Malicious Game Client Detection Using Feature Extraction and Machine Learning

# INTRO

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 for these games are also immensely popular. One popular repository for these types of extensions, 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.

Many studies have looked at key indicators for malware on infected machines (An et al., 2018; Canzanese et al., ; 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. There has only been limited research on malware indicators in Jar files in partial due to the complexity of 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 my 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).

# LITERATURE REVIEW

## 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.

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.

## 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 user’s install third party clients. The developer of Minecraft, Mojang, does provide their own launcher and you can add mods to it, but 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](http://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.

## 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.

## Anomaly Detection

Building a machine learning model for detecting video game malware will require 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.

## 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.

## Live Detection Summary

### 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. showed that malware was more likely to be transmitted over DNS requests to untrusted sources (2015). 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.

### 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 is also 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.). 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. Because of this 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 start up 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.

### Persistence

The goal of malware is usually beyond a one-time code execution. Attackers usually want to be able to maintain some sort of access or control over a system. Because of this, malware authors try to create persistent code. A previously mentioned 2022 study 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.

### Memory

Looking at how an application behaves in memory is crucial to understanding if 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.

### Anomaly Detection

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. All the features that will be used for live analysis are summarized in Figure 1.

Figure : Summary of live data collection features

## Static Detection Summary

Static analysis, when combined with live 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 live 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 can also 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 : 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. also noted that many of the suspicious API calls will result in an exception as malware is attempting to check its permissions scope. In order 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. In addition to entropy of strings, they also looked at the quantity of sensitive keywords in the code.

# 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 live analysis methods. From this data that used various machine learning classification techniques to discover which features are most prominent and create a model to recognize malicious and non-malicious clients.

Figure : Summary of methods.

## STEP 1: Obtain Malware Samples

This study will use 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 will need to obtain 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 will find as many Minecraft clients that are generally known to be safe and widely used which will later help distinguish between malware and non-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 : 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 show casing 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 normal clients and cheat clients, the distinction being cheat clients are meant to provide in-game hacks. Mods are not being used for analysis in this research. Each used software was manually researched through a combination of Google, Youtube, SpigotMC, CurseForge, and other Minecraft software repositories.

In addition to VirusTotal, client samples were found manually using common Minecraft software repositories, such as 9minecraft and cheatermad. Additionally, forums and discussion posts were used as reference points to locate clients individually, for example Lunar Client and Technic Launcher. Samples that were found manually were uploaded to VirusTotal to assemble more information on the number of submissions and submission dates.

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. The table below gives details on the quantity of samples that were submitted for each of the given years.

Table : 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 |

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 amount of vendors that flagged files as malicious.

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Mean | Median | Min | Max |
| Malicious File Size (MB) | 21.21 | 7.33 | 0.05 | 147.27 |
| Non-malicious File Size (MB) | 28.54 | 6.93 | 0.01 | 148.06 |
| Malicious Submissions | 498.31 | 16.5 | 1 | 10605 |
| Non-malicious Submissions | 20341.29 | 400 | 1 | 679802 |
| Malicious Detections | 6.88 | 5 | 1 | 39 |
| Non-malicious Detections | 0 | 0 | 0 | 0 |

In an effort to maintain closely matched samples, the non-malware samples were found within a similar timeframe to the malware samples, ranging from 2019 to 2023, with many more samples being included from 2022 and 2023 than the other years. The purpose of closely matching samples is to ensure that the machine learning algorithms has sufficient data from malicious and safe samples. Based on the useable samples, the non-malware clients are aggregated to generalize the activity of non-malicious clients.

## STEP 2: Malware Sandbox

A sandbox environment allows testing of live 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 marketshare (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 outlined in table X 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.

## STEP 3: Feature Extraction

### Live sample feature extraction

Based on the research presented in the literature review, the table below summarizes the features that will be extracted from the sandbox report. The sandbox report from Any.run is given in Json format and can easily be parsed with Python.

#### Network features

For extracting network features, the script iterates through all of 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.

#### System Calls

In order 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 also 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.

#### Memory

In order to find DLL calls that spawned from the root process, the script for parsing the live samples 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.

#### 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 : 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.

#### Anomaly Detection

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 live data is able to open the webpage using the Selenium Python library and collect this data automatically. Since the 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.

The table below provides a concise summary of all of the types of data that were aggregated from the sandbox samples.

