Research Article

iMedPub Journals
http://www.imedpub.com/

American Journal of Computer Science and Information Technology

2018
Vol.6 No.3:26

DOI: 10.21767/2349-3917.100026

Detecting Cheating in Computer Games using Data Mining Methods

Alexandre Philbert*

Institute for Information Systems Engineering, Concordia University, Montreal, QC, Canada

***Corresponding author:** Alexandre Philbert, Institute for Information Systems Engineering, Concordia University, Montreal, QC, Canada, E-mail: alexandre.philbert@hotmail.com

Received date: July 23, 2018; **Accepted date:** August 16, 2018; **Published date:** August 27, 2018

Citation: Philbert A (2018) Detecting Cheating In Computer Games Using Data Mining Methods. Am J Compt Sci Inform Technol Vol.6 No.3:26

Abstract

Cheating is prevalent in the gaming industry and is causing several types of uproars. Counter-Strike is a popular franchise which has had up to peaks of 850 thousand players at once playing. Professional events such as ESL One Cologne 2015 had 1.3 million concurrent viewers. Video games are a huge industry, sadly, integrity of the players is lacking. Professional players have been found to be using cheats at events, and cheaters in casual matches are expected to be found regularly. Current anti-cheat systems may use signature-based approaches and even heuristics, but none of them have explicitly stated to be using data mining techniques. Although signature methods may catch a lot of not technically adept cheaters, the ones we need worry about are the ones using metamorphic or polymorphic cheats that stay undetectable to these systems. To detect these kinds of cheats, another method needs to be put in place. Data mining techniques used for detecting zero-day malware as well as player behavior-based techniques for detecting cheaters are discussed in this proposal. The results of these methods are promising and will hopefully rid the gaming industry of ill-minded players.

Keywords: Data mining; Cheat detection

Introduction

Cheating in video games is commonplace and is expected on any and all platforms. A particular genre of game is affected, first-person shooters. This type of game is particular since a cheater has a multitude of ways that they can cheat and can even do so discretely in order to avoid suspicion. Aimbot, wall hack, no recoil, radar hack; these are just a few of the possible ways to cheat in games such as Counter-Strike. Cheaters are so prevalent in Counter-Strike that communities were created such as E-Sports Entertainment Association League. ESEA offers a service for players to participate in matches that are protected with more invasive anti-cheat software. On some days, there can be over 20 bans performed by this software [1-5]; this may seem minimal, but the overall number of players in a day isn't that high on this service. On top of that, a single cheater ruins the match for 10 players at once.

Although it may seem unimportant to some, cheating in video games greatly affects the profits and image of the company in charge. Players spend several hours playing video games to perfect themselves and become skilled. It is not enjoyable to go up against a player that hasn't spent as much time practicing and can beat you unfairly by using cheats. This discourages the player and ultimately makes them stop playing if it becomes too recurrent.

Cheating is not only an issue for casual players. Some people partake in competitions and professional events. Playing the video game is their full-time job. Having cheaters participate in these events without getting caught is an issue that has to be solved. The utmost important events have invasive anti-cheat and process analysis tools to ensure that the players do not have an unfair advantage; but this is part of an arms race [6]. Cheat manufacturers have found ways to bypass such security measures and have stated that some of the professional players today actually do use cheats.

Many anti-malware software currently use machine learning to detect and prevent new malware from causing harm to their client's systems (e.g.: Symantec [7], Kaspersky [8], Microsoft [9], etc.). Data mining techniques have been proven to be accurate enough to secure systems and provide information for experts to analyze more thoroughly when needed. Although this project is not about malicious executable, the same ideas should be applicable for cheat-related executable. Valve, the developers of Counter-Strike among other games, has announced their interest in using machine learning techniques in their anti-cheat software VAC to detect cheaters in their products [10].

There also exists software such as PunkBuster [11] which scans the memory of the player's local machine. As far as it's described online, they seem to be using a signature-based approach.

According to forums, heuristics seem to be used by ESEA's anti-cheat, PunkBuster, and probably many more; but no information is given on whether or not they employ data mining techniques.

