Snort Rule Generation for Malware Detection using the GPT2 Transformer

Mr. Ebenezer Nii Afotey Laryea

A thesis submitted to the University of Ottawa
in Partial Fulfillment of the requirements for the
Master of Science in Computer Science degree
in

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Abstract

Natural Language machine learning methods are applied to rules generated to identify malware at the network level. These rules use a computer-based signature specification “language” called Snort. Using Natural Language processing techniques and other machine learning methods, new rules are generated based on a training set of existing Snort rule signatures for a specific type of malware family. The performance is then measured, in terms of the detection of existing types of malware and the number of “false positive” triggering events.

Acknowledgements

Isaiah 40:31! I would like to thank Prof David Knox. Who through his dedication, patience and understanding made this work possible. I would also like to thank the Zane family whose financial support got me through this program. I wish Dr. Ronald Zane was alive to witness this glorious day! Thanks to my wife Judith who has been the wind under my sail and Ronald and Renora whose continuous running around and noise making kept me going. To my mom, dad, sister and my in-laws who took care of my family while I was in school, I pray God replenishes all that you have lost. And finally, to all my friends in Canada who kept me going through it all and gave me advice, I really appreciate you!

Table of Contents

List of Tables

List of Figures

List of Abbreviations

LSTM: Long Short Term Memory	44, 47
M	
MLP: Multi-layer Perceptron	15
N	
NLG: Natural Language Generation	45
NLP: Natural Language Processing	2
NN: Neural Network	4
P	
pcap: Packet Capture	4
PE: Portable Executable	17
R	
RNN: Recurrent Neural Networks	46
RNNs: Recurrent Neural Networks	44
S	
SAE: Stack Auto Encoder	17
SVM: Support Vector Machines	14
T	
TPU: Tensor Processing Units	58

1 Introduction

This work discusses the challenges associated with identifying malware using characteristic behaviour at the network level (i.e., signature-based detection) and the need for improved, automated ways of generating these signatures. As an alternative to the current manual methods, different ways for automating the task of generating Snort signatures, are analyzed. In this chapter, the motivation and research questions are presented, and the main contributions outlined. Lastly, the document structure is also described.

1.1 Motivation

Malware detection strategies are part of a "cat-and-mouse" game between the malware creators and the companies and researchers that are trying to keep the end-user safe [1]. Sadly, it appears that the malware writers are currently in the lead, and this will probably continue to be the case [2]. This can be attributed to the relatively rapid speed at which single errors or oversights can be found and exploited in deployed software and the time required for security professionals to reverse-engineer the new malware before writing antivirus signatures firewall rules to detect and block all new malware variants. Equally, finding one error is typically going to be an easier task than fixing all of them. These challenges beg the question of how malware signature generation can be automated to bring security professionals closer to the pace of malware creation. This work looks at the task of automating the generation of such signatures for Snort, a “language” often used to create signature-based rules for Intrusion Prevention Systems.

1.2 Objectives

Given the mundane and time-consuming task of writing Snort rules, this work aims to investigate how to apply Natural Language Processing (NLP) techniques to assist in writing Snort rules. If complete automation (i.e., removal of all human rule generation) is not possible, then our goal is to provide a tool that serves as an assistant to the expert in generating rules for specific malware. The generated rules can then be refined further to fulfil specific tasks. We propose an NLP technique to simplify the rule-writing process and so improve the speed and accuracy of malware detection.

Contributions

- Demonstrated a method which uses a selection of elements in an IP header to give context to classify if network traffic is malware or not (“*Malware Classification Using NLP on Packet Header Information*”).
- Demonstrated a novel method to train a neural network to generate partially effective malware rules for the Snort IPS (“*IPS Snort Rule Generation for Malware Detection using NLP*”).

- Showed that machine-generated Snort rules could be used with the Snort engine to detect malware in a network, without triggering any false positives when testing with malware-free traffic.
- Described various methods by which machine-generated rules can be sanitized to ensure they meet the strict requirements of the Snort language.
- Showed that a neural network can be used to train on old Snort rules and generate new rules to detect a limited set of malware.

1.3 Thesis Organisation

The remainder of this thesis is organised as follows:

Chapter 2 introduces the reader to the history of malware, how it has evolved over the years, and how the detection of malware has also evolved. Techniques used by various Intrusion detection/prevention systems for malware detection are described and analyzed. The Current literature about how AI is being used in the field of malware detection is reviewed and the approach used in this work for using AI NLP techniques for malware detection is justified.

Chapter 3 discusses the concept of using context information extracted from an IP Header to classify whether traffic is malware or not. The objective was to reduce the effort associated with deep packet inspection and sets up the later work that uses NLP techniques.

