Generating Cyber Forensic Data

Randall Dickinson and Sasha Petushkov

Abstract

Datasets containing system monitor (Sysmon) data are difficult to find and may not contain enough data for complex training tasks. This project aimed to explore the potential for a Wasserstein Generative Adversarial Network (WGAN) at generating fake system monitor (Sysmon) data. After installing a Sysmon and obtaining data, our project was able to encode most of the data into usable training data using a complex set of preprocessing steps. After training a WGAN on the input data, the generated data was converted back into Sysmon form using a reverse of the steps used to encode the data. Findings showed promising results for this model to generate realistic data for most of the fields from a Sysmon.

Introduction

Sysmon logs will often contain private information about a users' operating system. However, the Sysmon logs must be preserved in their original form if they are to have meaning. This can make datasets with Sysmon logs very difficult to come by. Furthermore, without any fake Sysmon log data generated, how could we train a classifier to detect fake Sysmon log data?

Our approach will center itself around the use of a WGAN to generate fake Sysmon data. GAN's create new data instances that resemble your training data [1]. GAN's are known for being effective models for image generation, but are less often used for text generation.

In particular, our project will face the difficulty of encoding the high volume of string data into a format which a GAN can train on while still retaining its meaning. For the model of WGAN we chose, this would involve organizing our dataset such that all observations for each variable had values between negative one and one. From our dataset, we identified that much of the string data could best be represented in a categorical format, so this would require much preprocessing to find a format which could encode a high volume of categorical strings into a meaningful real-valued format.

This project was centered more around the processing of data, but various WGAN architectures were tried as well. Out of nine architectures tried, we found that an autoencoder architecture resulted in data that had the fewest inconsistencies of incorrect category combinations being generated while still not suffering from mode collapse.

Following the generation of new data, much work was needed to translate that data back to its original format. During this postprocessing, counts were kept of a primary data inconsistency that could be found in the generated data as an empirical means of comparing various WGAN architectures. These inconsistencies were fixed during postprocessing, and the final dataset generated resembled the format of the original dataset very well.

Due to the fairly low percentage of inconsistencies to data generated and the fact that the model was able to avoid mode collapse, we conclude that a WGAN is an effective way to generate Sysmon data. In addition, future ideas for the improvement of Sysmon data generation include training separate models for generating process creation logs vs process termination logs.

Problem Definition

The purpose of this project will be to generate realistic Sysmon logs. Realistic is a subjective measure, and refers to the logs ability to appear like normal logs to a person viewing them. The generation of realistic Sysmon logs plays an important role in computer security. With a Sysmon installed, a user might be able to detect suspicious activity by viewing Sysmon logs. In the event of a security breach, it would be crucial to know if Sysmon logs found on an operating system are legitimate so that the breach can be properly investigated.

The single most difficult task in this project was finding a means to encode string data in such a way that a WGAN could be trained on it. Furthermore, generated data would have to be interpretable in such a way that it could be decoded back into string data while maintaining contextual integrity. Such encoding and decoding methods required deep understanding of the data.

This project was unique and interesting in that it maintained a very open-ended nature. The problem statement itself was to find a means to generate realistic Sysmon data using a WGAN. This allowed our team to explore the data and methods to do so, and gave great flexibility in our means to accomplish this task. As such, we were able to find unique ways to grow our critical thinking skills by breaking down the problem into small components that we ourselves determined and progressing towards the overall goal.

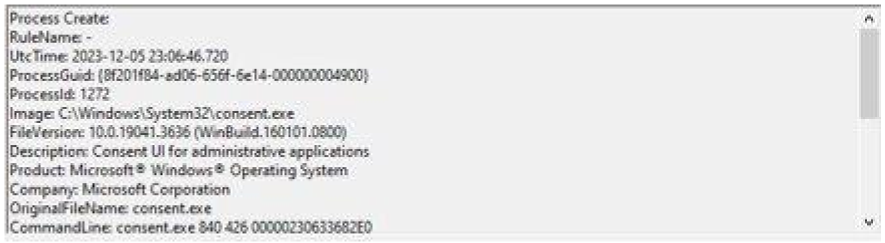
The generation of realistic Sysmon data is particularly useful in the cybersecurity field. By finding a means to generate data using a WGAN, the methods developed in this project could be further refined to supplement a larger project aimed at improving computer security.