Related Work

Background

- **Data mining methods for detection of new malicious executable [1]:** Schultz et al.worked on developing a classifier capable of accurately labeling malicious executable without the need of launching them. Information about the executable is gathered from the binary file itself, such as: used DLLs and function calls, strings found in the binary and byte sequences. The motivation behind this work is that previously used methods, such as the signature method, does not generalize well for new (a.k.a. zero-day) malware. Using the extracted features, they build several models using three algorithms: RIPPER, Naive Bayes and Multi-Naive Bayes. The results are then compared to each other and to a homemade signature method. The highest recorded accuracy was attained with Naive Bayes using the Strings feature: 97.11% with a 3.8% false positive rate. They believe that given more data and by modifying their byte sequence feature to be variable, that they'd achieve better accuracy.
- **Behavior-based cheating detection in online first person shooters using machine learning techniques [2]:** Alayed et al. argued that current cheating detection measures are a breach to the users' privacy. For this reason, they propose a server-sided behavior-based detection of players using data mining methods. To test their idea, they created a simple first-person shooter game called Trojan Battles and cheats to go along with it. The cheats included: Aim lock, auto aim, and auto fire. On top of these cheats, they added mechanisms to make them more difficult to be detected: Slow aim and auto miss. Several features were extracted from the server logs: Mean aiming accuracy, hit accuracy, mean view direction change, fire on aim ratio, fire on visible, time-to-hit rate, etc. The data mining method used had to be appropriate for time-series data since every log came with a timestamp and was only appropriate for a single frame. Extracted features would take into account a certain frame size, which were compared to each other to discover which gave the best results. They conclude by advising to use a separate model for each type of cheat (i.e. Auto aim, auto fire). In general, they get 90% accuracy when anti-detection mechanisms are put in place, and they get >96% for blatant cheaters. Another interesting point about this technique is that it's possible to detect cheaters online and offline using the recorded game logs.
- **Malicious sequential pattern mining for automatic malware detection [3]:** Fan et al. proposed using data mining techniques to extract and detect frequencies of malicious sequential patterns of instructions inside an executable to classify then as malware or as benign. They say that simple obfuscation techniques can easily bypass signature method detection and due to the incredible rate (thousands per day) of newly created malware, it's not an appropriate method to secure systems. To reduce the execution time, they perform feature selection techniques to filter redundant patterns and find those that are most expressive (i.e. Information gain, max-relevance, chi-square test); only the top 100 sequences were kept. In the case that an executable file is packed (i.e. ASPack, PECompact), they first unpack it. Many modeling algorithms were used and compared (i.e. Naive Bayes, Support Vector Machine, J48 Decision Tree). The best accuracy was attained using J48 with their MSPE algorithm for feature selection: 94.90% with a 6.25% false positive rate.

Discussion

The issue with most of these models is that a malicious executable creator could possibly bypass its detection by carefully manipulating their malware (i.e. Strings can be changed or encrypted; DLL could be provided, could be packed and obfuscated). This process is much more difficult when it comes to opcode sequences as seen in C, especially if it we are capable of unpacking the executable before static analysis. Although A also proposed the byte sequence feature, unless the binary is unpacked beforehand, it is unlikely that the sequence of bytes would be able to distinguish between the packer code and the actual executable's behavior; this method had a false positive rate of 6.01%. C extended off of A by using variable length of opcodes instead of the fix length of byte sequences.

A assumes that an executable is malicious if the majority of the strings inside the binary were never encountered before. This probably has an adverse effect on the false positive rate, especially if the software was simply packed for non-malicious reasons (i.e. A payable software that doesn't want people reverse engineering crackers to steal their product). A seemed to have better results than C by using the strings feature, but it's probably due to the fact that packed executable were considered malicious a priori. Given a data set with packed benign executable, it would have a high false positive rate.

An issue with B is when there is lag (a.k.a. Delay in the network). This affects the features being collected and may give false positives or other errors. Also, small frame sizes need to be used since smart cheaters only activate cheats for a small amount of time; this makes it more likely to catch unusual behavior. Also, there are prevalent cheats used in casual matches and online tournaments that are known as a wall hacks and radar hacks (among others, manipulated models and textures, removed smokes and other special effects, etc.). These types of cheats are not as easily detected using behavioral-based features; although it would be interesting to look at the data of an eye tracker, it would infringe the users' privacy and they could be tampered with.

