelot*, which are role-playing games where players control a character in a huge virtual world. The characters can explore the world in any order or way they'd like, fighting monsters, doing quests, visiting towns, making friends and many more actions. Many of these also employ a level system, where a characters level represents a character avatar's power, where a certain amount of experience points are needed in order to reach the next level. These are

(a) Dune II: The building of a dynasty (1992) (b) Total Annihilation (1997)

(c) Starcraft 2 (2010)

Figure 2.3: Screenshots of popular RTS games what were revolutionary in its time

Game	Subscribers	Game company	Release year	Sub-genre
World of Warcraft	10 million	Blizzard Entertainment	2004	RPG
Aion	2.5 million	NCSoft	2008	RPG
Star Wars: The Old Republic	1.5 million	BioWare	2011	RPG
EvE Online	361k	CCP Games	2003	RPG
Rift	250k	Trion Worlds	2011	RPG
RuneScape	750k	Jagex Game Studio	2001	RPG
Second Life	800k (2011)	Linden Labs	2003	Simulator
Age of Conan	150k (2010)	Funcom	2008	RPG
Lord of the Rings Online	250k	Turbine	2007	RPG

Table 2.2: The most popular MMO's as of 2012 (mmodata.net)

characteristics of a role-playing games, however, not all MMO's are role-playing games and it would thus be incorrect to declare MMO's and MMORPG's the same thing. MMORPG's would be more correctly viewed as a subset of MMO's. As we can see in table 2.2, virtually all the most popular MMO's are MMORPG's.

A central part of any MMORPG is killing monsters over and over for experience points and loot, popularly called grinding. It's a very repetitious and boring task, still people play $17 per month to be able to do it. This is deep-seated human psychology at play, and has to do with the rewards involved and the fact that your avatar increases in power as it is done. This creates a feeling of development and success that motivates you to keep killing the next monsters, and the next, and the next. However, once automated programs to do these borings parts of the game came the market players were quick to adopt them. Now, these programs called bots are the biggest threat to MMO games and the developers are doing what they can to stop it.

This is of particular interest, because items and money in these games are being sold for real money through third-party websites. This means that cheating the game can earn you real dollars and adds a new, strong incentive to cheaters. This secondary market revolving around MMO's is huge, and is estimated to $3 billion in 2011 [3].

2.2.4 Other genres

There are many other game genres, and there are still new genres popping as a result of game innovation and mixing of existing game genres. However, the three genres presented are the most popular ones when it comes to online multiplayer games. Therefore, no other genres will be described. Here is a list of other notable genres:

- Simulation games (*SimCity*, *The Sims*)

- Adventure games
- Puzzle games (*Portal*)
- Sports games (*FIFA, Pro Evolution Soccer*)

2.3 eSports

The increase in popularity of online game the last decades has also affected what is known as eSports, namely competitive gaming. For many of the biggest games, there are online leagues that players compete in as well as large events where players come from all over the world to challenge each other. One of the most notable eSports scenes is the one of *Starcraft II*. Due to its supreme balancing, wide range of viable strategies and spectator-friendly user interface it has become a big hit. There's an all-year Starcraft II league in Korea called *GOMTV Global Starcraft II League* (GSL), several competition events throughout the USA hosted by *Major League Gaming* (MLG) as well as some competitions in Europe, and the numbers are increasing. Matches from the GSL are broadcasted worldwide on the internet, commentated by both Korean- and English-speaking commentators and it has since it's birth in 2010 given away over $1.9 million in prize money [16].

eSports has in fact become so big that many people are able to make a living from gaming, through sponsor money and income from victories in tournaments. Many of the greatest players also stream themselves practicing on streaming sites such as Twitch TV, where the watchers will from time to time be presented with commercials, generating revenue for both the site and the individual streamer. Little is known about the amount earned from this streaming, but a stunt performed by SC2 professional Dario Wünsch, he is said to have earned $2487 in 24 hours of streaming. This number is definitely much higher than normal earnings as he played for 24 hours straight, and since the revenue gained went to charity there was possibly more people watching than normally.

Avid gaming fans are not only watching tournaments and professional practice sessions online, but also in bars and pubs. The phenomenon called BarCraft has become popular, where *fans of Starcraft II* meet up over a beer watch their favorite players fight for victory in the biggest tournaments. Even here in the current hometown of the author, Trondheim in central Norway, the bar Three Lions streamed the latest MLG final and to the bar-owner's big surprise, over 150 people showed up [17].

Chapter 3

Cheating

This chapter contains a broad introduction to game cheating. It begins with an historical view on cheating and discuss the evolution of cheats into what it is today. Furthermore, an overview of cheats available for the most popular online games is presented, along with a detailed description of these cheat types.

3.1 Introduction to cheating

Wikipedia defines cheating in online games as *"an activity that modifies the game experience to give one player an advantage over others."*. The advantage obtained can vary greatly, from showing enemies through walls, disconnecting your opponent from the server, augmenting player weapon aim, changing internal game variables like player speed or similar things. This game-modifying activity can be both in terms of modifying the game code or data, but also exploiting unintentional game behavior or other activities. To get a better understanding of cheating in games, let's start with some history.

3.1.1 History of game cheating

Cheating has existed since the very first games were released. As the game developers were making games and played them for testing purposes, it was often beneficial to have cheats in order to test game mechanics and to simplify difficult obstacles or bosses. This meant that the developers added cheat codes in the game source code, allowing them to achieve benefits like unlimited lives or restarts. In the game Manic Miner from 1983, a cheat mode was enabled by typing in "6031769", which happens to

Figure 3.1: The Konami Code

be the driver's license number of the game developer. Another famous cheat code is the Konami code, present in many games made by Japanese game maker Konami. The code was enabled by pressing the game controller's buttons in a special order, which can be seen in figure 3.1. This code was also created by a developer to start a new game with all available power-ups in order to reach the more difficult levels of the game more easily while playtesting. Over one hundred games on various game consoles make use of the Konami code, including newer games on newer consoles such as the Playstation 3, Xbox360 and Nintendo Wii.

Another way of cheating was by using game cartridges or extensions, which were hardware addons that would modify the game program code or memory before the game started or during the course of the game in order to achieve various benefits. One popular goal was to set the amount of player lives to infinite, which is a simple task when you have direct access to the game memory area. Cheat cartridges have been made for most if not all gaming consoles, such as the Game Genie for older consoles like Sega Genesis, Nintendo Entertainment System (NES) and the Super Nintendo, and GameShark for newer ones like the Playstation series. The game companies were mixed in their responses to these. Sega was acceptive and gave the Game Genie it's seal of approval, while Nintendo on the other hand was strongly opposed and actually sued the developer. This temporarily stopped Game Genie sales in the U.K, but the court judged in favor of the Genie. This was probably the first occurrence of cheating in video games where the game developer was not responsible for introducing the cheats and also not supportive of the cheating taking place. Nintendo tried to thwart the cheat cartridges by adding checksums of the program code in newer games, allowing the game to verify that it had not been tampered with. This is probably one of the very first times a game company tried to stop cheating by adding anti-cheat functionality in their software on such a large scale. However, as we shall see throughout this thesis, the dynamics between cheating and anti-cheating is an arms race similar to that of the virus- and antivirus-industry. The Game Genie developers did not take long to figure out how to bypass the software checksum verification procedure, and the cycle of game cheating had completed one of its first iterations.

Modern game cheating is much more complex, but inherits a lot from the cheat cartridges. Developer-added cheat codes still exists in many games, but the lion's share of these are single-player games, or games where the cheat codes are disabled in multiplayer mode. One example of this is Half-Life and it's mods Counter-Strike and Team Fortress Classic, where typing in "sv_cheats 1" in the console enables cheat codes like "godmode" in singleplayer. The cheat cartridges edited the game code and

memory, which is still the basis for cheating today. What has changed is that the games and thus the game code has become much more complex, making the programmatic task of achieving desired cheat effects harder for the cheat creators. Another notable difference is that today's modern online games are networked and very multiplayer-oriented, allowing exploitation of the network protocols and network code. Lastly, today's situation with more game genres and game types leave many more aspects and facets of games to be cheated. Early game cheating often revolved around infinite lives or unlocking all levels, while cheaters today achieve a great variety of effects.

3.1.2 Diablo - "How do I kill a monster?"

Diablo was the first game made by the legendary game developer Blizzard, and was released in 1998. It innovated the hack-and-slash role-playing genre, where the player assumed control of a lone hero fighting his way through the dungeons beneath the town Tristram to rid the world of the Lord of Terror, Diablo himself. The gameplay was simple yet captivating and integrated multiplayer gaming in an excellent way, making way for the games to follow. The game was hugely successful, not only reaching the goal of 100 000 sold copies but making it all the way to 2.3 million, which was a huge amount in 1998.

As it was the first game in the genre with a functioning multiplayer mode, experience with networked games was limited. The developers went for a peer-to-peer solution that minimized the load on the servers, and therefore stored all character data locally on the players' computers. This meant that changing some numbers in a text file could change your characters level, attributes and all other parameters [18]. Suddenly players with insanely good items and high levels showed up in games, asking questions like "How do I attack a monster?". If player A's computer tells player B's computer that his attack damage is 5000, there is no way for player B to verify this. Taking this even further, someone figured out that they could forge a message to another player saying that "I've just attacked you with 99999999 damage" and the player would instantly die, even if the players were not in combat range of each other. This forged the term "autokilling", and was done extensively on the online servers. Someone also figured out a way to fool the protocol by making a player's computer believe that the player was not in the town area, where player killing is not allowed. Thus the term "townkilling" came about, and this in combination with the autokilling cheat made online games unplayable. This rampant cheat use on the online servers led to the breakdown of the entire *Diablo* multiplayer scene, forcing people to only play with trusted friends on a LAN, or simply use the cheats themselves. The big advantage of this architecture as opposed to the now traditional client-server model, is speed. In fact, it was so fast and efficient that they chose to keep it for the sequel *Diablo II*, by allowing both online play which was client-server based, and offline play which was peer-to-peer. This way, users could choose if they wanted speed or security, but characters from one realm could not be played on the other

This is the first occurrence of cheating in a game becoming so serious that it affects the game ecosystem as a whole. It is widely accepted within the gaming scene that *Diablo* was completely ruined by cheats.

3.1.3 WordFeud - cheating on mobile platforms

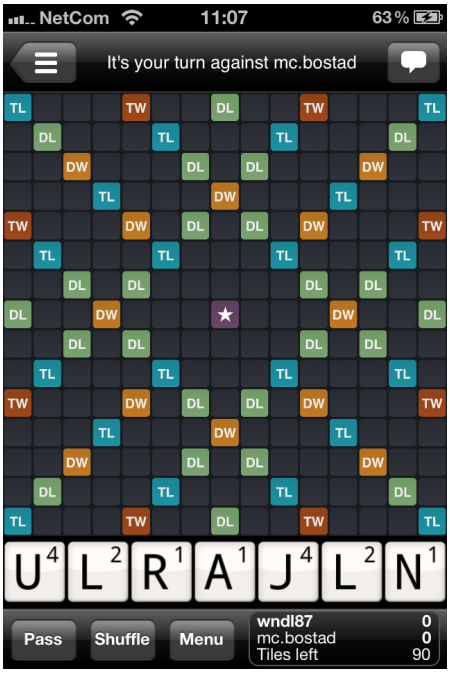
Even mobile games are not exempt from cheating. Mobile games on platforms such as Apple's iOS or the Android platform are somewhat protected in the way that the program code and memory is hidden from other programs, making it hard to fiddle with game data at run time. However, some games are prone to cheats that don't involve tampering with the game itself. In 2011, a Scrabble-like game called *WordFeud* was released in the Apple App Store, and it was a huge success. By late that year everybody was broadcasting their WordFeud-id on Facebook: *"Add me!"*. It drew the attention of the major Norwegian newspapers, and suddenly everyone with a smartphone was playing. The App Store reports over 15 million downloads by April 2012 [19].

The game is similar to *Scrabble* in the way that you create words and lay them out on a board to score points, and the board contains bonus tiles for double and triple letter points and word points. What makes *WordFeud* so popular is that the game is asynchronous - you have a full three days to respond to a opponents move. This way you could have several games with different friends running at the same time, and only use a few minutes here and there to play, waiting for the bus or for dinner to cook.

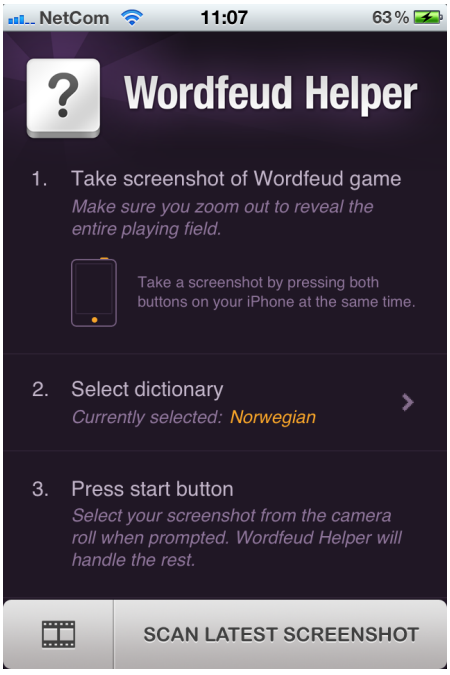
It didn't take long for someone to create an app to help you find the most valuable words to play in any given game session, and the App Store currently boasts over ten different helper applications for *WordFeud*. Most of these ask you to take a screenshot of the game board, import it in the helper application which then analyzes the board using image recognition software and calculates the possible playable word combinations and their points. One of them, Wordfeud Helper, claims to have over one million downloads. The app itself is free, but the two most valuable words are hidden when it shows you the possibilities - a premium version for NOK 14 is required for those. See Figure 3.2 for screenshots.

Is this cheating? Many of these apps are simply labelled as helper-apps, not cheats. Also, keep in mind that these apps are all approved by Apples strict regime to be let in the app store. There have been several articles on this in the Norwegian media as the use of helper applications became known. Some have proclaimed loudly that this is cheating and pathetic behavior [20], while a wordfeud guide suite online reassures that "although some may say this is cheating, its not - keep small lists of two- and three-letter words, and words starting with c, x, y etc" [21]. What's the difference between keeping lists of certain words and using a helper app, in principle? One could choose to play *WordFeud* with both players use a dictionary or a helper app, and it would be a totally different game. The focus would change from being a test of ability to recall words (often rarely used ones) starting with or containing certain letters or letter combinations, to a more strategic game. Being able to use all seven

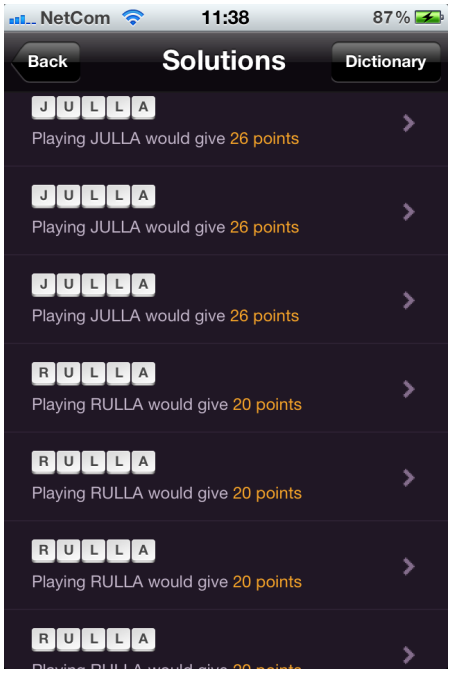
Figure 3.2: Screenshots of Wordfeud Helper, a cheat app for Wordfeud



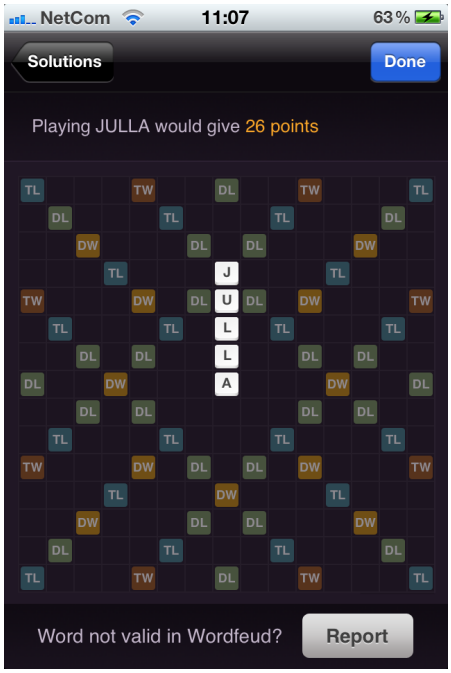
(a) The main game screen. Letters are placed in the tiles to form words, and the colored tiles are double and triple letter and word points.



(b) WordFeud Helper App main screen



(c) Word proposals from WordFeud Helper



(d) The App showing where to place a specific proposed word

letters a player has available in one word triggers a bonus of 40 points, many players choose to keep certain letters over others to maximize the probability of being able to lay a seven-letter word, even if it means playing an inferior word this round. The distribution of the game's 104 letters is also known, meaning you don't have to worry your opponent extending your "ear" into "wear" if the one w-letter per level is already played. This is an example of how cheats or game modifications may be used to create a different gaming experience if both players agree on the use before playing.

3.2 An overview of cheating types

This section presents an overview of currently available cheats for modern online games. Table 3.2 shows the games included in the research. The game genres are mostly MMOs and FPSs, with the odd games in between. Table 3.3 reveals the cheat types found for these games. Due to the large number of cheats, the cheat types are relabeled into short names, the translation to the real cheat names are found in table 3.4.

3.2.1 Cheat descriptions

This section will describe each cheat type discovered in detail. Screenshots of the cheats are included where applicable.

Aimbots

Aimbots are third-party programs that, as the name implies, aids a player in aiming the weapon in shooter games. Aiming is done by using the computer mouse, and at the higher levels of play it is done in the blink of a second. In most games, a shot in certain parts of the body like the head or the chest would produce a one-shot kill and is therefore sought after. These programs interact with the game or the video card directly to discover the location of visible enemies and in what direction a shot would have to be made in order to get a headshot of that player. When this is determined, the cheat inputs to the game the appropriate mouse movement needed to aim in the perfect direction and fires a shot.

These aimbots are usually very configurable, in terms of how fast or how accurate they are desired to be. A perfect aimbot would be easily detectable, as a player using one would be able to headshot enemies in all directions in a matter of milliseconds, which is simply not possible with human reaction times. In the early days of Counter-Strike when there were no or little anti-cheat mechanisms, a cheating player could kill you with a single shot the split second you spotted him, even if you came up on him from behind.

Wallhack

Wallhacks are programs that allow a player to either remove the textures of walls in order to see enemies through them, or make the outlines of enemies visible through walls. See figure 3.3 for a screenshot example. This wallhack renders enemies yellow even through walls - the player shown is crouching holding his weapon up to the right (the weapon not being shown). Wallhacking is possible when the server sends information about all players to the client, leaving the client with the responsibility of rendering the objects the player can see and not.

Figure 3.3: Example of a wallhack

ESP

ESP stands for extrasensory perception, and is a collective term used for describing cheats that provides the player with extra information about the playing environment. Common ESP features in cheats are information about enemy hitpoints, position, equipment etc. It is very often used in combination with a wallhack, so enemies are visible through walls. An example of ESP features is shown in figure 3.4, where all players are shown with colored rectangles and their current hitpoints below.. This is achieved because the server sends more information to the game client than what is currently revealed to the player, allowing this information to be obtained by reading the game memory or incoming game packets.

Figure 3.4: ESP cheat in the game *Battlefield 2142*

No spread

When a player in a shooter game fires a weapon, the bullets would normally spread out and hit over a small area. This spread is often dependent on the weapon used, and can be very big. This cheat disables the spread and makes all bullets follow the same trajectory. This is done by modifying the game code and removing the part of the code that randomizes the direction of the individual projectiles.

No recoil

Similarly, this cheat removes the recoil that normally appears when firing. Even the biggest of machine guns will stay perfectly still in its users arms with the cheat enabled. This is also achieved by modifying the game code.

No flash, smoke

Some games contain the game mechanic of flashbang and/or smoke grenades. Flashbangs will blind and deafen a player for a short period of time if struck, while a smoke grenade will fill an area with a non-transparent gas, clouding vision. Flashbang grenades are perfect for clearing out rooms or small, fortified areas where enemies have dug in. This cheat will disable the negative effects of these grenades. However, it is easily detectable if a player sitting on top of a flashbang grenade going off suddenly starts dropping headshots on all the players entering the room. These two cheats are grouped together as they virtually do the same thing, that is, remove negative visual effects from the game. Counter-strike has both flashbang and smoke grenades, while other games jus have one or the other. Screenshot

examples of these effects are shown in in figures 3.5 and 3.6.

Flyhack

This cheat exploits weak validation of player coordinate variables in a game and enables a player to change his z-position as well as the normal x- and y-positions. This means flying into the air, allowing the avatar to move in ways not intended by the ordinary game mechanics. This is usually exploited for transportation purposes in MMO's with very large game worlds. It is also used to reach hidden game areas that players are not supposed to reach, where the player then can attack other players with ranged weapons without them being able to hit back. This is useful in Player versus Player-areas in games that support this, such as *World of Warcraft*. The flyhack may also be used to skip parts of dungeons or quests, where an area from a later part of the dungeon is visible but unreachable, for example a cliff or an alcove in a cave, where the player may fly straight up and skip parts of the dungeon. Figure 3.7 shows a screenshot from an advertisement video of a flyhack for the game *RIFT*.

Speedhack

A speedhack alters a game characters movement speed, usually to increase the speed and thus be able to move around the map extremely fast. This is convenient for many purposes. First of all, it is also useful for transportation in large game worlds together with the flyhack. Second, it can be used to flee away from enemies when you are about to die in a MMO setting, or chase enemies that are trying to run away from you in the opposite scenario. Lastly, it can be used to reach certain areas of the game map that would not be reachable as quickly without the speedhack, hide there and be ready to ambush an unsuspecting enemy. There might also be a ton of different uses, all depending on the game.

Teleport

Teleport hacks are are closely related to the flyhack and the speedhack, but allows for instantaneous travel to other game locations by changing the player position variables. This is possible when a game does not protect the player position variables in memory or does not validate the data server-side. Using memory editors, hackers write programs to find and edit the player location memory areas.

No collision

This cheat allows a character to move through walls and objects in a game, also called wallwalking. This is especially useful in combination with a flyhack or speedhack, which makes you able to move

Figure 3.5: The smoke effect from a *Counter-Strike* smoke grenade

Figure 3.6: The flashbang effect in *Counter-Strike*, making the the entire screen white and slowly returning the brightness to normal. This picture shows the effect as it begins to wear off.

Figure 3.7: Screenshot example of a flyhack for the game *RIFT* where the game character is hovering in the air.

pretty much anywhere in the game world extremely fast. Many cheats bundle these together in the same cheat program. This is achieved by modifying the game code and bypassing or removing the collision detection checks.

Unlimited resources

In games with currencies, the player is able to bypass the code used to decrease the player resources when buying something or setting the resource variable to an infinitely high amount. This is usually achieved by modifying game memory. By using a program such as "Cheat Engine" you can first scan the memory for variables set to the same amount as your in-game resource count. Then use some resources, and scan the results from the first scan for the new amount. This leaves only two variables, which can then be set to a high value and frozen, so the game cannot change it again.

Radar

This mean enhancing the GUI by adding a map, or radar, of all the players positions. The radar interface element itself is added either in the game client by modifying the game code, or as a standalone application that is overlayed the proper game window. The information about enemy whereabouts is taken from the game memory.

Maphack

This kind of cheat can be found in games of the RTS genre, where parts of the map that the player does not currently have any units or building are displayed with a "fog of war". This fog means that the player can see the terrain layout of the map in this area, but now if there are any enemy presence. Removing the fog of war results in the player being able to see what is going on the entire map. In strategy games where map control and scouting actively to gain intelligence on your opponent is a key skill, gaining total vision of the map is extremely beneficial. Once you know what kind of strategy your enemy is employing, it is very simple to do the counter-strategy and win the game outright. This hack usually enabled by the fact that games send more information to the client than what the player can currently see, so the hackers can read the information from game memory.

Drophack

In RTS games like the Starcraft and Warcraft series, players battle in a one-on-one setting. Drophacks are hacks that manipulate the network code to send packets in a such a way that the enemy player is

dropped from the server, forcing a walkover win for yourself. In Starcraft II, there was a player called "*Pillage*" that made it to first place of the North American leaderboards with 69 straight wins and 0 losses - an impossible feat for any player. All of his matches were between one and two minutes long, ending with the enemy player being dropped.

This use of a drophack is easy to catch, and it didn't take long for the game developers to create a patch to stop the hack from working - this hack is reported patched by v1.3.3 [22]. What if some players in the absolute highest levels of play had access to a undisclosed, private drophack and used it in only every other game he was about to loose? Your opponent will only get a message he was dropped, and no signs of a drophack being used. As this happens from time to time, most players will probably just move on to start a new game. A discussion thread on *d3scene.com*, a game hacking website, indicates just this. This was posted by the user *ValiantChaos*, referring to the drophack cheater [22]:

> *Also, whoever this idiot is, you just wasted a huge f***king exploit, unless it's been an exploit for a while, because if you had actually used it intelligently,you could have gotten to GM [grandmaster league], without spamming it constantly, as you only need to have an 80-90% win ratio in masters 1v1 to have a chance to get bumped into GM either way, it will be hotfixed, hope some developers were able to take advantage before this guy leaked it.*

This illustrates the inner workings of the cheat scene, which is strikingly resemblant of the malware market. A previously unknown exploit is worth a lot of money, and is sold through the cheat vendors. This is just the same as unknown security holes in commercial software, known as zero-days, which are also being sold for large sums of money on the black market.