Models/Algorithms/Measures

In GANs, mode collapse happens when the generator focuses on producing a limited set of data patterns that deceive the discriminator. It becomes fixated on a few dominant modes in the training data and fails to capture the full diversity of the data distribution [2]. Simply put, the model will generate observations that are nearly identical repeatedly. One of the primary considerations when developing a generative model is to avoid such a situation. Thus, any models that appear to suffer from mode collapse would be labeled as unsuccessful.

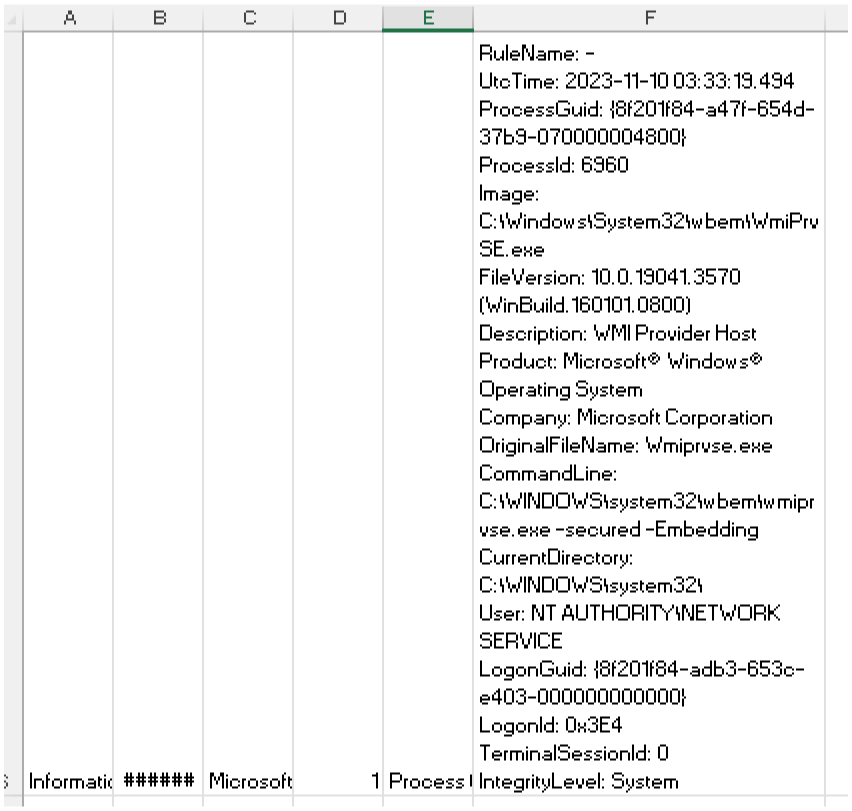
In inspecting the raw data gathered, we found a distinct split in how the data may be structured. The process termination logs were small logs that contained a given amount of information. Depending on whether a given log was for process termination or creation, there would be information on six or 24 variables, respectively. In addition, process creation logs contained all the same variables that were found with process termination. Due to this clear difference in how the data was structured depending on the log type, one of the measures we used to evaluate generated data was how well it kept to this structure of the 18 variables all being present with process creation, or not present with process termination. We will refer to this variable as discrepancies, and define a discrepancy as one of the 18 variables being present for the wrong type of log.

The data was gathered through installing a Sysmon on a single computer and collecting data for one week. Once data was successfully gathered, it was wrangled and tidying to a format that could be easily worked on. This involved rearranging the data into the appropriate number of columns. When the csv was generated of the Sysmon data, it consisted of six columns, of which two were identical for all observations and one could be gathered from the information provided by another column. The sixth column contained 5 or 17 individual variables that were grouped in the same variable, so tidying involved creating many more variables so that each separate piece of information could be represented by its own variable. An example of a Sysmon log and csv view of a Sysmon log can be seen below in **Figure 1** and **Figure 2**.



```
Process Create:
RuleName: -
UtcTime: 2023-12-05 23:06:46.720
ProcessGuid: {8f201f84-ad06-656f-6e14-000000004900}
ProcessId: 1272
Image: C:\Windows\System32\consent.exe
FileVersion: 10.0.19041.3636 (WinBuild.160101.0800)
Description: Consent UI for administrative applications
Product: Microsoft® Windows® Operating System
Company: Microsoft Corporation
OriginalFileName: consent.exe
CommandLine: consent.exe 840 426 00000230633682E0
```