Table : Summary of live analysis features and types of data.

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

### 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 in their differences is that header information from PE files provides a wealth of information about the executable, but the header information from Jars only shows basic archival information, such as the number of files it contains.

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.

#### 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.

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.[[1]](#footnote-1) 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.

#### PE file analysis

##### 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 was aggregated per sample and used as a numerical quantity.

##### 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. The table below shows the values used for the packer signatures.

Table : 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 below 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 : 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.

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

|  |  |
| --- | --- |
| String Analysis | |
| IP Addresses | Numerical quantity |
| URLs | Numerical quantity |
| DLL Calls | Numerical quantity |
| Header Info | |
| Packer signature | Boolean |
| Suspicious Imported Functions | Numerical quantity |

#### Jar file analysis

##### 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 : 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 conducted. First the text was tokenized using the NLTK Python module.[[2]](#footnote-2) A regular expression removed non-alphanumeric characters and then ran it through the GCLD3 language detector.[[3]](#footnote-3) 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 isn’t sufficient for small strings, however, and 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 considered to be 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 and method used to a list of suspicious API calls. Table 10 shows the 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 : 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 in order 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 of 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 : Summary of static analysis features and types of data for Jar files.

|  |  |
| --- | --- |
| Code Analysis | |
| IP Addresses | Numerical quantity |
| Suspicious API Calls | Numerical quantity |
| Empty Catch Clauses | Numerical quantity |
| Sensitive Keywords | Numerical quantity |
| High Entropy Strings | Numerical quantity |
| Jar Header Info | |
| File Count | Numerical quantity |

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 didn’t have any data points either but was kept for PE files.

## STEP 4: Data Analysis

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 separated sample sets were randomly divided up into a 70% training set and 30% test set. 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.

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 70% training set. These figures only show the top 5 countries for aggregated internet requests due to the quantity of countries that are contacted in each dataset. All countries are included in the dataset used in the machine learning algorithm.

Table : 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) |
| **STATIC ANALYSIS** |  |  |
| 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) |

Table : 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 heatmaps 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 narrowing down 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 an aggregate feature set will be more useful for detecting malicious samples. Many of the network features are strongly correlated with each other in both PE and Jar file sample sets.

