Malicious Game Client Detection Using

Feature Extraction and Machine Learning

Spencer J. Austad

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Electrical & Computer Engineering

Brigham Young University

Malicious Game Client Detection Using Feature Extraction and Machine Learning

Spencer J. Austad
Department of Electrical & Computer Engineering
Master of Science

Abstract

Minecraft, the world's best-selling video game, boasts a vast and vibrant community of users who actively develop third-party software for the game. However, it has also garnered notoriety as one of the most malware-infested gaming environments. This poses a unique challenge because Minecraft software has many community-specific nuances that make traditional malware analysis less effective. These differences include unique file types, differing code formats, and lack of standardization in user-generated content analysis. This research looks at Minecraft clients in the two most common formats: Portable Executable and Java Archive file formats. Feature correlation matrices showed that malware features are too complicated to analyze without advanced algorithms. The latest machine learning methods for malware analysis were employed to classify samples based on both behavioral features generated from running samples in a sandbox environment and static features through file-based analysis. A total sample set of 92 files was used and found that Portable Executable and Java Archive files have significantly different feature sets that are important for malware identification. This study was able to successfully classify 77.8% of all Portable Executable samples 84.2% of all Java Archive samples while maintaining high recall scores. This research, by shedding light on the intricacies of malware detection in Minecraft clients, provides a framework for a more nuanced and adaptable approach to game-related malware research.

Keywords: malware detection, cybersecurity, Minecraft, game mods, machine learning

Acknowledgments

I would like to express my sincere gratitude to my thesis chair, Dr. Justin Giboney, for his invaluable guidance, support, and unwavering commitment throughout this journey. I am also deeply thankful to my committee members for their insightful feedback and expertise. Additionally, I extend my appreciation to the Department of Electrical & Computer Engineering and the Information Systems Department for providing essential resources that facilitated the completion of this thesis.

Table of Contents

List of Figures

List of Tables

1 *Introduction*

The popularity of video games is enormous with games like Grand Theft Auto 5 (GTA 5) hitting 165 million sales by 2022 (Clement, 2022). Another popular game, Minecraft, has sold over 238 million copies through July 2021 (Clement, 2021). User-made extensions, or mods, for these games are also immensely popular. Modding is the process of making alterations to a game generally by fans or players of the game. One popular repository for mods, GTA5-mods, hosts tens of thousands of modifications for GTA 5 with hundreds of millions of downloads (GTA5-mods, 2022). Similarly, Minecraft has attracted thousands of unpaid developers to make extensions and clients for the game (Curseforge, 2022).

Because of the large player base and evident popularity of user-made extensions, it has already attracted malware authors using video games as a delivery mechanism for malicious software. Cisco Talos (talosintelligence.com) recently discovered a large campaign to push out malware using modding tools for two games, Point Blank and Crossfire. While these tools promised to provide tweaks to the game, they also installed a backdoor trojan known as XtremeRAT (Unterbrink, 2021). A survey from July 2020 to June 2021 found that Minecraft was the most infected game with game-related malware (Kaspersky, 2021). This data from the Kaspersky Security Network found over 184,000 infected users across 3 million detections for Minecraft-related malware. They report that in the top 5 most infected video games, 65% of affected users were Minecraft players.

Many studies have looked at key indicators for malware on infected machines (An et al., 2018; Canzanese et al., 2013; Liang et al., 2016). This study aims to add to malware research by finding key indicators for Minecraft clients, or launchers. A large portion of Minecraft launchers are in compressed Java archive files called JAR files. Research on malware indicators in JAR files has been limited, partly due to the complexity involved in decompiling these files back into Java code. Additionally, game clients present unique challenges from other forms of malware. Additionally, clients often advertise that they may trigger anti-virus notifications, especially clients that purport to give players hacks. This research will be beneficial for companies and institutions trying to catch video game related malware in user uploaded content. For this research, I intend to find what key indicators are indicative of malware in Minecraft clients.

Research Question: What are the prominent indicators of malware in game clients?

To answer the research question, I will look at current and past research to identify potential indicators of malware infection. I will then apply these indicators to video game mods to develop a set of prominent indicators to help identify infected video game mods. Finally, I will utilize machine learning algorithms to develop a model for detecting malware in the mods. This method was used by Smutz and Stavrou to detect malware in PDFs (Smutz & Stavrou, 2012).

2 Literature Review

2.1 Defining Clients

A client is a method for accessing a video game and is usually required for modern games. They are important because they allow for licensing and user-based authentication. Minecraft ships with its own launcher developed by the publisher Mojang. Often users want to extend the functionality of these launchers for means of modifying game mechanics, making routine tasks easier, or providing player hacks to give advantages in multiplayer servers.

Third-party Minecraft clients are a subset of mods and are different than in-game mods. In-game mods require a client, either a third-party or official client, and impact the gameplay as opposed to the authentication and start-up procedures of the game. Clients do not require a specific version of Minecraft to already be installed, whereas in-game mods do have specific version requirements.

Developers that alter the original game code have not always been in good favor with the game publisher. In 2017, TakeTwo, the publisher behind Rockstar's GTA V, delivered a cease and desist to the largest tool for modifying in-game API behavior, OpenIV (Livingston, 2017). Although TakeTwo eventually withdrew their legal complaint, it sent shockwaves through the community. Not as lucky, a group of developers who called themselves Apeiron had been developing a total conversion remake of the 2003 Knights of the Old Republic (KOTOR) for two years (Grayson, 2018). In 2018, Lucasfilm Ltd. ordered a cease and desist for the remake. Despite making no profits from the conversion and with no plans to generate revenue, the team had no choice but to accept the order and cease production.

Although game studios have a right to protect their content, developers of game-enhancing clients are fearful of repercussions and often seek anonymity. Most clients are made by unknown authors that only go by a username unlike the OpenIV tools and the KOTOR remake. Anonymity is great for malware attackers because it limits the ability of researchers to identify where malicious software may have come from and how it spread. When obtaining relevant video game malware, it will be necessary to try and document where it came from, but this can be made difficult due to the anonymity of those who publish altered game clients.

2.2 *Malware in clients*

Understanding why clients are created is important in understanding what attack vectors malware authors may use when writing malicious game clients. These developers are usually not employed by game studios and work on their software without pay (Poor, 2014). The disconnect between developers of game extensions and the publishing developers leaves room open for malicious app development. Any verification of game clients will have to be done at the repository level for a website hosting them.

Modding in Minecraft remains a popular reason users install third party clients. The developer of Minecraft, Mojang, does provide their own launcher, and you can add mods to it. However, it does not provide useful tools that other clients do, such as the ability to turn on and off mods with a simple GUI, so that the user doesn't have to touch the underlying file system. Another popular tool that clients provide are offering modpacks, simple ways to install a pack of many curated mods at once. This simplicity can be beneficial to many Minecraft players, but recent events have proved this can be harmful. In June 2023, malware called Fractureiser was found to be distributed through CurseForge (`www.curseforge.com`) and bukkit (`www.bukkit.org`) (Goodin, 2023). Malware propagation was primarily attributed to a few modpacks that users installed through the CurseForge launcher, such as the Feed the Beast (FTB) modpack. It is believed that this recent malware infestation affected around 6,500 users (Croft, 2023). Due to the spread of malware in Minecraft software, it is imperative that methods for detecting this type of malware are implemented.

2.3 *Cheat Detection*

Video game cheats that give an unfair advantage to players by modifying the game code are essentially unauthorized modifications and share similarities with malware. Many companies employ cheat detection software that looks for cheats and functions like anti-virus applications. Many methods used in cheat detection are applicable to this study for finding prominent indicators of malware in clients. Authors of cheat detection software are looking for process injection, unusual network communication from the game, and abnormal software behavior.

A literature review of cheat detection found five major types of detection methods: behavioral, verification, result, reputation, and hardware (Björkskog, 2019). Behavioral detection compares known cheaters with current players and looks for similarities in play style and behavior. Verification detection looks at the game state for illegal actions such as having a larger viewable area than allowed. Result detection assumes that players should be predictable and when dramatic changes in ability are noted over a short time, it is flagged. A newer method of detection is reputation based and has trusted clients act as referees for other clients. Fake data may be sent to clients, such as an invisible player. Cheat software may be able to detect this invisible player and react to it while trustworthy players will not. Lastly, hardware detection is perhaps the most common method and relies on monitoring CPU state and running processes. It is the most intrusive detection method.

Out of the above method detection methods, hardware detection is the most useful when looking at malware in game clients. A study looking at methods for detecting video game injector exchange notes that the "cornerstone" to making cheats is using a stealthy memory injector to insert the code into the running game process (Karkallis et al., 2021). Given the similarities between game cheats and malicious clients, looking for process injection will be of paramount importance when finding indicators for malware in Minecraft clients.

Cheating in video games is a constant game of hide and seek between cheaters and developers. As the developers get better at detecting running processes and injections, writers of cheat software get better at hiding them. Kanervisto et al. (2022) looked at methods for detecting cheats using machine learning instead of relying solely on memory analysis. This approach offers some insights into research about prominent indicators of malware in clients. Instead of only relying on what can be seen while the client is running, developing a holistic machine learning model for malware activity will likely be more effective.

Cheat detection has also caused security problems, for example, Capcom rolled out measures to detect cheating in Street Fighter V by implementing a kernel-level driver for monitoring running processes and injected code (Williams, 2016). This resulted in a severe rootkit due to poor programming and would allow attackers to inject any arbitrary code in the game's now kernel-level software. Although no known attacks were used during the period this driver was running, it shows how games can be leveraged to escalate access.

2.4 *Anomaly Detection*

Building a machine learning model for detecting video game malware requires looking for anomalies in infected systems. A 2018 study analyzing malware for the Echo device by Amazon found differences in the Linux kernel system calls using a one-class SVM-based algorithm (An et al., 2018). There were many unique system calls in the infected malware that did not show up in the clean systems. Additionally, they generated a pattern of calls based on the frequency and call number. This allowed their algorithm to quickly and accurately predict which samples contained malware.

An older study looking at Windows XP was able to show that malware infected systems have a higher number of NTQuery system calls than uninfected systems (Canzanese et al., 2011). Based on the IoT system call analysis and this Windows system call analysis, this is an excellent indicator to look at when researching malware.

A similar research study looking at IoT devices used a machine learning algorithm for detecting differences in network activity to find Mirai malware researchers aggregated packets captured across the network (Nguyen et al., 2018). They assigned each packet a symbol based on the characteristics and were able to detect malware with 95.6% accuracy and no false positives. This method of using machine learning to look at network packets will be a useful indicator to investigate clients that may contain malware.

2.5 *Non-signature Malware Detection*

Signature-based detection for malware has limited effectiveness when encountering novel malware. If the malware author changes the code enough that the signature changes, it will be more difficult to detect by routine scanners such as VirusTotal. One method for doing this involves extracting strings and analyzing them. A study of Windows malware extracted DLL calls from binary files as well as the function calls and number of calls that these DLLs appeared to make (Schultz et al., 2000). They found a significant difference in the number and type of DLL calls in infected systems compared to clean systems. Looking at DLL calls in client binaries could be an indicator of malware.

2.6 *Behavioral Detection Summary*

2.6.1 Network traffic

Analyzing packets individually is a complex task in modern malware analyses because of the extreme volume of network traffic on most modern devices. Based on research for a tool called MalAlert, some aggregate features can be extracted to detect the presence of malware. In this 2019 study, they showed that the quantity of bytes sent was an important feature in identifying malware compared to a baseline (Piskozub et al., 2019). Additionally, they showed that port features were important and aggregated two different sets of ports, those between 1 and 49152 and 49153 through 65535. These were compared against the 10 most common ports for abnormalities. Malware is more likely to be present when there is non-HTTP traffic being set over ports 80 and 443 and HTTP traffic is being sent over non-HTTP ports (Zhao et al., 2015).

Additionally, it has been shown that DNS requests and geographical regions can establish a trust measure that aids in detecting malware. Requests that are reaching untrusted domains more infrequently may indicate malware. Zou et al. (2015) showed that malware was more likely to be transmitted over DNS requests to untrusted sources. Although their model is more complicated than will be used in this thesis, it provides evidence that looking at trusted and untrusted regions for DNS requests is an important aspect of identifying potential malware.

2.6.2 System Calls

Malware has grown increasingly complex, and many forms of malware have detection capabilities that detect analysis engines and can terminate themselves. Although on the surface this can appear detrimental for analysis, this can be useful as applications that terminate a significant number of processes may indicate malware presence (Oyama, 2018) This research article shows that the number of API calls involved in process termination can be an indicator of malware presence. Additionally, the number of processes started also is a potential indicator of malware. This can be a tool that malware uses to hide itself by starting child processes, especially ones that normally look benign, such as explorer.exe.

Furthermore, research has shown that the context behind malware behavior in process generation is important. Wang et al. (2020) explained that the number of processes started as well as the type can indicate malware presence. Looking at the process tree is an important tool to identify potential malware. For example, a word editing program called texteditor.exe would appear less suspicious if it started a subprocess for an email application such as Outlook. It would appear more suspicious if the texteditor.exe application started a command terminal of Powershell, suggesting that it might be executing code.

A 2018 study on ransomware behavior showed that a large portion, 6 of 9, of the malicious API calls were directed at filesystem operations (Hampton et al., 2018). These operations included directory scans, requesting file types and sizes, and read and write operations. The malware significantly exceeded call rates of normal system operations. Due to the potential for malware to make an exaggerated number of file system API calls, the frequency will be an important metric for malware detection.

In addition to frequency of file changes, the locations of changes are important because changes to some areas of the file system may be more likely to indicate the presence of malware. For instance, a study looking at the file operation location and noted some areas, such as startup directory modifications, were more likely to indicate malware (Aslan & Erdal, 2022). Additionally, they showed that the frequency of registry changes and the location of registry changes, particularly those that relate to dynamically linked library (DLL) locations were indicative of malware. Given this information, it will be important to look at both the frequency and location of both file system and registry API calls.