Figure 1. *Example of Sysmon log as seen from the System Monitor.*



```
RuleName: -
UtcTime: 2023-11-10 03:33:19.494
ProcessGuid: {8f201f84-a47f-654d-
37b9-070000004800}
ProcessId: 6960
Image:
C:\Windows\System32\wbem\WmiPrv
SE.exe
FileVersion: 10.0.19041.3570
(WinBuild.160101.0800)
Description: WMI Provider Host
Product: Microsoft® Windows®
Operating System
Company: Microsoft Corporation
OriginalFileName: Wmiprvse.exe
CommandLine:
C:\WINDOWS\system32\wbem\wmipr
vse.exe -secured -Embedding
CurrentDirectory:
C:\WINDOWS\system32\
User: NT AUTHORITY\NETWORK
SERVICE
LogonGuid: {8f201f84-adb3-653c-
e403-000000000000}
LogonId: 0x3E4
TerminalSessionId: 0
IntegrityLevel: System
```

A	B	C	D	E	F
Informatic	######	Microsoft	1	Process	

Figure 1. *Example of the csv format of a Sysmon log in its original format.*

Once the data was tidied, additional operations were performed on the data to eliminate variables that could be determined by the value of another variable. Following the elimination of all redundant information, this project had to simplify some of the variables for the sake of time and complexity. These simplifications will be discussed further in later sections, but signify some of the shortcomings of the project. Further refinement of this work may improve upon these areas, and suggestions are made for future work in the conclusions.

Following the tidying and simplification of the dataset, our project began the largest work of data preprocessing. Upon researching how GAN's are able to train on a given dataset, we found that the data would have to be encoded into real-valued numbers. Through trial and error, first determinations were that these numbers had to be between 0 and 1.

For encoding the string data into numerical values, our project built python dictionaries to represent the possible values of the data as numerical numbers. These dictionaries would be essential in postprocessing to convert numbers back to string data. Further modification was required to change these encodings into one-hot encodings as will be explained in the next section.

The final format of the data consisted of almost 1,000 variables, a result caused by the rapid expansion of the dimension-space when representing a large amount of categorical values through one-hot encodings. Once this format was achieved, our project experimented with nine

different sizes of WGAN models of varying layer size and depth. The goal of this experimentation was to find an optimum architecture that would generate quality Sysmon data according to the measures previously described.
Once the newly generated Sysmon data was obtained, our project had to find a means to decode the generated data back into strings. The most difficult piece of data to decode was the file path strings. Fortunately, the framework for this was set up in the preprocessing step. In the preprocessing, file paths were converted to lists of folder strings. These were able to be used to rebuild the data back into full file paths from the numerical encodings. Subsequently decodings were found for the time data, the random data, and the easily decodable data that was represented in the main encoding dictionary.

Finally with all of the necessary string information gathered from the data, the data had to be converted back to its original nature. This ironically involved “untidying” the data so that it represented its original format. With some work, our script converts the data back into the format of the original csv of Sysmon logs

Implementation/Analysis

As explained previously, our dataset was gathered by installing a Sysmon on an operating system and collecting data for about one week. The first attempt at this produced a dataset full of observations that were nearly identical. By examining the data, we found that the logs appeared to be spammed by Visual Studio Code (VSCode). While VSCode was running, several to dozens of logs were produced each second, resulting in a dataset that consisted of observations that were almost all the same. The grep command from Windows Subsystem for Linux (WSL) was used repeatedly while VSCode was run, leading us to believe that the linter in VSCode was responsible for the anomaly. We observed that the limit for the number of logs was around 50,000, and only the most recent logs were kept. Thus we were able to collect new Sysmon data by continuing to run the Sysmon for another week while intentionally not launching VSCode or WSL on the given operating system. This was successful in collecting a more diverse dataset of over 28,000 observations.