NLP-based classification results are then discussed. Based on these results, our approach to generating malware rules (described in more detail in (in Chapter 5) is then justified.

Chapter 4 discusses the Structure of Snort, which is the specification language used for generating malware rules. The analysis of how the Snort engine works and how rules are written for Snort is discussed. A discussion and comparison on Artificial Language Processing (ALP) and NLP and a literature review of recent work related to text generation in ALP and NLP is given. NLP feasibility for achieving the intended task is demonstrated along with various NLP techniques that we believe are also relevant and useful.

Chapter 5 provides an explanation of the methods used for data collection and preparation of the dataset used in this work. The preparation refers to the cleaning of data and formatting to make the training of the Neural Network (NN) more effective.

Chapter 6 reviews the experiments that were performed and the results that were achieved. The training of the neural network and the generation of the required rules for SNORT signatures is discussed next, along with the cleaning and testing of these rules for the relevant malware packet capture (pcap) files. Finally, the quality of the machine-generated rules is analyzed and the impact of automated rule generation discussed.

Chapter 7 summarizes the work, highlights the main contributions and discusses future improvements.

2 Background

This chapter surveys the recent literature on malware and malware detection, using an intrusion detection system (IDS) or intrusion prevention system (IPS), highlighting the types of malware that are visible at the level of a computer network (e.g. with tools capable of "sniffing" network traffic). The key differences between these two types of security system, that work at a network level, are highlighted and our focus on signature-based IPS is justified. The application of AI to IPS is then motivated and the available, current methods are reviewed that use AI to generate the rules that are required in signature-based detection of different types of malware.

2.1 History of Malware

Malware, has existed for over three decades [3]. Malware is used to disrupt the regular operations of a computer and includes: viruses, trojans, ransomware, worms, spyware, adware and these are defined and discussed in the next section. The first recorded incident of malware was one written by two brothers in Pakistan called Brain. A. [3]. The brothers created software that replicated itself using floppy disks. The virus did not cause any harm apart from it continually replicating itself on any floppy disk inserted into the computer.

The Omega virus [3] would infect the boot sector of the computer, and on Friday the13th, it prevented all infected computers from booting. Code red [3] was one of the first internet worms (any self-replicating type of malware that can execute as a stand-alone entity). It spread via a vulnerability in the Microsoft Internet Information Server IIS, and its main malicious action was to launch Distributed Denial of Service DDOS attacks on various websites like whitehouse.gov etc.

In the past few years, malware has become more effective and more lucrative. Ransomware, where legitimate user files are encrypted, and victims must pay a “ransom” to get a promised (but not always delivered) decryption key. Similarly, malware that launches DDOS attacks has been used to bring down large networks. As an example of successful ransomware, the WannaCry ransomware infected machines in May 2017 [4], and demanded that victims pay between $300-400 US Dollars for the decryption key. It cost the UK National Health System NHS over £29 million pounds. Globally, it is estimated to have cost $ 29 billion US Dollars [5].

Several decades later, the available malware is sophisticated and more malicious in intent, when compared with Brain. A. Ransomware of a single type in 2016 was estimated to cost 1billion US dollars! [6]. The trend is for continuous evolution of different kinds of malware, with increasingly sophisticated attacks and behaviour patterns [7].

2.2 Kinds of Malware

Worms: Worms are self-replicating malicious software that infect computer systems. The main distinction between a worm and a virus is that a worm can self propagate without a host application or file. Because of this, worms usually spread via networks, whiles viruses tend to spread via applications that have either been downloaded from the internet or shared via USB storage media. As an example, the Morris worm [8], also called the “Internet

Worm", used a vulnerability in the Unix *sendmail*, *finger* and *rsh* utilities to spread. The payload behaved like a Denial of Service attack DOS. There were three key novelties in the Morris Worm: it performed password guessing attacks on infected nodes, it evaded notice by obscuring its process parameters, cleaning up any files that it created and, finally, it attacked one specific operating system (Unix) but two different computer architectures. [9]

Trojans: A trojan is a type of malware that disguises itself as a legitimate program but is maliciously performing illegitimate actions in the background. Trojans can be used for spying, collecting data about the user and in some cases, even espionage. Trojans come in various forms like games, fake antivirus software, remote access trojans etc. An infamous Trojan was the Zeus trojan. It infected windows machines and stole information like banking information, and was also logging keystrokes [10].

2.3 Evolution of Malware Detection

Due to the continuous evolution of malware, the various techniques used for detection have also evolved. Some of these methods include early-detection based techniques like signature-based detection and these have gradually evolved to using machine or deep learning techniques. As an example, the work in [11] analyzes and correlates features at four different levels; the kernel application, user and packages and uses these features to create acceptable and unacceptable behaviour patterns.