Implementation

Data acquisition

Cheats: We are targeting a franchise of games in particular: Counter-Strike. Since there does not exist any publically available data sets for cheating executable, even less for a particular game title, the first step was to develop a web scraper to acquire an acceptable amount of executable. This web scraper was developed using 'Java' alongside Selenium [12], a testing framework for web applications. It would have been simpler if we could have used tools that would have allowed us to perform

a GET request and simply download without going through the web page's user interface. The issue with this method is that the website had measures mitigate automated bulk downloading of their contents:

URL of download links were not obvious and indirect Must be logged in the website to have access to cheats Throttling was put in place (30 seconds)

Downloads wouldn't start without an actual browser opened and clicking on the links (**Figure 1-4).**

unknowncheats.me/forum/downloads.php?
do=file&id=id&act=down&actionhash=hash

Figure 1: Example of a download URL

The web scraper is composed of three steps:

Log in to a pre-registered account

Figure 2: Login page

Collect the list of all cheats for a particular game and their download page link and store to a file

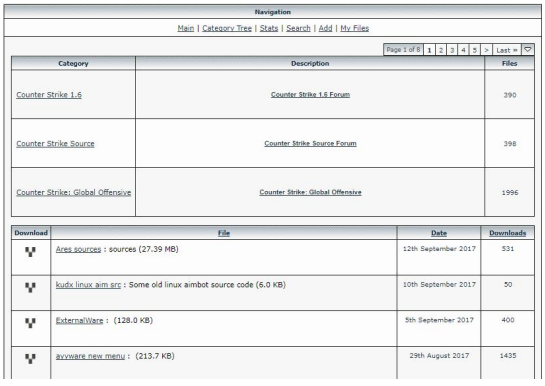

Figure 3: Cheat list page

Go through each download page and click on the download button

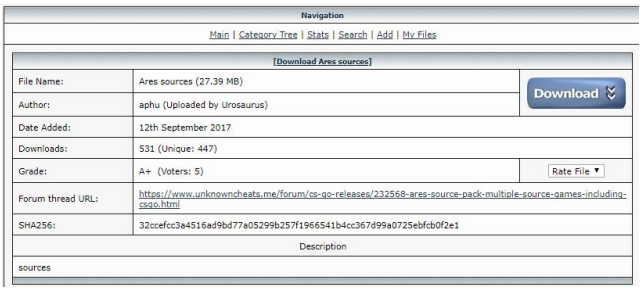

Figure 4: Cheat download page

Some precautions were taken to increase the robustness of the scraper. First, if the throttling wasn't done, it would wait some time before retrying. Secondly, in case it crashed, an index of the previously downloaded cheat was saved so it could resume its execution later [13]. Crashes would happen sometimes if the website was too slow; but later the waiting time was increased and crashes ceased happening (**Figure 5**). Another issue that would arise is that the chrome driver executable responsible for the execution of the scraper would be detected as a false positive by the anti-virus running on the machine; so it got deleted. We disabled the anti-virus for the remainder of the mining operation. We obtained a total of 527 cheats from the Counter-Strike family of games: 1.6, Source, and Global Offensive.

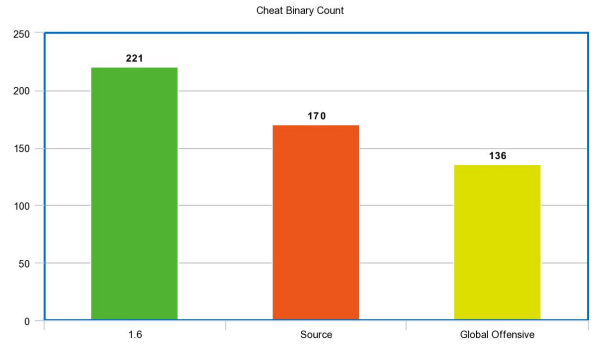

Figure 5: Cheat binary count

Benign: Gathering the benign executable was a much simpler task. A Bash script was made to list all the files on a Windows system, then the list was randomized, then the first 700 files are chosen for feature extraction. These files were checked to ensure that they weren't packed and that they weren't over a certain threshold of size. The threshold itself is not an issue in the case of cheats since they are typically small. Although, having a large file size could be an interesting way to bypass detection since it would take too long to process through feature extraction in the case of a hex dump. Since all the gathered benign programs are typical programs found on a veteran gamer's PC, it represents a comparable environment game developers would expect to run these checks in, which could benefit accuracy.