The first process of tidying the data was done in R in the Project.rmd file. This process involved separating variables by the presence of the ‘\n’ character. This decision was made following the observation of gathered data, which showed that the sixth column contains several to many pieces of useful information which belonged in separate columns, but was conveniently separated by newline characters.

While the first part of tidying was done in R, further observation and tidying was done in the Python preprocessing script. Upon further analysis, some of the variables were identified as redundant. The “Level”, “Source”, and “RuleName” variables were identical for all observations. The “Task Category” variable was directly correlated with “Event ID” and thus could be removed. Similarly, the “OriginalFileName” was a subsequence of “Image” that could always be determined and was therefore redundant. Finally, “UtcTime” represented a time that was eight hours past the given time in “Time and Date”, and could therefore be redundant. Thus, we removed all of these variables from the dataset.

Upon further analysis of the dataset we found that the “CommandLine” and “ParentCommandLine” variables were very complicated and varied widely based on the program that was the subject of process creation or termination. As every program has a unique set of command line arguments that may be used, attempting to account for the numerous possibilities would greatly multiply the difficulty of training a model to generate Sysmon data if these arguments had to be included. Given the time constraints of this project, our team decided not to include command line arguments in our trainable data. Instead, our project would include the command line invoked, which is the absolute path to the executable being launched / terminated but without the additional arguments. Recommendations for improvement in future models are provided in the conclusion.

The “Image” variable consisted of the absolute path to the executable being launched / terminated, and the “CurrentDirectory” string corresponded with the current working directory when the log was made. Sometimes these two were identical, although this was not always the case. In this instance, we did aim to capture the possibility of these two being distinct within our model. For each, we converted the file strings to lists of strings representing the various file folders. These lists would be used later in postprocessing. For the purpose of training the WGAN, we encoded only the final folder within the string into a numerical format. The advantage of this is that it allows the dataset used for training to be greatly simplified. Instead of encoding up to eight strings, our dataset only had to encode the final one. The disadvantage of this encoding is that it makes the assumption that a final folder string will only be used once. For example, folder “d” in " /a/b/c/d/” would not be the same folder as in “/c/b/a/d”. However, our encoding method would not discriminate between the two folders, and our postprocessing script would rebuild a generated file path using the first file string found in the original dataset where that particular folder is the final folder (thus our post-processed data would always favor ”/a/b/c/d” in the example provided).

The time data did have meaning of a continuous nature, and thus would be easily trainable for a WGAN. Early efforts on training a model with this data failed, and it was realized that all generated data from the WGAN only produced numbers between zero and one. Thus, we went back to the preprocessing of the dataset to normalize the data. As all of the data was gathered between November 9 - 14, we normalized the day of the month by subtracting nine and dividing by five, resulting in a scale to represent the day of the month that varied from zero to one and could directly be translated back to an appropriate day. Similarly, we normalized the time of day by changing the time to the number of minutes past midnight and dividing by the number of minutes in a day to obtain a scale that spanned from zero to one.

The final step in preprocessing involved the one-hot encoding of categorical data. While we successfully represented the potential string values for various columns by encoding them to numbers using Python dictionaries, these direct encodings could not be used for training. Below is an example encoding for the variable “IntegrityLevel”:

Encoding for IntegrityLevel: {'Low': 0, 'System': 1, nan: 2, 'Medium': 3, 'AppContainer': 4, 'High': 5}

While the time series data was as simple as dividing the encoded values by the total number of values to get a scale normalized between zero and one, this same method would not work for categorical data. Due to the nature of the GAN model, this method would impose an implicit bias on our model that generating a ‘1’ instead of a ‘0’ is better than generating a ‘3’ instead of a ‘0’. However, with categorical data the values are discrete and thus a different method must be used. Thus, upon further research we found the concept of one-hot encoding to be an effective way of encoding our data.