Bots

Bots are scripts or programs written with artificial intelligence specifically made to play a game for you. Bots can be very simple, specialized for one in-game task, or made to perform all game tasks and advance a players account or character to a desired level. They are commonly used to perform the boring or repetitious parts of a game, leaving only the fun parts for the player. Examples of this are found in pretty much every MMO, where the player would kill monsters and complete quests for experience points, in-game virtual currency and loot. Bots and the term bot originates from early FPS games from before online gaming had become as common and not everyone had access to cheap and fast broadband. Instead of playing online, people would play on a locally hosted server against computer-controlled opponents a, or robots, which spawned the bot name.

There are two general approaches to writing bots. The first is making a custom game client which interacts with the server in the way the normal client would. The other approach to botting is manipulating the original game interface solely through mouse gestures and keyboard inputs, as normal

playing would. The first approach is definitely more work as it involves reverse-engineering the game client and writing a new one. The new game client would have to communicate with the server in the exact same manner as the original, both in terms to message order and message content. Due to the the complexity of today's games, this approach is not very common. However, for some game types it is more feasible to do this, such as browser-based games. It is exactly the relatively simple communication pattern of this game that makes it a viable solution.

The other approach of manipulating the original game interface is by far the most popular because of its simplicity. The bot program would read input from the game through scanning pixel values or larger areas of the screen and using text recognition software on it. For instance, certain parts of the screen display the character's and the targeted enemy's health. A green value indicates full health, while a more red color value indicates lower health. Libraries for the large programming languages like Java and C# have methods for extracting the color value of a certain pixel, rendering this information extraction a very easy task. Similarly, there are libraries for simulating input events like key presses and mouse operations which is used for giving input to the game program. Algorithm 3.1 shows what a very simple bot could look like. This script will automatically drink a healing potion when the player health is low.

Algorithm 3.1

```
while (Game.isRunning()) {
 if (GetPixel(460, 120) == COLOR_RED) {
  IOLibrary.Keyboard.simulateKeyPresses(HEALINGPOTION_BUTTON);
 }
 sleep(500);
}
```

Many games include their own scripting language in which players can write small scripts called macros to perform simple tasks. These are used extensively in games such as *World of Warcraft*, where players create macros to simplify their gaming experience. The scripting API provided in *WoW* allows programmatical access to actions such as equipping items, sequencing spells, attacking or sending a message. Couldn't this be used to create a bot? There are two things stopping this. The first is a maximum character limit of 255. The second is that your macros are stored as a part of your character settings, which means that they will be disclosed to the game developer. This is interesting because making macros that will run unattended is considered cheating and is strictly forbidden in the gaming world. However, using small scripts to chain cast a series of spells or say *"Hello, stranger"* to a nearby player at the click of a button, is not.

Website	Games covered
AimJunkies	33
ArtificialAiming	40
Catalyst Hax	14
Call of Duty Cheats	5
DamnCheaters	29
Enhanced Aim	9
FPSCheats	11
Netcoders	9
Optimal Aim	12
Rage Hacks	3
Rent a Cheat	10
Simplex Cheats	7
TMCheats	9
Virtual Advantage	18
x22 Cheats	33
187ci	9

Table 3.1: Websites selling game cheats

3.3 Cheat distribution

Cheats are being sold to gamers through cheater websites. A list of websites can be seen in table 3.1. This is not an exhaustive list, but the result of a few days search. It is sufficient for the scope of this thesis as it shows that there are a significant number of websites selling cheats. Cheats are usually sold on a monthly subscription basis, allowing access to all cheats for a game or for all games while the subscription is active. Some sites are specializing in a game or game series, like the site *Call of Duty Cheats*. Other specialize in making a certain type of cheat, like *AimJunkies* who only make aimbots with accessories. Many of the sites have a game loader software that is used to load the games in a special way and inject the cheats undetected by the anti-cheat programs. These also check with the cheat developers' servers prior to starting the game, to verify that current cheat is still undetected.

In fact, there are so many cheat vendors online that someone has started a dedicated hack review site, where all the different actors are reviewed and rated on five categories: price, ease of use, customer service, community and detections [23]. The website is quite comprehensive, and contains reviews for 20 sites. At the time of the research, only 16 of these were online and only these are included in the table 3.1.

Review of Aim Junkies

Below is a review of the site Aim Junkies done by hackreviews.com, posted in June 2011.

Aim Junkies Revisited

Aim Junkies Revisited: By HR_Reviewer

Since my last review, Aim Junkies has done really well for themselves. They've added a new coder and offer hacks for quite a few game titles now. I'd be willing to bet that AJ will be one of the biggest hack companies in the world by this time next year. That said, I felt it was time to update my review because I feel their score has changed.

Price:

I still feel that AJ has one of the most competitive pricing models in the hack industry. With the addition of the new coder 'System Files', the price for his hack is $10.95. It's still pretty hard to complain about a price like that for one month of service.

Ease of Use:

This hasn't really changed much either in the past six months or so. The loader is still easy and simple to understand. The hack menu's are uniform and easy to use and understand. As a matter of fact the BC2 menu that I docked points for last time was updated and looks great. (Actually this did not change, I was lead to believe it had been fixed. After playing with the hack, you still must manually configure the colors in an .ini file. It's still very confusing and not at all user friendly. I am subsequently taking a 1/2 point back off of the score for this.)

Customer Service:

*Still great as always. This place really gives FPS cheats a run for it's money in this category. Response times are very low when someone has a technical problem. The chat box which is supposed to be for normal bull****ting, often turns into a support box. If you needed answers to a problem ASAP, you can always go onto their TS3 channel for instant support. There is a multinational group of tech mods ready to assist you, in any language.*

Community:

*This is a good community for the average gamer. However, the website has been dealing with a lot of first time hackers who do not know how to take a ban. This would be from the APB Reloaded group of people. They bitch and whine, and run their mouths about this and that, uugh, it's so annoying seeing that ****. Aside from that, there are lots of nice people on this board, and the support staff is nice and professional.*

Detections:

This is the reason for my re-review. Last time around this company was so new that they had not seen any bans. Since they've been around for a bit now, it's only realistic to assume there would be bans this time around. The BC2 hack has been detected twice I believe, which is still not bad at all. The main source of my angst has to do with the APB Reloaded hack. I got banned about every 9 - 10 days with it. It wasn't because I was raging on a daily basis, it wasn't because I had an abnormally high K/D, it was because the Punkbuster bypass made by system Files sucks major ass.

*Three to four bans a month is inexcusable, especially for Aim Junkies. The coder and the staff are very nonchalant about the game and have very little sympathy for people playing a free game. Their answer to a crappy bypass is simply to make a new account and to move on. Now, I don't blame them for being harsh to all the noobs who get banned using a premium account AND hacking, because that's just plain retarded. I'm going to deduct a full point for deductions just because of the lackadaisical treatment and management of the APB:R game. There is no excuse for that many detections. You need to work on a better bypass System Files and stop being so cavalier with the detection rate of this game, because it's just bull****.*

Final Score: 3.5/5

There is much to be learned from reading these reviews. It gives some insight into the world of cheaters, and what's important to them. Apparently, a price of $10.95 for a months' use for a cheat is decent. Customer service seems to be important, so when and if the cheat doesn't work or if there is a problem with configuration, it is possible to contact the people from the website and get help within a reasonable timeframe. Also, it is mentioned that *"three to four bans in a month is inexcusable"*. It also says that the website has been dealing with a lot of first time hackers who aren't experienced with getting banned, and that using the hack on a premium account (an account on which the player has used real money to buy in-game services or items) is plain stupid. This indicates that hackers does get banned from time to time, and that it's just a part of the business. We can also learn that there are bypass mechanisms for getting around the anti-cheat services protecting the games, and that this might be one of the biggest sources of error when hackers get detected and banned.

3.3.1 Economy

It is hard to find any information on how big this cheat market is, as the cheat sites do not reveal how many sales or subscribers they have. This thesis does not intend to do any economical analysis of the cheat market, but some numbers would contribute to understanding the scope of the industry. What we might possible find out, is how many people have been banned from certain games due to being detected by the anti-cheat programs. The developers of the game *All Points Bulletin:Reloaded* did just this and shared the results in a blog post [24]. They claim to have found a way to tell if a player is cheating with *"unequivocal certainty"*. This way they know the exact numbers of cheaters in their game and could estimate monthly earnings for the cheat makers. At the time of writing, there were three major cheat makers that supplied cheats for this game and the game developer estimate that they have made between $15 000 to $50 000 in revenue per month from this game alone. That is a lot of money for a small web company with only a few employees.

3.4 Results and discussion

3.4.1 Results

This section contains the results of the research on cheats and cheat availability. Table 3.2 shows the games included in the study, table 3.3 shows the cheats found for each game and table 3.4 contains a mapping for the cheat names to labels used in the big overview table. The sales or subscriber numbers of the games' listed in table 3.2 are found through web searches and the quality of the sources are debatable. These numbers must therefore be viewed as crude estimates. Still, they serve a purpose in describing the magnitude of a game.

Game selection

Only multiplayer games are being studied. Furthermore, only a subset of all available multiplayer games are included. It is not an exhaustive list of all vulnerable multiplayer games - that would also have made an interesting study, but that would be considerably more work and it is not the purpose of this thesis. The list is comprised of the most popular or most selling multiplayer games as of the latest years, in addition to some older or less popular games. These are included to illustrate the comprehensiveness of cheat availability. Other games could just as well have been used, but these were chosen for this study. The MMO's included are the ones with the highest subscriber numbers according to *mmodata.net*.

Game	Release year	Genre	Sales / Subscribers
Age of Empires Online	2011	MMO	Freemium
Aion	2009	MMO	2.5M
America's Army 3	2002	FPS	9M
All Points Bulletin: Reloaded	2010	FPS	Freemium
Battlefield 2142	2006	FPS	13M
Battlefield: Bad Company 2	2010	FPS	
Battlefield 3	2011	FPS	13M
Call of Duty: Black Ops	2010	FPS	24.5M
Call of Duty: Modern Warfare 2	2009	FPS	22M
Call of Duty: Modern Warfare 3	2011	FPS	25M
Counter-Strike	1999	FPS	25M
Darkfall	2009	MMO	-
EvE Online	2003	MMO	400K
Guild Wars	2005	MMO	6M
Heroes of Newerth	2010	DOTA	1M
Homefront	2011	FPS	2.6M
League of Legends	2009	DOTA	1.4M
Left 4 Dead	2008	FPS	11M
Left 4 Dead 2	2009	FPS	
PlanetSide	2003	MMO/FPS	-
Rift	2011	MMO	-
Star Wars: The Old Republic	2011	MMO	1.5M
Starcraft 2	2010	RTS	4M
Team Fortress 2	2007	FPS	3M
World of Warcraft	2004	MMO	10M

Table 3.2: Games included in the background study

Game	Genre	Bot	Ab	Wh	E	Ns	Nr	Nfs	Fh	Sh	Nc	T	Ur	Ra	Mh	Dh
League of Legends	DOTA															
Heroes of Newerth	DOTA															
Battlefield 2142	FPS		x		x	x	x									
Battlefield 3	FPS		x		x	x	x							x		
Battlefield: Bad Company 2	FPS		x		x	x	x									
Call of Duty: Black Ops	FPS		x		x	x	x									
Call of Duty: Modern Warfare 2	FPS		x	x	x	x	x	x		x						
Call of Duty: Modern Warfare 3	FPS		x	x	x		x	x								
Counter-Strike	FPS		x	x	x		x	x					x	x		
Homefront	FPS		x		x							x		x		
Left 4 Dead	FPS		x	x	x	x	x	x								
Left 4 Dead 2	FPS		x	x	x	x	x	x								
Team Fortress 2	FPS		x	x	x	x	x			x						
Age of Empires Online	MMO									x			x	x	x	
Aion	MMO	x														
Darkfall	MMO		x	x	x					x		x		x		
EvE Online	MMO	x														
Guild Wars	MMO															
Rift	MMO	x							x	x	x	x				
Star Wars: The Old Republic	MMO	x														
World of Warcraft	MMO	x														
PlanetSide	MMOFPS															
Starcraft 2	RTS														x	x

Table 3.3: Cheat availability overview, see table 3.4 for cheat types

Bot	Bot software
Ab	Aimbot
Wh	Wallhack
E	ESP
Ns	No spread
Nr	No recoil
Nfs	No flash, smoke
Fh	Flyhack
Sh	Speedhack
Nc	No collision
T	Teleport
Ur	Unlimited resources
Ra	Radar
Mh	Maphack
Dh	Drophack

Table 3.4: Cheat type map

3.4.2 Discussion

The most evident finding in this study of cheats is that thats there's a clear pattern of what cheat types are applicable to which game types. There exists aimbots and wallhacks for all FPS's, while bots are common in every single MMO. A cheat is by definition a program or an exploit of game procedure in order to gain a benefit over others, and this is of course dependent on the nature of the game. Since so many games are very similar to each other due to being of the same game genre, the applicable cheats will also be the same.

Public vs private cheats

There are usually two types of cheats. Public cheats that are made by a cheat developer and then sold to the masses, either as a single cheat or through a subscription. The second type is private cheats, made by gamers with above average programming skills that intend to use them themselves, and therefore do not disclose them, in order to prevent detection and patching of the exploit.

Theres a big market for cheats

There are very few available figures concerning the revenues generated by cheat sale, the number of people involved or simply how many people use cheats. However, the amount of websites dedicated to selling the cheats imply that cheating is happening on a grand scale, and that selling cheats is a big business.

Cheats have a due-date

As public cheats typically reach a broad audience, it's impossible to stop the game developers from learning about them. This means that cheats will follow the traditional cycle found in the (anti-)virus industry - a cheat would come out, sell to a lot of users, the game developers patches the game, a period of silence follow until someone manages to update the cheat and make it work again. Repeat.

Cheat visibility

The visibility of a cheat greatly affects its influence on a game economy. Extensive use of cheats like aimbots and wallhacks in an FPS is very visible, and will in most cases reveal the cheater to the other players in the game. When you are playing fair and square and keep getting blasted by a cheater, you will become fed up before long and either look for another server or somewhere to get your own cheats.

Other cheats like the drophack for *Starcraft II* or bots for various MMOs are less visible. If you get dropped in a tense moment in a battle of *Starcraft II*, there's no chance to know if you were the victim of a drophack or if there simply was a problem with your PC or internet connection. Disconnects does happen from time to time and as long as it's like that, it's hard to suspect your opponent of drophacking except if you're repeatedly dropped against the same opponent every time you are about to win the game. In the higher leagues of play, you are often matched against only a few dozens of players so meeting the same player happens pretty often.

Cheat-free games?

The only games in this study where no cheats or hacks were found are *Heroes of Newerth* (HoN) and *League of Legends* (LoL). The special thing about these games is that almost the entire game is run server-side [25]. On a game hacker forum, it's stated that the game is near impossible to hack because of this. The game client is very thin and only sends keyboard presses and mouse actions to the servers, and gets information from the server in return about everything that happens as a result. This way, it is impossible for a hacker to modify the game code or variables, as it is not accessible.

Chapter 4

Anti-cheating

Anti-cheating is the effort done by the game companies to try to stop cheating. Game security wasn't much of a issue before the late '90s, when games like *Diablo* were deeply troubled by cheaters in online multiplayer. Still, it wasn't before the huge success of *Counter-Strike* around 2001 that the first anti-cheats saw the light of day. Ever since, they have been fighting hackers worldwide who seek to hone their hacking skills on the domain of games. This chapter will present an overview of the anti-cheat industry, in terms of what software exists today and what games are covered by it.

Limitations

This chapter is somewhat limited because the anti-cheating companies are very reluctant to reveal information about their software. Both information regarding number of bans and the mechanisms used to detect cheaters are being withheld. There are some general information available and that information is summarized in this chapter. The original thesis intention was to delve deeper into the workings of anti-cheats, but due to this lack of information more importance was given to the case study instead.

4.1 Anti-cheat mechanisms

Anti-cheat software may employ different mechanisms in order to prevent cheating. The anti-cheat companies reveal very little of the inner workings of their software. Descriptions of the anti-cheat mechanisms that is publicly known follow.

File checksums: checksums are calculated of all critical game files and sent to the server prior to

joining a game, typically by using the MD5-algorithm. The server will have a list of all the correct checksum values and reject a player from joining if there is a mismatch. This way, a cheater won't be able to modify the game code before joining a game. After the checksum has been calculated, the file will be kept in an open state, so that no further changes to the file is possible while the game is running. This method is also used to detect older game versions in some games, automatically forcing a game update if an outdated client is trying to join a game server.

Process monitoring: Most game hacks will run as separate processes, and the anti-cheat programs will scan the running processes on the computer for known hacks. Some also scan all DLLs associated with a process, and the code inside the processes.

Memory scanning: Similarly, anti-cheat programs will scan the computer memory for known hacks, also in real-time while the game is playing.

Dynamic memory addressing: Many hacks are based on modifying game variables directly in memory. To do this, they first need to locate the position of the desired variable in the game's memory area. This make this harder, some games randomly move the important parts of the game data around in memory. This can be done either after each edit or at fixed time intervals. A common hacker approach to finding the memory address of a variable is to first search for the value of the desired variable in the memory, then change the value in the game by buying a weapon, changing team or similar and then doing a new search. This method is effectively stopped by dynamic memory addressing.

Ban lists: All anti-cheat software maintain lists of player ID's of the players that have been caught cheating, and bans them from the game. Both player ID's associated with the game purchase and signatures derived from the computer's hardware can be banned. The ban is in most cases permanent, and the anti-cheat companies are very determined at enforcing this rule. In some cases, a user can get banned from all games that are governed by a particular anti-cheat [26].

4.2 Current anti-cheat software

Wikipedia and a Google search reveals a large number of anti-cheat software in use or in development as shown in table 4.1. For many of these, all that can be found is an old wikipedia link that goes to a page that does no longer, exist or similar traces of previous existence on the internet. The DMW Anticheat has published on its webpage that they will no longer be selling subscriptions as of April 30th 2012. UCP Anticheat and EasyAntiCheat only support games that are already supported by the big Valve Anti-Cheat, and EasyAntiCheat hasn't been updated since 2011.

Anti-cheat software	Games protected	Release year
Valve Anti-cheat	**50**	**2002**
PunkBuster	**23**	**2001**
The Warden	**5**	**2004**
GameGuard	13	2002
Virtual Shield	unknown	-
DMW Anticheat	8	-
Codename Helsinki Anti-cheat	1	-
HackShield	8	2001
EasyAntiCheat	4	2006
UCP Anticheat	8	2007

Table 4.1: List of anti-cheat software

The conclusion is that the only active, functioning anti-cheat programs with continuous updates and support is the big three: Valve Anti-Cheat, PunkBuster and The Warden.

4.2.1 Valve Anti-Cheat

Valve Anti-Cheat (VAC) is the anti-cheat software of Valve Software, the company behind games like *Half-Life*, *Portal*, *Team Fortress* and *Left 4 Dead*. VAC is bundled with their online gaming platform Steam. Steam is estimated to have 70% share of the digital distribution market for games, and has 1504 games and 40 million active user accounts as of January 2012 [27]. Steam is used not only for buying and downloading games, but also as a social gaming network with user groups, friend lists, in-game voice and chat functionality. Many major game publishers have a big game catalogue available on Steam, including Bethesda Softworks (*Skyrim*), Activision (*Call of Duty* series), Rockstar Games (*Grand Theft Auto* series), Electronic Arts (*Battlefield* series), Square Enix, 2K games and Sega. Without fear of over exaggerating - Steam is a huge player in the game industry.

VAC was first released for *Counter-Strike* in 2002. *Counter-Strike* was at the time an extremely popular game, and cheats for the game started to emerge. Valve doesn't talk much about the VAC, but in 2006 they announced that "the new VAC technology", revealing a VAC 2.0, had caught over 10 000 cheaters in the previous week alone. Unofficial sources state that as many as 1.73 million Steam accounts have been banned by VAC to date [28]. As VAC is a component of the Steam platform and allows game developers to incorporate it in their games, VAC has the most comprehensive list of supported games of all the anti-cheat efforts.

When VAC detects a cheat, the account is flagged as cheating but no action is taken before after a certain delay. This is common practice in most anti-cheat programs, and prevents the cheaters from knowing exactly which cheat was detected. This is crucial as cheaters could just probe with different

cheats to see what cheat mechanism VAC detects and not. When banned, VAC usually only bans the player's Steam ID for that specific game. The exception is games running on the same game engine, in which case the player is banned from all games using that engine.

4.2.2 PunkBuster

PunkBuster is the second biggest anti-cheat initiative, and was actually the first one around. PunkBuster was started to fight cheaters in *Team Fortress Classic*, after some bad experiences by its founder Tony Ray. As *Team Fortress Classic* was a so-called mod of the original *Half-Life*, PunkBuster was suddenly the de facto anti-cheat service for all the games in the *Half-Life* family, like *Counter-Strike* and *Day of Defeat*. When Valve decided they wanted control of the fight against cheats themselves and started development of its own system, PunkBuster was replaced by VAC. Now, the most notable PunkBuster-protected games are the Battlefield game series.

4.2.3 The Warden

The Warden is the proprietary anti-cheat software made by Blizzard Entertainment for use in their own games. Blizzard is behind the popular game series of Diablo, Starcraft and Warcraft, which includes the game World of Warcraft. The Warden was first discovered a year after the release of World of Warcraft, but little was initially known about how it worked. Legitimate players had no problems with the fact that Blizzard had anti-cheat software running on players' machines along with the game client - quite the contrary, legitimate players benefit from this. Nobody likes to compete against hackers and cheaters for virtual currency and items. It was also reported to shield against rootkits, keyloggers and trojans, which is also more than welcome if you have no anti-virus software. All in all, a win-win situation.

A software security researcher, Greg Hoglund, disassembled the code of the Warden and discovered that it did more than just scan the game's data in the computer memory - it scanned the entire memory. Hoglund said the following in this blog (which unfortunately is not available anymore):

> *"I watched The Warden sniff down the e-mail addresses of people I was communicating with on MSN, the URL of several websites that I had open at the time, and the names of all my running programs."*

The Warden performed thorough investigations of your computer every fifteen seconds and sent results back to Blizzard if it found anything suspicious, according to Hoglund. As a result, the Electronic Frontier Foundation and other organization labeled the Warden as spyware [29]. Hoglund wrote and

publicized a program he called "The Governor" that monitored everything the Warden sent back to Blizzard. According to [29], Blizzard replied with the following arguments:

1. Warden does not collect any personal information, so what's the problem?
2. Everybody else is doing it as well, pointing to other game companies.
3. Read the EULA. Blizzard representative John Lagrave stated "People should read contracts" [30].

Concerning 1), we just have to take Blizzard words on that. The newer versions of the Warden are reported to send data encrypted to Blizzard, meaning that it is no longer possible to monitor what information goes out of your network adapter. The reasoning of argument two is rather easy to tear apart, considering the saying containing all your friends and a bridge, etc. Lastly, is installing a rootkit on your PC ok as long as it is mentioned in a hundred-page EULA?

This split the World of Warcraft community. Many gamers were supportive of the anti-cheat mechanism and accepted the "nothing to hide, nothing to worry about"-argument. Others saw it as a huge intrusion of privacy, and either started closing all other applications when playing or stopped playing altogether.

4.3 Game coverage

This subsection contains information about which games are covered by which anti-cheat services. Figure 4.1 portrays this information as a graphic representation of the game coverage.

VAC is clearly the largest anti-cheat service in terms of games covered, with PB as number two. However, the magnitude or importance of the different games covered differs greatly. The two biggest game franchises among these are the *Call of Duty*-series and the *Battlefield*-series, covered by VAC and PB respectively. VAC is also bundled with the world's biggest digital distributor for games, Steam, which account for 70% of the gaming market. This makes it much easier for a game developer to choose VAC over PB, and may be the main reason for the size difference of the two.

The Warden is different from the two others in the way that other game companies may not use it for their games. Still, it's the anti-cheat used by perhaps the most recognized game developer of them all in some of the best game franchises in the world.