Data preparation

Similarly to malware developers, smart cheat developers don't want people being able to easily perform reverse engineering tasks on their executable. Software called packers is used to make the task of decompiling or doing any other kind of static analysis slightly more difficult. Another purpose for these tools is to give a new signature to an executable which makes it avoid signature-based detection solutions for malware detection as well as cheat detection (**Figure 6**).

In order to perform feature extraction, we have to first ensure that the binaries are stripped off of their packer code. As mentioned earlier, if we omitted this step, our classifier could be fooled and simply detect packed versus not-packed files. This is not the behavior we are trying to detect since it would have a lot of false positives if we were to encounter packed benign software.

A list of tools gave us the information needed to determine what type of executable we were dealing with:

PEiD: A graphical interface that provided us with information about the packer being used, the section names in the binary, and the entropy of these sections. The entropy is a good indication of whether or not a file was compressed or not.

File: A command-line program which was used in bulk on the entire executable to determine the packers used and the type of application (i.e.: GUI, Console).

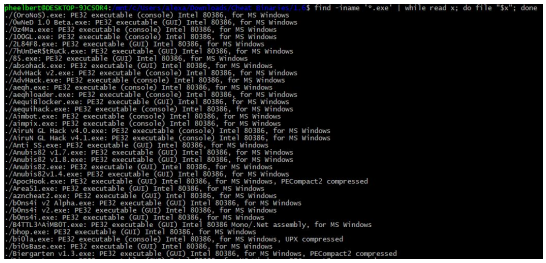

Figure 6: Example of find output

Exescan: A command-line program that gave similar information to the previous two tools. It was used to ensure that the information gathered from the other tools were accurate.

More specifically, we gathered information about which packer was used for each specific file. Using this, we can attempt to unpack them with the appropriate software (**Table 1**):

Table 1: Packer and Unpacker match

Packer	Unpacker
PECompact2	Unpecomp2.exe
UPX	upx.exe

Here is an example of a UPX packed cheat getting unpacked and a comparison of the packed versus unpacked contents (**Figure 7**):

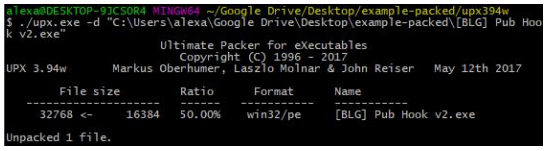

Figure 7: UPX unpacking output

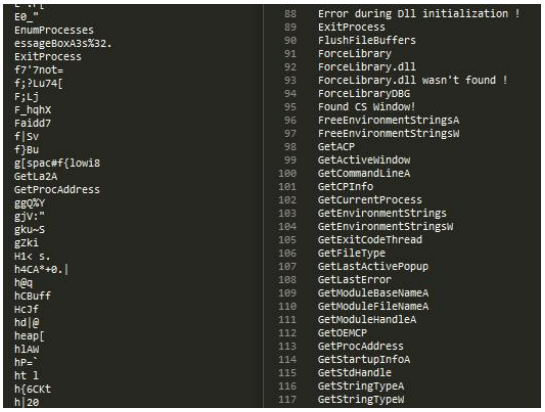

Figure 8: UPX packed versus unpacked strings

As we see in the previous figures, the information we get from an unpacked file is much more useful than one that was compressed and obfuscated by a packer (**Figure 8**).

Extraction

- **Hex dump:** A hex dump is the hexadecimal representation of a file's entire data in memory. We've dumped this information using od for every binary and kept it in a separate file. The main issue with this feature is that the dump itself could be ambiguous; it's unsure whether or not the sequences are meaningful or not.
- **Assembly instructions:** Tools such as objdump are sometimes capable of disassembling an executable into its corresponding assembly instructions. These instruction sequences are the logic of the executable, it shows the control flow, function calls, everything. Problems occurred when attempting to disassemble a PE32+ format executable, which is 64-bit. For this reason, we needed to use another tool called dumpbin.
- **DLL and API function calls:** Executable depends on dynamic-link libraries to interact with the user or with the system. The list of the DLLs and all the function calls being made can be extracted and give a good idea of what the executable is capable of doing. We can get this information using objdump again with the -p option. An issue with this is that sometimes the information isn't extracted properly (e.g. missing function calls).
- **Strings:** Functioning similarly to hexdump, strings will simply filter out what is considered to be characters meant to be read in a file. Information on the DLL and API function calls are also found in this output and it seems to be more robust than the previous method using objdump.