Figure : Heatmap of PE file features

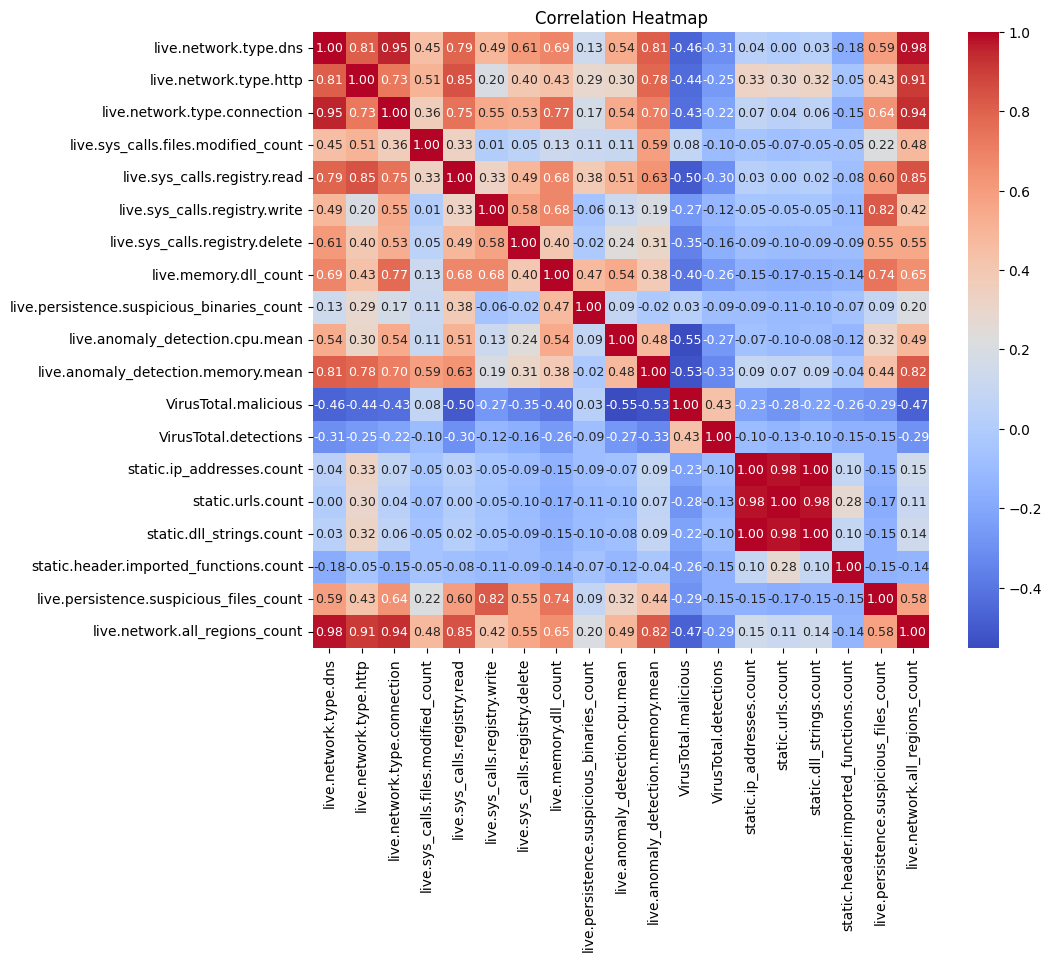
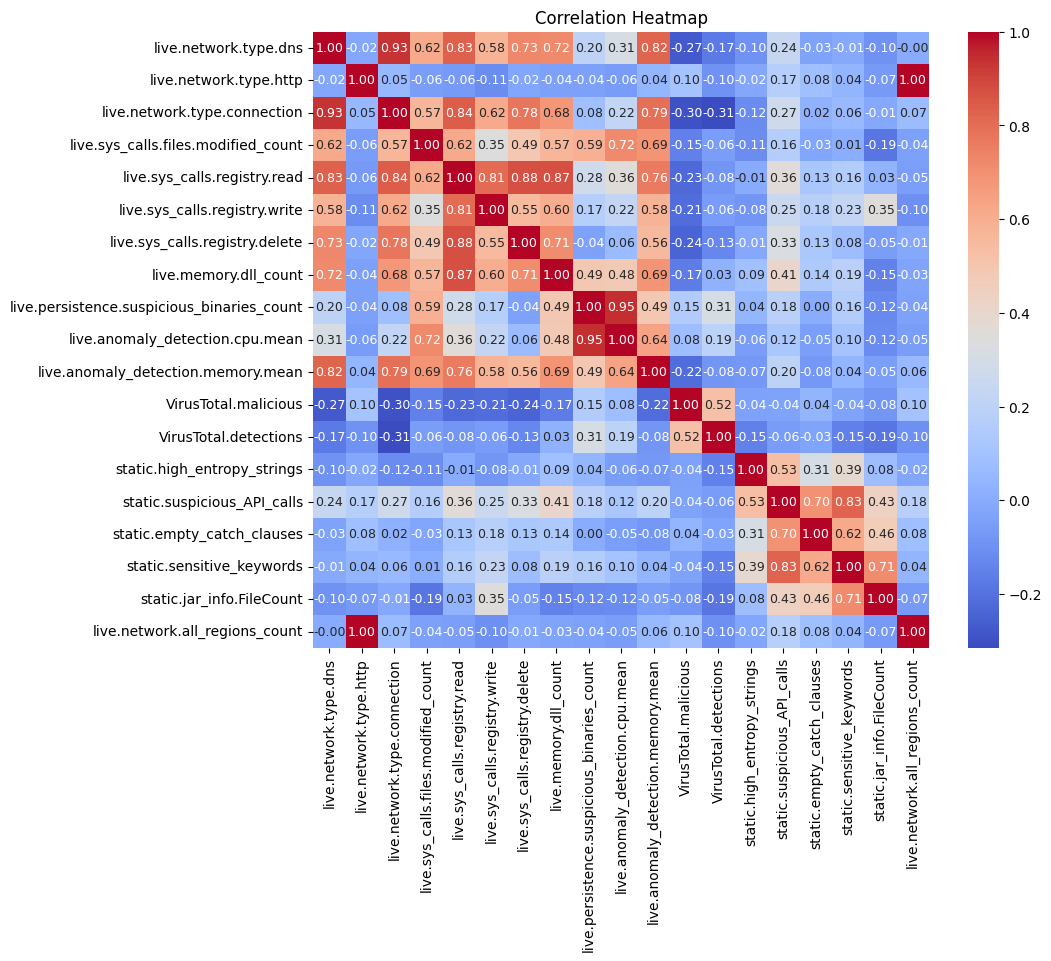


Figure : Heatmap of Jar file features



Six different machine learning classification algorithms were used in this study: classification tree, neural network (NN), naïve Bayes (NB), K’s nearest neighbor (KNN), support vector machine (SVM), and random forest. 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 in order to ensure consistency across all the classifier methods.