2.6.3 Persistence

The goal of malware is usually beyond a one-time code execution. Attackers usually want to maintain some sort of access or control over a system. Because of this, malware authors try to create persistent code. One important study about indicators of persistence looked at changes to the Windows startup directories and registry location for startup applications (Aslan & Erdal, 2022). They found that a particular piece of software was more likely to be classified as malware if it was making modifications to these areas.

Additionally, it is useful to look at executables accessed by suspicious software to determine if it is trying to remain persistent. A recent article on persistence methods highlighted that access to the binaries Reg.exe, Nslookup.exe, Regasm.exe, Runas.exe, Schtasks.exe, and Sc.exe indicated a strongly likelihood of malware trying to become persistent on a Windows machine (Barr-Smith et al., 2021). For this research, access to startup file locations and calls to these binaries will be monitored.

2.6.4 Memory

Looking at how an application behaves in memory is crucial to understand whether it contains potential malware. One of the most proven methods of looking at in-memory behavior for Windows machines is looking at DLL calls. Normal system usage generally has uniform DLL access, which means that no one DLL is called significantly more than another (Matsuda et al., 2020). This same article found that malware typically shows a significant amount of DLL calls to one particularly linked library, such as user32.dll. While the specific DLL called is not always the most reliable indicator of malware, the frequency can be a feature for malware. This thesis will look at DLL frequency to aid in malware detection.

2.6.5 Hardware Metrics

High level overviews of system resources have been shown to be an indicator of malware. A recent study looking that the Colonial Pipeline ransomware hack was able to show a method for detecting malware based on irregularities in system resource utilization (Kim & Park, 2022). Compared to baseline CPU utilization, malware showed more CPU usage over the same interval of time. Additionally, they showed that the patterns of RAM usage were different, although not necessarily higher due to both the control sample and the malware sample hitting the maximum RAM allocation at times.

A similar conclusion was reached when investigating cryptojacking, stealing resources for crypto mining, since it involved irregular CPU and RAM usage (Naseem et al., 2021). Particularly, this research noted that malware could throttle the CPU to avoid detection for high CPU usage. While this may work if only examining for high CPU usage, they suggest an approach that looks at CPU usage patterns compared to baseline. For this research, it will be important to look at CPU patterns, including high or low frequencies and usage, rather than just focusing on one aspect.

These five feature categories are summarized in Figure 1. along with their respective specific features for each category. The behavioral analysis for PE and JAR files is consistent across both file types based on the referenced studies.

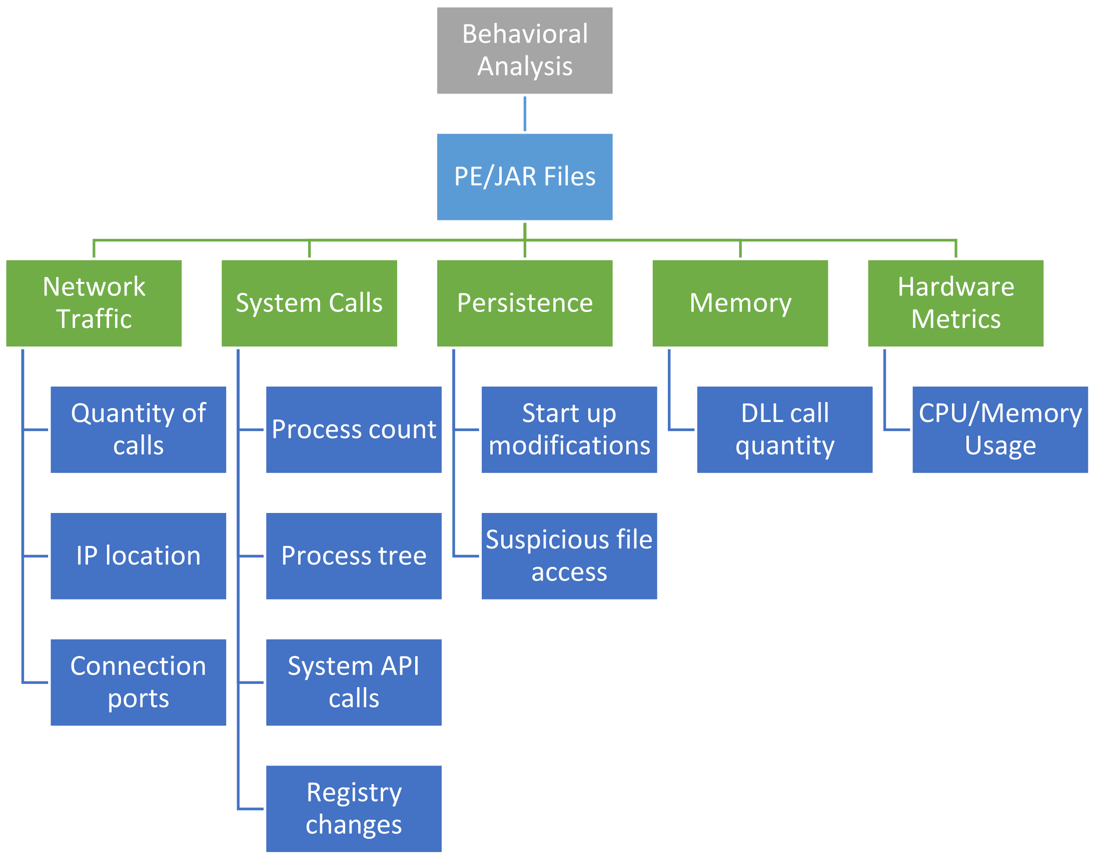

Figure 1: Summary of behavioral data collection features

2.7 Static Detection Summary

Static analysis, when combined with behavioral analysis techniques, is a powerful method of determining the behavior of potentially malicious software. The most basic method for performing static analysis involves looking at the strings contained within a file. One common marker of malware is that it often contains many DLL calls (Ahmadi et al., 2016). Based on this 2016 research by Ahmadi et al, they conclude that the number of DLL calls in a portable executable (PE) file is a useful metric for determining malware. Furthermore, looking at the packer signature of a PE is important because it can help provide useful information for obfuscated malware (Yuk & Seo, 2022). Research has shown that when a packer is used, it can be an indicator of malicious PEs (Shafiq et al., 2009).

Like behavioral analysis, the quantity of network calls is an important metric for determining if a sample is potentially malware. A 2011 study on new ways to look for malware showed that a raw increase of IP addresses and domain names could be an indicator of malware (Nadji et al., 2011). Since IP addresses and URLs also can be detected through string analysis, this can be another metric considered in static file analysis.

Inside the PE header information can be extracted about API calls that the program makes through imported functions. These functions are specific linked library function calls from various Windows DLLs. The characteristics of these calls can help identify malware because researchers have shown that certain types of imported function calls are more likely to be indicators of malware (Vyas et al., 2017). While the presence of these does not indicate malware by itself, the quantity of suspicious function imports can help establish a pattern for determining malware presence. Figure 2 summarizes the features that will be used in the static analysis of Minecraft clients.

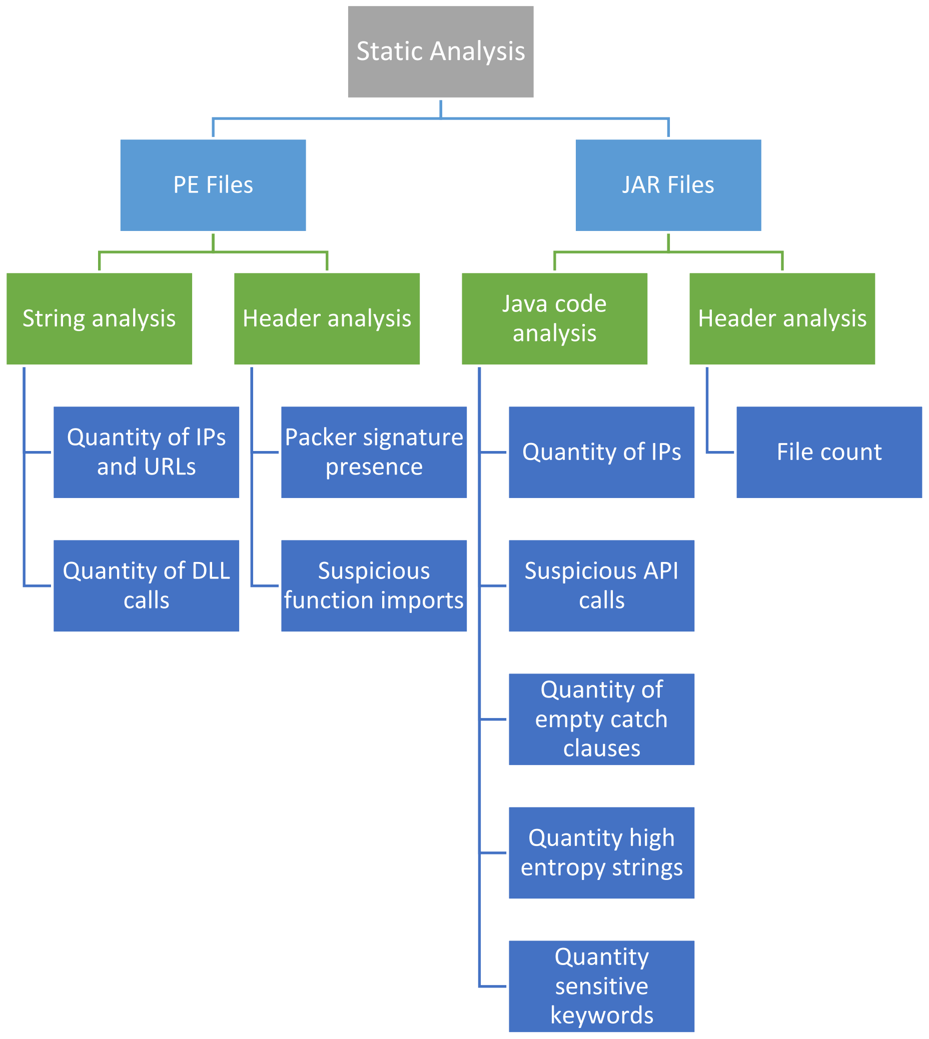

Figure 2: Summary of static data collection features

Minecraft clients generally come in two main file types, PE and JAR. PE files are generally compiled from C# or C++ code, while JAR files are written in Java. JAR files require different analysis because the format and function of the files is slightly different. Some general features, however, are shared between PE and JAR analysis, such suspicious function calls. In addition to IP addresses, effective analysis of Java code requires looking at suspicious API calls (Ladisa et al., 2022). These may range from file reads and writes to base64 encoding/decoding. While the presence of one or two suspicious API calls won't determine if a sample is malware, it helps establish a pattern of behavior. Ladisa et al. (2022) also noted that many of the suspicious API calls will result in an exception as malware is attempting to check its permissions scope. To not alert users of this, malware authors will often use empty try catch blocks to avoid user detection, making the quantity of empty catch clauses another useful metric for analyzing Java code statically.

The presence of high entropy strings can help indicate the presence of obfuscated code. Ladisa et al. (2022) used the Shannon entropy of each string to determine if a given string was a high entropy string. This approach involves a mathematical calculation to determine how random a group of characters in a string is (Shannon, 1948). Furthermore, the researchers applied a Kullbac-Leibler divergence metric to calculate relative entropy for smaller strings (Ladisa et al., 2022). In addition to entropy of strings, they also looked at the quantity of sensitive keywords in the code.

3 Method

There are three major parts to this study shown in Figure 3. I obtained a set of Minecraft games clients that are malicious as well as a set that are non-malicious. Using these two different types of clients I extracted a set of features from them using both static and behavioral analysis methods. I then used several methods of feature selection to narrow down the feature set based on correlation matrices. Using the data extracted from these features, I ran it through various machine learning classification techniques to discover which features are most prominent and create a model to recognize malicious and non-malicious clients. These methods were based on similar methods used in malware detection studies using machine learning to identify prominent malware features (Sgandurra et al., 2016; Tabish et al., 2009).

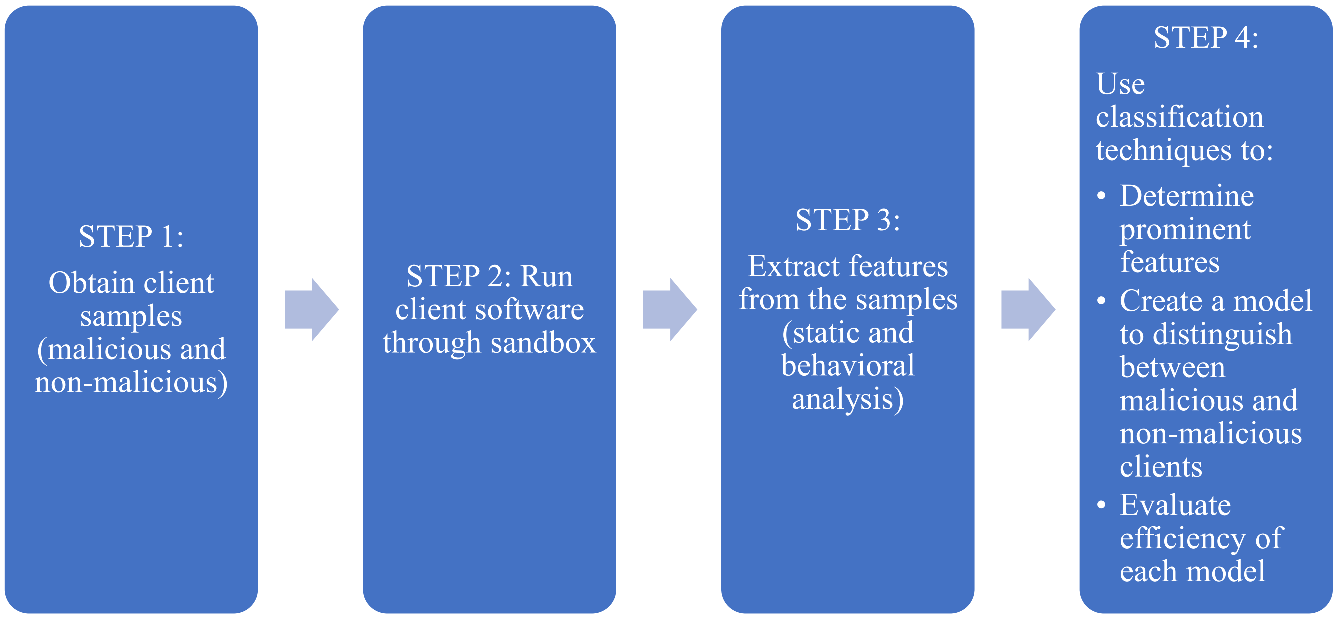
STEP 1: Obtain client samples (malicious and non-malicious)
STEP 2: Run client software through sandbox
STEP 3: Extract features from the samples (static and behavioral analysis)
STEP 4: Use classification techniques to:
• Determine prominent features
• Create a model to distinguish between malicious and non-malicious clients
• Evaluate efficiency of each model

Figure 3: Summary of Methods

3.1 STEP 1: Obtain Malware Samples

This study used the Java version of the game Minecraft to look at malicious clients given the available quantity of clients for this game and that it is one of the largest attack vectors for video game malware. I obtained malware samples of infected clients for the game Minecraft, usually in the format of

JAR (Java archive) files, although they are also packaged as portable executables too. Additionally, I found Minecraft clients that are generally known to be safe and widely used, which will later help distinguish between malware and non-malware. All samples were uploaded to VirusTotal to ensure non-malicious samples were not malware.