Signature-based detection is effective and simple to implement and use and has been in continuous use for a long time [12]. This technique employs the use of a unique sequence or pattern observed in the malware that differentiates it from other malware. Usually, these patterns are unique enough to identify a specific malware type or a particular family of malware. The challenge with this technique is that a slight deviation in the pattern will allow the malware to evade detection. Signature-based techniques can be considered to be reactive techniques [13] since they are not able to defend against new malware until new signatures are written for the as-yet-unseen malware types.

Heuristic-based detection has also been used for malware detection, historically. In heuristic-based detection, suspicious programs are run to obtain a defined pattern of behaviour, and a threshold is set for that behaviour. Whenever any application exhibits that behaviour pattern, it is classified as that specific malware. The main advantage of the heuristic technique is that it can detect new types of malware, based on similar behaviour patterns. This technique can lead to large numbers of false positives; hence a combination of heuristic-based and signature-based techniques are often used.

Increasingly, machine learning techniques such as [14], [15], [16], [17], [18], are being used for malware detection. The benefits of these techniques include: reducing the rate of false positives (i.e. something improperly identified as malware) greatly, being able to detect new and unknown malware variants, and providing faster overall detection rates.

2.4 Intrusion Detection and Prevention systems

Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS) are the two basic types of mechanisms used to detect malware at a network level. An IDS is usually an "offline" software system that automates the process of the detection of an intrusion [19]. Normally, the outputs of an IDS are alerts, and no restoration actions are taken.

The intrusions can arise with malware packets flowing into the network from the internet or could from software that is being executed inside the network, somehow, by an intruder. An IDS generates alerts whenever an attack is detected and requires the administrator to perform necessary actions to prevent further intrusion. While an IPS detects intrusions like an IDS, an IPS also prevents intrusions (i.e. takes a corrective action in real time). Because of their ability to assist in the prevention of intrusions, the detection techniques used by these systems can be classified into three major areas: Signature-based detection, Anomaly-based detection and Stateful protocol analysis [20].

Signature-based detection techniques deal with users writing rules that contain patterns or strings that correspond with a known attack or threat. Anomaly-based detection techniques are designed to characterize normal behaviours and then detect deviations from that characterized behaviour, using heuristics. The key advantage of such methods is being able to detect behaviour that is not yet known as malicious. The main limitation with them is the incidence of high false positives that are associated with “new” types of non-malicious behaviour that has not been adequately characterized.

2.5 AI-Based IPS

In an overview of the relevant literature that uses AI in IPS systems, most of the work to improve IDS system tends to focus on malware samples in their executable forms or in the form of IPS requests. Generally, there is agreement that AI technologies improve the rate of malware detection. Various malware sample features have been used to create accurate classification models. Finally, AI algorithms can be used for malware signature generation itself and this is the focus of the work described here.

2.5.1 Improving malware detection using AI techniques

Herani et al. [21] surveyed and analyzed different approaches for improving cybersecurity using AI. These methods included: expert systems, neural networks and intelligent agents. Their survey highlighted the increased accuracy and efficiency of detection when AI is used in intrusion detection and prevention systems. Swapnil et al. [22] did similar work but reviewed techniques like data mining, fuzzy systems, and pattern recognition systems. They concluded that using AI in such systems can help to adapt them to different cybersecurity situations, thereby increasing the accuracy and speed of decisions. Dalal et al. [23] discussed a novel architecture model based on machine learning for the prediction of a cyber security malware that is executed in a sandbox.

2.5.2 AI systems for generating malware detection signatures

Currently, signature rules for Intrusion Prevention Systems are written manually. Malware behaviour is studied at a network level, and corresponding rules are written to identify and flag them. This might be based on the detection of specific strings in the malware payload and varies with the type of malware. This task can be arduous and prone to error. Such errors might be non-detection of malware ("false negatives") or excessive numbers of "false positives" (i.e. malware that is flagged, when none is actually present).

Several researchers have attempted to automate signature-based, network-level malware detection methods. Different methods have been used to standardize the specification of signatures and to allow the manual cost of generating them to be shared by a larger community. Specification-based detection uses signatures written for the specific malware [24]. The semantics of the malware and its API calls can be combined to create a unique signature that can even be used for detecting new variants of that malware [25].
A specification language called Snort [26] has become the de-facto standard. This is explained briefly and justified as a representative signature-based 'language'. Some notable examples of similar work include the work by [27], who used the logs from a honeypot, specifically *honeyd*, to generate Snort rules. In their work, they wrote a script that would parse the log files from *honeyd* and generate the corresponding Snort rules. The rules that they generated were mainly from network scans and other probing scans. Their work differs from ours in that we are working on generating rules for malware which, per their behaviour and structure, are more diverse and specifically not IP-based. Thus, our rules do not require IP addresses to be part of the rules that are being generated since IP addresses for malwares can be changed easily.