Based on the heatmap data, many points are so strongly correlated they practically are the same data points. In order to account for this behavior, filters were used when processing the datasets. The first type of filter was a correlation filter with cutoffs at 90% and above correlation as well as 80% and above correlation. 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 from the ‘SelectKBest’ Scikit-learn function was used as another filter. These were tested on no K best feature filtering as well as 5, 10, and 18 best features. All tests were aggregated and run multiple times so that each K best feature was tested with each of the 3 correlation filters: none, 90%, and 80%.

# RESULTS

Table 14 and Table 15 show the results of each of these runs. Many of the classifiers in the PE samples had low precision and recall scores due to the relatively low sample size in the training set for PE files. Additionally, the 18 K best features and 80% correlation filter failed on both PE and Jar sample sets because it cut out too many features. A 10-fold cross validation score was given to each sample set and is represented in the column ‘CV\_avg’ as well as the standard deviation for cross validation. This provides a rough idea of how good the training set data is.

Table : Machine learning results for PE files

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Name | Classifier | Accuracy | Precision | Recall | F1\_score | CV\_avg | CV\_std\_dev |
| Exe - No correlation filter, no K best feature | | | | |  |  |  |
|  | classification tree | 0.555556 | 0.5 | 0.75 | 0.6 | 0.75 | 0.25 |
|  | neural network | 0.666667 | 0.571429 | 1 | 0.727273 | 0.55 | 0.269258 |
|  | naives bayes | 0.555556 | 0.5 | 1 | 0.666667 | 0.55 | 0.269258 |
|  | KNN | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.7 | 0.331662 |
|  | SVM | 0.666667 | 0.666667 | 0.5 | 0.571429 | 0.3 | 0.244949 |
|  | random forest | 0.555556 | 0.5 | 0.75 | 0.6 | 0.7 | 0.244949 |
| Exe - 90% correlation filter, no K best feature | | | | |  |  |  |
|  | classification tree | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.65 | 0.320156 |
|  | neural network | 0.555556 | 0 | 0 | 0 | 0.5 | 0 |
|  | naives bayes | 0.555556 | 0.5 | 0.75 | 0.6 | 0.55 | 0.35 |
|  | KNN | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.7 | 0.331662 |
|  | SVM | 0.777778 | 1 | 0.5 | 0.666667 | 0.5 | 0.316228 |
|  | random forest | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.75 | 0.25 |
| Exe - 80% correlation filter, no K best feature | | | | |  |  |  |
|  | classification tree | 0.555556 | 0.5 | 0.5 | 0.5 | 0.65 | 0.320156 |
|  | neural network | 0.555556 | 0 | 0 | 0 | 0.5 | 0 |
|  | naives bayes | 0.444444 | 0.4 | 0.5 | 0.444444 | 0.45 | 0.35 |
|  | KNN | 0.555556 | 0.5 | 0.5 | 0.5 | 0.5 | 0.387298 |
|  | SVM | 0.555556 | 0.5 | 0.75 | 0.6 | 0.6 | 0.2 |
|  | random forest | 0.555556 | 0.5 | 0.5 | 0.5 | 0.7 | 0.244949 |
| Exe - No correlation filter, 5 K best feature | | | | |  |  |  |
|  | classification tree | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.7 | 0.331662 |
|  | neural network | 0.555556 | 0.5 | 0.25 | 0.333333 | 0.55 | 0.15 |
|  | naives bayes | 0.777778 | 0.666667 | 1 | 0.8 | 0.7 | 0.331662 |
|  | KNN | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.7 | 0.331662 |
|  | SVM | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.65 | 0.320156 |
|  | random forest | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.75 | 0.25 |