Malware samples were obtained from multiple sources. The first source, especially for obtaining malicious samples, comes from VirusTotal advanced search. On VirusTotal advanced search, I filtered relevant samples by first using the “type” search modifier, which allows a specific filetype to be specified. For these searches, “JAR” files, or Java extensions, are how Minecraft clients are typically packaged. All files were also filtered with the “positives” search modifier, with searches conducted looking for at least one or five positives or more. The “metadata” search modifier looks for any file metadata that contains a particular word or phrase. The “content” search modifier looks for any word or phrase matches anywhere in the sample information page.

Minecraft clients may also purport to be able to automatically install mods for the game, so results matching patterns for typical mod platforms, such as Forge, Bukkit, Spigot, and Fabric are included. Table 1 shows the queries made and number of samples collected for each query from VirusTotal.

Table 1: Summary of VirusTotal searches performed.

Search Query	**Number of samples collected**
type:jar metadata:"minecraft" positives:5+	44
type:jar metadata:"minecraft" positives:1+	11
type:jar content:"minecraft" positives:5+	8
type:jar content:"forge" positives:5+	37
type:jar name:"forge" positives:5+	5
type:jar name:"bukkit" positives:5+	3
type:jar name:"spigot" positives:5+	3
type:jar name:"fabric" positives:5+	2
type:jar name:"optifine" positives:5+	1
type:jar content:"fabric" positives:5+	7
type:jar content:"bukkit" positives:5+	13
type:jar content:"spigot" positives:5+	10
type:jar content:"paper" content:"minecraft" positives:5+	3
Similar to queries	1
Total	**148**

Relevant samples were collected based on the following criteria. Files were first categorized based on their function by searching for the original source or a content creator showcasing the software. All obtained software in the sample set can be classified into two major categories, clients and mods. Clients can further be broken down into 68 normal clients and 24 cheat clients, the distinction being cheat clients are meant to provide in-game hacks. In-game mods are not being used for analysis in this research since they do not fall within the scope of client mods. Each software included in the sampleset was manually researched through a combination of Google, Youtube, SpigotMC (https://www.spigotmc.org), CurseForge (https://www.curseforge.com), and other Minecraft software repositories. The purpose of this research was to identify obtained samples as being clients and not another type of mod for Minecraft, such as in-game mods. Samples that could not be identified using any methods were discarded because they could not be verified to be valid client samples for this dataset.

In addition to VirusTotal, client samples were found manually using common Minecraft software repositories, such as 9minecraft (https://www.9minecraft.net/) and cheatermad (https://cheatermad.com/minecraft). Non-malicious samples were discovered through forums and discussion posts that referenced clients individually, for example Lunar Client and Technic Launcher. All these samples that were found manually were uploaded to VirusTotal to assemble more information on the number of submissions and submission dates and ensure that these files were non-malicious samples. Both malicious and non-malicious samples were aggregated for later use in the machine learning process.

Many of the samples could not be adequately identified as either a mod or a client and those samples were not used in this analysis to ensure that the data collected does not go out of scope. Samples were obtained from VirusTotal submissions starting in September 2006 and ending in June 2023. Most samples were uploaded to VirusTotal between 2022 and 2023.

Table 2 gives details on the quantity of samples that were submitted for each of the given years.

Table 2: Distribution of samples by first uploaded date to VirusTotal.

Year	Malicious	Non-malicious
2006	0	1
2015	1	0
2016	0	1
2017	0	1
2018	1	1
2019	1	0

2020	5	2
2021	8	5
2022	23	10
2023	17	15
Total	56	36

The samples averaged a file size of 24.08 MB and a median file size of 7.19 MB, with the largest sample at 148.06 MB and the smallest at just 0.005 MB. The average amount of user submissions was 8301.7 submissions per file and the median was 33, with the highest being 679802 and the lowest at only 1. Table 3 summarizes this data for both malicious and non-malicious software. Submissions refer to the amount of unique user uploads to VirusTotal and detection refers to the number of vendors that flagged files as malicious.

Table 3: Sample statistics of file size, total submissions, and detections.

	Mean	Median	Min	Max
Malicious File Size (MB)	21	7	0.05	147
Non-malicious File Size (MB)	29	7	0.01	148
Malicious Submissions	498	17	1	10605
Non-malicious Submissions	20341	400	1	679802
Malicious Detections	7	5	1	39
Non-malicious Detections	0	0	0	0

3.2 STEP 2: Malware Sandbox

A sandbox environment allows for behavioral testing of malware samples and is the key element that will provide much of the data needed for the analysis in this thesis. According to StatCounter, Windows 10 is the most popular operating system for Windows platforms, covering over 71% of the Windows market share (StatCounter, 2023). Because of this, Windows 10 was chosen for the malware sandbox operating system. The sandbox environment also needed to run interactively, meaning the user can interact with software samples. The environment also needed Java 8 available to run executable JAR files. The service Any.run (`https://any.run`) met all the requirements for data collection as well as the ability to run on a Windows 10 64-bit operating system interactively.

The general process of running samples involved using a template configuration profile for the analysis VM so that each run was kept consistent. The behavior of launchers can vary and require manual interaction. The configuration for Any.run was set to start the object from the Desktop directory with a maximum run time of 660 seconds, which was the maximum run time allowed for samples on Any.run. Network use was enabled because many launchers require downloading additional binaries. The operating system was set to 64-bit Windows 10 with auto confirm for user account control (UAC) prompts. The chosen pre-installed software set was "Complete" because it was the only software set that included Java 8, which is needed for a majority of the samples.

The following uniform protocol was used. If the launcher was an installer, an attempt was made to install the software and launch it (if it did not automatically do so). For launchers that did not require installation and those that did, an attempt was made to sign into Minecraft using an offline account and launch the game. Whether or not a software launched visually, once I was not able to make more progress in starting the game or installing the software, one minute was given for additional background activity to be analyzed. Once there appeared to be no more activity and at least one minute had passed, the sandbox was terminated.

For installers or launchers in other languages, the best attempt was made to follow the prompts in the application. If the application closed unexpectedly, the processes were restarted until I successfully reached an end to input that I can give the client or client installer.

The process ID (PID) for the client process was recorded for each sandbox analysis in order to ensure in the later data processing phase that the process being analyzed is the sample and not another process. For clients that spawned more processes, a list of PIDs was kept instead of a single PID.

It is difficult to discern between a successful client launch and an unsuccessful startup. It is possible that malware may be hiding in software that appears non-functional to the user. To account for this, analyses where the software did not appear visibly were still used in the data pool. Samples that did not run and immediately presented the user with a Java runtime error were excluded from the sample set, bringing the original 96 samples down to 92.

Due to file size limitations for the sandbox environment, files over 100 MB in size were uploaded to DropBox where I created an automatic download link for the file. Any.run allows for using a download link to analyze a file. These download links were used to analyze the file. Public DropBox links are created by first creating a link for the uploaded file in the right click context menu. Secondly, the URL requires a change from "dl=0" to "dl=1" to make it an automatic download.

Clients that were retrieved as directories were uploaded as a zip file so that the binaries could have all the necessary application extension files needed for using the client. These were unzipped and then run manually following the processes described in this section.

At the end of each run, after the sandbox had been stopped, the data needed for the run was downloaded in json file format for later processing. These were saved with the filename being the hash of the program that was being analyzed so they could easily be referenced later.

3.3 STEP 3: Feature Extraction

3.3.1 Behavioral feature extraction

The sandbox report from Any.run is given in Json format and can easily be parsed with Python. Additionally, my previous research concluded that behavioral analysis could be analyzed jointly for PE and JAR files, however, this is not the case for static analysis, which will be talked about later.

A.1.1.1 Network features

For extracting network features, the script iterates through all the network requests made during the run and separates them into their various types of activities: DNS requests, network connections, and HTTP requests. These are represented as a numerical quantity for each value. Additionally, using ipstack.com API, a request was made for each of these connection types to obtain the country or region which was assembled into an aggregate number to represent counts for country/region for all types of network activity.

A.1.1.2 System Calls

To determine a process chain from the root process and all spawned child processes, the script would recursively find all processes that had a parent ID (PPID) that match the root process. It then recursively finds all matching PPIDs for child processes and assembles a final chain that shows the root process and all spawned child processes. A CSV was used to track the samples, and it included a column for the root process ID. This was important because some samples that were archived started from File Explorer and were not indicative of the actual root process for the sample when it was executed manually. The process chain was important to establish so that the data regarding total child processes instantiated and total child process depth could be calculated.

The number of modified files and registry reads, writes, and deletes is a simple figure that is provided in the Any.run report so it didn't require further processing.

A.1.1.3 Memory

To find DLL calls that spawned from the root process, the script for parsing the behavioral analysis would search for process triggers using the process chain that was created previously. Inside this process chain, the modules were extracted from the process if they contained a referenced 'image' from the Any.run report. These were assembled into a list of all binary processes that were spawned from a process. A simple filter for files ending in .dll was used to assemble a list of DLLs that were spawned from a process. For the root process and all child processes these DLLs were aggregated into one list so a count could be determined of the total amount DLL calls made by the process and all children.

A.1.1.4 Persistence

Using the binary triggers obtained above, they were matched against a list of potentially suspicious binaries, shown in Table 4, that the program may have tried to access. These executable files can be used to maintain persistence, such as schtasks.exe which allows for the creation of automated and start up tasks. The total amount of accesses to these binary files was aggregated into a count. Accesses to these binaries have a higher likelihood of a sample trying to maintain persistence by accessing the registry, attempting privilege escalation, or creating scheduled tasks. These are assembled into a list of accesses and looked at for a total quantity of suspicious binary accesses.

Table 4: Windows suspicious binaries.

Windows suspicious binary access
Reg.exe
Nslookup.exe
Regasm.exe
Runas.exe
Schtasks.exe
Sc.exe

Additionally, file events were monitored to see if there were any changes made to the 'Start Menu' or 'StartUp' directories, which would further indicate possible methods of maintaining persistence. These are assembled into a list and looked at for total quantity of startup modifications.

A.1.1.5 Hardware Metrics

Any.run does not directly provide the CPU usage and memory usage in their Json reports. They do provide this data on the webpage, however. The csv used to track the samples contains a link to the Any.run sample page where the CPU and memory usage can be seen. A separate script used in tandem with the script for processing behavioral data can open the webpage using the Selenium Python library and collect this data automatically. Since this data is presented visually in CSS, I was able to confirm with Any.run that high values, 100% corresponded to a 0 CSS and the lowest value a 28, which corresponded to 0% activity. This is easily processed into a percentage format by subtracting the observed value divided by 28 from 1. These values are assembled into a list of CPU and memory usage that correspond to the run length. After converting these values to percentages, the mean and median were calculated. Table 5 provides a concise summary of all of the types of data that were aggregated from the sandbox samples.

Table 5: Summary of behavioral analysis features and types of data.

Network Traffic	
DNS requests	Count
Network connections	Count
HTTP requests	Count
Network locations	Count per country/region
System Calls	
Processes instantiated	Count
Process depth	Count
Modified Files	Count
Registry Reads	Count
Registry Writes	Count
Registry Deletes	Count
Memory	
DLL Calls	Count
Persistence	
Startup behavior modifications	List of values
Suspicious binary file accesses	Count
Hardware Metrics	
CPU usage	Average and median value
RAM usage	Average and median value

3.3.2 Static Feature Extraction

Research from the literature review concluded that Windows portable executable (PE) files and Java archives (JARs) could not be processed in the same way. One example of their differences is that header information from PE files provides a wealth of information about the executable, but the header information from JAR files only shows basic archival information, such as the number of files it contains. These differences can be seen in the summary of features for PE files in
Table 8 and in Table 11 for JAR files.

All files were processed through a Python script to do the static analysis. Two separate class files were made for analyzing the different file types, one for PEs and another for Jars.

A.1.1.6 File preparation

PE files only required string extraction using the Linux command 'strings' to find valid strings inside the binary file. These string files were cached into a directory so that they could be quickly reanalyzed for updated parameters or if bugs were encountered since running strings can be time consuming on larger binaries. The header was extracted using the pefile Python module.

JAR files are not as useful when strings are extracted from them. There is no limitation to being able to run strings on JAR files, but no evidence has shown that looking for strings extracted from a JAR file is a helpful way for analyzing samples for malware. Instead, a better method of analyzing JAR files is to perform a Java decomplication on the archive to revert the compressed archive back to its original Java code.

This process requires Open JDK to be installed on the machine, Open JDK 11 was used for this decomplication. The command line Java decompiler jd-cli was used on each JAR file and extracted to its own directory for later analysis (Cacek, 2023). This step will allow for more in-depth analysis on the underlying Java code and give better results for malware detection according to current research.