In one-hot encoding, we would instead create size new variables for the example provided above, one for each possible value. We would then place a ‘1’ in the variable that corresponded with the value of the entry, and ‘0’s in the other variables. For example, we would create variables “IntegrityLevelN”, where ‘N’ corresponds with a number 0 - 5. If a given observation were labeled as “Medium”, then we would place a ‘1’ in the column for “IntegrityLevel3”, and ‘0’s in the remaining columns for “IntegrityLevelN”. This method was used on all of the categorical columns. This resulted in the dimensionality of the data exploding from around two dozen dimensions to almost 1,000 dimensions. Such a large dimensional space could make training a model difficult.

After completing preprocessing, our project aimed to train WGAN models of various architectures on the dataset. Preliminary research on similarly-size datasets found that starting with a three layer, 512 node architecture would be a good starting point for training a model. For the various architectures, Tensorflow and Keras functionalities were used to build both the discriminator and the generator that would constitute the WGAN. Preliminary models were built online through a Google Colab notebook. Upon analyzing the first set of results produced by the WGANs, we found that the more complex models produced a wider variety of data and did not suffer from mode collapse. We were limited from producing too large of models in a Google Colab environment with strict random access memory (RAM) limits. Thus, six larger models, up to a maximum size of a six-layer, 1,024 node layer-wide architecture was developed on a desktop pc.

The newly generated data was then immediately translated back to its encoded form by converting the one-hot encoded variables into single variables with the encoded value. For each of the one-hot encoded values, there was a real-valued number between zero and one but never equal to either. However, as an experiment our team chose to find the one-hot encoded variable column which had the max value for each of the original variables and label this value as a ‘1’ for that given variable, and a ‘0’ for all remaining variables. Then we would collapse the one-hot encoded variables into a single variable with the value ‘N’ where the ‘N’ corresponds with the “OneHotEncodedN” variable that was the max for that given variable.

Following the generation of new encoded data from a total of nine different models, our team continued work by creating a postprocessing script to decode the generated data and transform it back into its original format. In order to decode the data back into its original string format, the encoding dictionaries would be required. Thus, the preprocessing script is included in the postprocessing Google Colab notebook and must be run prior to running the postprocessing modules. The first step in postprocessing included building dictionaries of reversed values from the preprocessing steps so that the encoded numbers may be decoded back to strings with meaningful value. This involved creating another dictionary “file_paths”, where the keys of the dictionary represented the encoded values of the last folders of “Image” and “CurrentDirectory” strings, and the values corresponded with a list of folders needed to build that given string. Thus, with a generated “Image” value, this could be used with the “file_paths” dictionary to rebuild the file string to that given folder. As noted before, this method assumes that no two final file folder locations have the same name, and any instance of such an occurrence would result in always generating data that corresponds with an absolute path to the first occurrence of the final folder as seen through iterations through the observations of the original data due to the way iterative method in which the “file_paths” dictionary was populated.

In the following module, the conversion of encoded “Image” values and encoded “CurrentDirectory” values back to file path strings was performed. Additionally, random variables, which included “ProcessId”, “ParentProcessId” (if applicable), “ProcessGuid”, and “ParentProcessGuid” (if applicable) were randomized within the ranges of data observed within the dataset, as no distinct correlation between process IDs / GUIDs and executables was found during EDA. While all of the GUIDs contained randomized sections, an exact prefix and suffix to these randomized sections was identified and maintained when generating new observations. In addition, the newly generated time data was encoded back into the original “Date and Time” and “UtcTime” formats.

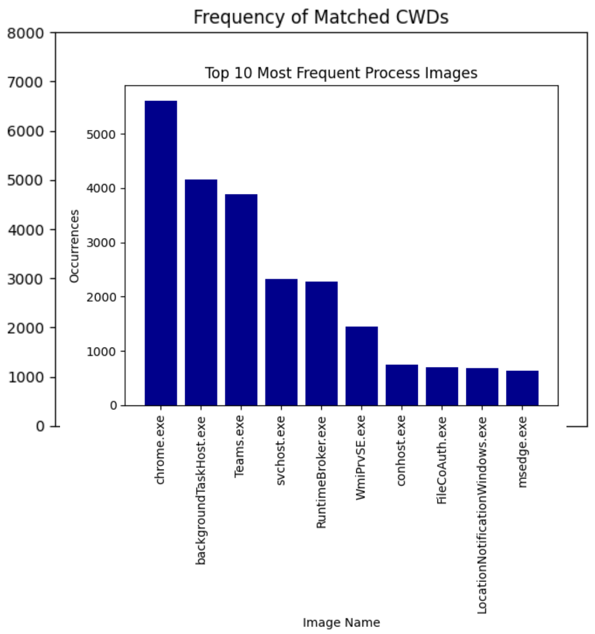
The next step in postprocessing the data involved converting all other one-hot encoded variables back to the correct strings. This was handled through a simple loop that iterates through “encoding_dicts”, a dictionary which holds dictionaries that encoded strings to the given numerical values for each of the one-hot encoded variables.