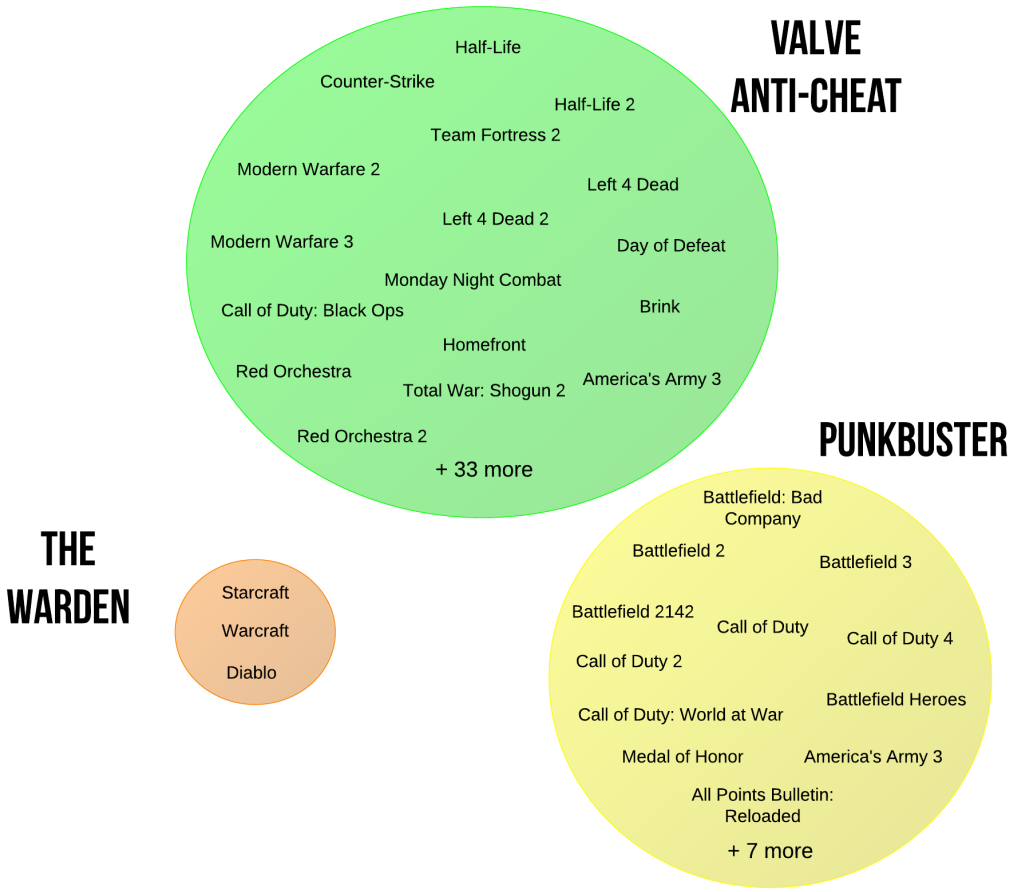


Figure 4.1: Figure showing game coverage of the anti-cheat actors Valve Anti-Cheat, PunkBuster and The Warden

Chapter 5

Bots in Browser-Based Multiplayer Games

This chapter describes how to make a bot for a browser-based game by running JavaScript code from the browser console window. Two bots for a specific, non-disclosed game are presented and their performance is analyzed.

5.1 Introduction and background

The game used in the case study is a browser-based MMO. Two self-developed bots have been used to cheat this game. The identity of the game is not disclosed, this is to avoid exposing the game's weaknesses to the public. Therefore, the game is described in a very general manner while still still revealing enough for the reader to understand the game genre and the usefulness of the bot. This target game for the case study is hereby just referred to as the game. This chapter describes the game, the bot development methodology and the results obtained in detail.

5.1.1 Introduction to browser games

So, what is a browser game? The name says it all - it's a game played in your web browser. They can be built with only the pure web technologies HTML, CSS and JavaScript, using browser plug-ins such as Flash or Java, or with new web graphics technologies such as SVG or WebGL. Browser-games come in all genres and types, both single-player and multi-player. Many websites specialize in offering a wide

range of browser-based games, such as *www.kongregate.com* and *www.badgegames.com*. However, most of these games are based on the Flash-plugin and are collectively called flash-games in gamer lingo, rather than browser games. They are played in a browser, but many use the term browser-games only for a certain type of browser games: text-based strategy MMO's.

Planetarion was the one of first of these games. It was released in 2000 and was made by Norwegian company Fifth Season. The game is still online, figure 5.1 shows a screenshot of the game interface as of 2012. With the introduction of HTML and new web graphic technologies, what people think of as browser-based games and browser-based MMO's is slowly changing as implementations of 3D games are being made in the web browser. As of 2012, there are several graphical equivalents of *World of Warcraft* and other MMO's that runs in the browser. In this context, the term browser-based multiplayer games will be used for the original definition of text-based strategy war games.

5.1.2 The game

The game is a hybrid between a RTS and a MMO, borrowing concepts from both genres. Players control bases in large, shared worlds of up to 50 000 players, building structures, researching technologies and controlling battle units. Like in an MMO, players continues where they left off each login but time keeps ticking in the game world also while offline. This means that building constructions, unit productions and researches will continue and the player may be attacked. In terms of base-building and economy management, the game is similar to a RTS, where the strategical part is crucial, spending your resources wisely and being able to defend off attacks and maintain your income. The game is on the other hand more similar to an MMO due to the fact that there are several thousand players on each server. Unique to this genre is that most game actions take rather long to complete. While a building is built in a matter of seconds in *Starcraft II*, construction work in this game can take up to 6 hours. Unit production can take as long as 18 hours, depending on the unit type. Travel times to other players for attacking and defending are also fairly long, from about an hour to several days depending on the distance between the bases.

The buildings

There are many different structures in the game, each having a distinct role or function in the game. Some of them are resource production buildings which contribute to hourly income while others enable production of certain troops or construction of other buildings. Other building functions include providing a means to gather intel on other players' bases, improving the base defense value or enabling resource trading with other players. All buildings start at level one when first constructed, and are upgradable to a certain maximum level. It is the sum of all building levels that constitute the points of a player. At account creation, a player is worth a few hundred points while a fully upgraded city is in the tens of thousands.

Figure 5.1: A screenshot of Planetarion, one of the first big browser-based MMO's

Demand time	Yield per demand	Yield per hour
5 minute	30	360
20 minute	70	210
90 minute	155	103
240 minute	300	75

Table 5.1: Resource farm returns by demand interval

The units

There are many types of battle units in the game, each with different statistics. All units have one attack type and strength value, and similarly, a defense value towards each of the attack types. This makes some units better than others at defending against some other units, some units better than attacking than defending and some units very specialized for certain tasks.

Research

The game has a wide range of researchable technologies, but a player can't research all of them and has to choose which ones to get. This creates many different possible strategies, and creates depth to the gameplay. Some possible research benefits are increased fighting power, shorter travel times and the possibility of taking over other players' bases by force.

Income

There are two main way of income in the game - the resource production buildings and farming. The resource production buildings produce a certain amount of resources every hour based on their level, from 8 resources at level one to 1050 resources at maximum level. This source of income is independent of a player's activity level, unlike the other type: resource farms.

Each base starts with access to one resource farm (RF), and can obtain a maximum of eight. A RF will upon demand supply a player with resources, but you can't demand from it again until after a certain time. The player can choose between four time intervals, with more resources being granted for longer time intervals. These intervals are 5 minutes, 20 minutes, 90 minutes and 240 minutes. Shorter demand intervals always give more resources per minute over time, so if you are going to be online for at least twenty minutes, there's a much greater yield to be had with four 5-minute demands rather than one 20-minute demand. This rewards active players, giving them much more resources than players who only log in a few times a day. Table 5.1 contains an overview over faming yields per hour for each type of demand for a farm of level 1.

The freemium model

The game uses the freemium business model, which means that the game itself is free but you may

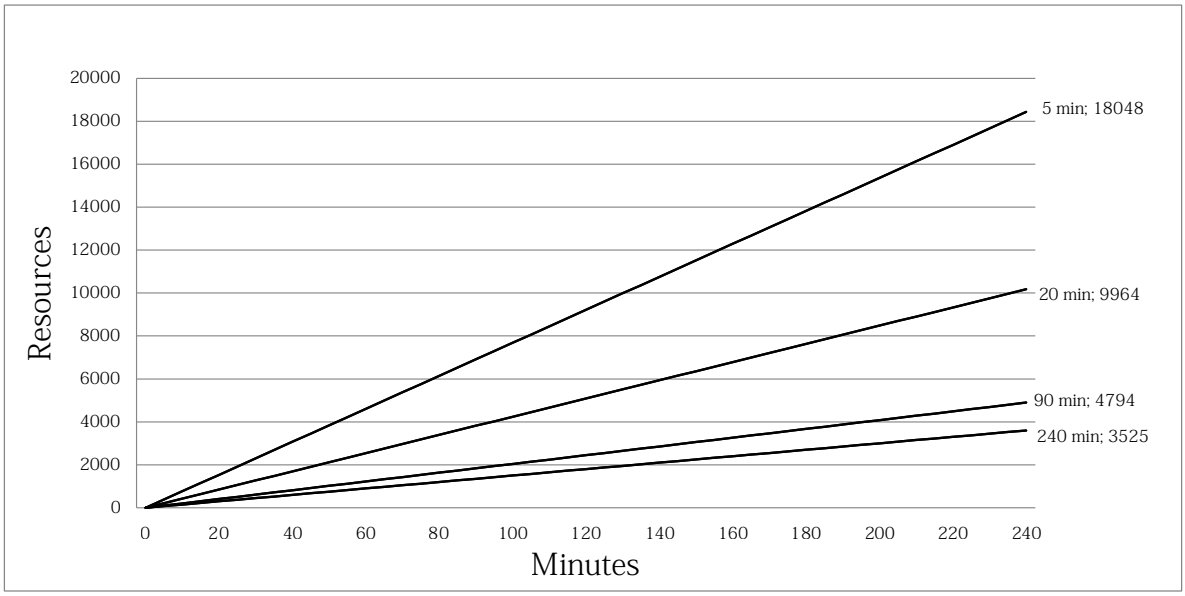


Figure 5.2: Resource yield per time unit for different demand intervals for resource farms.

subscribe to a premium version that costs real money but gives you benefits over the free version. The freemium business model is increasingly popular and is being used on relatively new but very popular web services such as Spotify, DropBox and Flickr. This business model is very new, but has increased in popularity in recent years due to the amount of software services targeted for the mainstream audience that has emerged. Freemium is especially suitable for software services as the marginal cost for selling the service is zero, and the free users serve as a marketing channel. For more information on freemium business models, read [31]. This model serves well for browser-games, where curious gamers try the game out, and when some get hooked, they end up paying for the premium service.

The battle system

There are three motivations to attack other players. Firstly, you will loot the target's stored resources. Second, you gain kill points for every unit killed, which is needed in order to be able to have control over more bases. Thirdly and most importantly, two successful attacks on an enemy base will force a base takeover if the required technology is researched. The battle system is entirely mathematical, and involves no player skill. The result is calculated based on the attackers' and defenders' attack and defense values, and a series of modifiers. Some examples of these are researchable technologies, base defense (a structure), premium features and a simple luck modifier (-30% to +30% fighting power) for each battle.

5.1.3 Methodology

This section will describe in detail the approach used to create and run the bot. It will contain detailed code examples of the scripts that were used, and a guide on how to replicate the procedure. Our cheat for this game is based on automating tasks, and creating bot code for the game that will perform the dirty work for us. This means that we will need to manipulate the game GUI in some way. As the game GUI is completely contained inside a web browser, let's start by introducing the browser tools used.

The web browser

Like all browser-games or games in general, some code will have to be run client-side. The game is based solely on pure web technologies, which is HTML for semantic layout (markup), CSS for styling and JavaScript (JS) for dynamic pages and interactivity. JavaScript makes up the dynamic, interactive web pages that is the game GUI, and all the JS code and functions is available in the browser console. All browsers have a developer console, where the user may inspect the markup of the page, the different stylesheets, script files, http requests and responses and much more. The author has been using the Google Chrome browser for this thesis, where the console is opened by Ctrl+Shift+J (Cmd+Alt+J on my mac).

Several tabs of the console are handy. First, the Elements-tab is used for inspecting the HTML of the page, and discovering the ID-s of HTML container elements that we wish to manipulate. Second, the Scripts-tab is used for inspecting the scripts that are either included in the web page directly or scripts that are being linked to remotely. Lastly, the console tab is where we all the code is inserted and executed. Before we start with code examples, we will need to present the JS-framework used: jQuery.

jQuery - the "do more, write less"-library for JavaScript

JavaScript is in many ways a controversial programming language, with its dynamic and weakly-typed styles. It's easy to misuse the bad parts of JS and get yourself in a lot of code problems as a programmer. However, many frameworks and libraries for JS have been released the last years, making it easier to use and adds some functionality. Some of these include MooTools, Scriptaculo.us, Prototype and jQuery. Especially jQuery has become extremely popular since its release in 2006 and it is the most popular JS-library today, being used in over 55% of the 10 000 most visited websites [32]. jQuery will be used primarily for it's simple and elegant way of selecting HTML elements, and some added functionality like iterating through groups of HTML elements. Note that all code written would be completely possible to do without jQuery - it simply allows for doing more with less code.

Manipulating the game GUI using mouse clicks

Manipulating the GUI normally means simulating key presses and mouse gestures through system calls in a ordinary desktop game, but most of the game navigation in this game is based on mouse actions. The forum and messaging systems will of course require keyboard input, as will menus where the user inputs the amount of troops to build or similar actions, but other than that there's no keyboard input required. We'll start off by presenting how to select a HTML element using jQuery in algorithm 5.1

Algorithm 5.1 Selecting HTML elements using jQuery

```
$(selector);

$('body');

$('div#main_container div.dataRow:first');
```

The dollar sign is a factory method for the jQuery object, which means that $ returns the jQuery object to which we will pass arguments. This argument is a jQuery selector string, which determines which HTML elements are returned. In the first example a string named selector is passed to the jQuery selector function. In the second we are being returned the HTML object "body", which is the main container for all content on the page except for meta information and such, which is in the header part of the page. In the last line we are selecting the first element of type div that has class *"dataRow"* and is the first child of a div with id of *"main_container"*. For more information on jQuery selectors, look up the jQuery documentation.

For simulating mouse clicks, we use a jQuery function *trigger(eventType)* to trigger an event of type "click" on the selected HTML element, see algorithm 5.2.

This will trigger a mouse click event just as if it was a real mouse click performed by human operation of the mouse. We can also make these happen at specified time by using the JS function *setTimeout(function, time)*, as shown in algorithm 5.3.

Algorithm 5.2 Simulating a mouse click on a HTML element using jQuery

```
$('div#retrieve_resources_button').trigger('click');
```

Algorithm 5.3 Delaying a mouse click

```
setTimeout(function () {
        $('div#retrieve_resources_button').trigger('click');
}, 10000);
```

Algorithm 5.4 Opening the construction

```
GameLayout.gameWindow.open('construction_menu');
```

Manipulating the game GUI programmatically

When a click event is fired on the specified HTML object, a JavaScript function is run that adds or changes an element on the web page. Often, these functions can be called directly from the console instead triggering them to be called by the event listeners. For instance, a certain area of the game screen will open the construction window where you can build or upgrade structures. The very same menu can be summoned by entering the code in algorithm 5.4 in the console..

By combining this with *setTimeout* and simulating a mouse click on the "Build"-button for a certain building, we have effectively created a bot that builds a certain building at a certain time. We could also easily extract the information about our current resources, our hourly resource income and the resource requirements to build the specified building to calculate the wait time in milliseconds needed in order to afford it. Here we are starting to see the outline of our bot. The script can be written in a separate text editor and copy-pasted into the console window to be run.

5.2 The farmbot

The first bot was created for use with developed accounts. The first stage of the game for a new player involves mostly base-building and preparing for what is to come, and many feel that the real game begins when a player can build units, fight other players and take over other's bases. The purpose of this bot is to automate the resource collection procedure, which drastically improves the daily income rate. The bot would farm all the resource farms in all bases every fifth minute, and the bot would run for as long as the pc was on. The author made sure to do non-bot activities in the game every now and then, like checking the alliance forum, sending messages or building something. This could mean running the bot for 8-10 hours per day, while doing school work, making dinner or doing housework at home and of course while just wasting time on the internet. Doing this continuos farming with the bot eliminated almost all effort involved, while doing it manually would require full attention for at least 30 seconds every five minutes. Without the bot, farming every five minutes would not be an

Algorithm 5.5 Example of code to add hotkey

```
$('body').keydown(function (event) { // BARRACKS hotkey
 if (!typingWindowOpen()) { // disable if messages/forum is open
  if (event.which == 66) { // B-key
   GameLayout.gameWindow.open('construction_menu');
  }
 }
});
```

option and the alternative would be to do longer farming intervals less often, like every one and a half hour, yielding far less resources over time. Figure 5.2 shows the difference in income when using long or short farming intervals. As we can see, the difference is dramatic.

With this, it was possible to create units much faster than the opponents, while still spending money on base building. It would be entirely possible to do the same without the bot, but it would require a much larger effort. Therefore, it is less likely that someone would get suspicious of the cheating, since it's far within the reach of possible human play. This is totally different to some other cheat forms, like aimbots, speedhacks or similar - these are simply not possible without modifying the game mechanics. Other players could be doing the same but it would require spending most of day playing, while the bot use reduced that dramatically.

Supplementary functionality

Some supplementary functionality was added along with the bot - the possibility of delaying attacks and keyboard shortcuts. It is desirable to attack opponents during the night, when they are not online to ask their friends for support or send away their battle-inept units. This was done very simply by using timeout-functions native to JavaScript to delay a simulated click on the attack button a certain amount of milliseconds. This was very helpful and was used extensively to attack opponents by surprise in the middle of the night, often catching offensive units in their bases which would fall very easily. A typical attack would be scripted to be initiated around three or four o'clock in the morning.

Adding keyboard shortcuts made it drastically easier to navigate around the game menus. The game is entirely mouse-based and has no keyboard hotkeys. Using a keyboard is much more effective than using the mouse, so keyboard listeners were added to the *<body>*-tag of the web page to open various menus or windows depending on what button was pressed. These hotkeys reduces the work required to play and thus allows you to do more in less time. In a game where time use is a major resource, this is important. Hotkeys were added for all the major menus in the game, allowing game interface navigation almost exclusively with the keyboard. One of the most helpful hotkeys were to add one to cycle through bases, significantly speeding up the process of building and producing units in all bases.

The effects of the bot and script use can be seen in figure 5.3. The information is gathered through a third-party website that collects player information about every player on every server in the game. Data is collected once per day from available API's from the game company's websites. Data was collected for the ten largest and most active players in the alliance the author played in, which was ranked 4th on the server. The reason there is not more than ten data sources is because there's a considerable effort involved in gathering the data. Also, the players considered are the most active ones and therefore an estimate of the fastest growing players in the world. As we can see in the graph, the game account using cheats has a considerable growth. It started as the smallest one in the data set, climbed one rank around March 25th, crossed the average line around April 2nd and became the largest player in the alliance around April 15th.

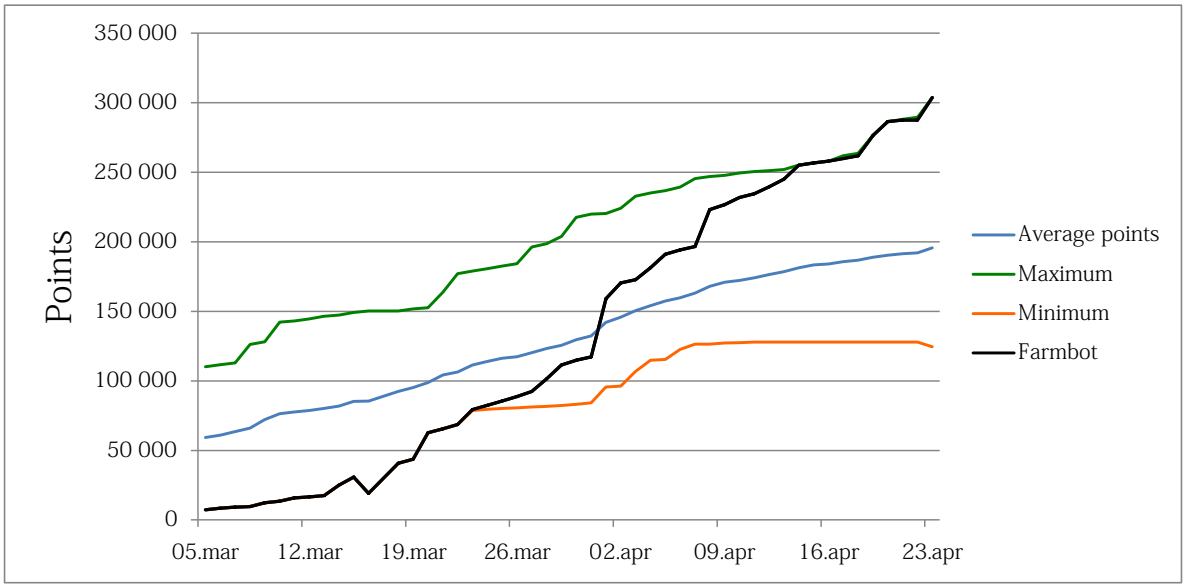


Figure 5.3: Player growth of ten most active players in the alliance

Analyzing the effects of the bot

In a RTS game the most important skill is the ability to think and assess a situation quickly, think strategically and lightning mouse reflexes. In a FPS game the latter is of most importance. In this game, the single most important factor is being active. If somebody attacks you while you are not online, you won't be able to ask alliance members for help or send away units stationed in the attacked city that are not suited for defensive duties. The resource collection mechanism also strongly favors active players. This becomes only more important as a player grows and controls more bases, as each additional base will involve more work in terms of building the appropriate structures to suit your base

plan, producing units, farming and more.

The bot helps in all these aspects. The farming script will farm automatically, eliminating the effort involved. Automatic event-triggered actions can cover much of the problems of being logged off for longer periods, like making a script to send away any offensive units in a city or post a pre-made support request in the alliance forum when being attacked. When it comes to the cumbersome work of managing a large number of bases, the improved GUI with hotkeys come in and significantly reducing the time spent.

5.3 The buildbot

The next step was to extend the bot to also perform automatic construction of buildings. In the early game phase, building and farming are the most important aspect of the game. Until a player develops his base to a certain level, he is unable to participate in fights, defend his alliance members, or capture enemy bases. Therefore, the first days of playing a new game server is a boring task of just building up your base. This is a simple task that the bot is extremely adept at,building and upgrading structures and farming. To measure the performance of such a bot in the early stage of the game compared to a human play style, three new game accounts were set up and played in parallel while logging account points. The goal was to construct and upgrade the required set of buildings to the levels where one can research the technology that allows you to take over other player's bases. This is used because it is a game-changing moment in the game for any player, and because it takes at least a few weeks to get there. Having such a long time span is needed to properly show the differences between the accounts. The experiment is very time-consuming as it requires full attention for up to an hour every day. This is the main reason it was not run longer, for example until the base had been fully developed, as this will take substantially longer.

Bot operation

The bot had a list of target building levels for all buildings in the game. Each time a construction was finished, it would issue a build order at random on a building that had not yet reached its target level. It would then read the remaining build time of all current queued orders, sleep for that amount of time and call the building procedure anew. This would repeat until the bot had upgraded all buildings to their target levels and the bot was done. After each build order placement, the script would also produce units. In order to make sure there would be enough money for the next build order, the bot would calculate the estimated earnings based on the time left until the current building operation was completed and the resources earned in the last farming operation. If the sum of the estimated earnings and the current resource levels exceeded the maximum storage capacity, these excess resources would

be used for producing a random defensive unit.

While the bot is farming, placing a build order or producing units, a boolean variable would be set to ensure that the bot would not start a second procedure while another was running. The function called while another was running would then call itself with a 10 second delay to wait. Without this, the bot could potentially be doing two things at one time and send messages to the server with very small time differences. Another possible outcome could be that one of the procedure would fail, as the game has some delay after each click before the menus are updated.

Practical execution

The accounts were to be played in a casual manner, just like the average player would. To emulate this, the accounts would be played for up to 30 minutes, three times each day. These times would be in the morning, at typical past-work time at around 16:00 and finally in the evening before going to bed. As the purpose of this was to compare the time required to develop a base to a certain level, a limited amount of time is needed each login in order to the required tasks of farming and building. Being logged on for longer periods is more beneficial in the later stages of the game when one has several bases and many units.

For each login the current points of each account was logged in a spreadsheet on Google Docs, along with the exact time of the login. Resource production buildings were prioritized in the beginning, to establish a large resource income. This was especially important in the non-botted accounts, as these would have significantly lower income than the botted account that farmed non-stop. Later, the high-cost buildings with long building times were prioritized in order to reach the goal as fast as possible.