A.1.1.7 PE file analysis

A.1.1.7.1 String analysis

Using the output from the strings Linux program the text was analyzed using regular expression for patterns that matched URLs and IP addresses. Using the Python ipaddress module, IP addresses were validated to be within an appropriate range reserved for IPs. The amount of IP addresses found as well as the number of URLs were totaled into a numerical count. DLL strings were also extracted using regular expressions. The totals for DLL calls were aggregated per sample and used as a numerical quantity.

A.1.1.7.2 Header info

The packer signature was checked against a list of packers that often indicate obfuscated code and thus a higher likelihood of containing malware. If the packer was present and, in this list, it was flagged as a suspicious packer, otherwise if there was no packer or it didn't appear malicious the Boolean value was false. Table 6 shows the values used for the packer signatures.

Table 6: List of suspicious PE packers.

Packer Name	**Packer Signature**
UPX	UPX\x00\x01\xF0\x88\xE2\xFA
NSIS	nullsoft install system
PECompact	PECompact
ASPack	ASPack
FSG	FSG!

tElock	tElock
Themida	Themida
ASProtect	ASProtect

The header info also provides a list of imported functions. These functions were checked against the list in Table 7 that was found during research into static analysis of PE files. While the presence of one or even a few of these imported functions on their own is not indicative of malware, it helps to establish a pattern when used in conjunction with other static analysis methods.

Table 7: List of suspicious imported functions for PE files.

RegCloseKey	RegOpenKey	RegQueryValue	RegSetValue
RtlCreateRegistryKey	RtlWriteRegistryValue	CheckRemoteDebuggerPresent	FindWindow
GetLastError	IsDebuggerPresent	sleep	OutputDebugString
GetAdaptersInfo	FindWindow	GetTickCount	NtSettInformationProcess
DebugActiveProcess	QueryPerformanceCounter	NtQueryInformationProcess	VirtualAllocEx
LoadLibrary	VirtualFree	GetProcAddress	LdrLoadDll
LoadResource	VirtualProtectEx	CommandLineToArg	ShellExecute
system	WinExec	SetWindowsHook	RegisterHotKey
GetKeyState	MapVirtualKey	listen	socket
accept	bind	connect	send
recv	FtpPutFile	InternetOpen	InternetOpenUrl
InternetWriteFile	ConnetNamedPipe	PeekNamedPike	gethostbyname
inet addr	InternetReadFie	BitBlt	GetDC
CryptDecrypt	CryptGenRandom	CryptAcqureContext	SetPrivilege
LookupPrivilege	CreateRemoteThread	WriteProcessMemory	ReadProcessMemory
OpenProcess	NtOpenProcess	NtReadVirtualMemory	NtWriteVirtualMemory
CreateFile	CreateFileMapping	CreateMutex	CreateProcess
CreateService	ControlService	OpenSCManager	StartServiceCtrlDispatcher
CreateRemoteThread	WriteProcessMemory	ReadProcessMemory	OpenProcess
NtOpenProcess	NtReadVirtualMemory	NtWriteVirtualMemory	MapViewofFile
Module32First	Module32Next	OpenMutex	OpenProcess
QueueUserAPC	SetFileTime	SfcTerminateWeatherThread	SuspendThread
Thread32First	Thread32Next	WriteProcessMemory	ResumeThread
DllCanUnloadNow	DllGetClassObject	DllInstall	DllRegisterServer
DllUnregisterServer	NetScheduleJobAdd	FindFirstFile	FindNextFile

Table 8 summarizes the data that were collected for static analysis of PE files. There are five total static features, two from the header and three that come from the strings extracted from the strings program. Together, these static features will work with the behavioral features to provide a more complete picture of the samples.

Table 8: Summary of static analysis features and types of data for PE files

String Analysis	
IP Addresses	Count
URLs	Count
DLL Calls	Count
Header Info	
Packer signature	Boolean
Suspicious Imported Functions	Count

A.1.1.8 JAR file analysis

A.1.1.8.1 Code analysis

The directories containing the decompiled Jars were iterated through to analyze the code source contained in the java files. The decomplication left some artifacts that rendered the code unable to parse, so all the Java code was run through a text cleaner that converted non-ascii displayable characters as their Unicode representation in ascii. This allows the code to preserve its flow while still being able to analyze it. IP addresses were simply extracted from each java file that was processed in the decomplication directory like the analysis performed on PE files.

Sensitive keywords were found using a basic search and match feature looking for strings in the decompiled code. The number of strings that were sensitive was aggregated into a total count. Table 9 is a list of the sensitive keywords that were used in the analysis which includes keywords that try to establish network connections and modify files.

Table 9: Sensitive keyword list used in JAR file static analysis.

runtime.exec	processbuilder	system.exec
runtime.getruntime().exec	process.start	cmd.exe
powershell.exe	inetaddress	httpurlconnection
httpsurlconnection	datagramsocket	multicastsocket
java.net.url	jshell	scriptengine
eval	javascript	randomaccessfile

filewriter	bufferedwriter	writeobject
readobject	native	jni
http://	https://	tcp://
udp://	smtp://	ftp://
smb://	tomcat	jetty
undertow	http://	https://
tcp://	udp://	smtp://
ftp://	smb://	bypass
ignoresecurity	disablesecurity	tomcat
jetty	undertow	

As mentioned in the literature review, high entropy strings can indicate the presence of malware. To calculate the presence of malware, I used the same method that Ladisa et al (2022) conducted. First the text was tokenized using the NLTK Python module (Aarsen, 2023). A regular expression removed non-alphanumeric characters and then ran it through the GCLD3 language detector (Google, 2023). Strings that could not be confidently identified as belonging to a language were identified as suspicious.

These suspicious strings from the code were run through a function to calculate the Shannon entropy, which is the overall entropy of the string values in relative randomness. This is not sufficient for small strings, so relative entropy was also calculated using the Kullbac-Leibler divergence metric. If values had a relative entropy above 2.0 or a Shannon entropy above 4.0, they were high entropy strings.

The Python module javalang was used to parse each java file so that individual features could be extracted for the remaining analyses. Empty catch clauses were identified using the javalang node called CatchClause, and, if there was no code inside the catch clause, it was considered an empty catch clause.

Suspicious API calls were found by iterating through the javalang processed Java code and checking against the class-method used to a list of suspicious API calls. Table 10 shows the class-methods considered suspicious for each of the corresponding classes. These are the same methods used in the research of Ladisa et al. (2022) to identify suspicious Java code. The total amount of suspicious API calls was aggregated for each JAR file.

Table 10: Suspicious methods for Java classes used in static JAR analysis.

Class	Methods
runtime	exec
processbuilder	processbuilder, command, start
system	load, loadlibrary
desktop	open
jshell	eval
scriptengine	eval
base64$decoder	decode
base64$encoder	encode, encodetostring
socket	socket, getinputstream, getoutputstream
url	url, openconnection, openstream
uri	uri, create
urlconnection	getinputstream
httprequest$builder	get, post
urlclassloader	urlclassloader
classloader	loadclass
class	forname, getdeclaredmethod, getdeclaredfield, newinstance
method	invoke
introspector	getbeaninfo
system	getproperty, getproperties, getenv
inetaddress	gethostname
fileoutputstream	fileoutputstream, write
file	file
files	newbufferedwriter, newoutputstream, write, writestring, copy
filewriter	write
bufferedwriter	write
randomaccessfile	write
fileinputstream	fileinputstream, read
filereader	read
scanner	scanner
bufferedreader	read
randomaccessfile_read	read, readfully

The file count for each archive was included as a metric to provide a baseline for the other quantities. Some JAR archives have more than 40,000 files and it would follow suit that they would likely have a higher count for all the values that were aggregated. By using the file count, I can more accurately measure each file by a ratio of the file count to the number of observed values for each metric. Table 11 summarizes the values that were gathered for the static analysis of Java code.

Table 11: Summary of static analysis features and types of data for JAR files.

Code Analysis	
IP Addresses	Count
Suspicious API Calls	Count
Empty Catch Clauses	Count
Sensitive Keywords	Count
High Entropy Strings	Count
JAR Header Info	
File Count	Count

Not all the collected data was useful, and some changes were made after looking at the dataset. In the JAR dataset, the count for static IP addresses was zero for all samples, so this data was removed for the JAR dataset, but kept for the PE dataset. Additionally, no suspicious packers were found for any of the PE samples, so they were removed. Finally, suspicious file count was removed for JAR files because it did not have any data points either but was kept for PE files.

3.4 STEP 4: Data Analysis and Feature Selection

Once all the data was processed according to the metrics used in the previous section, summary statistics were generated for both the PE and JAR file datasets. Data was ingested into Python from Json files into Pandas data frames. Samples with at least 1 VirusTotal detection were placed into the malicious software set, while samples with no detections were put into the non-malicious dataset. These data frames were stored on disk to allow for faster data retrieval.

The VirusTotal detections datapoint was removed from the training and test sets because it cannot be used as a predictor for malware when looking at novel samples. Summary statistics are detailed in Table 12 and Table 13 and were generated from the training set.

Table 12: Summary statistics of PE files

Feature	*Non-malicious*	*Malicious*
LIVE ANALYSIS		
CPU Usage	0.14 (0.1)	0.06 (0.02)
Memory Usage	0.53 (0.04)	0.47 (0.05)
DLL Count	693.5 (618.6)	289.0 (297.75)
Australia Traffic	0.0 (0.0)	0.1 (0.45)
Brazil Traffic	0.5 (2.24)	0.0 (0.0)
Canada Traffic	1.6 (6.1)	0.1 (0.45)
China Traffic	0.9 (4.02)	0.0 (0.0)
Columbia Traffic	1.9 (4.21)	1.0 (2.83)
Czechia Traffic	0.2 (0.89)	0.0 (0.0)
Denmark Traffic	0.2 (0.89)	0.2 (0.89)
Finland Traffic	0.0 (0.0)	0.2 (0.89)
France Traffic	0.8 (2.61)	0.2 (0.89)
Germany Traffic	42.2 (22.12)	16.5 (26.88)
Ireland Traffic	8.2 (6.23)	7.8 (2.59)
Netherlands Traffic	17.6 (29.52)	7.9 (4.15)
Unknown Region Traffic	3.8 (0.89)	3.1 (0.89)
Russia Traffic	0.0 (0.0)	0.9 (4.02)
Singapore Traffic	0.3 (1.34)	0.0 (0.0)
Slovenia Traffic	0.1 (0.45)	0.0 (0.0)
Spain Traffic	6.5 (17.97)	0.6 (2.68)
Sweden Traffic	0.3 (1.34)	0.4 (1.1)
Switzerland Traffic	1.0 (2.45)	0.0 (0.0)
United Arab Emirates Traffic	0.9 (4.02)	0.3 (1.34)
United Kingdom Traffic	1.5 (2.24)	0.0 (0.0)
United States Traffic	53.1 (25.74)	24.1 (38.1)
Network Connections	39.3 (19.28)	24.9 (11.85)
DNS Requests	67.4 (45.55)	29.0 (30.09)
HTTP Requests	34.9 (34.78)	9.5 (16.57)
Suspicious Binaries	2.6 (7.55)	3.0 (8.18)
Suspicious File Access	1.1 (1.6)	0.4 (0.7)
Modified Files	678.7 (1016.5)	1026.0 (2962.58)
Registry Deletes	37.6 (70.51)	2.0 (3.27)
Registry Reads	18576.9 (16840.8)	4614.5 (6317.83)
Registry Writes	339.9 (880.79)	10.4 (16.21)
STATIC ANALYSIS		
DLL Strings	194.0 (562.78)	22.5 (29.06)
Imported Functions	368.5 (753.07)	92.6 (67.51)
IP Addresses	479.3 (1470.99)	12.9 (23.4)
URLs	719.7 (1760.18)	23.6 (49.08)

Table 13: Summary statistics of JAR files

Feature	*Non-malicious*	*Malicious*
LIVE ANALYSIS		
CPU Usage	0.08 (0.05)	0.1 (0.15)
Memory Usage	0.49 (0.07)	0.46 (0.05)
DLL Count	369.69 (655.28)	201.59 (360.34)
Brazil Traffic	0.0 (0.0)	0.03 (0.45)
Canada Traffic	0.62 (2.3)	0.0 (0.0)
China Traffic	0.0 (0.0)	0.97 (7.7)
Columbia Traffic	0.77 (3.08)	0.31 (2.68)
Finland Traffic	0.38 (2.24)	0.0 (0.0)
France Traffic	0.15 (0.55)	0.38 (2.17)
Germany Traffic	18.31 (26.72)	9.59 (11.65)
India Traffic	0.15 (0.89)	0.0 (0.0)
Ireland Traffic	6.85 (1.87)	6.07 (4.15)
Japan Traffic	0.0 (0.0)	0.0 (0.0)
Netherlands Traffic	7.62 (2.92)	8.41 (23.45)
Unknown Region Traffic	3.69 (1.3)	3.14 (1.0)
Russia Traffic	0.0 (0.0)	0.31 (4.02)
Singapore Traffic	0.0 (0.0)	0.03 (0.45)
Spain Traffic	0.85 (2.86)	0.0 (0.0)
Sweden Traffic	0.08 (0.45)	0.24 (3.13)
Switzerland Traffic	0.0 (0.0)	0.1 (1.34)
United Kingdom Traffic	1.85 (7.46)	0.0 (0.0)
United States Traffic	24.62 (30.67)	495.62 (6277.2)
Network Connections	26.69 (18.52)	18.66 (8.09)
DNS Requests	32.62 (36.41)	18.24 (16.3)
HTTP Requests	6.62 (10.74)	488.31 (2610.03)
Suspicious Binaries	0.15 (0.55)	1.17 (3.72)
Modified Files	1569.46 (3113.63)	720.03 (2335.48)
Registry Deletes	3.0 (10.22)	0.07 (0.26)
Registry Reads	5667.54 (10937.64)	2381.28 (3526.76)
Registry Writes	21.0 (28.15)	11.79 (15.31)
STATIC ANALYSIS		
Empty Catch Clauses	81.69 (80.98)	89.97 (107.39)
High Entropy Strings	365.46 (729.29)	316.1 (515.62)
File Count	11210.31 (17550.36)	8976.1 (10305.01)
Sensitive Keywords	347.69 (463.03)	317.1 (368.31)
Suspicious API Calls	76.38 (73.66)	71.14 (65.96)
VirusTotal Detections	0.0 (0.0)	7.1 (6.53)

Correlation tables were generated for PE and JAR files from the training set, which are shown in Figure 4 and Figure 5. These heatmaps were useful in feature selection for the final dataset to account for data with high correlation. The correlation heatmaps show interesting patterns. For instance, "VirusTotal.malicious," which is a binary value for malicious and non-malicious samples using the detection count, is not strongly correlated with any one of the other features on its own (except for "VirusTotal.detections" which is where the binary value was derived). This suggests that no single feature is strongly correlative of a malicious sample and that machine learning algorithms will be required to adequately analyze these samples. Many of the network features are strongly correlated with each other in both PE and JAR file sample sets.