As was previously mentioned, one of the main ways in which generated data could easily be observed as varying from real data lies in the differences between process creation and process termination logs. As such, a counter of inconsistencies was kept for instances where the generated data did not fit to that format. We identified eleven possible variables where the data could incorrectly have a value for a given observation that does not align with process creation / termination. With one hundred generated observations, this amounted to 1,100 potential chances for variables to not agree with expected norms compared to the original data. We would increment the number of inconsistencies for each one of these instances found as an empirical measure of how well the WGAN captured one of the main patterns in the data. Thus we could also divide (1,100 - inconsistencies) / 1,100 to determine the percentage of appropriately generated fields.

In the final module of our project, we would “untidy” the data to convert it back to its original format. While the first five variables in the original format were straightforward, the sixth variable (which encompassed over 90% of all information) involved using complex string building processes. Thus we created functions that would rebuild either a process termination or process creation string from all of the fields which we had recreated. Our process would iterate over each of the observations in the generated dataset, identify whether it was a process creation or termination, build a string using the appropriate function, and then place that string in the correct observation and variable for the final dataset. The sixth column was without any name in the original dataset, so it was named ‘deleteName’ in the final dataset so as to label that the name should be manually deleted from a text editor if the csv were to be used to represent Sysmon data in its original format.

Results and Discussion

To begin, we will discuss the results found in EDA. Although the generation of Sysmon data revolved around the encoding, training, and decoding of data, further EDA was performed to better understand some of the trends seen in the data. Further data analysis was performed to compare many of the questions asked during the processing steps. First, “How often did the current working directory match the directory in which the process is in?” Next, “What is the ratio of processes created to processes terminated?” Then, “What were the most frequent programs run during the data collection?” Finally, “How often did individual process ID’s get used during the collection of the data?” The findings of these questions are shown in **Figures 3 - 6** below.



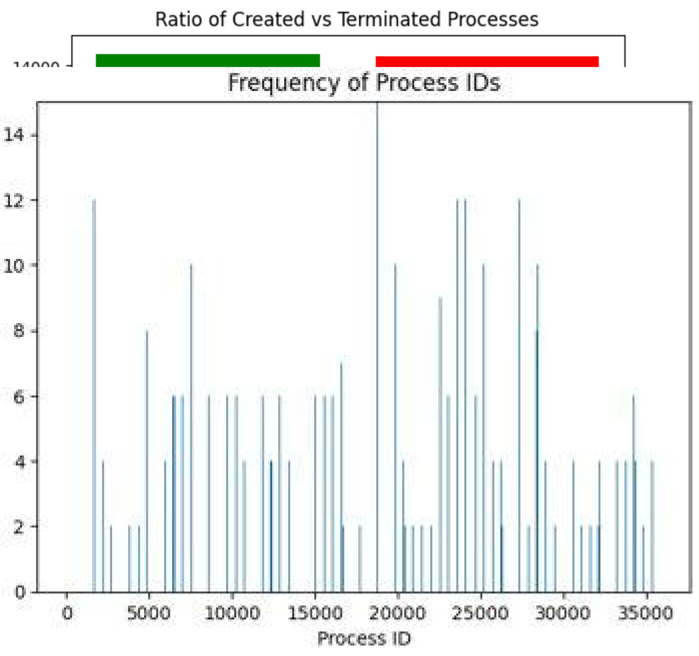


Figure 3 *(top left). Frequency that the current working directory matched the parent directory for a given process creation.*
Figure 4 *(top right). Frequency of process creation logs and process termination logs.*
Figure 5 *(bottom left). Frequency of the top 10 most run images.*
Figure 6 *(bottom right). Frequency of occurrence of individual process Id's.*