5.3.1 Results and discussion

The results are visualized in figure 5.4. The fastest growing curve is the bot, while the second one is the premium account and the last one is the non-premium account. The bot and the premium account both grow significantly faster than the last one. This is quite understandable, as this one could only place two build orders in queue per login, while the premium one could place seven and the bot would continually place orders, even when the PC was unattended. When all buildings reach a certain level, the build time to upgrade to the next level gets so high that placing seven build orders would keep the base busy until the next login, even over night. When the premium account reached this point it was a problem that farming all RFs for 240-minute intervals wouldn't give enough resources to fill the building queue. Normally, only three or four orders could be issued, and this would mean a few hours right before the next login that the base wouldn't be building or upgrading something. This is

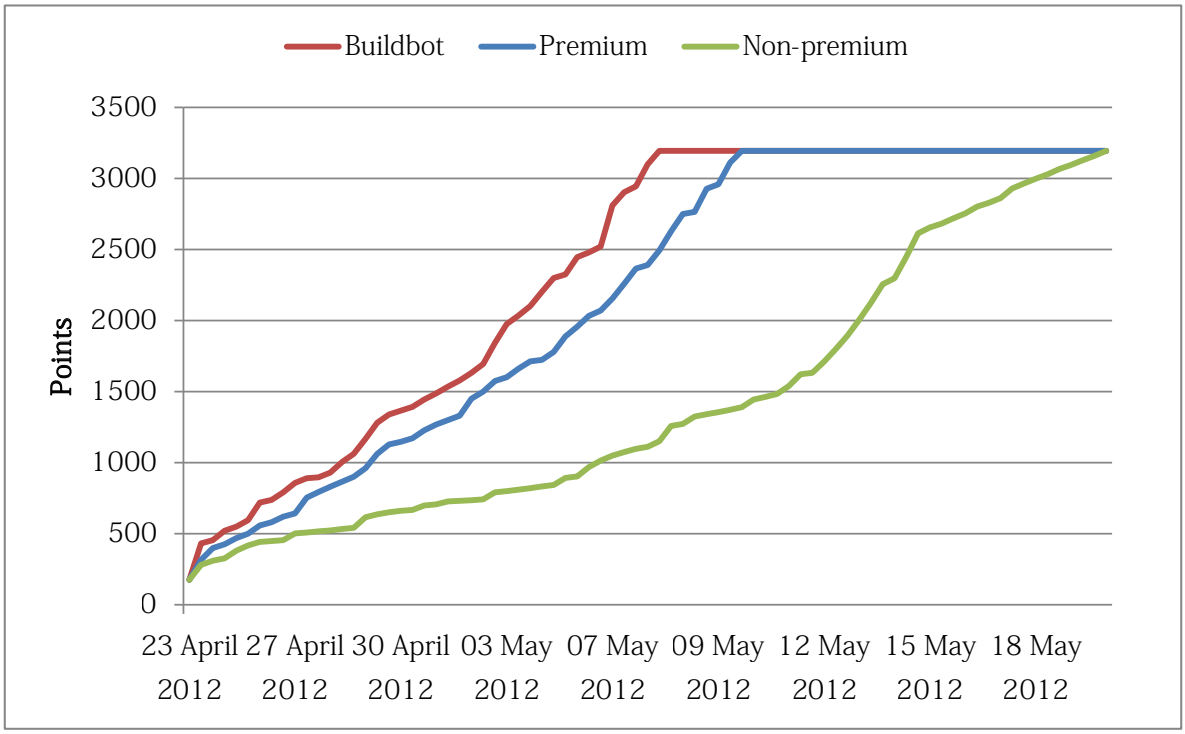


Figure 5.4: Player growth in points, botted, premium and non-premium accounts

essentially the difference between the bot and the premium account performance-wise. Of course, the premium account had to be played which was at least 15 minutes worth every login, while the bot account would only rarely need to be looked after.

Another difference is that the bot would afford considerably more units because of the automatic farming. It was doing 5-minute demands nonstop, while the other accounts could only use the 240-minute demands every login. The non-bot accounts could barely afford the building upgrades, while the bot was stinking rich. As the bases become more developed, they became takeover targets for other players on the server. A base with low defenses and few units is a juicy target, while a base with a lot of defensive units is ignored.

The most important result from this test is that the bot wins over a premium account, and the bot will run unattended while the premium account needs to be played.

5.4 Probing for bot detection

As the list of scripts grew, a fear of detection and getting caught by the game company grew similarly. Therefore, an experiment was conducted to attempt to discover if the game company had any countermeasures in play to detect botting. In order to this, an aggressive version of the bot was run on a spare computer in the office. This computer had a browser open twenty-four seven on a separate game account, to avoid losing the main account if the experiment was detected.

The bot would farm continually every five minutes throughout the day even if the storage capacity was full or the daily resource limit of the farm was reached. A design goal of this script was to be as simple, machine-like and detectable as possible, there was no point to add any smartness to it. The bot would check every twenty minutes if there were any possible build orders to issue and pick one at random if there was. No matter if a build order actually was placed or not, the bot would wait twenty minutes and repeat. This in combination with continuos farming made sure that there were always something being constructed as new orders would be placed in queue while one was in progress, and that there would always be enough money to place build orders.

The bot was started on February 29th. The computer was checked several times daily to make sure the script was running, and that there were no other problems. In addition to checks during the day, remote connections were made at nighttime to make sure the bot would run during the night. It was monitored this closely to make sure that the footprint of the bot would be as large as possible so as to maximize the chance of triggering any detection mechanisms.

5.4.1 Results and discussion

The bot was run continuously for about three weeks. At this point it had around 4000 points but had no units due to the nature of the bot. This meant it was a juicy target and ended up being captured by another player. Still, the bot had been running for over 20 days and did not receive any attention from the game company. As the bot was active every single hour of the day for this many days, they can not be monitoring this variable or else it would surely trigger some alerts in their detection system. Similarly, the bot would farm every five minutes throughout the day which means that they're not monitoring that either. It could be that they're monitoring total resource gain per day, which is low in this case since the bot only controlled a single base.

5.5 Discussion

The results clearly shows that using the cheat is far superior to normal play, through both base-building in the early stage of the game and farming also in the later game stages . The early game play is especially suitable for the bot, in which it outperforms the premium account even without using premium itself. As a player becomes larger in points and number of bases, the benefits of using the bot fades a bit due to the game focus changing from base building to battling other players. In a war scenario, the challenges a player faces are to plan the attacks well, launch attacks at the correct time and dodge enemy attacks. While my bot is more adept in the early game play, it could easily be extended to help deal with some of the later challenges in the game. The reason for not creating a more advanced bot as a part of this thesis is time restraints - further development of the bot would not contribute much to the point being made. Browser games are easily cheated with simple programming. However, an outline of possible further development of the bot is found in section 7.2.

5.5.1 Bot implementation

A standalone bot for the game has been developed by a dutch university student named Jos Demmers. His bot is made entirely in C#, and runs as a desktop application on your pc. This approach to botting is the first one mentioned in subsection 3.2.1, where the programmer implements his very own game client and imitates the network protocol between the real client and the game server. Demmers claims his bot to have about 5000 users per month. This bot has been developed over the course of two years, and has much more functionality and a more user-friendly interface than the one presented here. It is also easier to run for longer periods of time, as the application can be set to a run on startup of your OS and automatically start after a crash. The JS-bots will need to be inputted to the console to start them, and they will not restart themselves after a browser or system failure. However, the standalone bot has some problems when it comes to detection.

The bot use was never detected by the game company. On his blog, Demmers has received many comments from his bot users who have gotten banned. Compared to the total number of users they are rare, but it does happen. When so many people use the same software, it will become easier for the game company to discover patterns and anomalies and thus detect the bot users. The resulting comment threads from the banned users conclude that they've used the farming function too much, or set up the client incorrectly so that it will send too many misformed requests to the server. Another problem is that the client will not update its communication protocol automatically when there is a new game version out. This results in bans when a message type is deprecated and no longer used in the new version, but all the bot users keep sending these to the server. Playing through the original website would never generate the old requests, but the standalone bot does and this reveals it immediately.

The approach presented here does not have to deal with this as it is only manipulating the GUI of the real game and thus sending up-to-date HTTP messages.

5.5.2 Replacing premium benefits

Most premium benefits are hard to obtain without buying them the legitimate way. Benefits like 20% increased resource production or fighting power are stored server-side and the client checks with the server that the player in question has the required premium services before using them. However, some of them can be achieved through using scripts. These are listed along below with a description of how do obtain them technically.

1. **Increased building queue:** One premium feature is an increased building queue of seven, instead of the normal two. With the bot, the queue is virtually infinite as we can schedule the bot to add new orders as they are finishing.

2. **In-game overviews:** One premium feature is some interface menus which show the current situation in all bases with regard to resources, production and incoming attacks. A big player will have control over many bases, and keeping track of all of these is time-consuming without these premium overviews. All the information displayed is available to the user by clicking through your bases, but also programmatically by calling the client side javascript functions in the console. By copying the HTML from the overviews, we can add the data ourselves by calling the appropriate game code functions and toggle the overview by adding a hotkey.

3. **Farming overviews:** One premium feature is a simplified interface to farming the resource farmings. Each base has many resource farms, and a big player has many bases. This premium feature adds an overview that displays all farms belonging to a base on one page, and adds a button to farm all farms with just one click. We can't do the same here as with the previous item in this bulletlist, because this button fires a special message primitive that is validated server-side. If the player doesn't have the premium feature, it won't work. However, using the farming bot, the effort of farming is removed nonetheless and the premium feature is unnecessary.

This is especially interesting in regard to the economical impact of cheating. This browser-game is a business, and exists to generate revenues for their developers. As there are no advertising in the game, the only way they make money is by selling the premium service. When significant parts of the premium service is easily obtainable without actually buying it, it is quite serious. What if a hacker with bad intents wishes to hurt the game company, and publishes scripts like these on the web? For all we know, many players are already doing this, causing a large loss for the game developer.

5.5.3 Bot detection mechanisms

The aggressive, ban-seeking bot did not succeed in making the game developers ban it, but this doesn't necessarily means that there are no detection mechanisms in play. Botting is the main cheat problem in MMO's and games with repetitious, boring tasks and it is highly unlikely that the game developer is unwary of the threat. They have several other browser games in their backlog and will no doubt have met the problem of bots before. This means that they either detected the bot but chose to not act on it, or that they simply did not detect it. Let's discuss each of these possibilities.

Bot detection is an automated process - it is simply too costly to have someone do this manually. Most likely, the system will monitor some parameters for suspicious behavior and flag accounts that trigger these. Then, flagged accounts will either be automatically banned or sent to support for manual inspection. Even though the game developers have total control over which users they will allow and not and that they may ban whoever they want with no warning, it's not good for business to be banning innocent players. Most players will have invested a lot of time in the game and would be furious to be banned due to a malfunctioning detection system. Thus, the game company should be really sure of the violation before they take action against a player. This will require human inspection of the violated account before banning, which is costly compared to automated system. This may imply that the game company concentrate their efforts on the most serious cheaters, leaving the small fish to cheat as they wish. As the account using the buildbot was small in points, not a part of any alliance and certainly not a game-defining player on the server, this may be the reason it was not detected. On the other hand, the main account had a lot of points, was ranked top 65 and was certainly a big player. It did not use the building bot but it used the farming bot extensively as well as the other scripts and was not detected. The main account would most likely face repercussions if detected, being such a significant player on it's server.

To recap, this means that the farming bot was not detected and the main bot were *probably* not detected. Let's say that they weren't. This means that they could not have triggered the filters that are being used to flag suspicious accounts. Since the bot was being active in the game throughout the day, every day, the game can't possibly be monitoring account activity times for being in a certain range. No player will play the game for that many hours every single day without pause - everyone has to sleep. The bot also has a very machine-like playing pattern, mostly doing in-game actions at intervals of whole seconds. If the time between actions were monitored, it is possible that the bot would be identifiable using this.

Chapter 6

A Bot Detection System for Browser-based Multiplayer Games

This chapter presents a Bot Detection System (BDS) for browser-based multiplayer games (BBMG's). The BDS is general and is applicable to all BBMG's. The goal of the BDS is to detect the use of bots in BBMG. The proposed system is easily automatable and can be used to flag suspicious game accounts, which may in turn be inspected manually.

The chapter is structured as follows. Section 6.1 starts with a general description of the BDS. Section 6.2 applies the proposed BDS on the bots from chapter 5, before section 6.3 wraps up the chapter with a discussion of the BDS.

6.1 General model

The BDS wishes to reveal players using bots by identifying a set of attributes and comparing these to average human behavior. A score system ranging from 0 to 100 is introduced, where the average human behavior is defined as 0, while increasing non-human behavior is given a score > 0, with max 100.

First, three categories of player behavior are defined in the set CAT as follows.

$$CAT = \{PA, GV, NT\} \tag{6.1}$$

1. Player Activity (PA). This category contains attributes regarding the playing pattern of a player,

such as login times, session lengths, or total daily playtime.

2. Game variables (GV). This category contains attributes regarding in-game measurements or properties, such as experience points or resources. These will vary greatly from game to game, but most games employs some of sort resource used to build, buy or upgrade, whether its minerals, crystals, resources or just good old dollars. Role-playing games will usually have a character level system, where experience points are needed to increase a character's level. The main incentive for a bot in many games is increased resource production or experience point gain, often resulting in expectionally high values.

3. Network Traffic (NT). This category contains attributes regarding the in-game behavior of a player, that is, what happens when a player is logged in. This includes analysis of the network traffic from the client to the server, as all in-game actions will generate an outgoing HTTP request. From this the time between each game action, the types of game actions or the order of game actions can be analyzed.

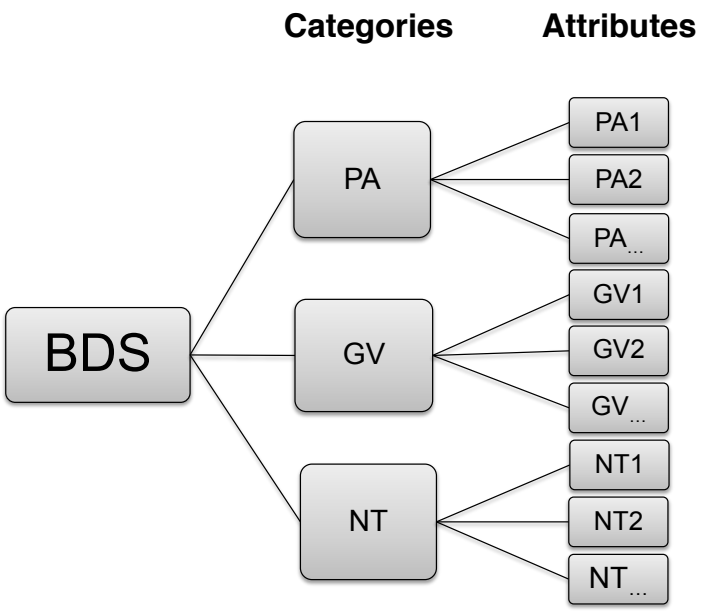


Figure 6.1: Categories and attributes

We then define A as the set of all attributes as follows:

$$\begin{aligned} A &= \{A_{PA}, A_{GV}, A_{NT}\} \\ &= \{PA_1, PA_2, PA_{...}, GV_1, GV_2, GV_{...}, \\ &= NT_1, NT_2, NT_{...}\} \end{aligned} \tag{6.2}$$

The number of attributes and number of attributes in a given category are defined as $|A|$ and $|A_{CAT}|$, respectively. We define a player p_i with $|A|$ associated game variables as follows:

$$p_i = \{x_{j,i}\}, \forall_{j \in A} \tag{6.3}$$

For each attribute j in set A, a human average h is computed by using m samples from human players:

$$\forall_{j \in A} : h_j = f(x_{j,1}, x_{j,2}, x_{j,...}, x_{j,m}) \tag{6.4}$$

An attribute score for attribute A for player p_k is $a_{A,k}$. It will have a value in $[0, 100]$ where 0 represents human average behavior and 100 is as bot-like as possible. In order to detect whether a certain player p_k is using a bot or not, we calculate the attribute score $a_{A,k}$ for every attribute.

$$a_{j,k} = f(h, x_{j,k}) \tag{6.5}$$

Depending on the nature of the attribute, a suitable attribute function $f(h, x)$ will need to be chosen. Linear, exponential or lognormal functions work well for this purpose. Examples of attribute functions are given later in this chapter. The function must be scaled and possibly shifted to only return non-zero values for the appropriate deviation values. It is crucial to have sufficient amounts of player data in order to learn what deviation value range is standard human behavior and what is bot behavior. Attributes may differ in importance in various BBMG, and this can be accounted for by factoring in a weight factor $\mu_j \in [0, \rightarrow]$ for each attribute j. The sum of all weights in a category CAT is μ_{CAT}.

Finally, all attribute scores in each category are summed and normalized to find the category scores S_{CAT}. These scores are in turn summed and normalized to obtain the final score $S_{TOTAL} \in [0, 100]$. This will be the variable that a game company's automatic system will monitor for values exceeding a fixed threshold. Accounts with too high S_{TOTAL}-values are automatically flagged and scheduled for manual inspection.

$$S_{CAT} = \frac{1}{|A_{CAT}|\mu_{CAT}} \sum_{j \in A_{CAT}} \mu_j a_j \tag{6.6}$$

$$S_{TOTAL} = \frac{S_{PA} + S_{GV} + S_{NT}}{3} \tag{6.7}$$

Variable	Description
x	Direct gaming variable
a	Attribute score, in [0,100]
h	Human average value for a certain attribute
μ	Weight factor
A	The set of attributes
CAT	The set of categories
A_{CAT}	The set of attributes in category CAT
PA	Category: Player Activity
GV	Category: Game variables
NT	Category: Network Traffic
CAT_1	Attribute i in category CAT, e.g. PA_1
p_i	Player i

Table 6.1: Terminology overview

6.2 Example BDS implementation

In this section, the proposed BDS is implemented on the game and bots from chapter 5. Table 6.2 shows the attributes used.

Category	**Variable**
Player Activity	Number of logins
	Aggregated session length
Game Variables	Resource gain
	Times farmed
Network Traffic	Request interarrival times
	Request type distribution
	Request order

Table 6.2: Attributes used in example BDS implementation

To make the approximation of what is normal, human game behavior, data was gathered from six players. The players involved are people the author has played with during the research work done for chapter 5. They were instructed to set up Wireshark using a certain capture filter, to record only outgoing HTTP packets with destination IP that of the game server. The instructions the participants were given can be seen in Appendix B. They had Wireshark running while playing for some days until they had captured at least 2000 packets, and then mailed back the Wireshark capture files. The same method was done to record data for the two bots. Table 6.3 shows the amount of data captured for each player and bot. Ideally, data should be collected for a significantly larger number of players, but will suffice for the sake of demonstrating the proposed BDS. A game company using the framework will obviously have access to a very large amount of data.

no	minutes recorded	packets recorded
#1	954	4346
#2	260	3935
#3	827	2795
#4	583	2215
#5	1 429	10253
#6	592	2407
Farmbot	3753	4448
Buildbot	13 318	16702

Table 6.3: Data basis for

6.2.1 Player activity

6.2.1.1 PA_1 - Number of logins

The first attribute in the Player Activity category is the number of logins done daily. There are two reasons to monitor game logins. The first is that some less sophisticated bots will try to reconnect continuously if they are not configured properly, resulting in a high number login attempts in a short time interval. This would never happen for a player, making it an obvious give-away for a bot. The other reason is that to reset the scripts that have been injected to the console, the browser window must be refreshed, resulting in a new login request. These logins also stand out, as the game has been played actively up until the point of the new login unlike normal logins that come after a period of not playing. There is no point in refreshing the browser when playing, so it will normally only happen when someone accidentally hits the refresh button, or, when someone wants to clean the time-driven injected scripts.

To attempt to model the human average in this regard, data was collected from seven players in the game by simply asking them for estimated values. These are of course very crude and might be far from accurate, but still serve purpose in the pursuit of an human average. No player reported more than four logins per day, and the average was around three. Based on this data, a negative exponential function was chosen as attribute function - this is shown in figure 6.2 and formula 6.8. The function is shifted by five which is the mean plus three standard deviations of our data set. The exponent is divided by a scaling factor to get a less steep curve. This is done to avoid having the function reach the maximum value of one too fast, so it is still possible to separate values that are both very high but not close together.

$$a_{PA_1} = 1 - e^{\frac{-(x_{PA_1}-5)}{20}} \tag{6.8}$$

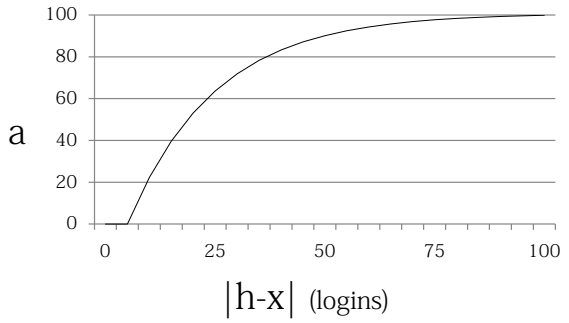


Figure 6.2: Attribute function for PA_1

i	Daily average logins	$\frac{h-x}{\sigma}$	a_i
1	3	0.1	0
2	4	0.8	0
3	3.5	0.4	0
4	1.5	1.4	0
5	5	1.7	0
6	2.5	0.5	0
(7)	2	1.0	0
Average	3.07	-	-
Standard deviation	1.12	-	-
Farmbot	15	10.7	39
Buildbot	<1	0	0

Table 6.4: Results for PA_1

The results of the calculations are shown in table 6.5. Only the farmbot obtained a non-zero attribute score, with its daily login rate of about 15. The farmbot has this login rate because the game window was refreshed when the script was to be stopped temporarily. The reasons for stopping the script could be many, for example that all bases have maximum resources, or that it is inconvenient to have the farming procedure starting every five minute in tense situations and thus blocking gameplay. The buildbot on the other hand, ran for several days on just one login, which is also suspicious. This results in a daily login count between 0 and 1, and this is hard to automatically label as suspicious behavior because this would also include all players who doesn't play on daily basis. Therefore, this anomaly of very long game sessions is detected in the next attribute - a_{PA2} .

6.2.1.2 PA_2 - Aggregated session length

The other attribute in the Player Activity category is the amount of time a player is logged on. Each time a player logs in, a new session ID is used. A timeout system is employed, so a player will get logged out after a certain time period of inactivity. The game company can use this to monitor the time a player is logged in. It is similar to the time a person is playing, but this is slightly incorrect because timeout period is unknown and it may be as much as an hour. Still, if the timeout period is an hour, being logged in for more than 16 hours means that there is less than 8 hours remaining that the player can be sleeping. Most people will have school or work, and not be anywhere close to playing this much but some groups of people may be able to play a lot, for example people who are retired, in jail, unemployed or on various welfare services. Thus, aggregated login times of over 16 hours is very unlikely and certainly over time - everyone has to sleep.

A shifted negative exponential function is used also here as the attribute function, as seen in figure 6.3 and formula 6.9. This formula is shifted by 16 and divided by a scaling factor 2 for the same reasons as in PA1.

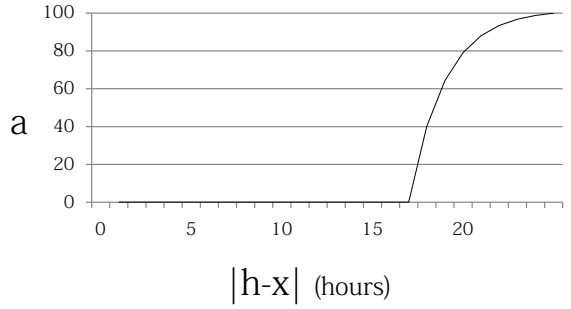


Figure 6.3: Attribute function for PA_2

$$a_{PA_2} = 1 - e^{-\frac{(x_{PA_2} - 16)}{2}} \tag{6.9}$$

The results of the calculations are shown in table 6.5. Again, data was gathered from seven players and these are their own recollections from memory and are in no way accurate, but acts as a general estimate. The buildbot stands out with its twenty-four seven operation and correspondingly receives an attribute score of 100.

i	Aggregated session length (hours)	a_i
1	14	0
2	9	0
3	9	0
4	12	0
5	15	0
6	7.5	0
(7)	3.5	0
Average	11.7	0
Standard deviation	5.5	0
Farmbot	14	0
Buildbot	24	100

Table 6.5: Results for PA_2

6.2.2 Game variables

In this game, the most interesting variable to monitor for suspicious activity is resources. Resources are used for many purposes in the game, and limits the rate of base expansion and battle ability through unit production. It is also one of the prime targets of botting, by automating the task of farming the resource generation rate is significantly increased. Two attributes will be used to cover the resource monitoring - the daily resource gain and times farmed. These two are both required in order to catch all suspicious behaviors. This is because a small player can farm continuously all day and still not have a particularly high income, due to being a small player with few bases and few resource farms. A large player with tens of bases and even more farms would have a larger daily income than the small player just by making one resource demand. Therefore, the amount of times farmed also has to be monitored.

6.2.2.1 GV_1 - Daily resource income

The data available shows that most people farm rarely, with long demand intervals. One 8-hour demand before bed and three 4-hour demands during the day is a very popular variant. This is a good start to where the attribute function should begin to kick in. In order to earn more than this, one has to do shorter demands more often. The two persons that farm most often is still within one standard deviation of the mean income, so the attribute function is shifted three standard deviations from the mean also for this attribute. The attribute function is yet again negative exponential, shown in figure 6.4 and formula 6.10. To achieve an income high enough to get much response from the attribute function one must do 5-minute demands for a very long period of time, in addition to long demands at night and in the morning. In other words, you have to have exceptional endurance or be a machine

i	Daily average income ($x_{GV_1,i}$)	$\frac{h-x}{\sigma}$	$a_{GV_1,i}$	Times farmed ($x_{GV_2,i}$)	$\frac{h-x}{\sigma}$	$a_{GV_2,i}$
1	21 532	1.4	0	11	1.0	0
2	21 350	1.3	0	17	2.4	0
3	13 720	-1.2	0	7	0.1	0
4	17 962	0.2	0	4	0.5	0
5	17 962	0.2	0	4	0.5	0
6	19 236	0.6	0	3	0.7	0
7	17 962	0.2	0	4	0.5	0
8	14 252	1.1	0	5	0.3	0
9	12 824	1.5	0	2	1.0	0
Average	17 422	0.9	-	17 422	0.9	-
Standard deviation	3 007	0.53	-	3 007	0.53	-
Farmbot	60480	14.3	74	60480	14.3	92
Buildbot	20790	1.1	0	20790	1.1	95

Table 6.6: Results for GV_1 and GV_2

to earn this much in a day.

$$a_{GV_1} = 1 - e^{1\frac{-(x_{GV_1}+\mu+3\sigma)}{10}} \tag{6.10}$$

The results are shown in table 6.6. As players can have a different amount of bases, the numbers presented are per base. The highest possible daily income for one base is around 120 000 resources,

All players go under the radar while the farmbot with its 60 000 resources per day, a whopping 14 standard deviations from the mean yields an attribute score of 74.