The correlation-based filter works to reduce the feature set and improve the accuracy of the machine learning algorithms (Abawajy et al., 2021). A correlation threshold value is applied, in the case of this study either 0.80 or 0.90, which was chosen based on the correlation tables in Figure 4 and Figure 5. Each feature is compared with every other feature in a pair-wise fashion. For example, the number of DNS requests was compared with network connections which have a correlation of 0.95 for PE files. If the 0.90 correlation filter is applied, then the second value is removed from the dataset. In this example, the network connections feature is removed from the dataset. The value for network connections is stored for later analysis since it has the same importance as the feature that was not removed, DNS requests. Only the top half of the correlation table is used to avoid analyzing the same pairs twice.

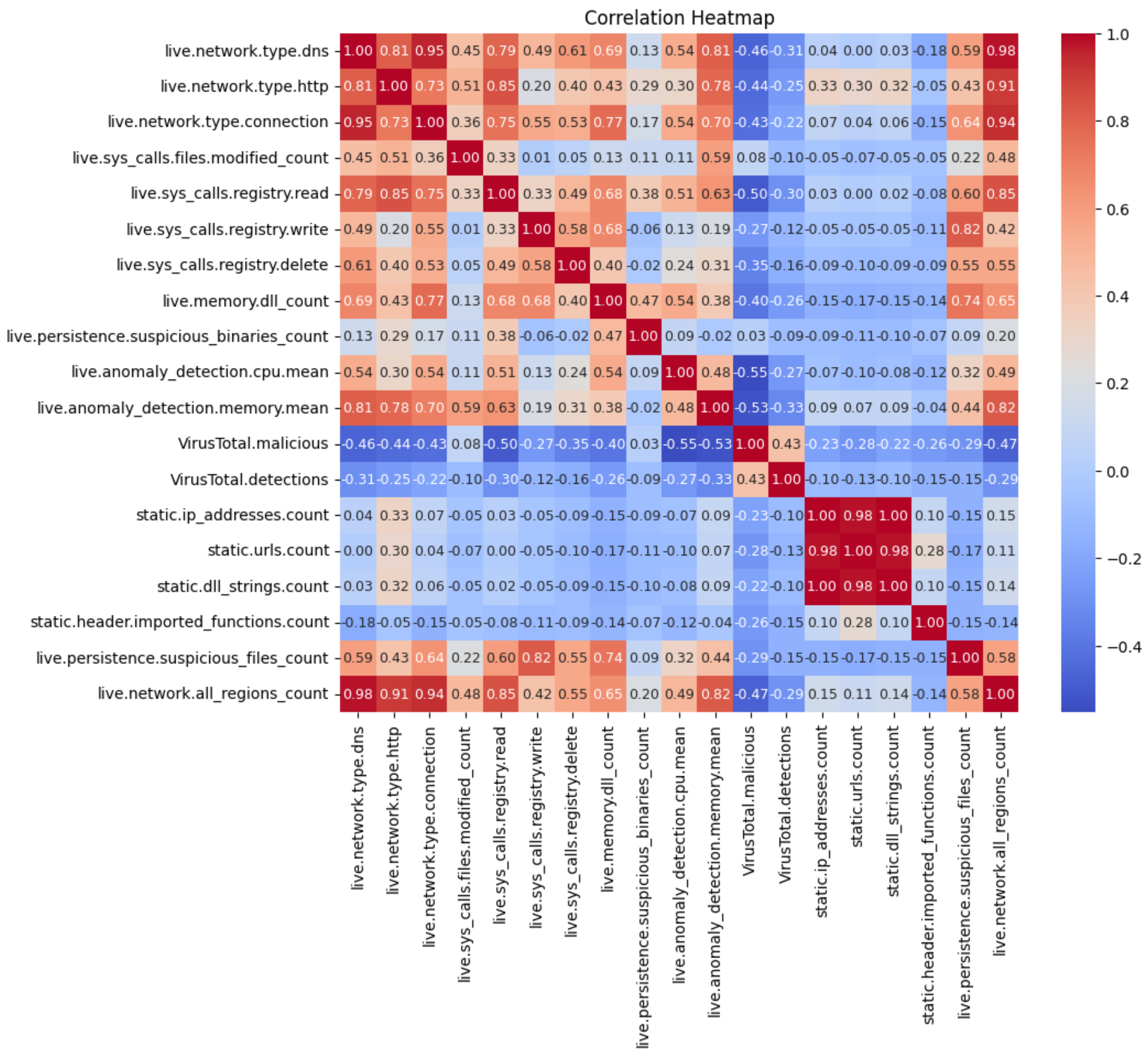

Figure 4: Heatmap of PE file features

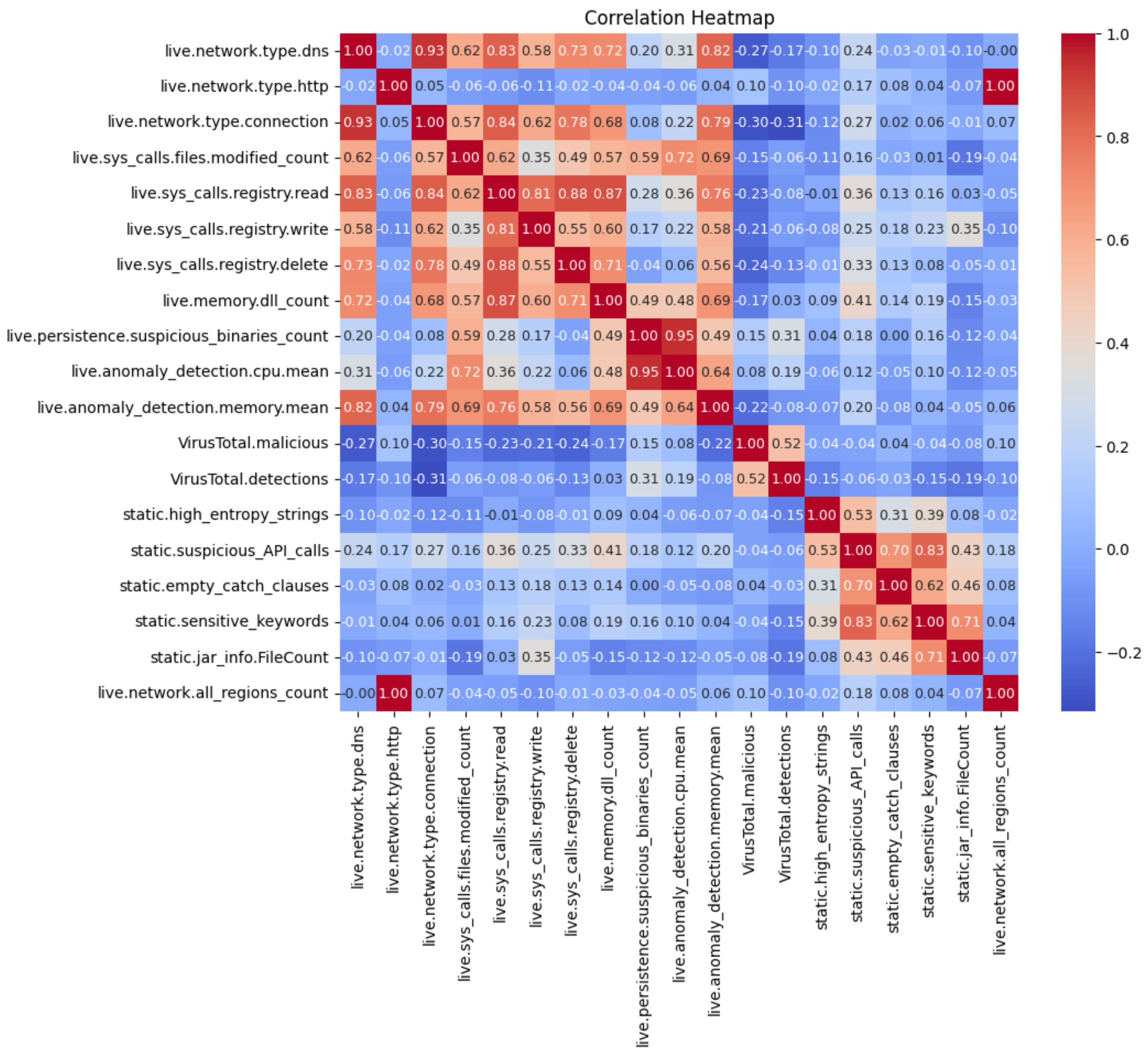

Figure 5: Heatmap of JAR file features

Based on the heatmap data, two correlation filters were used, 0.80 and 0.90. The first type of filter was a correlation filter with cutoffs at 90% and above correlation as well as 80% and above correlation. For PE files there are a total of 38 features. The 90% correlation filter reduced it to 27 while the 80% filter further reduced it to 19 features. For JAR files, there are a total of 35 features and the 90% filter reduced it to 27 features and the 80% filter reduced it to 20. Each classification algorithm was run through no correlation filter, 90% filter, and 80% filter for both the PE and JAR datasets.

Additionally, the K best feature algorithm, based on the chi squared test, from the 'SelectKBest' Scikit-learn function was used as a secondary filter (Gibert et al., 2020). The K best features algorithm reduces each set to the number of features specified in the K best features algorithm. For example, 10 K best features will reduce the set to the best 10 features. In tests where the K best features algorithm is used conjointly with the correlation filter, the correlation filter was run first and then the best features applied after. Because the 80% correlation filter on PE files reduces the features set to 9, the 10 K and 18 K best features algorithm was not run on PE files with an 80% correlation filter. Additionally, since the 80% filter brought the total features below 18 for JAR files, the 18 K best features algorithm was not used.

The PE and JAR sample sets were randomly divided up into a 70% training set and 30% test set. To assess the performance of a machine learning model, 10-fold cross-validation was conducted on the training set. Within each fold, the training set was further stratified into a 75-25 split. This process was repeated for 10 iterations, each time using a different 75-25 split, to thoroughly evaluate the model's performance and assess the reliability of the training set. The values for the cross-validation scores for the PE and JAR file sets are shown in Table 14 and Table 15.

For JAR files, there are 42 samples in the training set and 19 in the test set, totaling 61. In the 42 sample JAR training set, 29 samples were malicious and 13 non-malicious. The JAR test set included 12 malicious files and 7 non-malicious files. PE files had 20 samples in the training set and 9 in the test set, totaling 29 samples. In the 20 sample PE training set there were 10 malicious and 10 non-malicious files. The test set included 4 malicious and 5 non-malicious files.

Six different machine learning classification algorithms were used in this study based on similar malware research: classification tree, naïve Bayes (Burnap et al., 2018), neural network (Tekerek & Yapici, 2022), K’s nearest neighbor, support vector machine, and random forest (Verma et al., 2020). The Python library Scikit-learn provides all these classifiers and was used to process the data for machine learning. The same consistent training set and testing set was used to ensure consistency across all the classifier methods.

Table 14: Cross-validation Scores for PE Files Training Set

Name	Classifier	Cross-validation Score	Cross-validation Std. Dev.
Exe - No correlation filter, no K best feature			
	classification tree	0.75	0.25
	neural network	0.55	0.27
	naives bayes	0.55	0.27
	KNN	0.7	0.33
	SVM	0.3	0.24
	random forest	0.7	0.24
Exe - 90% correlation filter, no K best feature			
	classification tree	0.65	0.32
	neural network	0.5	0.22
	naives bayes	0.65	0.32
	KNN	0.7	0.33
	SVM	0.45	0.35
	random forest	0.7	0.24
Exe - 80% correlation filter, no K best feature			
	classification tree	0.7	0.24
	neural network	0.5	0.32
	naives bayes	0.5	0.32
	KNN	0.55	0.35
	SVM	0.6	0.3
	random forest	0.65	0.23
Exe - No correlation filter, 5 K best feature			
	classification tree	0.7	0.33
	neural network	0.55	0.15
	naives bayes	0.7	0.33
	KNN	0.7	0.33
	SVM	0.65	0.32
	random forest	0.75	0.25
Exe - 90% correlation filter, 5 K best feature			
	classification tree	0.7	0.33
	neural network	0.5	0
	naives bayes	0.65	0.32
	KNN	0.7	0.33
	SVM	0.55	0.42
	random forest	0.7	0.24
Exe - 80% correlation filter, 5 K best feature			
	classification tree	0.75	0.25
	neural network	0.6	0.2

naives bayes	0.75	0.34
KNN	0.55	0.35
SVM	0.6	0.3
random forest	0.65	0.32
Exe - No correlation filter, 10 K best feature		
classification tree	0.65	0.32
neural network	0.5	0.22
naives bayes	0.6	0.37
KNN	0.7	0.33
SVM	0.55	0.35
random forest	0.7	0.24
Exe - 90% correlation filter, 10 K best feature		
classification tree	0.7	0.33
neural network	0.5	0.22
naives bayes	0.65	0.39
KNN	0.7	0.33
SVM	0.6	0.37
random forest	0.7	0.24
Exe - 80% correlation filter, 10 K best feature		
classification tree	0.65	0.32
neural network	0.35	0.39
naives bayes	0.7	0.33
KNN	0.55	0.35
SVM	0.45	0.27
random forest	0.65	0.32
Exe - No correlation filter, 18 K best feature		
classification tree	0.75	0.25
neural network	0.55	0.15
naives bayes	0.7	0.24
KNN	0.7	0.33
SVM	0.35	0.23
random forest	0.7	0.24
Exe - 90% correlation filter, 18 K best feature		
classification tree	0.65	0.32
neural network	0.6	0.37
naives bayes	0.75	0.25
KNN	0.7	0.33
SVM	0.4	0.3
random forest	0.7	0.24
Exe - 80% correlation filter, 18 K best feature		
classification tree	0.7	0.24

	neural network	0.5	0.32
	naives bayes	0.5	0.32
	KNN	0.55	0.35
	SVM	0.6	0.3
	random forest	0.65	0.23