| Exe - 90% correlation filter, 5 K best feature | | | | |  |  |  |
|  | classification tree | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.65 | 0.390512 |
|  | neural network | 0.666667 | 0.571429 | 1 | 0.727273 | 0.7 | 0.244949 |
|  | naives bayes | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.65 | 0.390512 |
|  | KNN | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.7 | 0.331662 |
|  | SVM | 0.666667 | 0.666667 | 0.5 | 0.571429 | 0.65 | 0.320156 |
|  | random forest | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.7 | 0.244949 |
| Exe - 80% correlation filter, 5 K best feature | | | | |  |  |  |
|  | classification tree | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.8 | 0.244949 |
|  | neural network | 0.555556 | 0 | 0 | 0 | 0.55 | 0.15 |
|  | naives bayes | 0.555556 | 0.5 | 0.75 | 0.6 | 0.75 | 0.33541 |
|  | KNN | 0.555556 | 0.5 | 0.5 | 0.5 | 0.5 | 0.387298 |
|  | SVM | 0.444444 | 0.428571 | 0.75 | 0.545455 | 0.65 | 0.320156 |
|  | random forest | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.6 | 0.374166 |
| Exe - No correlation filter, 10 K best feature | | | | |  |  |  |
|  | classification tree | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.65 | 0.320156 |
|  | neural network | 0.444444 | 0.444444 | 1 | 0.615385 | 0.5 | 0.223607 |
|  | naives bayes | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.6 | 0.374166 |
|  | KNN | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.7 | 0.331662 |
|  | SVM | 0.444444 | 0.333333 | 0.25 | 0.285714 | 0.55 | 0.35 |
|  | random forest | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.7 | 0.244949 |
| Exe - 90% correlation filter, 10 K best feature | | | | |  |  |  |
|  | classification tree | 0.555556 | 0.5 | 1 | 0.666667 | 0.55 | 0.35 |
|  | neural network | 0.555556 | 0.5 | 0.25 | 0.333333 | 0.55 | 0.269258 |
|  | naives bayes | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.6 | 0.374166 |
|  | KNN | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.7 | 0.331662 |
|  | SVM | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.5 | 0.387298 |
|  | random forest | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.7 | 0.244949 |
| Exe - 80% correlation filter, 10 K best feature | | | | |  |  |  |
|  | FAILED |  |  |  |  |  |  |
| Exe - No correlation filter, 18 K best feature | | | | |  |  |  |
|  | classification tree | 0.555556 | 0.5 | 0.75 | 0.6 | 0.75 | 0.25 |
|  | neural network | 0.555556 | 0 | 0 | 0 | 0.55 | 0.15 |
|  | naives bayes | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.7 | 0.244949 |
|  | KNN | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.7 | 0.331662 |
|  | SVM | 0.666667 | 0.571429 | 1 | 0.727273 | 0.35 | 0.229129 |
|  | random forest | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.7 | 0.244949 |
| Exe - 90% correlation filter, 18 K best feature | | | | |  |  |  |
|  | classification tree | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.65 | 0.320156 |
|  | neural network | 0.555556 | 0 | 0 | 0 | 0.5 | 0 |
|  | naives bayes | 0.555556 | 0.5 | 0.75 | 0.6 | 0.7 | 0.244949 |
|  | KNN | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.7 | 0.331662 |
|  | SVM | 0.555556 | 0.5 | 0.5 | 0.5 | 0.6 | 0.3 |
|  | random forest | 0.666667 | 0.6 | 0.75 | 0.666667 | 0.7 | 0.244949 |
| Exe - 80% correlation filter, 18 K best feature | | | | |  |  |  |
|  | FAILED |  |  |  |  |  |  |