6.2.2.2 GV_2 - Times farmed

Monitoring the daily income does only catch the big fish, namely those with many bases. A small player could farm just as often and use a bot, but not earn as much. Resources are even more scarce in the early game stages because a player is constantly upgrading his base while producing troops. Bigger players will reach a stage where all their bases are fully upgraded and troop production is the only resource sink. Therefore, it is important to monitor also the number of times demands are made from resource farms.

Using data from the same players as in the previous attribute, a shifted, negative exponential attribute function is found appropriate. The shift is also here equal to three standard deviations from the mean. The attribute function is seen in figure 6.5 and formula 6.11.

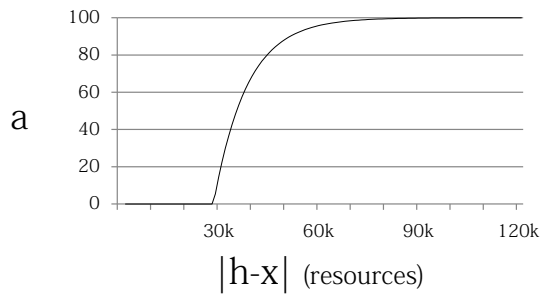


Figure 6.4: Attribute function for GV_1

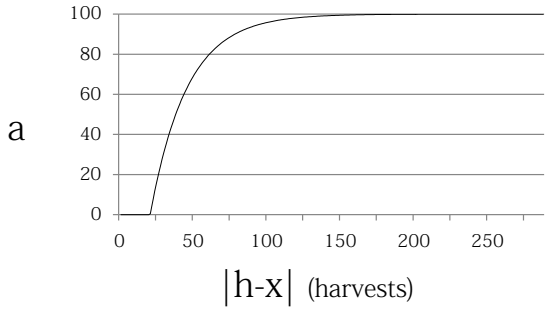


Figure 6.5: Attribute function for GV_2

$$a_{GV_2} = 1 - e^{1\frac{-(x_{GV_2}+\mu+3\sigma)}{25}} \tag{6.11}$$

It is possible to do a total of 288 5-minute demands each day, if absolutely no time is lost between each demand. This is very hard, so the real maximum is probably a little lower. Still, this would imply spending up to a minute every five minutes throughout a whole day, which no sane person would do for a game.

6.2.3 Network traffic analysis

In this category, the outgoing HTTP packets to the game server are analyzed. These are sent each time an action is performed in the game, such as opening a game menu, placing a build order, producing units, selecting a base or moving around the map. The packets are captured by using Wireshark, a free software tool that allows recording of all network packets going through the co2mputers network interface card (NIC). By using a particular capture filter, we set Wireshark to only record HTTP packets having a certain destination IP (the game server). This allows us to capture only the packets we are interesting in. These capture files are then exported in .csv format and opened in Excel for calculations and statistical analysis. We are looking at three aspects of the outgoing packets: the time between each request, the request types and the request orders.

6.2.3.1 NT_1 - Request interarrival times

By analyzing the time between each outgoing game packet, some interesting statistics can be found. Here, the mean time, the variance and standard deviation between requests can be used. These reveal significant differences in the play styles of the human participants and the bots - see figure 6.7. The

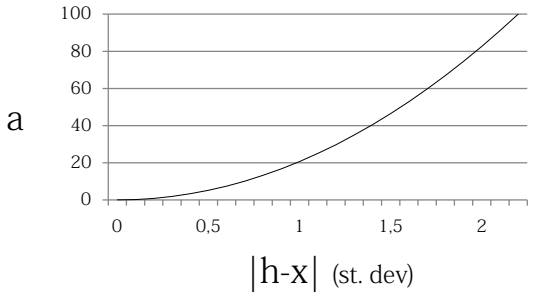


Figure 6.6: Attribute function for NT_1

human average has decreasing relative frequencies for higher time intervals, with a significant peak in [0, 0.15] of 18%. The bots does not have this long tail at all, the highest value for both being around 2 seconds. The bots themselves also have very different distributions, where the farmbot has values in all pillars from 0 to 2 and the main bot is concentrated mostly about 0.8 seconds. This shows big differences in play styles, from a natural, human-like distribution to a very mechanical distribution for the buildbot. These results stem from the delays that are built in the bots' source code, where they wait a certain amount of time between each action. The difference between them is while they both farm, the farmbot uses the premium function and the buildbot does not. This creates differences in the sequence of actions, where some actions require a longer delay than others.

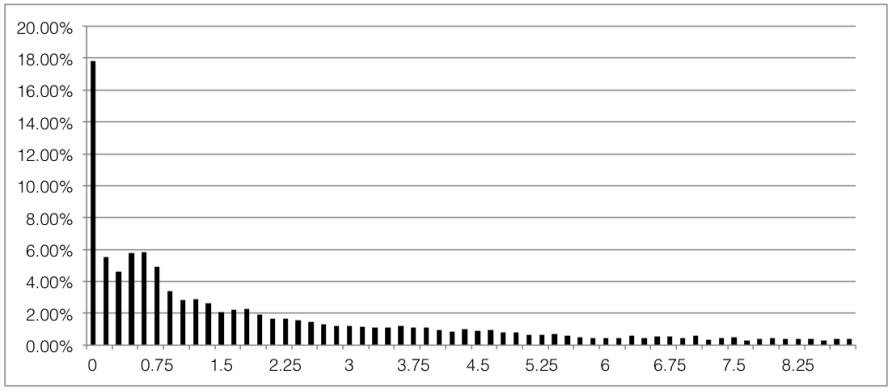
Only requests with interarrival times under 9 seconds are included in this analysis. This is to be able to have a fine granularity in the low intervals to reveal time patterns of the bots. The time distribution of longer intervals can also be interesting, but is not used here because there are already large differences between humans and bot in the short time intervals.

It is easily seen from the graphs in figure 6.7 that there are clear differences between the human average and the bots. To measure this numerically, the standard deviation is used. This is because the standard deviation is a very suitable measure to catch the low spread of values. Table 6.7 shows the results of these computations. Formula 6.12 is used to calculate the attribute score a_i from the standard deviations.

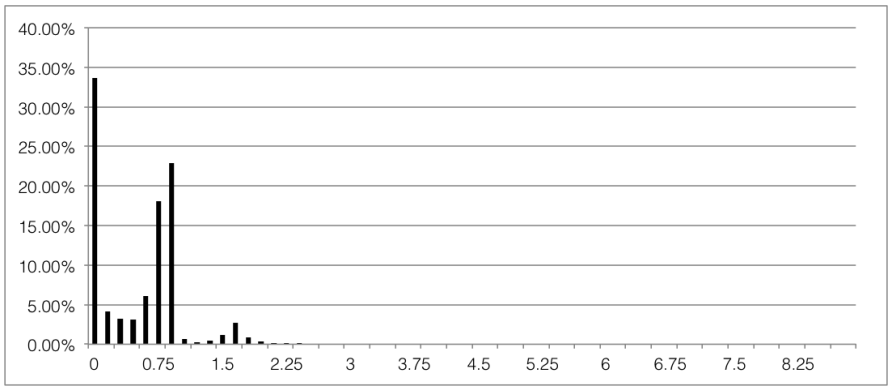
$$a_{NT_1} = \frac{|h - x_{NT_1}|^2}{|h - x_{NT_1}|^2_{MAX}} = \frac{|2.4 - x_{NT_1}|^2}{2.2^2} \tag{6.12}$$

i	X	$Var(X)$	σ	a_i
1	1.4	3.4	1.8	6
2	2.5	6.0	2.5	0
3	2.3	4.9	2.2	1
4	2.8	7.9	2.8	4
5	1.6	4.6	2.2	1
6	2.7	7.7	2.8	3
Average	*2.2*	*5.7*	*2.4*	-
Standard deviation	*0.5*	*1.6*	*0.3*	-
Farmbot	0.7	0.2	0.5	76
Buildbot	1.1	0.6	0.8	54

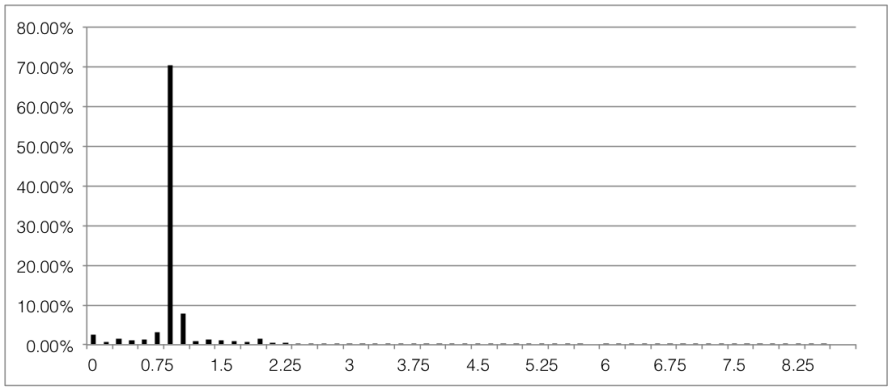
Table 6.7: Results for a_{NT1}



(a) Human average



(b) Farmbot



(c) Buildbot

Figure 6.7: HTTP Request Inter-arrival Time Distribution

6.2.3.2 NT_2- Request types

By analyzing the types of requests and their relative frequencies, it is possible to create a profile of a player's play style. Players play game differently, and this is reflected in the amounts of each request type being sent. For example, some players use the forum more than others, some players attack a lot, some players farm a lot. To find the relative frequency, the number of each request type was counted and divided by the total amount of requests. Figure 6.9 shows the relative frequencies of all requests in a sector diagram. The request names are slightly obfuscated to avoid disclosing the game identity. For display reasons, the request names are mapped to letters and this mapping can be seen in table 6.9. A large number of requests are generated each game session, and the most frequent ones account for a lion's share of the total. The ten most frequent request types make up for 86%, while the ten least frequent are just 2%. The most frequent ones are requests related to base management, using the in-game notification system, moving and looking around the world map, seeing information about other bases and the alliance forum.

A bot will typically not use as many game functions as a player since they are specially made to perform certain tasks. The difference between human and bot behavior can therefore be spotted by examining the variation of requests being sent. To measure this request type complexity, formula 6.13 is used. S_i is the relative frequency of request type i and n is the number of request types. This number $\mathbb{C} \in [0, 100]$ represents the complexity of the values - the more equal they are, the higher it will be. In mathematical terms, $S_1, S_2, S_3, \ldots, S_n = x \rightarrow \mathbb{C} = 1$. The $\mathbb{C}_i$ for all players was calculated and can be seen in table 6.8. The player complexity values are fairly close together, with an average of 35.0 and standard deviation of 2.5. An attribute function is needed to convert the complexity numbers $\mathbb{C}_i$ to attribute scores a_i. An exponential attribute function is suitable for this purpose, show in Figure 6.8. As all attribute functions it is normalized between 0 and 1.

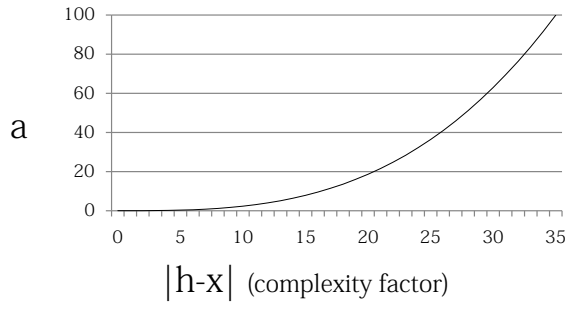


Figure 6.8: Attribute function for NT_2

i	Max_i	$< 1\%$	$\mathbb{C}_i$	a_i
1	20.80%	11	35.89	0
2	18.62%	14	29.6	0
3	17.87%	14	35.9	0
4	17.02%	10	37.0	0
5	20.14%	12	35.7	0
6	20.62%	13	35.8	0
Average	*19.23%*	*12*	*35.0*	-
Standard deviation	*1.53%*	*1.5*	*2.5*	-
Farmbot	98.25%	19	4.1	75
Buildbot	81.67%	19	5.8	64

Table 6.8: Results for NT_2

$$\mathbb{C} = x_{NT_2} = 100 \cdot \frac{(\sum_{i=1}^{n} S_i)^2}{n \cdot \sum_{i=1}^{n} S_i^2} \tag{6.13}$$

$$a_{NT_2} = \frac{|h - x_{NT_2}|^3}{|h - x_{NT_2}|^3_{MAX}} = \frac{|35 - x_{NT_2}|^3}{35^3} \tag{6.14}$$

The $\mathbb{C}$ values of the two bots are significantly lower: 4.1 for the farming bot and 5.8 for the main bot. These produce attribute scores of 75 and 64, meaning that the attribute function successfully caught these bots' suspicious behavior. These values stem from the fact that the bots only use a small subset of all request types, and that they are used almost exclusively. The most frequently used request types for the two bots make up for 98.25% and 81.67% of all requests, which is very far from the human average. They also have zero or very few occurrences of many of the request types, such as the request for the alliance forum. However, an even less sophisticated bot could be made, which would produce an even lower complexity value and means that the values produced by the bots from the case study shouldn't be scaled to 100.

Label	Request function
A	Popup menu
B	Index
C	Alliance forum
D	Notify
E	Base info
F	Update map
G	Premium farming overview
H	Farm overview
I	Message
J	Notification
K	Building 1
L	Unit production
M	Premium base overviews
N	Construction menu
O	Autocomplete
P	Command info
Q	Player info
R	Rankings
S	Base group overviews
T	Get image
U	Get data
V	Research
W	Login
X	Building 2
Y	Premium features

Table 6.9: Request type mapping for figure 6.9

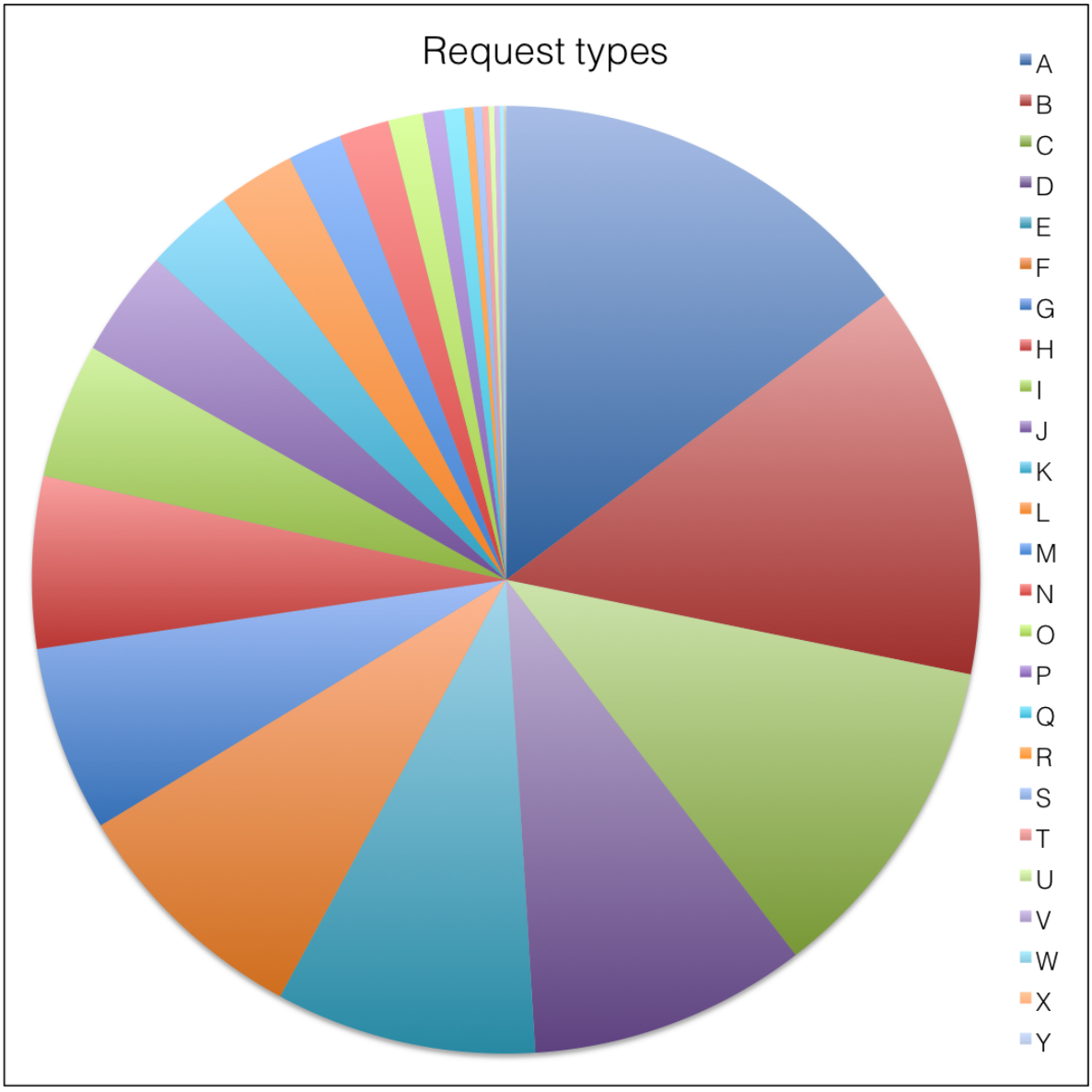


Figure 6.9: HTTP Request types, human average

6.2.3.3 NT_3- Request order

In this part, the sequence of requests is studied. This way it is possible to analyze the way a player plays the game, through looking at which action types follow other types more often. Even though many players tend to do certain actions in a fixed succession, there is always a degree of randomness to the play. Bots however, tend to repeat the same patterns over and over. This is discovered easily with this method of analysis.

These results can be presented with varying degrees of visualization, and three methods will be used here. The first is no visualization, through just mathematically crunching it down to a number representing the complexity of the request patterns. This is done through using a similar function to that used in the previous attribute, originally stemming from the field of network complexity analysis. The formula is then used to represent the equality (or inequality) of the traffic loads across different network links. In network complexity calculation, $S_{i,j}$ is the relative traffic load on the link from node i to node j. In this setting, $S_{i,j}$ represents the relative frequency of a request of type i being immediately succeeded by a request of type j. If all the relative frequencies are the same, the number will be 100. The less equal they are, the closer to 0 the result will be. The mathematical expression is shown below in formula 6.15. A simple second degree polynomial is used as attribute function, as shown in figure 6.10 and formula 6.16.

$$\mathbb{C} = x_{NT_3} = 100 \cdot \frac{(\sum_{i=1}^{I} \sum_{j=1}^{J} S_{i,j})^2}{I \cdot J \cdot \sum_{i=1}^{I} \sum_{j=1}^{J} {S_{i,j}}^2} \tag{6.15}$$

$$a_{NT_3} = \frac{|h - x_{NT_3}|^2}{|h - x_{NT_3}|^2_{MAX}} = \frac{|h - x_{NT_3}|^2}{9.9^2} \tag{6.16}$$

The second way of presenting the results is more graphic. It is a greyscale bitmap, made from the

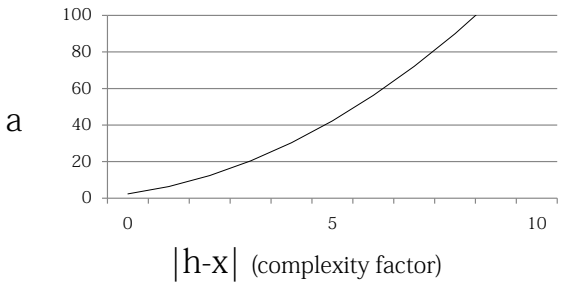
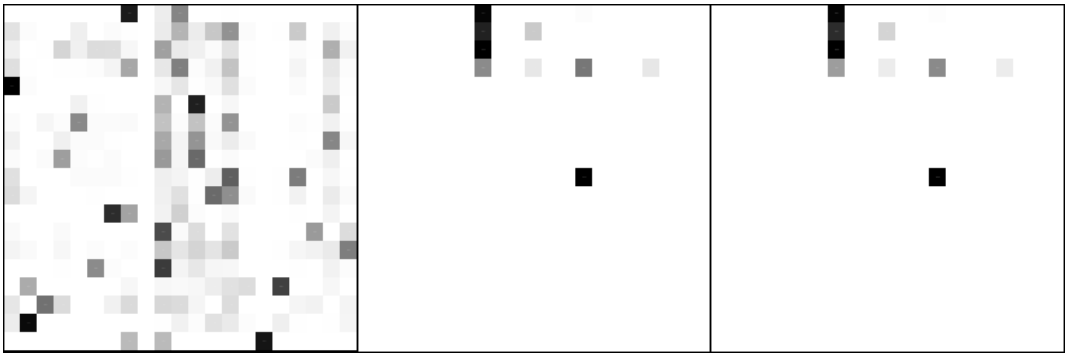


Figure 6.10: Attribute function for NT_3

i	$\mathbb{C}_i$	a_i
1	12.23	5
2	7.979	4
3	9.84	0
4	9.84	0
5	10.73	1
6	9.8	1
Average	*9.95*	-
Standard deviation	*1.32*	-
Farmbot	1.54	70
Buildbot	2.78	51

Table 6.10: Results for NT_3



(a) Human average (b) Farmbot (c) Buildbot

Figure 6.11: Representing NT_3 as a color-coded, two-dimensional table

two-dimensional array of transition probabilities between request types. A probability of 100% is represented as black, 0% as white and everything in between as varying shades of grey. The result is seen in figure 6.11. This representation method provides a very visual way to represent the information.

The third and most visualized representation method is to make a state diagram graph of the data, which is a complete model of a player's play style. The graph becomes quite complex for an average player even if only including $S_{i,j} > 5\%$ and it is not obvious how to extract information from the graph automatically. However, it is an easy way to see the general complexity of a player, as we see in the examples below. Figures 6.12 and 6.13 show the graphs for the human average and the buildbot. Note that the graph for the human average is only meant to demonstrate the complexity of the human behavior and not to show details. This same graph for the farmbot only contains one state, as there is only one $S_{i,j} > 5\%$.

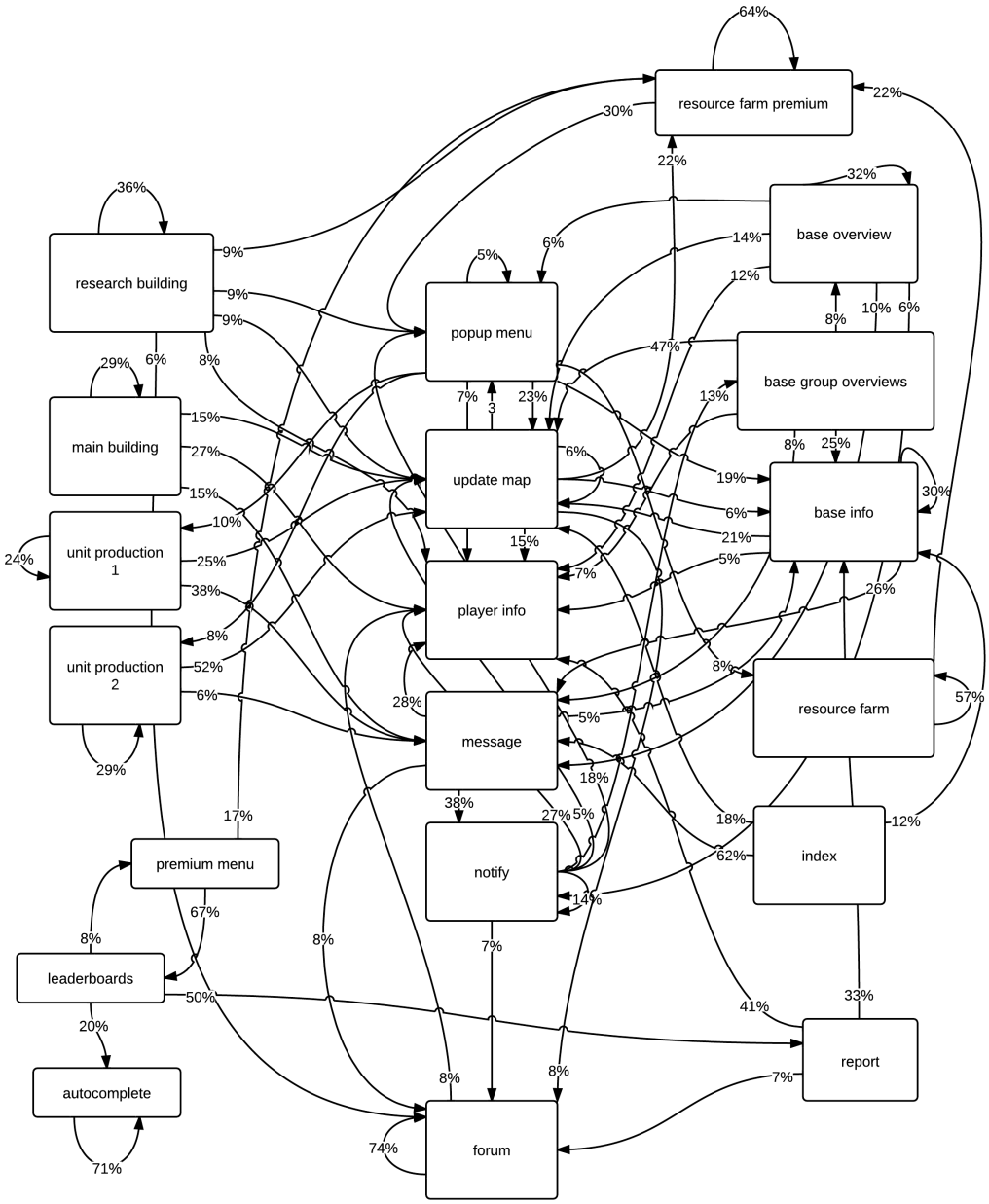


Figure 6.12: State diagram of NT_3 for the human average

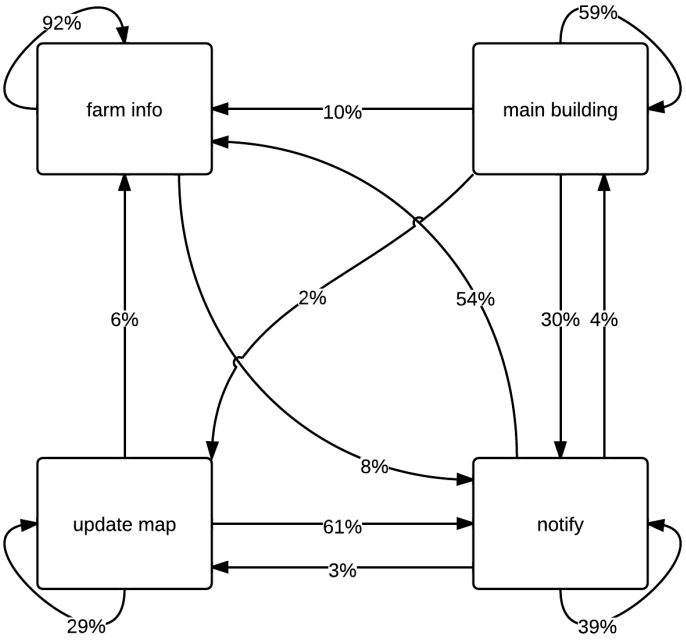


Figure 6.13: State diagram of NT_3 for the buildbot

The complexity value difference between the farmbot and the buildbot is bigger in a_{NT3} than a_{NT2} ([4.1, 5.8] vs [1.54, 2.78]). In the current attribute, the buildbot complexity is almost twice that of the farmbot, while only about 40% larger in the previous attribute. This is simply because of the farmbot's less sophisticated behavior pattern. Almost every request is followed by another of the same type, namely, a farming request. The buildbot is a tad more complex because it switches between farming and building, and this is represented in the complexity value. This shows that these two attribute of the NT category cover different aspects of a play style and complement each other. It also shows that the measurement method works!