Table 15: Cross-validation Scores for JAR Files Training Set

Name	**Classifier**	**Cross-validation Score**	**Cross-validation Std. Dev.**
Jar - No correlation filter, no K best feature			
	classification tree	0.6	0.17
	neural network	0.57	0.26
	naives bayes	0.48	0.23
	KNN	0.7	0.14
	SVM	0.63	0.26
	random forest	0.7	0.16
Jar - 90% correlation filter, no K best feature			
	classification tree	0.51	0.22
	neural network	0.48	0.25
	naives bayes	0.55	0.25
	KNN	0.7	0.14
	SVM	0.42	0.23
	random forest	0.68	0.21
Jar - 80% correlation filter, no K best feature			
	classification tree	0.55	0.23
	neural network	0.36	0.31
	naives bayes	0.38	0.21
	KNN	0.67	0.15
	SVM	0.48	0.22
	random forest	0.7	0.16
Jar - No correlation filter, 5 K best feature			
	classification tree	0.58	0.28
	neural network	0.77	0.17
	naives bayes	0.8	0.15
	KNN	0.62	0.13
	SVM	0.77	0.13
	random forest	0.62	0.25
Jar - 90% correlation filter, 5 K best feature			
	classification tree	0.58	0.18
	neural network	0.68	0.12
	naives bayes	0.74	0.15

KNN	0.62	0.16
SVM	0.7	0.09
random forest	0.53	0.23
Jar - 80% correlation filter, 5 K best feature		
classification tree	0.55	0.16
neural network	0.57	0.17
naives bayes	0.72	0.13
KNN	0.64	0.11
SVM	0.7	0.14
random forest	0.58	0.24
Jar - No correlation filter, 10 K best feature		
classification tree	0.6	0.11
neural network	0.6	0.16
naives bayes	0.8	0.15
KNN	0.7	0.09
SVM	0.77	0.13
random forest	0.6	0.16
Jar - 90% correlation filter, 10 K best feature		
classification tree	0.6	0.13
neural network	0.52	0.2
naives bayes	0.72	0.16
KNN	0.65	0.13
SVM	0.6	0.11
random forest	0.65	0.13
Jar - 80% correlation filter, 10 K best feature		
classification tree	0.52	0.18
neural network	0.68	0.12
naives bayes	0.67	0.28
KNN	0.64	0.11
SVM	0.67	0.1
random forest	0.55	0.16
Jar - No correlation filter, 18 K best feature		
classification tree	0.66	0.12
neural network	0.68	0.17
naives bayes	0.74	0.12
KNN	0.65	0.13
SVM	0.64	0.16
random forest	0.62	0.13
Jar - 90% correlation filter, 18 K best feature		
classification tree	0.48	0.14
neural network	0.58	0.27

naives bayes	0.72	0.18
KNN	0.65	0.13
SVM	0.67	0.1
random forest	0.6	0.17
Jar - 80% correlation filter, 18 K best feature		
classification tree	0.55	0.26
neural network	0.55	0.28
naives bayes	0.38	0.21
KNN	0.67	0.15
SVM	0.48	0.22
random forest	0.7	0.16

4 Results

It is evident that the impact of various dataset filtering techniques is crucial, as reflected in Table 16 and
Table 17. Choosing the best model based on performance relies on looking at all the scores collectively and not just the best scores for one category. For instance, a model may have performed adequately in precision but very poorly in recall. Depending on the circumstances, it may be better trade off some precision for better recall. In the case of malware data, I found the model with the best balance between accuracy, precision, and recall. Additionally, I ensured that the cross-validation scores were not extremely low for the training set. The range of cross-validation scores was from 0.30 to 0.75 for PE files and 0.36 to 0.80 for JAR files, so models with scores closer to 0.75 and 0.80 were used over low scoring models.

Table 16: Machine learning results for PE files

Name	Classifier	Accuracy	Precision	Recall	F1 Score	AUC
Exe - No correlation filter, no K best feature						
	classification tree	0.56	0.5	0.75	0.60	0.57
	neural network	0.67	0.57	1	0.73	0.85
	naives bayes	0.56	0.5	1	0.67	0.55
	KNN	0.67	0.6	0.75	0.67	0.72
	SVM	0.67	0.67	0.5	0.57	0.7
	random forest	0.56	0.5	0.75	0.60	0.7
Exe - 90% correlation filter, no K best feature						
	classification tree	0.56	0.5	0.75	0.60	0.57
	neural network	0.67	0.67	0.5	0.57	0.65
	naives bayes	0.56	0.5	0.75	0.60	0.65
	KNN	0.67	0.6	0.75	0.67	0.72
	SVM	0.44	0.4	0.5	0.44	0.25
	random forest	0.67	0.6	0.75	0.67	0.6
Exe - 80% correlation filter, no K best feature						
	classification tree	0.56	0.5	0.5	0.50	0.55
	neural network	0.56	0.5	0.25	0.33	0.4
	naives bayes	0.56	0.5	0.75	0.60	0.6
	KNN	0.56	0.5	0.25	0.33	0.4
	SVM	0.44	0.33	0.25	0.29	0.55
	random forest	0.44	0.33	0.25	0.29	0.45

Exe - No correlation filter, 5 K best feature					
classification tree	0.67	0.6	0.75	0.67	0.68
neural network	0.56	0.5	0.25	0.33	0.75
naives bayes	0.78	0.67	1	0.80	0.7
KNN	0.67	0.6	0.75	0.67	0.68
SVM	0.67	0.6	0.75	0.67	0.6
random forest	0.67	0.6	0.75	0.67	0.65
Exe - 90% correlation filter, 5 K best feature					
classification tree	0.67	0.6	0.75	0.67	0.68
neural network	0.56	0	0	0.00	0.7
naives bayes	0.78	0.67	1	0.80	0.7
KNN	0.67	0.6	0.75	0.67	0.68
SVM	0.56	0.5	0.5	0.50	0.7
random forest	0.67	0.6	0.75	0.67	0.68
Exe - 80% correlation filter, 5 K best feature					
classification tree	0.67	0.6	0.75	0.67	0.68
neural network	0.56	0	0	0.00	0.5
naives bayes	0.56	0.5	0.75	0.60	0.6
KNN	0.44	0.4	0.5	0.44	0.4
SVM	0.33	0.38	0.75	0.50	0.35
random forest	0.67	0.57	1	0.73	0.6
Exe - No correlation filter, 10 K best feature					
classification tree	0.67	0.6	0.75	0.67	0.68
neural network	0.44	0.44	1	0.62	0.55
naives bayes	0.67	0.6	0.75	0.67	0.65
KNN	0.67	0.6	0.75	0.67	0.68
SVM	0.44	0.33	0.25	0.29	0.7
random forest	0.67	0.6	0.75	0.67	0.62
Exe - 90% correlation filter, 10 K best feature					
classification tree	0.67	0.6	0.75	0.67	0.68
neural network	0.67	0.57	1	0.73	0.9
naives bayes	0.56	0.5	0.75	0.60	0.65
KNN	0.67	0.6	0.75	0.67	0.68
SVM	0.56	0.5	0.5	0.50	0.5
random forest	0.67	0.6	0.75	0.67	0.7
Exe - 80% correlation filter, 10 K best feature					
classification tree	0.67	0.6	0.75	0.67	0.68
neural network	0.44	0.4	0.5	0.44	0.35
naives bayes	0.56	0.5	0.75	0.60	0.65
KNN	0.44	0.4	0.5	0.44	0.4
SVM	0.44	0.43	0.75	0.55	0.6

	random forest	0.56	0.5	0.75	0.60	0.57
Exe - No correlation filter, 18 K best feature						
	classification tree	0.56	0.5	0.75	0.60	0.57
	neural network	0.56	0	0	0.00	0.7
	naives bayes	0.67	0.6	0.75	0.67	0.65
	KNN	0.67	0.6	0.75	0.67	0.68
	SVM	0.67	0.57	1	0.73	0.4
	random forest	0.67	0.6	0.75	0.67	0.75
Exe - 90% correlation filter, 18 K best feature						
	classification tree	0.67	0.6	0.75	0.67	0.68
	neural network	0.67	0.57	1	0.73	0.95
	naives bayes	0.56	0.5	0.75	0.60	0.65
	KNN	0.67	0.6	0.75	0.67	0.72
	SVM	0.33	0.25	0.25	0.25	0.75
	random forest	0.67	0.6	0.75	0.67	0.7
Exe - 80% correlation filter, 18 K best feature						
	classification tree	0.56	0.5	0.5	0.50	0.55
	neural network	0.56	0.5	0.25	0.33	0.4
	naives bayes	0.56	0.5	0.75	0.60	0.6
	KNN	0.56	0.5	0.25	0.33	0.4
	SVM	0.44	0.33	0.25	0.29	0.55
	random forest	0.44	0.33	0.25	0.29	0.45

Table 17: Machine learning results for JAR files

Name	**Classifier**	**Accuracy**	**Precision**	**Recall**	**F1 Score**	**AUC**
Jar - No correlation filter, no K best feature						
	classification tree	0.63	0.67	0.73	0.70	0.61
	neural network	0.47	0.57	0.36	0.44	0.49
	naives bayes	0.68	0.73	0.73	0.73	0.64
	KNN	0.74	0.71	0.91	0.80	0.82
	SVM	0.47	0.54	0.64	0.58	0.4
	random forest	0.63	0.62	0.91	0.74	0.83
Jar - 90% correlation filter, no K best feature						
	classification tree	0.68	0.78	0.64	0.70	0.69
	neural network	0.42	0.5	0.18	0.27	0.44
	naives bayes	0.68	0.67	0.91	0.77	0.69
	KNN	0.74	0.71	0.91	0.80	0.82
	SVM	0.53	0.58	0.64	0.61	0.7
	random forest	0.74	0.71	0.91	0.80	0.76

Jar - 80% correlation filter, no K best feature					
classification tree	0.84	0.9	0.82	0.86	0.85
neural network	0.58	0.59	0.91	0.71	0.57
naives bayes	0.47	0.67	0.18	0.29	0.67
KNN	0.84	0.83	0.91	0.87	0.84
SVM	0.42	0.5	0.45	0.48	0.38
random forest	0.68	0.67	0.91	0.77	0.75
Jar - No correlation filter, 5 K best feature					
classification tree	0.53	0.57	0.73	0.64	0.57
neural network	0.58	0.6	0.82	0.69	0.41
naives bayes	0.58	0.6	0.82	0.69	0.4
KNN	0.53	0.56	0.91	0.69	0.45
SVM	0.63	0.61	1	0.76	0.47
random forest	0.47	0.53	0.73	0.62	0.64
Jar - 90% correlation filter, 5 K best feature					
classification tree	0.58	0.64	0.64	0.64	0.61
neural network	0.63	0.62	0.91	0.74	0.61
naives bayes	0.58	0.59	0.91	0.71	0.5
KNN	0.58	0.59	0.91	0.71	0.55
SVM	0.63	0.61	1	0.76	0.53
random forest	0.58	0.62	0.73	0.67	0.66
Jar - 80% correlation filter, 5 K best feature					
classification tree	0.58	0.64	0.64	0.64	0.64
neural network	0.68	0.65	1	0.79	0.72
naives bayes	0.63	0.62	0.91	0.74	0.52
KNN	0.58	0.58	1	0.73	0.63
SVM	0.63	0.61	1	0.76	0.44
random forest	0.53	0.56	0.82	0.67	0.57
Jar - No correlation filter, 10 K best feature					
classification tree	0.74	0.69	1	0.81	0.69
neural network	0.58	0.59	0.91	0.71	0.56
naives bayes	0.58	0.6	0.82	0.69	0.53
KNN	0.63	0.61	1	0.76	0.45
SVM	0.63	0.61	1	0.76	0.7
random forest	0.68	0.65	1	0.79	0.5
Jar - 90% correlation filter, 10 K best feature					
classification tree	0.68	0.65	1	0.79	0.62
neural network	0.74	0.71	0.91	0.80	0.61
naives bayes	0.63	0.62	0.91	0.74	0.74
KNN	0.58	0.58	1	0.73	0.74
SVM	0.58	0.58	1	0.73	0.58

random forest	0.63	0.61	1	0.76	0.69
Jar - 80% correlation filter, 10 K best feature					
classification tree	0.58	0.64	0.64	0.64	0.64
neural network	0.58	0.58	1	0.73	0.38
naives bayes	0.58	0.59	0.91	0.71	0.4
KNN	0.58	0.58	1	0.73	0.63
SVM	0.63	0.61	1	0.76	0.78
random forest	0.58	0.58	1	0.73	0.64
Jar - No correlation filter, 18 K best feature					
classification tree	0.63	0.62	0.91	0.74	0.58
neural network	0.58	0.58	1	0.73	0.17
naives bayes	0.58	0.6	0.82	0.69	0.55
KNN	0.58	0.58	1	0.73	0.74
SVM	0.58	0.58	1	0.73	0.35
random forest	0.63	0.61	1	0.76	0.52
Jar - 90% correlation filter, 18 K best feature					
classification tree	0.74	0.69	1	0.81	0.69
neural network	0.47	0.56	0.45	0.50	0.51
naives bayes	0.58	0.59	0.91	0.71	0.74
KNN	0.58	0.58	1	0.73	0.74
SVM	0.53	0.56	0.91	0.69	0.69
random forest	0.63	0.61	1	0.76	0.7
Jar - 80% correlation filter, 18 K best feature					
classification tree	0.79	0.82	0.82	0.82	0.78
neural network	0.58	0.58	1	0.73	0.65
naives bayes	0.47	0.67	0.18	0.29	0.67
KNN	0.84	0.83	0.91	0.87	0.84
SVM	0.42	0.5	0.45	0.48	0.38
random forest	0.74	0.71	0.91	0.80	0.79

PE files showed the best results for the Naïve Bayes (NB) classifier with no correlation filter and 5 K best feature. For JAR files K's nearest neighbor (KNN) with 80% correlation filter with no K best features performed the best. The ROC graph in Figure 6 highlights the best classification algorithm for PE files with no correlation filter and 5 K best features, based on the previously observed data in Table 16. In the figure this is 'GaussianNB' and represents the NB classification method.