Table : Machine learning results for Jar files

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Name | Classifier | Accuracy | Precision | Recall | F1\_score | CV\_avg | CV\_std\_dev |
| Jar - No correlation filter, no K best feature | | | | |  |  |  |
|  | classification tree | 0.631579 | 0.666667 | 0.727273 | 0.695652 | 0.605 | 0.173853 |
|  | neural network | 0.473684 | 0.571429 | 0.363636 | 0.444444 | 0.575 | 0.255196 |
|  | naives bayes | 0.684211 | 0.727273 | 0.727273 | 0.727273 | 0.48 | 0.227156 |
|  | KNN | 0.736842 | 0.714286 | 0.909091 | 0.8 | 0.695 | 0.142215 |
|  | SVM | 0.473684 | 0.538462 | 0.636364 | 0.583333 | 0.63 | 0.26096 |
|  | random forest | 0.631579 | 0.625 | 0.909091 | 0.740741 | 0.7 | 0.156525 |
| Jar - 90% correlation filter, no K best feature | | | | |  |  |  |
|  | classification tree | 0.631579 | 0.7 | 0.636364 | 0.666667 | 0.535 | 0.229183 |
|  | neural network | 0.789474 | 0.769231 | 0.909091 | 0.833333 | 0.595 | 0.108282 |
|  | naives bayes | 0.736842 | 0.6875 | 1 | 0.814815 | 0.625 | 0.173566 |
|  | KNN | 0.736842 | 0.714286 | 0.909091 | 0.8 | 0.695 | 0.142215 |
|  | SVM | 0.684211 | 0.647059 | 1 | 0.785714 | 0.42 | 0.169115 |
|  | random forest | 0.736842 | 0.714286 | 0.909091 | 0.8 | 0.7 | 0.192354 |
| Jar - 80% correlation filter, no K best feature | | | | |  |  |  |
|  | classification tree | 0.736842 | 0.75 | 0.818182 | 0.782609 | 0.525 | 0.271339 |
|  | neural network | 0.421053 | 0.5 | 0.454545 | 0.47619 | 0.53 | 0.196469 |
|  | naives bayes | 0.684211 | 0.666667 | 0.909091 | 0.769231 | 0.605 | 0.186346 |
|  | KNN | 0.842105 | 0.833333 | 0.909091 | 0.869565 | 0.67 | 0.151987 |
|  | SVM | 0.631579 | 0.611111 | 1 | 0.758621 | 0.675 | 0.229401 |
|  | random forest | 0.789474 | 0.769231 | 0.909091 | 0.833333 | 0.65 | 0.204939 |
| Jar - No correlation filter, 5 K best feature | | | | |  |  |  |
|  | classification tree | 0.526316 | 0.571429 | 0.727273 | 0.64 | 0.585 | 0.282002 |
|  | neural network | 0.578947 | 0.6 | 0.818182 | 0.692308 | 0.77 | 0.170587 |
|  | naives bayes | 0.578947 | 0.6 | 0.818182 | 0.692308 | 0.795 | 0.145688 |
|  | KNN | 0.526316 | 0.555556 | 0.909091 | 0.689655 | 0.625 | 0.132759 |
|  | SVM | 0.631579 | 0.611111 | 1 | 0.758621 | 0.77 | 0.128841 |
|  | random forest | 0.473684 | 0.533333 | 0.727273 | 0.615385 | 0.62 | 0.2502 |
| Jar - 90% correlation filter, 5 K best feature | | | | |  |  |  |
|  | classification tree | 0.631579 | 0.611111 | 1 | 0.758621 | 0.62 | 0.11225 |
|  | neural network | 0.684211 | 0.777778 | 0.636364 | 0.7 | 0.505 | 0.213834 |
|  | naives bayes | 0.631579 | 0.625 | 0.909091 | 0.740741 | 0.74 | 0.122066 |
|  | KNN | 0.526316 | 0.555556 | 0.909091 | 0.689655 | 0.645 | 0.157242 |
|  | SVM | 0.684211 | 0.666667 | 0.909091 | 0.769231 | 0.625 | 0.206458 |
|  | random forest | 0.684211 | 0.647059 | 1 | 0.785714 | 0.645 | 0.192938 |
| Jar - 80% correlation filter, 5 K best feature | | | | |  |  |  |
|  | classification tree | 0.631579 | 0.642857 | 0.818182 | 0.72 | 0.69 | 0.154596 |
|  | neural network | 0.684211 | 0.647059 | 1 | 0.785714 | 0.6 | 0.130384 |