6.2.4 Result summary

Table 6.11 shows the scores for all attributes for all players and the two bots. It also shows the category scores S_{CAT} and the total score S_{TOTAL}. In this application of the BDS, no attribute weights μ are used.

	Farmbot	Buildbot	P1	P2	P3	P4	P5	P6
a_{PA_1}	39	0	0	0	0	0	0	0
a_{PA_2}	0	1	0	0	0	0	0	0
S_{PA}	**20**	**50**	**0**	**0**	**0**	**0**	**0**	**0**
a_{GV_1}	97	0	0	0	0	0	0	0
a_{GV_2}	99	1	0	0	0	0	0	0
S_{GV}	**98**	**50**	**0**	**0**	**0**	**0**	**0**	**0**
a_{NT_1}	76	54	6	0	1	4	1	3
a_{NT_2}	75	64	0	0	0	0	0	0
a_{NT_3}	70	51	5	4	0	0	1	1
S_{NT}	**74**	**56**	**4**	**1**	**0**	**1**	**1**	**1**
S_{TOTAL}	**64**	**52**	**1**	**0**	**0**	**0**	**0**	**0**

Table 6.11: Summarized results for all attributes

6.3 Discussion

The results clearly show that the bot detection system is successful in detecting the bots. Both bots obtain high attribute scores for several attributes, and the buildbot even obtains a score of 100 on two attributes. Also, the system is successful in not registering false positives from the players. All players receive individual attribute scores below 6, and most have scores of zero for many attributes. The fact that the bots have so high values mean that they could actually have been behaving closer

to the human average and still obtain relatively high scores. Similarly, the players could also have had larger deviations without getting high scores.

The single most important reason this is worked is because it was possible to make good attribute functions. This was possible because there was enough data to recognize what is human behavior and what is bot behavior. It might be possible that in the universe of all the game's players there are players with larger deviations than the ones in the data set used here. This would mean that the curve of the attribute function would have to be less steep than the ones used, and the chance of error will become larger. On the other hand, the game developer will naturally have access to a much larger sample size in order to decide the boundaries of human and bot behavior. With this information, they would be able to create perfectly tailored attribute functions.

Finding the behavioral patterns of the known bots is also essential to creating good attribute functions. For the game used in the case study, there are currently two available bots on the market that can be easily found with a web search. Since these bots are publicly available, the game company can download and analysis them themselves. Also, as there probably are many people using these bots it is easier to detect them as opposed to scripts being used by only one person. This might just be the reason that the bots presented in this study has not been detected - because the game company is only focusing on the known bots, making detection trigger specifically for these instead of having more general parameters. With a detection system like the one proposed here, it is easier to catch the small-scale cheaters because there are more parameters being monitored.

It is also important to remember that the system is meant to flag suspicious accounts which are in turn inspected manually. This way, a false positive will not have as large consequences as if the player were automatically banned. Banning an innocent player is very bad publicity for the game company. The only cost involved for a false positive here is the cost of the wasted time for the person doing the manual inspections. The attribute function can be fine-tuned over time as false positives are discovered. In the end, it is a trade-off between catching as many bots as possible and having as few false positives as possible.

6.3.1 A hybrid solution

There is reason to believe that the game company have a few automated detection triggers running. The author has been in contact with the creator of one of the available bot programs, the dutch coder Jos Demmers, who shared his experience with the game company's automated detection methods. One of the ways the users of his bot got detected was that when the game was updated and the HTTP request format changed, his bot would keep sending requests of the old format. The game company would detect this and ban the users who still sent outdated requests. This is a simple, binary parameter that is easy to monitor - either a user has sent an outdated packet, or the user has not. Yet, it is a sure

sign that a player is using a bot, as the normal game client never would send an outdated request as the game client (the webpage) is forced to load the new version immediately. There are several binary parameters like this, and the bot detection system could be extended to also include these.

This could be done by adding a new attribute type b_i for binary attributes, and putting these attributes in appropriate categories. The category scores would be extended to include the number of enabled binary attributes in addition to the regular, numeral category score. Alternatively, they could modeled like a normal attribute but weighted with factor of two or three, so that a positive binary attribute would kick in and be visible in the category score.

The reason that binary attributes have not been included in this chapter is simple. Binary parameters are easy ways to surely detect bots, but also relatively easy to avoid. The BDS proposed here is more complex and can perform a more thorough comparison of play styles. Binary attributes can be seen as the first step in a bot detection scheme where a more detailed system like the one proposed in this thesis would be the next step.

Other possible binary attributes:

Short packet interarrival times: A bot could issue many more commands per minute than any human and thus be sending out packets at suspiciously fast intervals. The dutch bot developer claims that the early versions of his bot was detected because of this, and that he had to set a delay of 1200 ms to avoid getting caught.

Simultaneous actions: When using a script to automate the farming procedure, the player could be running this script not only when away from the game but also while playing. If the player is doing some other action in the game while the script is farming, the request pattern will reveal this as doing two separate things simultaneously which is not possible without superhuman mouse skills. This is clear sign that a player is using scripts, and can be monitored by looking at packet interarrival times. If they are very small for a number of seconds and the packets are varying between farming packets and some other action, it is a bot for sure.

Impossible orders: When ordering units, the unit production menu will have to be opened first. The request generated when placing an order can only be generated by using the unit production menu, but a bot may create such a HTTP request from scratch without opening the menu. This means that if a player suddenly places an order without opening the corresponding menu in advance, it is a sure sign that the player is not using the original game GUI.

6.3.2 Play style variations

As a user begins using a bot, their play style will most likely change. It is also reasonable to assume that most bot users start playing the game naturally and discover the use of bots at a later stage. This means that the play style will at some point change drastically. This change can be detected by the proposed system by storing average player scores over time and monitoring for sudden changes.

6.3.3 Selecting attributes

When selecting these attributes, it's vital that they are measurable server-side. In a regular, non-browser game, each keyboard input or mouse action will result in a network packet bound for the server and is therefore monitorable for the server. In our game, this is not the case. Some in-game clicks will not generate network traffic, as they only set local variables which are then included in HTTP messages that are sent later, triggered by clicks in other areas of the interface. Thus, it is possible that only a subset of the users actions in a browser-game initiates traffic to the server. This is why Wireshark is used to capture outgoing traffic instead of measuring time between clicks in the game directly.

6.3.4 Possible sources of error

1. A fellow master student at the Department of Telematics have also been using Wireshark for her thesis. She reported to have had some problems with Wireshark where some packets were not recorded. There were periods of time of up to a minute where no packets were captured, although it was certain that packets were going in through the network interface. It is possible that this bug may have happened during the recording of packets for this thesis. This is hard to find out, as a period of no recorded packets could just as well be that the recording player was not playing in that time period. However, with the number of packets and play time recorded for the people participating in the data gathering for this thesis, it is not likely that this error would have any significant impact on the results.

2. The same fellow master student also reports that Wireshark buffers packets and this adds a slight source of error to the packet timestamps. The magnitude of error was reported to be in nanoseconds, so this influence on the result will be discarded.

3. The local packet timestamp differences on local outgoing packets is probably not identical to the timestamp differences of incoming packets on the server. There are many possible sources of delay, both in the network and in processing delays in the source and destination node.

Chapter 7

Discussion

This chapter will discuss the findings of the preceding chapters. All chapters contain their own sections where the results are discussed, while this chapter intends to reflect upon the findings at a higher and broader level.

7.1 Background study

The background study clearly shows that there are lots of different hacks and cheats in circulation, and that there's also a large number of cheat vendors or cheat websites distributing these cheats. It is hard to find any numbers on cheat sales or cheat users, but the sheer number of cheat websites suggest that there are a substantial amount of cheaters. There are however a few figures available. The site *VACBanned.com* contains information about the current status on account bans made by the anti-cheat software Valve-Anti Cheat. According to them, currently 1.73% of the total 100 000 000 Steam users are banned, which results to 1 730 000 banned users. It must also be taken into consideration that this is only the cheaters that have gotten caught. The other source of information is the game developers behind *All Points Bulletin: Reloaded*, who after a large wave of bans could reveal that the three major cheat developers for their game would had made between $15 000 and $50 000 monthly. Using a default cost of $15, we would end up between 1000 and 3500 users monthly of just this cheat alone. Although there is a large degree of uncertainty in these numbers, it is clear that the extent of cheating is severe.

Game hacking dynamics

The dynamics between cheat makers and anti-cheat makers work very much like the dynamics of the

malware and anti-virus industry. As soon as one part releases something new, it is only a matter of time before the counterpart will respond. So goes the development cycles of both cheats, anti-cheats, malware and anti-virus. However, it is the bad guys who initiate each iteration by launching a new threat on the market and the good guys who will have to respond. The notion of malicious software is nothing new, and neither is software designed to keep your system clean. Still, this doesn't mean that we have malware-free computers in 2012, and it is tempting to draw the same conclusion for game cheats.

Cheaters = premium customers?

The game developers of *All Points Bulletin: Reloaded* also revealed that nearly 50% of the banned accounts were premium accounts, where the players had used real money to buy in-game weapons and items. This brings attention to an interesting notion - for premium games, there's a chance that banning cheaters may include losing paying customers and valuable revenues. The work involved with finding and banning cheaters is also costly, and it might be that the total costs of anti-cheat development, practical costs of banning the users and lost revenues from premium users outweigh the benefits of removing cheaters. This will probably depend greatly on the nature of the cheats used. In FPS's, players with aimbots are very visible and quickly break the motivation of their opponents when they headshot you while jumping from a cliff and looking in the opposite direction. On the other extreme, using a bot in a browser-based game like the game in the case study can be nearly invisible if used with moderation. This possibility that costs outweigh the benefits of removing cheats related to VISA card security. VISA has decided that the security of the credit card is good enough, and that it is cheaper to reimburse customers who have been scammed or ripped off, than it is to make a more security but less user-friendly system.

The way games are meant to be enjoyed

One of the hacker websites had the slogan *"The way games are meant to be enjoyed"*. During the research for the background study, it became clear that there are people who don't cheat to increase their chances of winning but because they see it as a part of the game experience to be able to fiddle with game code, change parameters and in various other ways tweak the game. This allegedly prolongs the lifespan of a game and adds excitement after a game has lost its initial appeal. The Teamkill and Cheat Community (*www.tkc-community.com*) say in their FAQ that "*Games, above all else must be fun for the player ... When a game has lost its entertainment value, there ceases to be a purpose to play. Players have the right to due whatever is necessary to perpetuate or resurrect enjoyment in a game for as long as desired, and this right need not be sacrificed for the enjoyment of fellow players.*". This act of cheating is thus not always performed in order to increase win rates and crush opponents but to change and rejuvenate the game experience. This philosophy of game cheating must also be taken into account when considering cheating.

The anti-cheat efforts

The anti-cheat firms are continuously struggling to catch cheaters, however, the general consensus on game forums is that the game hackers heavily outnumber the game security employees. Statements such as *"the hackers will just find a way around the anti-cheat programs anyway"* are common. The number of cheats and the information on the cheat websites about current detection status also suggest that the cheats are rarely down due to new detection methods. The game hack review website stated in a review that "*Three to four bans a month is inexcusable, especially for Aim Junkies.*". This implies that bans should be occurring more rarely than several times a month, and if a user is able to play with cheats for a month without being banned, is the anti-cheat really working? The fact that you may get banned and lose your game account will possibly scare off a few would-be cheaters, but one ban per month isn't much of a hindrance to a determined cheater.

Server-side games

A particularly interesting finding is the games Heroes of Newerth and League of Legends, where almost the entire game is run server-side. This means that the game client is an extremely thin software, only sending information about key presses and mouse gestures to the server, and the server performs all calculations and decision-making. The information about the resulting events in the game is returned to the client which is responsible for displaying it graphically on the screen. By moving all the logic and data storage to the server, there isn't much left on the client to be hacked. These games would normally be subject to maphacks or hacks that give you unlimited resources or hitpoints but when all this information is stored server and strictly validated, it is very hard or impossible to hack it without hacking the game server.

The drawback of this scheme is that it places a significantly higher load on the servers and increases the hosting costs for the game. The only reported cheats for these games was an exploit that found an unintentional loophole in the game code of League of Legends that allowed players to continuously cast a spell that would normally have a several minute cooldown [33]. This was quickly fixed by the game developers and the offenders were banned. This is cheating of a quite different nature than the ordinary hacks being sold, where the hacks and anti-cheat software are updated every other day in the eternal arms race.

Will games ever be cheat-free?

All games except the server-sided ones have working cheats being sold for them on the cheat market. Will these ever be cheat-free? Is the only possible solution to move all security-critical parts of the game into the cloud to obtain secure games? One could also picture a solution where the game company rootkits your PC and assumes administrator privileges, but this is drastic intrusion on your personal computer just to secure your gaming experience. Is it worth it? It would require substantial research on the topic to justify such an initiative.

7.2 Case study

Chapter 5 showed that it is easy to cheat the target browser-game by using javascript to automate game tasks. It also showed quantitatively that the gains from using cheats are substantial, even if only automating a few parts of the game. The cheat potential in the game is huge, and the scripts presented is only a small part of what is possible. Writing scripts like these thesis requires only a basic understanding of HTML and JavaScript, and can be written by anyone with a little programming experience.

The resource collection mechanism in this game makes bot use especially effective, but it is a threat to all games of the same genre. The game company has a backlog of several other similar games, where these also contain game mechanisms that are well suited as bot work. Most games favor active players who spend much of their free time playing, and the aspects of the games that reward this are typically building, researching, attacking other players or some sort of resource collection or management.

Unknown extent of script use

There are currently 43 scripts made for the game from the case study available on the site *www.userscripts.org*. These scripts are pure JavaScript files that are installed in the browser and functions as extensions of the web browser code library. Many of these scripts perform game functions that are not allowed to be scripted according to the game's terms of use, and some scripts even automate the resource collection just like the bots presented in this thesis. Although the script site reports 1085 downloads of this particular farm script, it's hard to say how extensive this kind of script use is among the players.

Selling game accounts

For many of the big MMO's, there is a big secondary economy around buying and selling in-game items and currency. Also game accounts with well-developed characters are being sold for real money on third-party websites such as *www.vbarracks.com*. Character of maximum level with many rare items represents hundreds of hours of playtime and are thus very sought after among players, and can sell for several hundreds dollars. A much-played game account in this game can also represent similar amount of playtime, and could possibly be sold. This adds another dimension to the cheating when a cheat could use the bots to develop game accounts to a certain point and sell them, or even create accounts on certain servers on demand. The global secondary economy with regard to online games is huge, and was estimated to $3 billion in 2011 [3].

Further development of the bot

The bots written as a part of the thesis could have been made much more sophisticated. This bot was merely a tool to simplify parts of the game and aid the player in normal playing, but there's much more functionality that could be added. Some of these are listed below.

- Automatic attacking. Make the script launch attacks, so by only specifying target and which bases to be included in the attack the bot would perform the rest. It would calculate the travel times from all bases, and launch the attacks at the appropriate times so the attacks would land together.
- Bypassing premium. Implementing the premium overviews mentioned in item 2 in section 5.5.2.
- Automatic trading. Make the bot automatically trade resource from bases with surplus resources to bases in demand.

Detection avoidance

Much could be done to make the bots harder to detect. One obvious example is making the time between request more random, or even try to emulate the human average time distribution. Another measure could be to make the bots use all the game function instead of just the few, for example opening the alliance forum every now and then, looking at player profiles, etc. This would require the BDS to be even more sophisticated in return, and adapting the attribute functions to the new bot behaviour. The arms race between cheat developers and anti-cheat developers is never-ending.

A completely unattended bot

The bot could be extended to perform all tasks involved with playing the game, without human interaction. Also the strategic parts of the game, like choosing which players to attack and which bases to take over could be programmed with a bit of artificial intelligence. To continue this train of thought, one could set up several bot players like this that could be controlled through a central GUI and populate a game server with a large botnet of accounts controlled by the player. With this many bases behind your back the bot developer would be a force to be reckoned with. The playing ability of such a complete bot would depend only on the game knowledge of the bot developer, as the bot would win over any player in terms of playtime and attack precision.

Chapter 8

Conclusion

This chapter contains a summary of the thesis, and answers the research questions presented in section 1.2. These are also answered throughout the thesis, with a chapter dedicated to each of the questions Q1 - Q4. These chapters form the broad background needed and contains information used to answer the research question , while a more brief and summarized version is presented in this chapter.

8.1 Q1: What cheats exist for modern online games? Which games and game genres are being cheated? How are cheats distributed?

Chapter 3 contains the background information needed to answer this question. This section summarizes the findings.

Twenty-four games and the cheats available for them were studied, and the result can be seen in table 3.3. Table 3.2 shows the games that were included in the study. The research revealed that cheats are available for nearly all online games. Games where all code is run server-side is the only exception, such as Heroes of Newerth and League of Legends. The cheats available for a given game depend heavily on the game's nature and genre. Some cheat types are very common and are found for a wide range of games. Also, some cheat types exists for all games of a certain genre. For FPS games, these are aimbots and wallhacks. For MMO's, there is always some sort of bot available that will automate the boring parts of the game.

Cheats are made by game hackers and sold through cheater websites. A total of 16 such cheat vendors

were found, the largest selling cheats for 44 different games. Cheats are sold either separately or through monthly subscriptions where cheats for all games are included.

It was the intention to describe how the cheats work on a rather technical level, however, this turned out to be too time-consuming and was dropped to rather spend more time on the other research questions.

8.2 Q2: What anti-cheat services exists today? What countermeasures are being used?

Chapter 4 contains the background information needed to answer this question. This section summarizes the findings.

Anti-cheat measures are employed both by the game developers and as third-party anti-cheat services. Some games use their own proprietary software exclusively, while others rely on the use of a third-party service. The most notable third-party services are Valve Anti-Cheat and PunkBuster, covering 50 and 23 games respectively. Other services exist, but only support games already supported by VAC or PunkBuster or have very outdated websites, suggesting that they are not being developed anymore. The biggest proprietary software is The Warden, a system developed by Blizzard Entertainment for use in their own game series *Starcraft*, *Diablo* and *Warcraft*, including *World of Warcraft*. Detailed information on these services are not available, as the companies responsible do not wish to reveal anything that might help the cheaters in their cheat development.

However, some basic information on their detection mechanisms is known. The anti-cheat software tries to search for known cheats by scanning the computer memory, open processes and game directory files. When a cheat is found, the systems will ban the offending player based on either a player ID in the particular gaming platform or on a value based on the system hardware. The bans can be just for the game in question or for all games. The anti-cheat companies are very strict in enforcing the bans and does not reopen a banned account under any circumstances.

Algorithm 8.1 Examples of the three most important mechanisms used to make a bot

```
// Simulates a mouseclick
$('HTML-element-to-select').trigger('click')

// Delay the execution of doSomething() by 10 seconds
setTimeout(function() { doSomething() }, 10 000)

// Execute doSomething() every 10 seconds
setInterval(doSomething, 10 000)
```

8.3 Q3: How can one create a bot for use in browser-based multiplayer games? What benefits are achieved from bots?

Chapter 5 contains the background information needed to answer this question. This section summarizes the findings.

How can one create a bot for use in browser-based multiplayer games?

Browser-game bots are either made as standalone applications that mimic the game's HTTP requests, or as a part of the web browser, manipulating the original game GUI. The first variant is more work as it involves learning the structure of all HTTP request and response messages going to and from the game server, while the other approach only requires basic knowledge of HTML and JavaScript. With this method, JavaScript code is run from the browser console window to simulate clicks on various HTML elements. These clicks will trigger game actions, and the JavaScript library can be used to delay clicks by an amount of time, or to create repeating actions. The code shown in algorithm 8.1 constitute the basis of a browser-game bot. All in all it is a simple procedure and can be done by anyone with a little programming experience.

What benefits are achieved from using a bot?

The advantages gained by using a bot in a browser-game depends on the game, but most browser-games have some exploitable game mechanisms. This is because these games heavily favor active players, and a bot can be playing non-stop throughout the day. One example of these game mechanics is that building construction, upgrades or researches take some time to complete and it is desired to start a new one right after the previous one finishes. Another one is resource income, where active players earn more resources due to being logged on more.

Two game bots were made in order to perform quantitative analysis of the performance gain in a

particular game. The first bot was made to automate the resource collection procedure on a already well-developed game account. With this, it was possible to build much more battle units and build them much faster than the opponents, resulting a very fast game progression. In-game rankings were monitored for a month, where the game account used had the highest growth rate of all ten players in the study.

The second bot was made specifically for the early-game, which revolves around building and upgrading the base to a point where you can build troops and attack other players.. It would perform resource collection continuously and place a new build order as soon as the previous one finished. To measure its performance, three game accounts were set up and played until reaching a certain point in base development. One account used the bot and the other two were played manually, one with premium features and one without. The bot would complete the fastest, beating the premium account by 2 days and 12 hours and the normal account by nearly 12 days.

8.4 Q4: How can bot use in browser-based multiplayer games be prevented?

Chapter 6 contains the background information needed to answer this question. This section summarizes the findings.

In order to stop bots, it is necessary to find aspects of the bots behavior that stands out from normal behavior. Only if one can successfully distinguish human players from bots will a bot detection scheme succeed. Thus, a very sophisticated bot that successfully imitates the playing style of a human will be undetectable by passive monitoring. By studying the play style of humans and bots, significant differences in some parts of the game behavior was identified. To show how these differences can be used to find bots, a Bot Detection System (BDS) is proposed. The BDS calculates average values for each attribute based on test data from a a set of guaranteed human players. A specific player is analyzed by comparing his gaming variables values for each parameters to the human average and calculating a severity score based on the deviation from the average value. The BDS is general, highly customizable and is applicable to all browser games. It is also fully automatable, in contrast to the bot detection schemes employed by most browser-based games today where players are manually inspected.

The BDS is then applied to the bots from the previous chapter. On a scale from 0 to 100 where 100 is most bot-like and 0 is the human average, all human participants scored lower than 2 and the bots scored 64 and 52, respectively. This verifies that the BDS works and can be used to detect bot use in browser-based multiplayer games.

Bibliography

[1] GameZone. Why Gaming Is Bigger Than Movies And Music. www.mweb.co.za/games/ViewNewsArticle/tabid/2549/Article/1474/why-gamings-bigger-than-movies-and-music.aspx (accessed June 6th 2012).

[2] ESA Annual Gaming Report. http://www.theesa.com/facts/pdfs/ESA_EF_2011.pdf (accessed June 10th 2012), 2011.

[3] V. Lehdonvirta and M. Ernkvist. Knowledge Map of the Virtual Economy. The World Bank, Washington, DC, 2011.

[4] A Systematic Classification of Cheating in Online Games. http://www.research.ibm.com/netgames2005/papers/yan.pdf (accessed June 7th 2012).