Figure 7 highlights the relative performance of each of these classification methods for JAR files with the 80% correlation filter and 18 K best features. This ROC graph highlights the best classification algorithm for JAR files with an 80% correlation filter and 18 best features, based on the data from

Table 17. In the figure this is KNeighborsClassifier' and represents the KNN classification method.

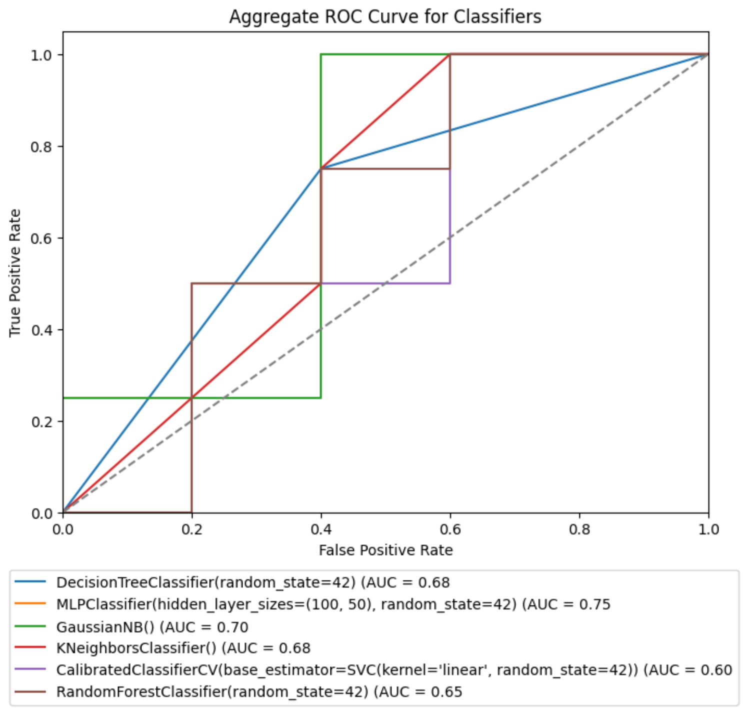

Figure 6: ROC graph for PE files with no correlation filter and 5 K best features

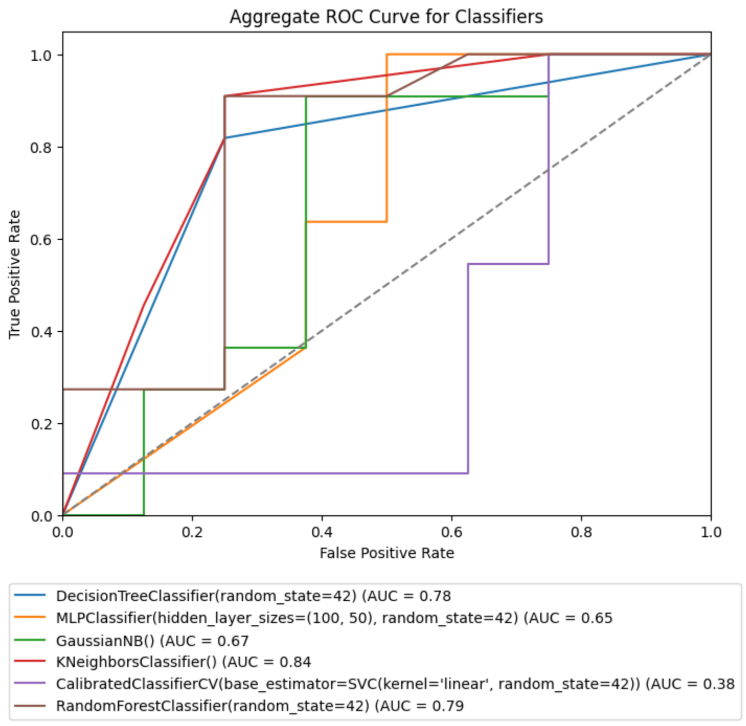

Figure 7: ROC graph for JAR files with 80% correlation filter, 18 K best feature

This data suggests that for PE files using the same feature set of behavioral and static features as presented, it is best to not filter out best directly on correlation values but instead use the 5 best features algorithm with the NB classifying method. It is worth noting that the 90% correlation filter with 5 best features performed the same as the chosen model, however, the feature set between both were the same except for one feature. Because both models had nearly identical feature sets, the simpler feature reduction model was chosen. JAR files, which use a slightly different feature set for static features than PE files, are best analyzed in this dataset when all values that are correlated above 0.80 or below -0.80 are filtered out and then using the KNN classifying method.

For JAR files, the model was able to accurately classify 84% of all samples in the training set. The model, with a precision score of 83%, shows that it is relatively good at not making false positive predictions. False positive are cases where the model predicts that something is malware when it is not malware. The recall score of 91% is impressive because it shows the model can accurately identify most samples with malware while infrequently predicting false negatives, or samples that contain malware that the model predicted did not have malware. The F1 score balances precision and recall and scored 87% for this classifier, indicating a good balance between accurately predicting real positives against false positives and false negatives.

The cross-validation score of 67% for JAR files reflects the model's performance in generalizing its predictions to unseen data by providing an indication of variance within the training set. Specifically, this score indicates that the classifier exhibits a moderate level of generalization capability, implying that it can make reasonably accurate predictions on new, unseen data. However, it's essential to note that the model is not flawless in its generalization, leaving room for potential improvements. The 67% cross-validation score suggests that the model strikes a balance between underfitting and overfitting, with potential for further fine-tuning to enhance its performance. Overfitting occurs when a model learns the training data too well, capturing noise and leading to poor generalization, while underfitting is when a model is too simplistic and fails to capture the underlying patterns in the data, also resulting in poor generalization.

For PE files, the accuracy scored 78% on the best algorithm, NB. It scored 67% on precision showing that it can make malware prediction more often than not, but sometimes declares false positives. The high recall score of 100% indicates that it was able to accurately predict malware without making any false negatives. This resulted in an F1 score of 80% suggesting a minor trade-off between precision and recall. The cross-validation score of 70% implies that the model's performance on the training set, when subjected to 10-fold cross-validation, is reasonably consistent. This suggests that the model is likely to perform similarly to the JAR model when presented with new, previously unseen data. It's important to note that a 70% cross-validation score indicates a moderate level of generalization, and while it shows the model's potential, there is still the possibility of overfitting.

5 Discussion

In this research I set out to identify the prominent indicators of malware in game clients. Using past academic research, industry insights, and other available tools, I was able to identify 5 prominent features for PE files and 18 features for JAR files. I was also able to establish a methodology others can follow to obtain measurements from malware samples. By using common machine learning techniques, I was able to narrow down the list of indicators and build a processing pipeline others can use to classify malware. I also show differences in indicators, processing, and models between different build frameworks. I will explain what this research contributes to industry and science as well as explain the limitations of this research and proposed future research. The code and data set used in this research can be found here: https://github.com/sjaustad/minecraft-malware-research.

I ran multiple models with various features to answer my research question, which asks what the prominent indicators of malware are in game clients. Table 18 provides a list of features for PE and JAR files that I found were the most important indicators of malware in Minecraft clients. The NB model performed best for PE files and produced 5 important features. For JAR files the KNN model produced 18 key features.

Table 18: Summary of most important features for PE and JAR Minecraft clients

File Type	Features
PE (5 features)	- Number of DNS connections and registry reads - Average CPU and memory usage - Network traffic from the United Kingdom
JAR (18 features)	- Number of DNS and HTTP connections - Number of modified files and suspicious binaries - Number of empty catch clauses, high entropy strings, and suspicious API calls - Archived file count - Network traffic from Netherlands, Ireland, Canada, Sweden, Colombia, India, Brazil, Singapore, Switzerland, and unknown regions

In addition to the features listed in Table 18, there are also additional features that may be of importance. These additional features are due to the pair-wise correlation filtering where one of each pair of highly correlated features are removed and are shown in Table 19. When analyzing the data from this research these features can be considered significant as well.

Table 19: Additional important feature for PE and JAR files

File Type	Other Significant Features
JAR	- Number of network connections and registry reads - Average CPU and memory usage - Network connections from United States, Germany, France, China, and Russia - Number of sensitive keywords

Importantly, the F1 score (87.0%) for the KNN JAR classifier suggests that it balances the prediction of false positives and false negatives well. Additionally, the F1 score of 80% for PE files similarly shows that it has a decent balance between false positives and false negatives, while trailing a little bit behind the JAR model. Most importantly, however, is the recall score. The recall score indicates how often false negative samples are predicted. A false negative would be a sample that contains malware, but the classifier algorithm believed that it did not contain malware. This is more dangerous than a false positive because it could allow a user to believe a sample is safe when it is not safe. The JAR model scored a 91% in recall, meaning only 9% of samples were false negatives and the PE model scored a 100%, meaning that no false negatives were predicted. The high recall performance in these models suggests that they can be useful for real world use when looking at unseen Minecraft clients.

It's crucial to consider the potential limitations of overfitting. Overfitting occurs when a model becomes too tailored to the training data, which can lead to high performance on the training set but reduced generalization to new, unseen data. The high recall scores are indicative of the model's ability to accurately predict malware. Still, they also raise concerns about potential overfitting, where the model may memorize the training data rather than generalize well on unseen Minecraft clients. The cross-validation scores of 67% for JAR files and 70% for PE files indicate they perform well but indicate there is room for improvement which could be done with a larger sample set. While high recall performance is valuable, it should be balanced with other metrics and validated with cross-validation scores to ensure robustness and generalization.

5.1 Contributions to industry

This research could allow websites that host Minecraft related software such as CurseForge (curseforge.com) and 9minecraft.net to implement Minecraft-specific malware detection techniques, especially through behavioral analysis. The method provided in this research provides a framework that can be used to test user-generated content with a live sandbox and static file-based analysis and then automate the machine learning code to detect potentially malicious samples. With recent malware spread on the sites for Minecraft software, this could potentially impact thousands of players who have seen infections from using third-party Minecraft software. One Minecraft mod distributor that I contacted for samples indicated interest in the findings of my research so they could potentially improve their malware detection capabilities of user-uploaded content. Implementing these models in Minecraft software repositories would be beneficial for both website administrators and players.

In my research I contacted several websites that host Minecraft software, and, while I could not get specifics from any of them on their exact procedure for analyzing user-uploaded content, I gathered that much of the heavy lifting is done from signature-based malware analysis. While this method of malware analysis is useful for quickly identifying possible threats and will catch some malware, it is not nearly as robust at behavioral and static analysis of samples. There are many ways that malware authors can cover their tracks and quickly change signatures of malware files. The analysis method outlined in this thesis would provide a great addition to current methods of file analysis that Minecraft software hosting websites are currently using.

A key takeaway from this research is that analyzing video game related software for malware presents some key challenges compared to traditional malware analysis. The use of JAR files coded in Java means that many traditional methods of statically analyzing novel files do not work effectively since most static analysis research is centered on PE files. Additionally, the diversity of third-party Minecraft clients from independent developers makes it challenging to notice subtle behavioral differences between malicious and non-malicious software. Even some of the non-malicious clients, such as MultiMC, showed many common indicators of malware like high registry access, many modified files, and high network activity. This amount of nuance suggests that there is utility in looking at Minecraft client malware separately from traditional malware.

5.2 Contributions to science

In addition to the results produced in this research, I also present a novel method for analyzing malware in the context of Minecraft clients that can be used as a pattern for other areas of research. While many articles provide rough frameworks on how to conduct malware research, I wanted to present a complete process including a code base that other researchers could use and even improve upon. The dataset is provided in full and could continue to provide new insights when studied in different ways. This thesis provides an adaptable framework that can serve as a benchmark for security research in the gaming industry and provides a comprehensive methodology for analyzing gaming-related software.

Furthermore, this research suggests that understanding the intricacies of a particular software community is crucial for effective malware detection. I demonstrated that the features that most effectively encapsulate malware for PE files are vastly different from the features that were predictors of malware in JAR files. This specificity suggests that different gaming communities can exhibit behaviors and patterns that may not align with conventional malware indicators. The findings presented encourage researchers to tailor their approaches to the community where malware is encountered.

The methods used in this research are based on the latest techniques of malware analysis with a heavy focus on machine learning. This approach continues to be beneficial because malware is becoming increasingly difficult to recognize based on simple patterns. I re-emphasize through my findings the importance of machine learning in malware analysis. It is an important tool that helps narrow a wide set of features down to a smaller set of statistically significant features. My research adds to the breadth of machine learning focused malware analysis by using it in a novel context, further demonstrating the robust nature of these techniques.

5.3 Limitations and future research

These models also could be further refined if there was a collaboration between security researchers and hosting websites with malware samples. I was not able to obtain any samples from websites that host Minecraft software created by independent developers. While I was able to find an adequate number of samples for this research, further adding to the sample pool could only work to improve the reliability of these models and make them more robust for looking at novel samples. Currently, finding specific malware samples on the internet is a difficult task. Many studies that I researched used generic large sample sets of malware for their research. It is easy to find many repositories that have general malware for well-known platforms, such as Windows or Android. It was much more difficult to find specific samples for Minecraft clients, especially when hosting websites are not currently working with security researchers. Hopefully a benefit of this study will be to encourage groups that have these samples to work with people willing to develop malware analysis models.