|  | naives bayes | 0.631579 | 0.625 | 0.909091 | 0.740741 | 0.72 | 0.169115 |
|  | KNN | 0.578947 | 0.578947 | 1 | 0.733333 | 0.62 | 0.15843 |
|  | SVM | 0.684211 | 0.647059 | 1 | 0.785714 | 0.67 | 0.151987 |
|  | random forest | 0.578947 | 0.6 | 0.818182 | 0.692308 | 0.715 | 0.14151 |
| Jar - No correlation filter, 10 K best feature | | | | |  |  |  |
|  | classification tree | 0.736842 | 0.6875 | 1 | 0.814815 | 0.595 | 0.108282 |
|  | neural network | 0.578947 | 0.588235 | 0.909091 | 0.714286 | 0.595 | 0.155644 |
|  | naives bayes | 0.578947 | 0.6 | 0.818182 | 0.692308 | 0.795 | 0.145688 |
|  | KNN | 0.631579 | 0.611111 | 1 | 0.758621 | 0.695 | 0.087892 |
|  | SVM | 0.631579 | 0.611111 | 1 | 0.758621 | 0.77 | 0.128841 |
|  | random forest | 0.684211 | 0.647059 | 1 | 0.785714 | 0.595 | 0.155644 |
| Jar - 90% correlation filter, 10 K best feature | | | | |  |  |  |
|  | classification tree | 0.684211 | 0.647059 | 1 | 0.785714 | 0.625 | 0.173566 |
|  | neural network | 0.473684 | 0.555556 | 0.454545 | 0.5 | 0.62 | 0.213542 |
|  | naives bayes | 0.631579 | 0.625 | 0.909091 | 0.740741 | 0.695 | 0.168003 |
|  | KNN | 0.578947 | 0.578947 | 1 | 0.733333 | 0.65 | 0.130384 |
|  | SVM | 0.578947 | 0.588235 | 0.909091 | 0.714286 | 0.67 | 0.208806 |
|  | random forest | 0.631579 | 0.611111 | 1 | 0.758621 | 0.58 | 0.181934 |
| Jar - 80% correlation filter, 10 K best feature | | | | |  |  |  |
|  | classification tree | 0.789474 | 0.769231 | 0.909091 | 0.833333 | 0.6 | 0.204939 |
|  | neural network | 0.631579 | 0.611111 | 1 | 0.758621 | 0.435 | 0.249048 |
|  | naives bayes | 0.684211 | 0.666667 | 0.909091 | 0.769231 | 0.715 | 0.212191 |
|  | KNN | 0.842105 | 0.833333 | 0.909091 | 0.869565 | 0.67 | 0.151987 |
|  | SVM | 0.578947 | 0.578947 | 1 | 0.733333 | 0.67 | 0.102956 |
|  | random forest | 0.789474 | 0.769231 | 0.909091 | 0.833333 | 0.635 | 0.219146 |
| Jar - No correlation filter, 18 K best feature | | | | |  |  |  |
|  | classification tree | 0.631579 | 0.625 | 0.909091 | 0.740741 | 0.665 | 0.118427 |
|  | neural network | 0.578947 | 0.578947 | 1 | 0.733333 | 0.675 | 0.166208 |
|  | naives bayes | 0.578947 | 0.6 | 0.818182 | 0.692308 | 0.74 | 0.122066 |
|  | KNN | 0.578947 | 0.578947 | 1 | 0.733333 | 0.65 | 0.130384 |
|  | SVM | 0.578947 | 0.578947 | 1 | 0.733333 | 0.645 | 0.157242 |
|  | random forest | 0.631579 | 0.611111 | 1 | 0.758621 | 0.625 | 0.132759 |
| Jar - 90% correlation filter, 18 K best feature | | | | |  |  |  |
|  | classification tree | 0.684211 | 0.692308 | 0.818182 | 0.75 | 0.61 | 0.27 |
|  | neural network | 0.578947 | 0.666667 | 0.545455 | 0.6 | 0.55 | 0.315436 |
|  | naives bayes | 0.684211 | 0.666667 | 0.909091 | 0.769231 | 0.65 | 0.130384 |
|  | KNN | 0.736842 | 0.714286 | 0.909091 | 0.8 | 0.695 | 0.142215 |
|  | SVM | 0.631579 | 0.666667 | 0.727273 | 0.695652 | 0.445 | 0.2067 |
|  | random forest | 0.736842 | 0.714286 | 0.909091 | 0.8 | 0.66 | 0.216564 |
| Jar - 80% correlation filter, 18 K best feature | | | | |  |  |  |
|  | FAILED |  |  |  |  |  |  |