[5] Kuan-Ta Chen, Andrew Liao, Hsing-Kuo Kenneth Pao, Hao-Hua Chu. Game Bot Detection Based on Avatar Trajectory. http://mmnet.iis.sinica.edu.tw/pub/chen08_gamebot.pdf (accessed June 7th 2012).

[6] Steven Gianvecchio, Zhenyu Wu, Mengjun Xie, Haining Wang. Battle of Botcraft: Fighting Bots in Online Games with Human Observational Proofs. http://www.gianvecchio.com/uploads/1/0/7/8/10784991/ccs09.pdf (accessed May 23rd 2012).

[7] Christian Platzer. Sequence-Based Bot Detection in Massive Multiplayer Online Games. http://www.iseclab.org/papers/ICICS2011.pdf (accessed June 7th 2012).

[8] Stefan Mitterhofer, Christian Platzer, Christopher Kruegel, Engin Kirda. Server-Side Bot Detection in Massive Multiplayer Online Games. http://iseclab.org/papers/botdetection-article.pdf (accessed June 7th 2012).

[9] Kuan-Ta Chen, Jhih-Wei Jiang, Polly Huang, Hao-Hua Chu. Identifying MMORPG Bots: A Traffic Analysis Approach. http://mll.csie.ntu.edu.tw/papers/bot\ident.pdf (accessed June 7th 2012).

[10] A. A. Berger. *Media Analysis Techniques.* SAGE, 2005, page 1.

[11] The Verge. Modern Warfare 3 sales edge out Avatar by reaching $ 1 billion in record time. http://www.theverge.com/2011/12/12/2629959/call-of-duty-modern-warfare-3-record-sales (accessed on June 1st 2012).

[12] DFC Intelligence. Online Sales Expected to Pass Retail Software Sales in 2013. http://www.dfcint.com/wp/?p=311 (accessed July 13th 2012).

[13] Market Size Blog. Worldwide movie industry market size 2010. http://www.marketsize.com/blog/index.php/category/movies/ (accessed May 11th 2012).

[14] Wikipedia. Global music industry market — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/wiki/Global_music_industry_market_share_data (accessed May 11th 2012).

[15] Adweek. The DVD Dies Before Our Eyes. http://www.adweek.com/news/technology/dvd-dies-our-eyes-131970 (accessed May 11th 2012), 2011.

[16] Wikipedia. Global Starcraft II League — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/wiki/GOMTV_Global_Starcraft_II_League (accessed May 14th 2012).

[17] BarCraftig utvikling (in Norwegian). http://www.underdusken.no/reportasje/2012/7/1523540/barcraftig+utvikling (accessed May 14th).

[18] Andy Kuo. A Very Brief History of Cheating. http://www.stanford.edu/group/htgg/cgi-bin/drupal/sites/default/files2/akuo_2001_2_0.pdf (accessed June 1st).

[19] Apple. Wordfeud Downloads. http://itunes.apple.com/us/app/wordfeud/id428312806?mt=8 (accessed April 26 2012).

[20] Teknofil.no. Raging against Wordfeud-cheating (in Norwegian). http://www.teknofil.no/artikler/rasermotwordfeud-juks/105252 (accessed April 26th 2012).

[21] Ordbok.com. Wordfeud tips og tricks (in Norwegian). http://www.ordbok.com/wordfeud/tips.html (accessed April 26 2012).

[22] Forum post, Starcraft II Drophack Use. http://www.d3scene.com/forum/starcraft-2-hacks/53698-desync-hack.html (accessed May 3rd 2012).

[23] HackReviews. http://www.hackreviews.com/forums/forum.php (accessed June 2nd 2012).

[24] TechMech. All Points Bulletin:Reloaded ban wave. http://apbreloaded.gamersfirst.com/2011/09/do-we-need-some-sort-of-cheater-amnesty_30.html (accessed June 2nd 2012).

[25] Forum post about Heroes of Newerth. http://www.d3scene.com/forum/heroes-newerth-hacks/40138-any-hacks-our-there.html (accessed June 3rd 2012).

[26] PunkBuster — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/wiki/PunkBuster` (accessed June 3rd 2012).

[27] Wikipedia. Steam — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/wiki/Steam_(software)` (accessed May 3rd 2012).

[28] VACBanned. VAC Ban Statistics. `http://vacbanned.com/view/statistics` (accessed May 3rd 2012).

[29] Electronic Frontier Foundation. A New Gaming Feature: Spyware. `https://www.eff.org/deeplinks/2005/10/new-gaming-feature-spyware` (accessed April 26 2012).

[30] CNet News. Game Player Say Blizzard Invades Privacy. `http://news.cnet.com/Game-players-say-Blizzard-invades-privacy/2100-1043_3-5830718.html` (Accessed April 26 2012).

[31] Anderson, C. Free: The Future of a Radical Price: The Economics of Abundance and Why Zero Pricing Is Changing the Face of Business. *New York, US and London, UK: Random House*, 2009.

[32] Wikipedia. jQuery — Wikipedia, The Free Encyclopedia. **http://en.wikipedia.org/wiki/JQuery** (accessed May 8th 2012).

[33] Riot Games. Forum post about League of Legends exploit. `http://na.leagueoflegends.com/board/showthread.php?t=2088065&highlight=tamat` (accessed June 5th 2012).

Appendix A

Source code

This section contains the code used in chapter 5. This includes the JavaScript code for the farmbot, the buildbot and the extra scripts used, including keyboard shortcuts.

A.1 Supplementary scripts

```
// adding hotkey for selecting next base in the list
$('body').keydown(function (event) {
// figure out if either forum or messages are open,
// because then we dont want to bind the space key
var windows = GameWindowManager.getAllOpen();
var forumOpen = false;
var messagesOpen = false;
for (var i = 0; i < windows.length; i++) {
if (windows[i].type == 7 && windows[i].getTitle() == "Forum") forumOpen = true;
if (windows[i].type == 16 && windows[i].getTitle() == "Messages") messagesOpen = true;
}
// if event is space and forum and messages not open
if (event.which == 32 && !forumOpen && !messagesOpen) {
var windows = GameWindowManager.getAllOpen();
for (var i = 0; i < windows.length; i++) {
 if (windows[i].type == 7 && GameWindowManager.getAllOpen()[0].getTitle()=="Forum")
  { return undefined; }
```

```
}
event.preventDefault();
nextBaseInGroup();
}
});

// For discovering which keys are which integers
$('body').keydown(function (event) {
    // console.log(event.which); // uncomment to enable
    });


$('body').keydown(function (event) {
if (!typingWindowOpen()) {
if (event.which == 65) { // A opens RESEARCH
javascript:GameWindowManager.buildingWindow.open('research')
}
else if (event.which == 66) { // B opens UNIT PRODUCTION
javascript:GameWindowManager.buildingWindow.open('unit_production1')
}
else if (event.which == 67) { // C opens [removed to hide identity]
openCultView();

}
else if (event.which == 68) { // D unsets Base group
deselectBaseGroup();
}
else if (event.which == 72) { // H opens UNIT PRODUCTION 2
javascript:GameWindowManager.buildingWindow.open('unit_production1')
}
else if (event.which == 79) { // O opens [removed to hide identity]
openCommandView();
}
else if (event.which == 83) { // S opens building menu
javascript:GameWindowManager.buildingWindow.open('building_construction')
}
else if (event.which == 86) { // V opens [removed to hide identity]
javascript:GameWindowManager.buildingWindow.open('intelligence_center');
```

```
}
}
});



function typingWindowOpen() {
var windows = GameWindowManager.getAllOpen();
for (var i = 0; i < windows.length; i++) {
if (windows[i].type == 7 && windows[i].getTitle() == "Forum") return true;
if (windows[i].type == 16 && windows[i].getTitle() == "Messages") return true;
}
return false;
}

function unitOrderConfirm() {
$('#unit_order_confirm').trigger('click');
}

function getWindowByType(stringType) {
var intType = -1;
switch (stringType) {
case "unit_production1":
intType = 10;
break;
case "unit_production2":
intType = 10;
break;
case "farm":
intType = 34;
break;
case "fv":
intType = 2;
break;
default: alert("Not found!");
}
var windows = GameWindowManager.getAllOpen();
```

```
for (var i = 0; i < windows.length; i++ ) {
if (windows[i].type == intType) return windows[i];
}

return undefined;
}

function nextBaseInGroup() {
var activeGroupId = IBases.getActiveBaseGroup().id;
var BasesInGroup = IBases.getSortedBaseGroupMapByName(activeGroupId);
var activeBase = Game.BaseId;
var activeBaseIndex = -1;

for (var i = 0; i < BasesInGroup.length ; i++ ) {
if (parseInt(BasesInGroup[i]) == activeBase) {
activeBaseIndex = i;
break;
}
}
if (activeBaseIndex == -1) {
console.log("ERROR: Active Base not found in active group? wtf."); }
// safeguard

var targetBaseId = BasesInGroup[(activeBaseIndex + 1) % BasesInGroup.length];
toggleBaseSelector();
setTimeout(function () { changeBase(targetBaseId, activeGroupId); }, 500);
}

function toggleBaseSelector() {
$('#Base_name_link').trigger('click');
}

function changeBase(id, group) {
$('#t' + id + '_g' + group + ' a:first').trigger('click');
}

 // CULTURE
```

```
function pointInProduction() {
return $('.point_progressbar div').length;
}

function openOverviews() {
if ($('.menu_wrapper').length == 0) {
$('#overview_link').trigger('click');
return false;
}
return true;
}


function openView1() {
var time = 500;
if (openOverviews()) { time = 50; }
setTimeout(function() {
 $('[removed to hide game identity]').trigger('click'); }, time);
}

function openView2() {
var time = 500;
if (openOverviews()) { time = 50; }
setTimeout(function() {
 $('[removed to hide game identity]').trigger('click'); }, time);
setTimeout(function() {
 overlay([removed to hide game identity] }, time + 500);
}
function deselectBaseGroup() {
var time = 500;
if (openOverviews()) { time = 50; }
setTimeout(function() {
 $('#Base_group_overviews-Base_group_overview').trigger('click'); }, time);
setTimeout(function() {
 $('#Base_group_unset_active a').trigger('click') }, time + 500);

}
```

```
// adds information temporarily to the top left section of the screen
function overlay(txt) {
if ($('tekstboks').length == 0) {
$('body').append('<div id="tekstboks"></div>')
$('#tekstboks').css('opacity','0.0');
}

$('#tekstboks').css('position','absolute');
$('#tekstboks').css('opacity','0.6');
$('#tekstboks').show();
$('#tekstboks').css('top','72px');
$('#tekstboks').css('left','170px');
$('#tekstboks').css('height','17px');
$('#tekstboks').css('backgroundColor','white');
$('#tekstboks').css('zIndex','1000000000');
$('#tekstboks').css('border','1px solid black');


$('#tekstboks').css('-webkit-box-shadow','0px 0px 8px rgba(0, 0, 0, 0.3)');
$('#tekstboks').css('border','1px solid black');
$('#tekstboks').css('border','1px solid black');

$('#tekstboks').text("   " + txt + "   ");
setTimeout(function() { $('#tekstboks').hide(); }, 10000);

}



function delayedAttack(time) {
setTimeout(function() {
nextBaseInGroup();
setTimeout(function() { $('#unit5').trigger('click'); }, 3000);
setTimeout(function() { $('#unit1').trigger('click'); }, 3000);
setTimeout(function() { $('#unit2').trigger('click'); }, 3000);
setTimeout(function() { $('#unit3').trigger('click'); }, 3000);
setTimeout(function() { $('#unit4').trigger('click'); }, 3000);
setTimeout(function() { $('#attack_button').trigger('click'); }, 6000);
```

```
}, time);
}
```

A.2 Farmbot source code

```
// generates a time offset in range +- variance
 function rnd() {
    var variance = 40; // tidsoffset
    return Math.round(Math.random()*variance*2 - variance, 1)
    }

    function closeFarmWindow() {

    var windows = GameWindowManager.getAllOpen();
    var index = -1;
    for (var i=0; i < windows.length ; i++) {
    if (windows[i].type == 34) {
    index = i;}}
    if (index != -1) windows[index].close();
    }

    var BaseNo = 0;
    var BaseList = [];

    var delayTime = 900;

    function farmArea() {
    var BaseId = BaseList[BaseNo];
    var time = delayTime + rnd();
    $('.Base'+BaseId+'.fto_Base').trigger('click');

    setTimeout(function (){
    var moods = $('.fp_text').text().split('%')
    var minimumValue = 100;
```

```
    for (var i=0 ; i < moods.length ; i++) {
    var mood = parseInt(moods[i]);
    if (mood < minimumValue) { minimumValue = mood };
    }
    if (minimumValue > 85) {
    $('#fto_pillage').trigger('click');
    }
    else if (minimumValue < 80) {
    clearInterval(timerId);
    console.log('ERROR: SAFETY VARIABLE BELOW 80%, SOMETHING IS WRONG, FARMING STOPPED.');
    }
    else {
    $('#fto_claim').trigger('click');
    }
    $('#fto_claim_button .middle').trigger('click');
    }, time);


    BaseNo++; // these must be the two last lines of farmArea()
    if (BaseNo >= BaseList.length) {BaseNo = 0;}
    }


    function farm() {
$('a#farm_Base_overview_icon').trigger('click');
     setTimeout(function () {
            BaseList = [];
            $('.fto_Base').each(function () {
            BaseList.push($(this).attr('class').split(" ")[3].substr(4, 5)); });
            for (var i = 0; i < BaseList.length; i++) {
                setTimeout(function () { farmArea(); }, 2 * (1 + i) * delayTime);
            }
            setTimeout(function () {
             closeFarmWindow();
             }, (BaseList.length + 2) * 2 * delayTime + rnd());

        }, 1000);
```

```
}

function timerMethod() {
    farm();
}

var timerId = setInterval(farm, 320343 + 40*rnd());
farm();
```

A.3 Buildbot source code

```
// adds function max() to the type Array
Array.prototype.max = function() {
var max = this[0];
var len = this.length;
for (var i = 1; i < len; i++) if (this[i] > max) max = this[i];
return max;
}
// adds function min() to the type Array
Array.prototype.min = function() {
var min = this[0];
var len = this.length;
for (var i = 1; i < len; i++) if (this[i] < min) min = this[i];
return min;
}


// building levels the bot will upgrade to
// names are obfuscated
var maxLevels = new Object();
maxLevels['building_1'] = 24;
maxLevels['building_2'] = 15;
maxLevels['building_3'] = 20;
maxLevels['building_4'] = 28;
maxLevels['building_5'] = 30;
maxLevels['building_6'] = 40;
maxLevels['building_7'] = 25;
maxLevels['building_8'] = 40;
maxLevels['building_9'] = 10;
maxLevels['building_10'] = 25;
maxLevels['building_11'] = 10;
maxLevels['building_12'] = 10;


// list of building names as used in the game's javascript
// names are obfuscated
var buildingPool = ["building_1", "building_2", "building_3", "building_4",
```

```
 "building_5", "building_6", "building_7", "building_8",
  "building_9", "building_10"];
var busy = false;
var resTimer;
var resYield = -1;

function resourceAgent() {

// dont farm if full
if (storageIsMaxedOut()) { return; }

var farmIds = [1,2,3];
var farmNames = ["name1", "name2", "name3", "name4"];
var UIopt= {'action':'claim_info'};

GameWindowManager.window.Create(GameWindowManager.TYPE_RESOURCE_FARM,
'Demand from <span class=\"farm_town_title_postfix\">'+farmNames[0]+
'</span>',UIopt,farmIds[0]);
setTimeout(function(){ $('.farm_claim_resources_button:first').trigger('click'); }, 1000);

setTimeout(function(){
GameWindowManager.window.Create(GameWindowManager.TYPE_RESOURCE_FARM,
'Demand from <span class=\"farm_town_title_postfix\">'+farmNames[1]+
'</span>',UIopt,farmIds[1])}, 2000);
setTimeout(function(){ $('.farm_claim_resources_button:first').trigger('click'); }, 3000);

setTimeout(function(){
GameWindowManager.window.Create(GameWindowManager.TYPE_RESOURCE_FARM,
'Demand from <span class=\"farm_town_title_postfix\">'+farmNames[2]+
'</span>',UIopt,farmIds[2])}, 4000);
setTimeout(function(){ $('.farm_claim_resources_button:first').trigger('click'); }, 5000);

setTimeout(function(){
GameWindowManager.window.Create(GameWindowManager.TYPE_RESOURCE_FARM,
'Demand from <span class=\"farm_town_title_postfix\">'+farmNames[3]+
'</span>',UIopt,farmIds[3])}, 6000);
setTimeout(function(){ $('.farm_claim_resources_button:first').trigger('click'); }, 7000);
```

```
setTimeout(function() { busy = false; }, 9000);
setTimeout(function() { getWindowByType("fv").close(); }, 10000);
}

function maxStorageCapacity() {
return IBase.getBase(Game.baseId).resources().storage;
}

function getWoodCount() {
return IBase.getBase(Game.baseId).resources().wood;
}
function getStoneCount() {
return IBase.getBase(Game.baseId).resources().stone;
}
function getSilverCount() {
return IBase.getBase(Game.baseId).resources().iron;
}

function storageIsMaxedOut() {
var max = IBase.getTown(Game.townId).resources().storage;
return max == getWoodCount() && max == getStoneCount() && max == getSilverCount();
}


function findPossibleBuildOrders() {
var buildings = [];
$('.build.small').each(function() {
var type = "building_main_" + $(this).attr('onclick').split('\'')[1];
var newLevel = parseInt($(this).attr('onclick').split('\', ')[1]);
if (isIn(buildingPool, type)) {
if (newLevel <= maxLevels[type]) {
buildings.push(type);
}
}
} );
return buildings;
}
```

```
function getFreePopulation() {
return parseInt($('#pop_current').text());
}

function buildingAgent() {
if (busy) {
setTimeout(buildingAgent, 10000); // poll for free time
log("Client is busy, waiting for free time (10s) [building constructor]");
return;
}
Layout.buildingWindow.open('main');

setTimeout(function innerBuildLoop() {
if (getFreePopulation() < 3) {
build('farm'); // build farm if free population is lower than three
sleepMethod();
return;
}
var builds = findPossibleBuildOrders();
if (builds.length == 0) {
sleepMethod();
return;
}
var rnd = parseInt(Math.random()*builds.length);
var buildingToUpgrade = builds.length == 0 ? "none" : builds[rnd].split("_")[2];
build(buildingToUpgrade);
builds = findPossibleBuildOrders(); // if more possible build orders, recursive call

if (builds.length > 0 && buildingQueueCount() < 2) {
setTimeout(buildingAgent, 15000);
}
else {
sleepMethod();
}
}, 2000);
}
```

```
function isIn(list, target) {
for (var i=0 ; i<list.length;i++) {
if (list[i] == target) {
return true;
}
}
return false;
}


function sleepMethod() {
var remaining = remainingBuildTime();
var wait;
if (remaining == 0) { // if no buildings in queue and were justing waiting resources
wait = 600000 // wait ten minutes (two rounds of farming)
}
else { // else wait until queue is empty
wait = remaining + parseInt(Math.random()*5);
}
setTimeout(buildingAgent, wait);
log("Sleeping for " + parseInt(wait/3600000) + " hours " + parseInt((wait%3600000)/60000) +
" minutes and " + parseInt((wait%60000)/1000) + " seconds. ("+wait+" ms)");
}

function unitAgent() {
if (busy) {
setTimeout(unitAgent, 10000); // poll for free time
log("Client is busy, waiting for free time (10s) [unit producer]");
return;
}
if (getFreePopulation() < 20) {
return;
}

var types = ["unit1", "unit2", "unit3"];
// if player doesnt have technology to build unit type, remove from list
```

```
if ($('#unit2').css('display') == "none") types.splice(1,1);
if ($('#unit3').css('display') == "none") types.splice(types.length-1, 1);
var rnd = parseInt(Math.random()*types.length);
var unitType = types[rnd];
$('#unit_order_tab_' + unitType).trigger('click'); // select type of unit

// calculate amount of surplus cash that can be spent on units
var remainingFarmDemandsBeforeEmptyBuildQueue = parseInt(remainingBuildTime()/300000)
var expectedYield = remainingFarmDemandsBeforeEmptyBuildQueue*resYield;
var expectedRes = [getWoodCount() + expectedYield, getStoneCount() + expectedYield,
 getSilverCount() + expectedYield];
var surplus = [expectedRes[0] - maxStorageCapacity(),expectedRes[1] - maxStorageCapacity(),
 expectedRes[2] - maxStorageCapacity()];

var unit1Cost = [95, 0, 85];
var unit2Cost = [120, 0, 75];
var unit3Cost = [0, 75, 150];

var unit1ResCount = [surplus[0]/unit1Cost[0], surplus[1]/unit1Cost[1], surplus[2]/unit1Cost[2]];
var unit2ResCount = [surplus[0]/unit2Cost[0], surplus[1]/unit2Cost[1], surplus[2]/unit2Cost[2]];
var unit3ResCount = [surplus[0]/unit3Cost[0], surplus[1]/unit3Cost[1], surplus[2]/unit3Cost[2]];

var unitCounts = [unit1ResCount.min(), unit2ResCount.min(), unit3ResCount.min()];


var amount = 1;
$('#unit_order_input').val(amount);

Layout.buildingWindow.open('unit_production');



return undefined;
}

function remainingBuildTime() {
var secsToMs = 1000;
var minToMs = 60000;
```

```
var hoursToMs = 3600000;

var hours = 0;
var mins = 0;
var secs = 0;
var ms = 0;

$('#building_tasks .remanining_time').each(
function() {

var array = $(this).text().split(":");
hours += parseInt(array[0]);
mins += parseInt(array[1]);
secs += parseInt(array[2]);


});
ms = hours*hoursToMs + mins*minToMs + secs*secsToMs;
ms = ms;
return ms;
}


function findHighestPopulationRequirement() {
var reqs = [];
$('.possible_build_order').each(
function() {
$(this).mouseenter();
reqs.push(findPopulationRequirement());
$(this).mouseleave();
} );
var max = -1;
for (var i=0 ; i<reqs.length; i++) {
if (reqs[i] > max) max = reqs[i];
}
return max;
}
```

```
function build(buildingType) {
if (buildingType == "none") return;
var nextLevel = 1;
if (parseInt($('#building_type_' + buildingType +
 ' .level').text()) != 0) nextLevel = ('#building_type__'+buildingType+
' .possible_build_order:first').text().split("Expansion to")[1].trim();
log("Expanding " + buildingType + " to level " + nextLevel + " with building time " + findConstructio
$('#building_type__' + buildingType + ' .possible_build_order:first').trigger('click');


}

function cancelBuilding() {
$('.cancel').trigger('click');
}


function findPopulationRequirement() {
var popReq = parseInt($('.popup_middle_middle').html().split("Food\">")[1].substr(0,2));
return popReq;
}

function buildingQueueCount() {
return parseInt($('#building_tasks h4').text().split("(")[1].substr(0,1));
}



function findConstructionTime(buildingType) {
$('#building_type_'+buildingType+ ' a').mouseenter();
var time = $('#popup').html().split('Time">')[1].split('<br>')[0];
$('#building_type_'+buildingType+ ' a').mouseleave();
return time;
}

function getWindowByType(stringType) {
    var intType = -1;
```

```
    switch (stringType) {
        case "unit_production2":
            intType = 10;
            break;
        case "unit_production1":
            intType = 10;
            break;
        case "farm":
            intType = 34;
            break;
case "fv":
intType = 2;
break;
        default: alert("Didnt find specified window type");
    }
    var windows = GameWindowManager.getAllOpen();

    for (var i = 0; i < windows.length; i++ ) {
        if (windows[i].type == intType) return windows[i];
    }

    return undefined;
}

function log(logString) {
var d = new Date();
console.log("[GAMEBOT: "+d.toDateString().replace(" 2012","") + " " +
 d.toTimeString().replace(" GMT+0200 (W. Europe Daylight Time)", "")+"]" + ": " + logString);
}

function startBot() {
resourceAgent();
resTimer = setInterval(resourceAgent, 305333);
buildingAgent();
}
```

Appendix B

Case study participant instructions

This section contains the written instructions that was sent to the players who constituted the human average used in chapter 6.

1. Download Wireshark for your operating system at www.wireshark.org

2. Install and run Wireshark.

3. Under 'Capture' on the front page, click 'Interface list'

4. Wait a few secs and see which interface is sending and receiving packets and click the 'Options'-button to the right. Fill in the following:

Capture Filter: (a wide input field in the top box) : type in "tcp port http and dst host [game server IP]" without the quotes. Capture File(s) -> File:(input field with a Browse-button to the left): select a file where the capture will be saved.

Then click start!

5. You are now recording all network packets going out from your computer to the game server! Isnt that fun? :)

6. Play the game! Play the game just as you normally would, and leave Wireshark running for as long as you play. If you play a little now and then during the day, it is great if you can just leave the program running. If you play more concentrated a few times a day, you can stop the curring capture in Wireshark after logging out of the game, save the capture and start over the next time you play.

If 4. was done succesfully, Wireshark should now save the capture continously to your harddrive! Just

make sure Wireshark is open and capturing when you play, and when displayed packet count (should say the number at the bottom of the screen) exceeds 1000 it should be enough data to send me. If you can get more, thats even better! :)

One important note: do not log in while Wireshark is running, as the packets sent during login will contain password in cleartext. I am not interested in your passwords, but still you do not want to send them to me so make sure you are logged in from before when you start Wireshark!