Figure 3 shows that the current working directory did not match the parent directory of the image by a factor of about 3:1. This was a surprise as initial glances at the data appeared to show the opposite to be true, but this must have been a localized case of contrary results. **Figure 4** shows that the processes terminated and created were equal. While this was not expected when EDA was run, it is a completely logical finding as every process that is created will be terminated exactly once. **Figure 5** shows that Chrome.exe was the most commonly run image, followed by backgroundTaskHost.exe, and then Teams.exe. Given the nature of usage of the OS from which the data was collected, this is unsurprising as well. **Figure 5** yields perhaps the most surprising results. In **Figure 5**, we see that a large number of process Id's were used repeatedly. Process Id's have the potential of being any integer within the range of the computer (about 1 - 40,000 for this given OS), yet many of the process Id's were reused in multiple instances while the overwhelming majority were not used. This would indicate that the OS tends to favor specific process Id's for reasons unknown.

Finally, another question posed was "Do the Sysmon logs follow any diurnal patterns?" That is, do they follow a specific fluctuation pattern from one day to the next? To answer this question, we conducted further EDA to determine the frequency of all logs over the five day period of data collection. The results are shown on the next page on **Figure 7.**

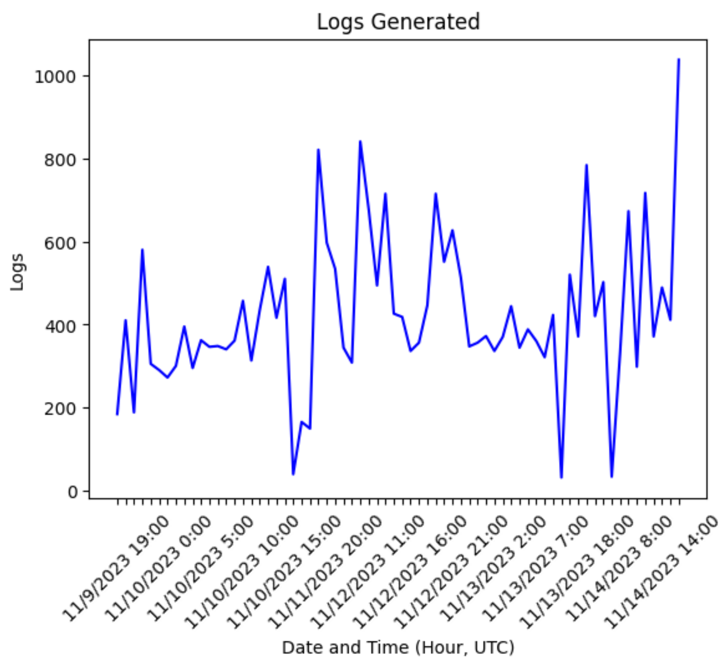


Figure 7. *Sysmon log generation over the course of the five day collection period.*

Figure 7 shows a surprising result. The Sysmon logs do not appear to follow any given daily pattern over the course of the five days. In addition, relatively high numbers of Sysmon logs were generated continuously over the entire collection period. This would indicate that many processes were being created and terminated even when the computer was not actively being used.

The results of comparing various WGAN architectures found that the architecture of a five layer hourglass shaped architecture, with 1,024 node layers one and five, 512 node layer two and four, and a 256 node layer three had the most effective results in generating a wide variety of Sysmon data with low inconsistency values. **Table 1** below shows the findings of various WGAN architectures.