PE files showed the best results for the SVM classifier with a 90% correlation filter and no K best feature. For Jar files 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 a 90% correlation filter, based on the previously observed data in Table 14. Figure 7 highlights the relative performance of each of these classification methods for Jar files with a 90% correlation filter and 10 K best features. The ROC graph in Figure 7 highlights the best classification algorithm for Jar files with an 80% correlation filter, based on the data from Table 15.

Figure : ROC graph for PE files with 90% correlation filter and no K best features

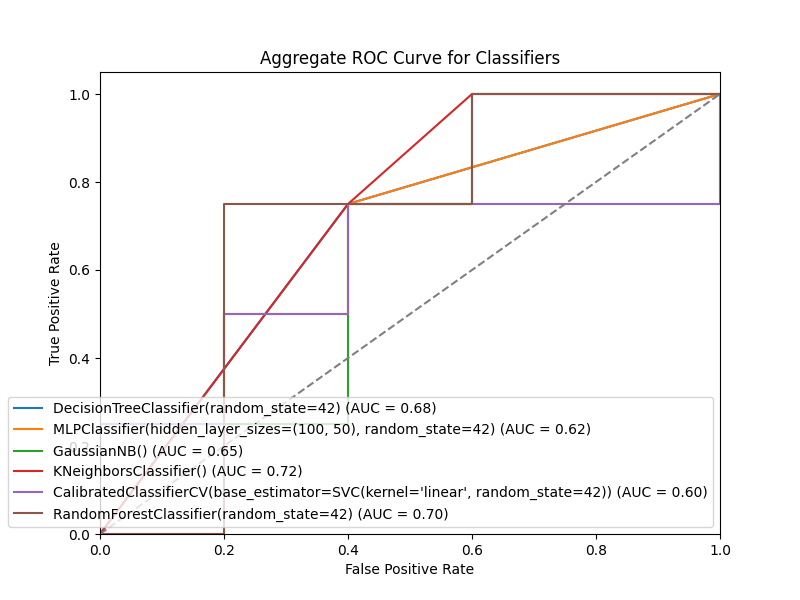
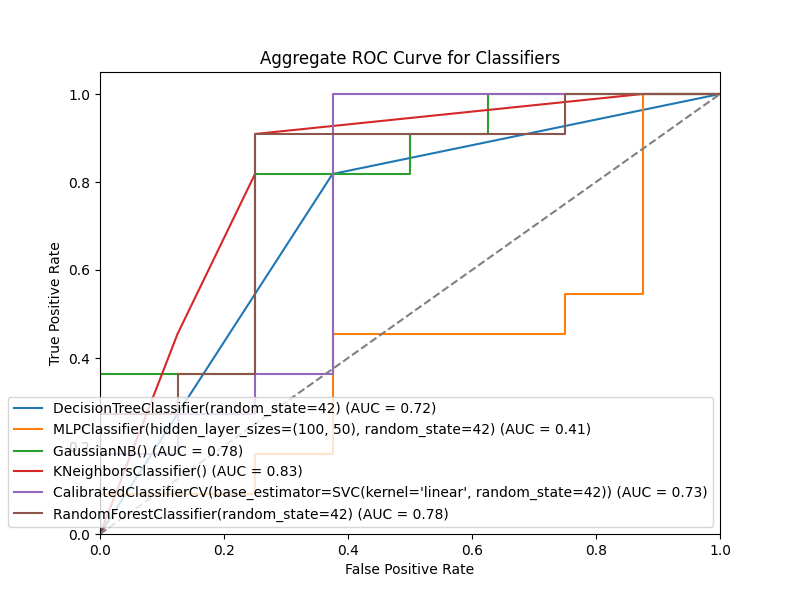


Figure : ROC graph for PE files with 80% correlation filter and no K best features



# CITATIONS

References

Ahmadi, M., Ulyanov, D., Semenov, S., Trofimov, M., & Giacinto, G. (2016). (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). (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). (2021). Survivalism: Systematic analysis of windows malware living-off-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>

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>

Canzanese, R., Kam, M., & Mancoridis, S. (2011). (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.

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.

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

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). (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). (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). (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). (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). (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.

Shafiq, M. Z., Tabish, S., & Farooq, M. (2009). (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>

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

Vyas, R., Luo, X., McFarland, N., & Justice, C. (2017). (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). (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.

1. https://github.com/intoolswetrust/jd-cli [↑](#footnote-ref-1)
2. https://www.nltk.org/ [↑](#footnote-ref-2)
3. https://github.com/google/cld3 [↑](#footnote-ref-3)