I would like to receive the capture files on my email address erikwe@stud.ntnu.no

How does this sound? :)

IF THERES ANY QUESTIONS AT ALL, DONT HESITATE TO CONTACT ME ON SKYPE: wndl87

Appendix C

Scientific paper

A scientific paper was written based on the research from chapter 6. The paper is six pages long and describes the proposed Bot Detection System and its application on the bots from the case study. It has been submitted to CCSW 2012: The ACM Cloud Computing Security Workshop, a conference in conjunction with the ACM Conference on Computer and Communications Security (CCS). The following pages contain the document in its entirety.

Game Security in the Cloud

A Bot-Detection System for Browser-Based Multiplayer Games

Erik Wendel
Department of telematics
Norwegian University of Science and Technology
Trondheim, Norway
erik@wendel.no

Harald Øverby
Department of telematics
Norwegian University of Science and Technology
Trondheim, Norway
haraldov@item.ntnu.no

ABSTRACT

Browser-based multiplayer games (BBMG) have gained huge popularity in recent years due to ubiquitous game access and appealing game mechanics. Most BBMG are based on the client-server architecture and utilize the HTTP protocol for game data communication. This creates an opportunity for using bots to enhance playing performance, often in conflict with the rules set by BBMG providers. In this paper we present a bot-detection system (BDS) for BBMG, enabling BBMG providers to detect the use of bots among its players. The BDS identifies key attributes of player behavior and compare bot behavior with human player behavior. We employ the BDS, showing the difference in behavior between two different bots and human players for a selected BBMG. Our results show that there are differences between bot and human behavior, however, depending on the complexity of the bot.

Keywords

Browser-Based Games, Bots, Bot-Detection

1. INTRODUCTION

The gaming industry has enjoyed a huge rise of popularity the past years, and is now valued higher than the movie industry if both hardware and software are counted together [1]. Innovation has spawned new game genres and introduced gaming to new audiences. Social gaming platforms are making gaming a more social and sharing experience. According to ESA, 72% of all U.S households play computer or video games, the average gamer is about 37 years old and about 42% of all players are female [7].

During the last years, we have seen new games deployed as Browser Based Multiplayer Games (BBMG), exploiting both the added functionality and performance of modern web browsers for game experience. These games have become popular due to their ubiquitous game access and low entry barrier. BBMG come in all game genres and themes, appealing to both casual and hardcore gamers.

A challenge in BBMG's is the use of bots to increase player performance. Such bots may be developed by the user, or bought via a third party provider of bots. Many BBMG providers view the use of bots as an act of cheating, thus conflicting with the terms of use of the game. However, as bots do not directly alter game data or intrude game mechanics, it can be hard to detect for a BBMG provider if a player is using a bot or not.

In this paper we present a Bot Detection System (BDS) for BBMG. The proposed BDS identifies key attributes associated with a BBMG player, and how these attributes may be used to distinguish between a human player and a bot. We employ the proposed BDS to analyze two bots for a selected BBMG.

The field of game security has been subject to various research, including bots and bot detection. [3] analyze various network network parameters of players to reveal bots in MMORPG's, while [5], [2] and [4] look at the in-game movement patterns of players. [6] on the other hand, uses the variations in keyboard and mouse inputs to distinguish bots, also in MMORPG's. There have been no research on the topic of bot detection in browser games, however, there exists a tutorial on how to make your own bot for a browser-game [8].

The rest of the paper is organized as follows: Section 2 presents BBMG and the use of bots in BBMG. Section 3 presents a cheat detection framework for detecting bots in browser based games. A case study of a selected browser based game utilizing the proposed framework is presented in section 4. Finally, section 5 concludes the paper.

2. BACKGROUND

2.1 Browser-Based Multiplayer Games (BBMG)

BBMG come in all blends and flavors, however one of the most popular game genres is the real-time or tick-based large-scale war strategy games. One of the first popular games of this genre was Planetarion, released in 2001. In this game players would assume control of a planet in an intergalactic space war, managing economies, building structures and controlling battleships. The genre borrows the base-building and economy management aspects from real-time strategy games but also borrows elements from massively multiplayer online games (MMOG) by pitting players against

each other in a large, shared world.

Some BBMGs of this genre include Travian, Planetarion, Tribal Wars and Hyperiums. Figure 1 shows the game interface of Planetarion, which is representative for the genre.

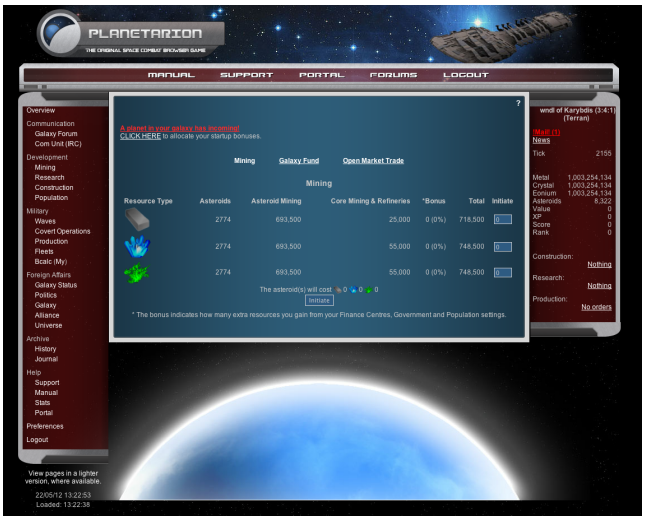


Figure 1: Screenshot of the game interface of Planetarion

The game interface is usually implemented in pure web technologies, using HTML, CSS and JavaScript. The communication with the game server is hence traditional HTTP, with a request-response communication protocol. The protocols are often simple and easy to mimic and this makes it easy to make a bot that communicates with the game server just like the web page does, bypassing the original game interface.

It is also feasible to implement a bot to manipulate the game interface by running JavaScript code snippets from the browser console, that will simulate mouse clicks in various parts of the web page often used in conjunction with JavaScript's own method library in order to delay actions by a certain amount of time.

2.2 The use of bots in BMMG

A bot is a program used to automate or optimize certain parts of the game. BBMG often contain game mechanics that greatly favors active players who invest a lot of time in the game, like logging in often to start a building construction or harvest resources. Doing these tasks regularly and often greatly increases player performance and this creates a large incentive for automation. A bot is therefore used both to maximize performance and reduce the effort involved.

A bot can either be a custom made game client that mimics the ordinary interaction pattern with the game server, or an add-on that simply manipulates the original game interface. Due to the increasing complexity in games, the first option is very rare except for BBMG. Here, the games' network protocols are usually so simple that they are easily analyzed and copied. A bot will consist of some artificial intelligence, a main loop function that will initiate various actions at set time intervals. Typical examples of these actions are attacking other players at night time when they are offline, placing a building construction order, harvesting resources or similar.

The use of bots in online games is debated. Most games' terms of use forbid their use explicitly, but some games allow use of macros or small scripts to simplify the gaming experience. World of Warcraft is an example of this, where bots written in the scripting language lua is accepted, but automating game functions by using external programs is not. There is also a divide between attended and unattended script use. Bots that run with human attendance are often allowed, but unattended bot use is forbidden in practically all games.

In BBMG, script or bot use is heavily regulated. The game used in the case study has a white- and blacklist of scripts or script functions that are allowed and forbidden. The whitelist is very short, only including scripts that e.g. puts the game server name in the browser titlebar, or modifies the colors of teammates on the map.

3. A BOT DETECTION SYSTEM (BDS) FOR BBMG

3.1 General model

This section presents a general Bot Detection System (BDS) for BBMG. The goal of the BDS is to detect the use of bots in BBMG by identifying a set of attributes and comparing these to average human behavior. These attributes are grouped in three categories as shown in figure 2.We introduce a score system ranging from 0 to 100, where the average human behavior is defined as 0, while increasing non-human behavior is given a score >0, with max 100.

The BDS will record and compute a human behavior pattern, based on a set of human players playing the game and their player characteristics.

We define three categories of player behavior in the set CAT as follows.

$$CAT = \{PA, GV, NT\} \quad (1)$$

1. Player Activity (PA). This category contains attributes regarding the playing pattern of a player, such as login times, session lengths, or total daily playtime.

2. Game variables (GV). This category contains attributes regarding in-game measurements or properties, such as experience points or resources.

3. Network Traffic (NT). This category contains attributes regarding the in-game behavior of a player, that is, what happens when a player is logged in. This can be the time between each game action, the types of game actions or the order of game actions.

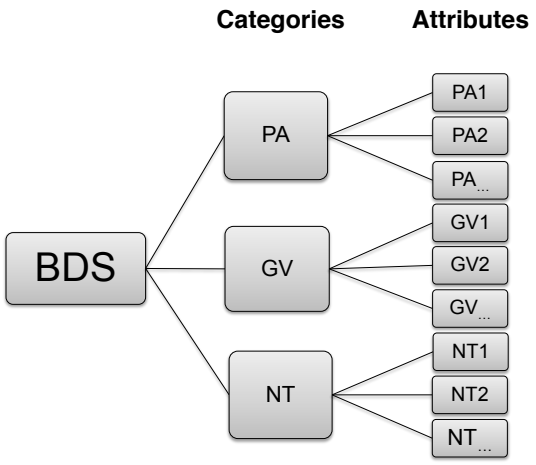


Figure 2: Categories and attributes

We then define A as the set of all attributes as follows:

$$\begin{aligned} A &= \{A_{PA}, A_{GV}, A_{NT}\} \\ &= \{PA_1, PA_2, PA_{...}, GV_1, GV_2, GV_{...}, \\ &= NT_1, NT_2, NT_{...}\} \end{aligned} \quad (2)$$

The number of attributes and number of attributes in a given category are defined as $|A|$ and $|A_{CAT}|$, respectively.

We define a player p_i with $|A|$ associated game variables as follows:

$$p_i = \{x_{j,i}\}, \forall_{j \in A} \quad (3)$$

For each attribute j in set A, a human average h is computed by using m samples from human players:

$$\forall_{j \in A} : h_j = f(x_{j,1}, x_{j,2}, x_{j,...}, x_{j,m}) \quad (4)$$

An attribute score for attribute A for player p_k is $a_{A,k}$. It will have a value in $[0, 100]$ where 0 represents human average behavior and 100 is as bot-like as possible.

In order to detect whether a certain player p_k is using a bot or not, we calculate the attribute score $a_{A,k}$ for every attribute.

$$a_{j,k} = f(h, x_{j,k}) \quad (5)$$

Depending on the nature of the attribute, a suitable attribute function $f(h, x)$ will need to be chosen. Linear, exponential or lognormal functions work well for this purpose. Examples of attribute functions are given later in this paper.

Variable	Description
x	Direct gaming variable
a	Attribute score, in [0,100]
h	Human average value for a certain attribute
μ	Weight factor
A	The set of attributes
CAT	The set of categories
A_{CAT}	The set of attributes in category CAT
PA	Category: Player Activity
GV	Category: Game variables
NT	Category: Network Traffic
p_i	Player i

Table 1: Terminology overview

The function must be scaled and possibly shifted to only return non-zero values for the appropriate deviation values. It is crucial to have sufficient amounts of player data in order to learn what deviation value range is standard human behavior and what is bot behavior.

Attributes may differ in importance in various BBMG, and this can be accounted for by factoring in a weight factor $\mu_j \in [0, \rightarrow]$ for each attribute j. The sum of all weights in a category CAT is μ_{CAT}.

Finally, all attribute scores in each category are summed and normalized to find the category scores S_{CAT}. These scores are in turn summed and normalized to obtain the final score $S_{TOTAL} \in [0, 100]$.

$$S_{CAT} = \frac{1}{|A_{CAT}|\mu_{CAT}} \sum_{j \in A_{CAT}} \mu_j a_j \quad (6)$$

$$S_{TOTAL} = \frac{S_{PA} + S_{GV} + S_{NT}}{3} \quad (7)$$

3.2 Descriptions of example attributes

3.2.1 PA_1 - *Number of logins*

Bots may in some cases keep trying to reconnect when there's a programming error in the code, resulting in a large number of logins over a short period of time. Also, bots who are run from the browser console will need a browser refresh to clean the browser window's running code, also resulting in a new game login.

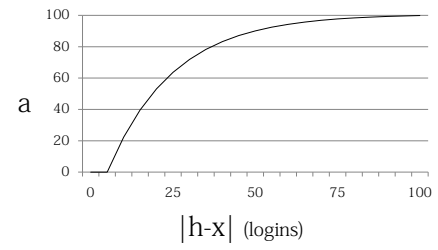


Figure 3: Example attribute function for PA_1

3.2.2 PA_2 - Aggregated session length

Some players are casuals and login perhaps once a day or even rarer, while other players are logged in throughout the whole day. Still, everyone has to sleep and eat, so playing continuously for more than 16 hours per day is definitely not human-like behavior. This can be caught by monitoring the aggregated session length of a player over a day.

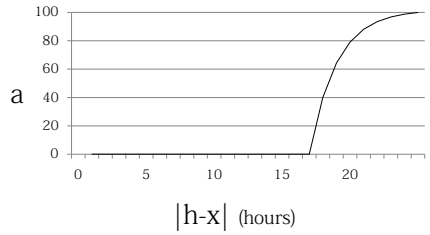


Figure 4: Example attribute function for PA_2

3.2.3 GV - Game-dependent variables

A big incentive of bot use is increased production of resources or experience points. Many games use some kind of resources be it gold, minerals or dollars, that is used to buy in-game services, items or benefits. Other games use an experience point system where a player avatar gains power by obtaining experience points through killing monsters or doing quests. These variables will vary greatly from game to game, but most games have one and they go in this category.

3.2.4 NT_1 - Packet interarrival times

By looking at the time between each outgoing packet to the game server it is possible to measure the time between each click in the game. Bot and human behavior vary in this regard, depending on the sophistication level of the bot. By calculating the standard deviation of the packet interarrival time distribution we discover notable differences between humans and bots. This is because bots are programmed to perform actions at certain time intervals, while humans are more random in their behavior. This attribute is applicable for all BBMG. Figure 5 shows the attribute function used in the case study.

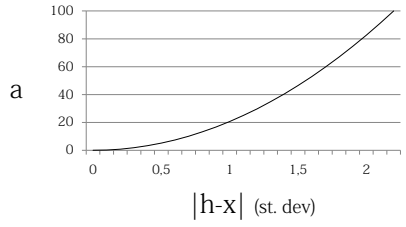


Figure 5: Example attribute function for NT_1

3.2.5 NT_2 - Request type distribution

Bots are usually less complex in their game behavior than their human counterparts, as they are programmed for specific tasks. It is not easy nor desired to make a bot that will perform every part of the game. These differences in game behavior complexity can be measured by examining the content of the outgoing data packets. Each packet contains a certain request type, and through calculating the relative frequencies for all request types we can create a game behavior complexity factor.

Formula 8 is used for this purpose, stemming from the field of network complexity. It will return a number in $[0, 100]$ that represents the equality of the different S_i. If all S_i are equal then it will be 1, decreasing with increasing entrophy. The original function is two-dimensional and it is used in NT_3. This attribute is applicable for nearly all BBMG. Figure 6 shows the attribute function used in the case study.

$$x_{NT2} = 100 * \frac{(\sum_{i=1}^{n} S_i)^2}{n \cdot \sum_{i=1}^{n} S_i^2} \tag{8}$$

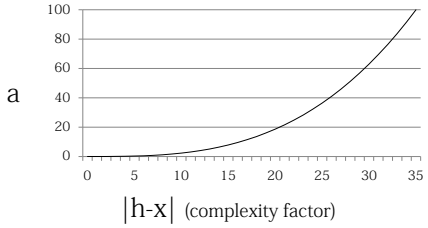


Figure 6: Example attribute function for NT_2

3.2.6 NT_3 - Request order

This attribute is similar to NT2, but instead of looking at the variation of request types, the order of requests is studied. For every request type, the succeeding packet types are studied. Example: Packet type A is followed by type B 50% of the time and type C 50% of the time. With this data, we can analyze the playing pattern of player. The data is compressed to single number representing the overall complexity of the data similar to NT2, shown in formula 9. This complexity number in $[0, 100]$ will be higher the more equal the relative frequencies are. This attribute is applicable for nearly all BBMG. Figure 7 shows the attribute function used in the case study.

$$x_{NT3} = 100 \cdot \frac{(\sum_{i=1}^{I} \sum_{j=1}^{J} S_{i,j})^2}{I \cdot J \cdot \sum_{i=1}^{I} \sum_{j=1}^{J} {S_{i,j}}^2} \tag{9}$$

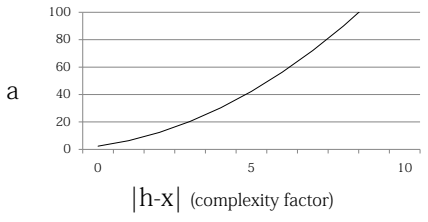


Figure 7: Example attribute function for NT_3

4. CASE STUDY

This section applies the proposed BDS on a non-disclosed BBMG. To approximate a human average behavior, data was gathered from six players. The data was collected on the players' own machine without supervision, but is suffices for the scope of the case study. Also, two bots was written and their behavior was analyzed. Table 2 shows the minutes and game packets recorded from each player and bot.

no	minutes recorded	packets recorded
#1	954	4346
#2	260	3935
#3	827	2795
#4	583	2215
#5	1 429	10253
#6	592	2407
Bot 1	3753	4448
Bot 2	13 318	16702

Table 2: Data basis

The first bot focuses on increasing the resource income rate for advanced players, while the second bot is a base builder specifically made for the first phase of a game.

4.1 Game-specific attributes

The example attributes defined in section 3.2 are used in the case study implementation, as well as two game-specific variables as explained in the following subsections.

4.1.1 GV1 - Resources earned

The main rationale for using a bot in this game is the increased resource production. Obtaining daily income in the value range of that of a bot is very hard or impossible for a human, so monitoring the resources earned per day is helpful in this regard.

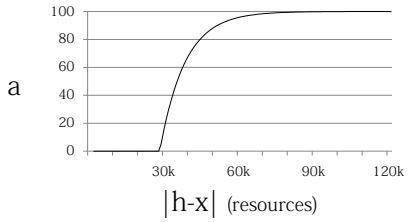


Figure 8: Example attribute function for GV_1

4.1.2 GV2 - Times harvested

Small players using bot will also gain a lot but not as much in absolute value, which is why it is also necessary to monitor the number of times the resource collection procedure is done, not only the total amount of resources earned.

4.2 Results

Table 3 shows the results of the case study for each attribute for the two bots and the human maximum. The bots obtained significant a-values for 6 out of 7 attributes and thus got a average scores of 64 and 52, respectively. The human players on the other hand received a non-zero score on only two of the attributes, where the largest scores were 6 and 5. This shows that these bots are easily distinguished from human players by the use of this bot detection system. Figure 10 shows a graphical representation of the total scores S_{TOTAL}.

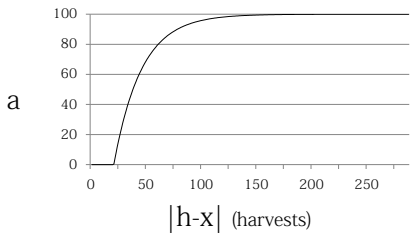


Figure 9: Example attribute function for GV_2

Attribute	Bot 1	Bot 2	Player max
a_{PA1}	39	0	0
a_{PA2}	0	100	0
a_{GV1}	82	13	0
a_{GV2}	92	94	0
a_{NT1}	76	54	6
a_{NT2}	75	64	0
a_{NT3}	70	51	5
S_{TOTAL}	64	52	1

Table 3: Summary of results

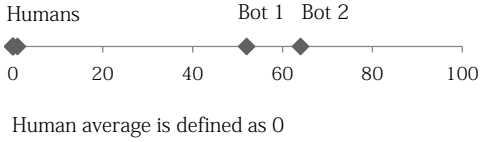


Figure 10: A graphical representation of the total scores

4.3 Discussion

The differences between humans and bots are striking and result in high attribute scores for the bots in all but one attribute. These are PA_2 for bot 2 and PA_1 for bot 1. This is because bot 1 earns a lot of resources daily but doesn't use the resource collection as many times, while bot 2 farms more often but results in a low total because it is used in the early stages of the game.

The most interesting attributes are the ones in category NT. For NT1, the packet interarrival times for human players are more spread out and follow a long-tail curve, while the bots are more clumped together. When it comes to NT2 and the presence of various request types, we see that the bots only use a small subset of the possible requests. This is because they only perform some of the game actions, while the humans use all of them. Table 4 shows the five most

Request type	Bot 1	Bot 2	Human average
Resource collection	98%	82%	15%
Notify	1%	16%	13%
Building construction	0.5%	1%	11%
Get map data	0.04%	0.22%	9%

Table 4: The four most frequent request types

used request types from NT_2 for the two bots and the human average.

Lastly, the packet order also reveals big differences in playstyle. The human playstyle is far more complex and random by nature, whereas the bots are predictable and have a simpler game interaction pattern. Figure 11 shows the request order for bot 2, the most complex bot, modeled as a graph, where each state represents a request type and a transition between two states is the relative frequency of this request type succession. This graph contains only four states while the human equivalent contains over 20 states and 266 non-zero transitions.

This system provides a detailed and thorough system that enables a game company to fine-tune the attributes function to catch certain types of suspicious behavior. The attribute set can also be extended or modified to better suit a specific game. The game used in this case study employs a simple detection scheme where only a few parameters, which are easy to monitor, sure signs of bot use but also simple for the bot makers to circumvent. One example of this is that when the game is updated, some requests may be modified to contain other parameters or that the parameter names changes. A bot client will keep sending the old requests until the bot is updated, and this is a clear giveaway that the request is originated from a bot, as the ordinary game client (web page) would always be updated.

The BDS proposed could easily be extended to include these binary parameters, but the proposed framework can be viewed as the second iteration of a bot detection system because of the increased granularity of behavior awareness and more thorough monitoring.

5. CONCLUSIONS

This paper proposes a bot detection system (BDS) for browser-based multiplayer games (BBMG). The system monitors different attributes of player behavior and groups these in three categories: Player activity, game variables and network traffic. Each player examined will obtain an average score $S_{TOTAL} \in [0, 100]$ that represents the deviation of the player being a bot. In the case study presented, the average devation of the human participants receive a score of 1 while the bots examined receive 64 and 52.

This shows that it is possible to distinguish humans and bot by analyzing their game behavior with this bot detection system. The critical success factor is having enough user data to estimate a human average and accurately define suspicious behavior.

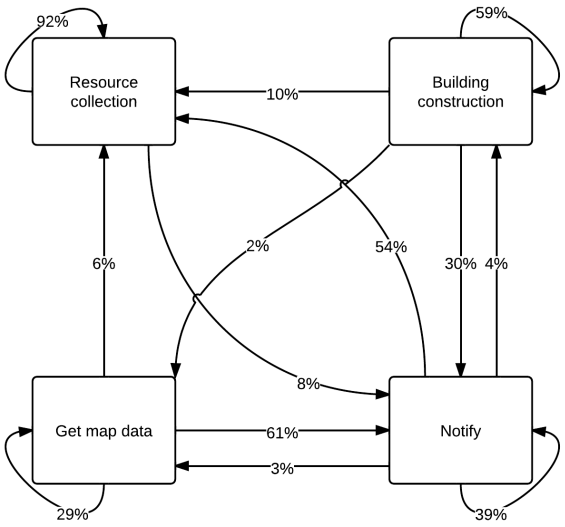


Figure 11: Behavior of bot 2 as a state diagram

6. REFERENCES

[1] GameZone. Why Gaming Is Bigger Than Movies And Music. `www.mweb.co.za/games/ViewNewsArticle/tabid/2549/Article/1474/why-gamings-bigger-than-movies-and-music.aspx` (accessed June 6th 2012).

[2] Kuan-Ta Chen, Andrew Liao, Hsing-Kuo Kenneth Pao, Hao-Hua Chu. Game Bot Detection Based on Avatar Trajectory. `mmnet.iis.sinica.edu.tw/pub/chen08\_gamebot.pdf` (accessed June 6th 2012).

[3] Kuan-Ta Chen, Jhih-Wei Jiang, Polly Huang, Hao-Hua Chu. Identifying MMORPG Bots: A Traffic Analysis Approach. `mll.csie.ntu.edu.tw/papers/bot\_ident.pdf` (accessed June 6th 2012).

[4] C. Platzer. Sequence-Based Bot Detection in Massive Multiplayer Online Games. `www.iseclab.org/papers/ICICS2011.pdf` (accessed June 6th).

[5] Stefan Mitterhofer, Christian Platzer, Christopher Kruegel, Engin Kirda. Server-Side Bot Detection in Massive Multiplayer Online Games. `iseclab.org/papers/botdetection-article.pdf` (accessed June 6th 2012).

[6] Steven Gianvecchio, Zhenyu Wu, Mengjun Xie, Haining Wang. Battle of Botcraft: Fighting Bots in Online Games with Human Observational Proofs. `http://www.gianvecchio.com/uploads/1/0/7/8/10784991/ccs09.pdf` (accessed June 6th 2012).

[7] The ESA. Video Game Industry Facts, 2011. `www.theesa.com/facts/pdfs/ESA\_EF\_2012.pdf` (accessed June 26th 2012).

[8] Uthar. Browser-Bame Bot Tutorial. `bots.uthar.nl/35/mybot-tutorial-part-1` (Accessed June 6th).