Architecture	Discrepancy Score	Mode Collapse Rating
222	N/A	5
555	88	4
5555	60	3
1111	70	3
15151	243	2
15251	87	2
25152	22	4
555555	212	3
111111	1100	5

Table 1. *Discrepancy scores for various WGAN architectures and ratings of the level of mode collapse visualized. For the architecture, each number corresponds with the size of the neural network at a given layer, where 1 = 1024, 5 = 512, and 2 = 256. For example, 15251 is a five layer neural network where the first layer is 1,024 nodes, the second layer is 512 nodes, the third layer is 256 nodes, the fourth layer is 512 nodes, and the fifth layer is 1,024 nodes.*

From **Table 1**, you will see that the large and small neural network sizes both performed worse and were more likely to mode collapse. Having few nodes in the outer layers of the network also made the model perform worse in terms of mode collapse. The best results were found with the hourglass-shaped five layer neural network. This architecture was found through trial and error, and a search online shows that this is commonly referred to as an autoencoder [3]. By dividing the number of correctly generated data points (as measured by our inconsistency counting) by the total generated data points, we find the percentage of appropriately generated fields to be 100((1,100 - 87) / 1,100) = 92.09%. This shows promising results for those looking to expand upon this project and improve its ability to generate Sysmon data.

An important note is that our research focused primarily upon the processing of data. Undoubtedly, better architectures may be found and the model improved through hyperparameter tuning, but that was not within the scope of this project.

Related Work

We are not the first to analyze Sysmon data. Cybersecurity firm Varonis describes its own take on how Sysmon data can be analyzed to detect cyberattacks, acknowledging the wealth of information provided in contrast to Windows's Event Viewer [4]. Varonis's approach entails converting Sysmon logs into JSON data and using graph analysis. Using that information, Varonis detects threats to a customer's computer. While their end goal is similar to ours, their approach for parsing and analyzing the data differs significantly from ours.

Another example is a research study done by the University of Oslo, which focused on analyzing Sysmon data by culminating all the data into NoSQL database systems and analyzing the processes against known bad actors [5]. This use case is more akin to what an antivirus would do, comparing the hash of an executable to known hashes of malware or other viruses. This is a good way of detecting and neutralizing threats on a target system, but it also differs significantly from our approach.

Conclusion

As previously stated, the best performing WGAN architecture found was an autoencoder architecture. This architecture produced realistic Sysmon logs with no hyperparameter tuning during this project. This project also did not have time to exhaustively try different architectures other than the nine previously stated. Therefore, this project shows there is much room for model improvement

We also drew the following conclusion from the analysis of the generated data: Much of the discrepancies could be avoided through dividing the training of process creation and process termination into two separate tasks to be performed by two separate WGANs. By doing so, it would allow each model to be evaluated more upon factors other than whether it had the inconsistencies we saw in our model. This division would entirely eliminate all of the inconsistencies we observed, and is recommended for training any model to produce Sysmon logs.

Furthermore, if the goal of a future project is to create Sysmon logs that can replicate command line arguments, then a separate model should be trained specifically on the arguments which may appear with a given image, and that model should be used in conjunction with the model for process creation (as “CommandLine” and “ParentCommandLine” are only present in process creation logs). Command line arguments have a complex structure due to the wide variety of arguments that may occur, and as we observed that this model struggled at times with the complexity of our encoded dataset, compounding the dataset complexity would result in poor performance.

All of the code used in this project will be available at the GitHub repository available at this address [6].

Bibliography

[1] “Introduction | Machine Learning | Google For Developers,” Google, https://developers.google.com/machine-learning/gan (accessed Dec. 12, 2023).
[2] M. TOPAL, “WHAT IS MODE COLLAPSE IN GANS?,” Medium, Jul. 23, 2023. https://medium.com/@miraytopal/what-is-mode-collapse-in-gans-d3428a7bd9b8 (accessed Dec. 13, 2023).
[3] A. Dertat, “Applied deep learning - part 3: Autoencoders,” Medium, https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798 (accessed Dec. 12, 2023).
[4] “Sysmon Threat Analysis Guide,” www.varonis.com. https://www.varonis.com/blog/sysmon-threat-detection-guide (accessed Dec. 13, 2023).
[5] V. Mavroeidis and A. Jøsang, “Data-Driven Threat Hunting Using Sysmon,” Proceedings of the 2nd International Conference on Cryptography, Security and Privacy - ICCSP 2018, 2018, doi: https://doi.org/10.1145/3199478.3199490.
[6] [1] R. Dickinson and S. Petushkov, “Randalld3/generating_cyber-data,” GitHub, https://github.com/randalld3/Generating_Cyber-Data.git (accessed Dec. 12, 2023).