The relatively low sample size does introduce a possibility of overfitting within the model and is a limitation of this research. A larger sample set would improve the cross-validation scores and help narrow down a better model based on those scores. Overfitting in this context would result in the model being too tailored to the training data, which generally only happens with smaller datasets. Broadly speaking, overfitting results in a model being less proficient at classifying never before seen software samples. I made the best effort to include every possible client sample I could find, however, future work with mod repositories could help increase the number of samples to study.

Future work can improve the precision of the model, reducing the occurrences of false positives. While the security concerns of false positives are not as severe as false negatives, they can erode user trust in the reliability of these predictive algorithms. Research into more features could help improve these models further. For instance, in both PE and JAR samples, the network traffic was so highly correlated that most of it was cut out when the 80% and 90% correlation features were applied. Investigating which specific types of traffic are better indicators of malware could be useful in helping narrow down specific features for Minecraft client malware. In PE file static analysis, the number of DLL strings, IP addresses, and URLs were almost all perfectly correlated with each other. This would be an interesting relationship to investigate and may tell a deeper story about PE Minecraft clients.

While this study analyzed 38 features on Minecraft PE files and 35 features for Minecraft JAR files, there may be other important features that could be indicators of malware. For behavioral analysis features, more hardware level features such as disk I/O features and network bandwidth could prove beneficial. Additionally, while Any.run does have some basic features for combating anti-analysis techniques, more research could be conducted on feature collection with malware with anti-analysis properties. Regarding static analysis, code complexity for PE files would be an excellent additional feature for Minecraft launcher analysis. This would look for encoding techniques such as Base64 and encryption detection.

Studying more features would not only benefit the detection of malware in Minecraft clients but would pave the way for a broader understanding of how independently produced software in the gaming industry can be analyzed for security threats. As this research has demonstrated, the complexities observed within Minecraft clients are likely to find parallels in other gaming ecosystems. Furthermore, a collaborative effort between gaming and security communities could contribute to a broader understanding of how user-generated content can be both creative and secure.

More research should be conducted on the detection of malware in Minecraft mods in addition to the clients studied in this research. As recently as June 2023, mods have been found to contain malware which was spread to thousands of players (Goodin, 2023). Many of the same features for analyzing Java code in this research could likely be useful when analyzing JAR files for Minecraft mods. Furthermore, malware research would benefit from investigating other video game related software for popular games like GTA V, Skyrim, Garry's Mod, and many others. This research showed that there are complexities when analyzing Minecraft launchers compared to traditional malware. It is possible that other games exhibit similar nuance when analyzing user-made content.

This thesis provides useful ways for looking at Minecraft malware in clients and may be useful for similar Minecraft-related software. It is, however, worth noting that a significant outcome of this study was understanding how complicated malware research can be. The generated model from these findings may not be directly applicable to clients or mods for other games since it was specifically tailored to Minecraft clients. The methodology still provides a useful framework for how a security researcher could investigate other video game software. Having an intimate relationship with the game and its community proved to be effective for this Minecraft client research. This leaves open the possibility of educational initiatives aimed at raising awareness among players and independent software developers about potential security research. There would be value in security researchers working with game scholars to get comprehensive insights into the malware landscape in gaming.

6 Conclusion

This research has shed light on the complexities of analyzing malware in the unique context of Minecraft clients. It is beneficial to understand the context around gaming and the independent software development community to analyze specific malware (rather than solely relying on traditional analysis methods). The features that suggest malware presence were too complicated to analyze without machine learning techniques because no single feature or features correlate highly to malware classification. Combined behavioral and static analysis techniques are a powerful tool for this approach when coupled with effective classifiers. Portable Executable (PE) files performed best with the Naïve Bayes classifier using a set of 5 best features. PE files attained an 80% combined precision and recall score. This model scored 100% in recall, demonstrating the ability to not misidentify malware as non-malicious. Java Archive (JAR) files performed best with the K's nearest neighbor classifier using a set of 18 best features. JAR files attained an 87% combined score with a 91% recall score, also demonstrating good performance in correct malware identification.

This research has emphasized the significance that both the PE and JAR models attained high recall scores in malware detection meaning they perform well at correctly identifying actual malware. This high recall performance for each model suggests the potential utility in safeguarding users of Minecraft clients from unseen threats. Minecraft and other similar gaming communities are incredibly dynamic with hundreds of newly-developed, independent software uploaded every month. It is important that we continually develop methods for analyzing novel and evolving malware.

The approach demonstrated in this research can serve as an exemplary benchmark for not only the Minecraft community but also for similar gaming platforms and communities. By showcasing the importance of understanding the nuances of each gaming community, I encourage others in the gaming industry to adopt similar security measures. Prioritizing user protection is not just a duty but also a demonstration of a platform's dedication to delivering a secure and enjoyable gaming environment. By following this benchmark, gaming communities can fortify their defenses against evolving threats and establish a standard of excellence in securit

References

Aarsen, T. (2023). Natural language toolkit [computer software]. https://www.nltk.org/:

Abawajy, J., Darem, A., & Alhashmi, A. A. (2021). Feature subset selection for malware detection in smart IoT platforms. *Sensors, 21*(4), 1374.

Ahmadi, M., Ulyanov, D., Semenov, S., Trofimov, M., & Giacinto, G. (2016). Novel feature extraction, selection and fusion for effective malware family classification. Paper presented at the *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy,* 183-194.

An, N., Duff, A., Noorani, M., Weber, S., & Mancoridis, S. (2018). Malware anomaly detection on virtual assistants. Paper presented at the *2018 13th International Conference on Malicious and Unwanted Software (MALWARE),* 124-131.

Aslan, Ö, & Erdal, A. (2022). Malware detection method based on file and registry operations using machine learning. *Sakarya University Journal of Computer and Information Sciences, 5*(2), 134-146.

Barr-Smith, F., Ugarte-Pedrero, X., Graziano, M., Spolaor, R., & Martinovic, I. (2021). Survivalism: Systematic analysis of windows malware livingoff-the-land. Paper presented at the *2021 IEEE Symposium on Security and Privacy (SP),* 1557-1574.

Björkskog, G. (2019). Detecting cheaters who utilise third-party software to gain an advantage in multiplayer video games. https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1322397&dswid=4210

Burnap, P., French, R., Turner, F., & Jones, K. (2018). Malware classification using self organising feature maps and machine activity data. *Computers & Security, 73*, 399-410.

Cacek, J. (2023). Jd-cli - command line java decompiler [computer software]. https://github.com/intoolswetrust/jd-cli:

Canzanese, R., Kam, M., & Mancoridis, S. (2011). Inoculation against malware infection using kernel-level software sensors. Paper presented at the *Proceedings of the 8th ACM International Conference on Autonomic Computing,* 101-110.

Canzanese, R., Kam, M., & Mancoridis, S.Toward an automatic, online behavioral malware classification system. Paper presented at the *2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems,* 111-120. https://ieeexplore.ieee.org/abstract/document/6676498

Clement, J. (2021). *Cumulative number of copies of minecraft sold worldwide as of april 2021*. Statista.

Clement, J. (2022). *Lifetime unit sales generated by grand theft auto V worldwide as of may 2022*. Statista.

Croft, P. (2023). *How to get rid of minecraft malware and stay safe while playing minecraft.* https://allaboutcookies.org/how-to-remove-minecraft-launcher-malware#:~:text=CurseForge%20reported%20that%20the%20infected,anytime%20you%20download%20files%20online.

Curseforge. (2022). Curesforge minecraft mods. https://www.curseforge.com/minecraft/mc-mods

Gibert, D., Mateu, C., & Planes, J. (2020). HYDRA: A multimodal deep learning framework for malware classification. *Computers & Security, 95*, 101873.

Goodin, D. (2023). *Dozens of popular minecraft mods found infected with fracturiser malware.* https://arstechnica.com/information-technology/2023/06/dozens-of-popular-minecraft-mods-found-infected-with-fracturiser-malware/

Google. (2023). Compact language detector v3 [computer software]. https://github.com/google/cld3:

Grayson, N. (2018). Star wars: KOTOR fan remake shutting down after cease and desist from lucasfilm. https://kotaku.com/star-wars-kotor-fan-remake-shutting-down-after-cease-a-1829720602

GTA5-mods. (2022). *5Mods*. GTA5-mods.

Hampton, N., Baig, Z., & Zeadally, S. (2018). Ransomware behavioural analysis on windows platforms. *Journal of Information Security and Applications, 40*, 44-51.

Hautamaki, Anssi Kanervisto and Tomi Kinnunen and Ville. (2022). GAN-aimbots: Using machine learning for cheating in first person shooters. *IEEE Transactions on Games,* https://doi.org/https://doi.org/10.48550/arXiv.2205.07060 Focus to learn more

Karkallis, P., Blasco, J., Suarez-Tangil, G., & Pastrana, S. (2021). Detecting video-game injectors exchanged in game cheating communities. Paper presented at the 305-324.

Kaspersky. (2021). *Minecraft most malware-infected game on the market with 228k users affected*. Kaspersky.

Kim, J., & Park, K. (2022). Ransomware classification framework using the behavioral performance visualization of execution objects. *Computers, Materials & Continua, 72*(2), 3401-3424.

Ladisa, P., Plate, H., Martinez, M., Barais, O., & Ponta, S. E. (2022). Towards the detection of malicious java packages. Paper presented at the *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses,* 63-72.

Liang, G., Pang, J., & Dai, C. (2016). A behavior-based malware variant classification technique. *International Journal of Information and Education Technology, 6*(4), 291.

Livingston, C. (2017). GTA modding tool OpenIV shuts down due to cease and desist from take-two (updated). https://www.pcgamer.com/gta-modding-tool-openiv-shuts-down-claiming-cease-and-desist-from-take-two/#:~:text=News-,GTA%20modding%20tool%20OpenIV%20shuts%20down%20due%20to%20cease,from%20Take%2DTwo%20(Updated)&text=The%20tool%20has%20been%20essential,due%20to%20a%20legal%20notice.

Matsuda, W., Fujimoto, M., & Mitsunaga, T. (2020). Detection of malicious tools by monitoring DLL using deep learning. *Journal of Information Processing, 28*, 1052-1064.

Nadji, Y., Antonakakis, M., Perdisci, R., & Lee, W. (2011). Understanding the prevalence and use of alternative plans in malware with network games. Paper presented at the *Proceedings of the 27th Annual Computer Security Applications Conference,* 1-10.

Naseem, F. N., Aris, A., Babun, L., Tekiner, E., & Uluagac, A. S. (2021). MINOS: A lightweight real-time cryptojacking detection system. Paper presented at the *Ndss,*

Nguyen, T. D., Marchal, S., Miettinen, M., Fereidooni, H., Asokan, N., & Sadeghi, A. (2018). (2018). DÏoT: A federated self-learning anomaly detection system for IoT. Paper presented at the *2019 IEEE 39th*

International Conference on Distributed Computing Systems (ICDCS), 756-767.

Oyama, Y. (2018). Trends of anti-analysis operations of malwares observed in API call logs. *Journal of Computer Virology and Hacking Techniques, 14*(1), 69-85.

Piskozub, M., Spolaor, R., & Martinovic, I. (2019). MalAlert. *Acm Sigmetrics Performance Evaluation Review,* https://doi.org/10.1145/3308897.3308961

Poor, N. (2014). Computer game modders' motivations and sense of community: A mixed-methods approach. *New Media & Society, 16*(8), 1249-1267. https://doi.org/10.1177/1461444813504266

Schultz, M. G., Eskin, E., Zadok, F., & Stolfo, S. J. (2000). Data mining methods for detection of new malicious executables. Paper presented at the *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001,* 38-49.

Sgandurra, D., Muñoz-González, L., Mohsen, R., & Lupu, E. C. (2016). Automated dynamic analysis of ransomware: Benefits, limitations and use for detection. *arXiv Preprint arXiv:1609.03020,*

Shafiq, M. Z., Tabish, S., & Farooq, M. (2009). PE-probe: Leveraging packer detection and structural information to detect malicious portable executables. Paper presented at the *Proceedings of the Virus Bulletin Conference (VB), , 8*

Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal, 27*(3), 379-423.

Smutz, C., & Stavrou, A.Malicious PDF detection using metadata and structural features. Paper presented at the *Proceedings of the 28th Annual Computer Security Applications Conference,* 239-248.

StatCounter. (2023). *Desktop windows version market share worldwide .* https://gs.statcounter.com/windows-version-market-share/desktop/worldwide/#monthly-202307-202307-bar

Tabish, S. M., Shafiq, M. Z., & Farooq, M. (2009). Malware detection using statistical analysis of byte-level file content. Paper presented at the *Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics,* 23-31.

Tekerek, A., & Yapici, M. M. (2022). A novel malware classification and augmentation model based on convolutional neural network. *Computers & Security, 112*, 102515.

Unterbrink, N. L. a. H. (2021). *Cheating the cheater: How adversaries are using backdoored video game cheat engines and modding tools*. Cisco.

Verma, V., Muttoo, S. K., & Singh, V. B. (2020). Multiclass malware classification via first-and second-order texture statistics. *Computers & Security, 97*, 101895.

Vyas, R., Luo, X., McFarland, N., & Justice, C. (2017). Investigation of malicious portable executable file detection on the network using supervised learning techniques. Paper presented at the *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM),* 941-946.

Wang, Q., Hassan, W. U., Li, D., Jee, K., Yu, X., Zou, K., Rhee, J., Chen, Z., Cheng, W., & Gunter, C. A. (2020). You are what you do: Hunting stealthy malware via data provenance analysis. Paper presented at the *Ndss,*

Williams, C. (2016). *Double KO! capcom's street fighter V installs hidden rootkit on PCs*. The Register.

Yuk, C. K., & Seo, C. J. (2022). Static analysis and machine learning-based malware detection system using PE header feature values. *International Journal of Innovative Research and Scientific Studies, 5*(4), 281-288.

Zhao, G., Xu, K., Xu, L., & Wu, B. (2015). Detecting APT malware infections based on malicious DNS and traffic analysis. *IEEE Access, 3*, 1132-1142.

Zou, F., Zhang, S., Rao, W., & Yi, P. (2015). Detecting malware based on DNS graph mining. *International Journal of Distributed Sensor Networks, 11*(10), 102687.