Selection

Several Java programs were coded to extract sequences of bytes and instructions as well as strings from these files. The program gathers the frequency of each sequence or string across all dumps of a specific class cheats for example. A single dump can only contribute a maximum of one frequency [14]. This way, we can find out which sequences or strings are common across the whole class. Given this data, we can then compare it to the other class's results. The idea is to remove any common sequences between the cheats and benign files and only keep the ones who are distinguishing features.

Learning

Several algorithms were used and compared to train on the data set features that were previously collected. Default settings from the R language caret package are used for every model. We did not optimize the parameters since the purpose was to find a good enough model as quickly as possible.

Some of the algorithms were already implemented in R libraries:

- Classification And Regression Trees (CART) k-Nearest Neighbors (kNN)
- Random Forest (RF)
- Support Vector Machine (SVM)

Two simpler algorithms were implemented in R from scratch in order to compare and to learn their intricacies:

Naive Bayes (NB)

A Native Bayes algorithm was developed from scratch in R. In order to make the algorithm more robust, smoothing was done. A small value was added to 0-frequency features and the products were changed to sum of logs to avoid under-flowing. Also, a Gaussian Bayes was tried using the packages and obtained a measly 71% accuracy.

Results

The accuracy of the classifier libraries in R given the features gathered previously yielded some impressive results. Random Forests have proven to be an effective solution when dealing with classifying cheats against benign binaries. Cross validation (10-fold) and predetermined random seeds were used to ensure that the results of the library-provided algorithms weren't due to chance. Random sampling (average of 10-times) was used for the algorithms that were implemented.

Counter-Strike: 1.6 is the game that had the most samples and is also the one with the best accuracy for the specialized model. In practice, it doesn't make much sense to have a model capable of classifying cheats for multiple games; we did it here to see if there was a correlation between them for educational purposes.

Counter-Strike: Source had the most false negatives, about 6 times more than the other two games individually.

The strings features are the most accurate of the two methods. This is expected since the hexdump method might be taking slices of the hexdump that isn't appropriate. To attain a higher accuracy, one could test out several slice sizes and possible offsets (**Table 2**).

		Accuracy			
Algorithm	Feature	1.6	Source	GO	All
CART	Strings	96.19%	92.39%	92.80%	89.28%
kNN	Strings	96.40%	92.95%	94.19%	92.80%
RF	Strings	99.37%	96.64%	98.95%	97.76%
SVM	Strings	98.52%	95.64%	97.44%	96.88%
NB	Strings	94.52%	73.13%	83.22%	79.39%
CART	Hexdump	88.93%	87.84%	90.51%	80.07%
kNN	Hexdump	92.01%	88.86%	91.23%	84.81%
RF	Hexdump	94.36%	90.89%	93.45%	90.21%
SVM	Hexdump	92.44%	89.08%	88.76%	87.07%
NB	Hexdump	82.83%	75.25%	67.59%	59.35%

Table 2: Comparison of accuracy of features and algorithms

Confusion Matrix (All, RF)			
	Actual		
Prediction	Benign	Cheat	Accuracy
Benign	710	14	98.07%
Cheat	14	513	97.34%
Total	724	527	97.76%

Table 3: Confusion matrix for random forest with strings

To better visualize the data in a crude manner, we reduced the string features to simply be the sum of frequencies for each class. The accuracy of this model was 6% lower than the full-fledged bag of words (**Table 3**). Nonetheless, being able to visualize the data set is helpful to understand the results better. As shown in tables, there is an overlap between the cheat and benign executable. More specifically, there is a dense group of benign executable that have low benign feature frequencies and much very little cheat features. Also, the plot was scaled so it would look nicer, but some benign executable had a large amount of cheat features and very little benign ones (**Figure 9**).

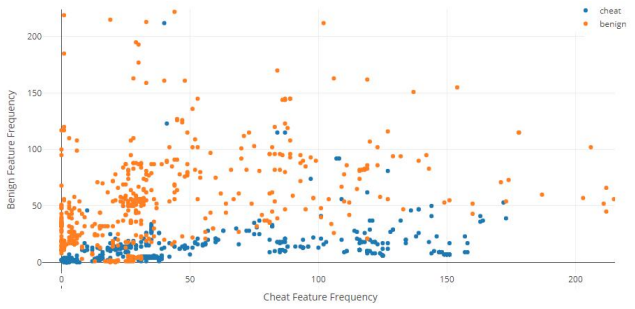

Figure 9: Reduced dimension strings feature set

Conclusions and Future Work

Cheating is so prevalent in the gaming industry and the interest in cheats is as well. New cheats are being developed all the time and only a data mining solution seems possible to counteract these facts. As mentioned before, signature-based methods aren't capable of detecting new cheats and are easily

bypassable. Although, due to the base rate fallacy, the 98% detection rate might not seem that great, this method of detection could be used in conjunction with other signature-based and behavior-based methods as well as white lists for known false positives. The action taken when cheat detection is made could also be a simple warning, which would allow the user to close that program and then continue playing. To conclude, the features and models used in this paper could be improved, even though the results are already acceptable as is.

Glossary

Executable: A file that can be executed on a computer (e.g. typically .exe file extension on windows, but can vary).

Binary: Short-hand of binary file, which is a file that is only interpretable by a program which knows its structure (e.g. Human-readable file is one which contains text in the form of ASCII characters).

Malware: An executable which performs malicious behavior (e.g. encrypt all the files on a system; log all your key strokes, etc).

Benign executable: An executable which performs the expected non-malicious behavior (e.g. Executable that are present on a fresh copy of windows).

Cheat: Un-intended changes in behavior to gain an advantage in a video game by tampering files or injecting code. Most cheats are in the form of executable and are typically referred to as hacks.

Dynamic-link library: Microsoft's solution to share libraries across programs. Cheat developers perform DLL injection attacks to change the behavior of a video game.

Opcode: Abbreviation of operation code which is a machine language instruction that can be found by disassembling an executable (e.g. push, pop, mov, add).

Packer: Short-hand of software packer, which are used to compress executable and to slightly increase the difficulty of reverse engineering tasks such as disassembling.

API: Short-hand of application programming interface is the interface that programmers use to interact with a system (e.g. An operating system).

References

1. Schultz GM, Eskin E, Zadok E, Stolfo JS (2001) Data Mining Methods for Detection of New Malicious Executables. IEEE Security & Privacy.
2. Alayed H, Frangoudes F, Neuman C (2013) Behavioral-Based Cheating Detection in Online First Person Shooters using Machine Learning Techniques. IEEE Conference on Computational Intelligence in Games.
3. Fan Y, Ye Y, Chen L (2016) Malicious sequential pattern mining for automatic malware detection. Expert Systems with Applications 52: 16-25.
4. Counter-Strike: Global Offensive - Steam Charts. steamcharts.com/app/730
5. ESEA. play.esea.net/index.php?s=support &d=ban_list&type=1
6. Lahti E, CS:GO competitive scene in hacking scandal, 3 players banned. pcgamer. www.pcgamer.com/csgo-competitive-scene-embroiled-in-hackingscandal-as-three-players-are-banned/
7. Symantec.com., www.symantec.com/security_response/writeup.jsp?docid=2016-051811-2400-99 &tabid=2
8. Usa.kaspersky.com., usa.kaspersky.com/small-to-mediumbusiness-security/endpoint-windows
9. docs.microsoft.com/en-us/windows/threat-protection/windowsdefender-antivirus/configure-protection-features-windows-defender-antivirus
10. Valve to use Machine Learning to detect CS:GO cheaters - KitGuru. www.kitguru.net/gaming/matthew-wilson/ valve-to-use-machinelearning-to-detect-csgo-cheaters/
11. En.wikipedia.org. PunkBuster. en.wikipedia.org/wiki/PunkBuster
12. Strike-cs.my1.ru.strike-cs.my1.ru/cheats.jpg
13. Assets.change.org. assets.change.org/photos/2/wt/ee/WUWtEEQwkPShTSh-800x450-noPad.jpg?1486292297
14. Oldschoolhack.me., www.oldschoolhack.me/hackdata/screenshot/7a50440099cc1947d08853d294f73cf1.jpg