The work in [28], uses two data mining algorithms to generate rules for five types of attacks, which include Brute Force and DOS attacks. Again, it can be observed that these types of attacks are not sophisticated and are easily mitigated by blocking the offending IP address that is launching the Brute-Force or Denial of Service attacks. This approach is less effective for the DDOS types of attacks discussed already, which are launched from multiple IP source addresses. The work of [29] used a genetic algorithm solution for rule Snort generation, using anomalous ICMP packets as inputs. The authors then designed an algorithm to generate Snort rules. The malicious traffic being detected was ICMP packets, which are simple types of attacks that can usually bethwarted by using IP-based rules.

Jeyasingam et al. [30] looked at the dynamic generation of Snort rules for SCADA Systems. The authors used descriptions provided by system administrators who used simplified natural language to capture relevant features and other constraints like IP addresses, etc. This description was passed to a parser to interpret the language definitions, which then generated the required rule. This technique did not use Machine Learning methods, per se, but is the closest related work where an IDS rule generation has been automated. Their work is different from ours in that they were still requiring a human to describe the characteristics of the system, such as: network topology, protocol, etc. before it could be used. Another limitation of their work is they were specifically focused on network-based attacks like DDOS and not on malware. Our goal is to generate signatures for both malware and network-based attacks.

We believe our work to be among the first to use machine learning techniques in combination with Natural Language Processing (NLP) methods for Snort rule generation. We do this for various families of malware, starting with existing Snort rules. Our objective is to do this without requiring any additional human processing steps, specifically related to re-writing the rules or re-interpreting them.

2.6 Signature-Based Detection

Signature-based detection is currently used in both the leading IDS and IPS systems [31]. Some of the benefits of signature-based systems include having high processing speed for attacks that have been identified and also having a very low false-positive detection rate (i.e. benign traffic classified as being malware). The goal of all IDS and IPS systems is to reduce the number of false positives and the number of false negatives (i.e. malware traffic not classified as such) too.

3 Related Work

3.1 Malware Detection using AI

Using machine learning techniques to detect malware is not new. In [18], the authors collected about 250 malware and benign samples, which they submitted to an online automatic dynamic analysis service called Anubis. This generated a report in XML. Using XML, they created sparse Vector models for each report, and Attribute Relation File Format files (ARFF). ARFF files are ASCII text files that describe a list of instances and the attributes they share. This file format was specifically designed for a Machine Learning tool called WEKA which the researchers used. The learning and classification were based on the ARFF files. Various Machine Learning techniques were applied to these files. Their key findings showed that the best performance was achieved on the J48 algorithm on both binary weight and term or occurrence frequency-weight datasets. The J48 Algorithm is an algorithm used to generate decision trees. Decision Trees are graphical representations of possible solutions based on specified conditions. There were slightly different performances between k-Nearest Neighbour k-NN, Support Vector Machines SVM and J48 algorithms. The main limitation with their technique is that malware detection is not performed in real-time, because samples need to be of the time required to reformat the samples used for training and then testing.

A framework for detecting malware on Android devices was proposed in [32]. This is a "lightweight" malware detection system because it uses very minimal CPU, memory, and battery power. Different metrics are monitored, like CPU consumption, packets sent via Wi-Fi, the number of processes running, the battery level and other system usage parameters. The data collected is processed in stages, a threat assessment generated, and then appropriate notifications are sent to the user about specific applications or processes. The goal of the report is to lead the user to either uninstall the application, kill a specific process or encrypt their data [32]. The authors found that Naïve Bayes and logistic regression algorithms were superior to other classifiers in most of the configurations.

The work in [33] is an evaluation of different classifiers for detecting mobile malware [33]. Two experiments are conducted. The first experiment captures network packets from the Google Play Store "Top 20 clean applications" list. Next, data from the *MalGenome* dataset (which was the largest malware dataset at the time of the publication) is gathered. The authors use the common behaviour of malware, as defined at the time of publication, where 97% of the malware studied, established network connections to their developers (phoning home) as the basis for their study. Features are extracted from the samples, including frame length, frame number source port, destination port etc. Their results showed that Bayesian Network and Multi-layer Perceptron MLP classifiers provided an almost perfect Area Under Curve AUC value of 0.995 (a basic measure of classification accuracy indicator, with the highest value being equal to 1) with Random Forest having an AUC value of 0.991followed by KNN and J48.

The researchers in [34] took raw pcap files as their input data and then used CNN to propose a new traffic classification model. They converted the collected pcap files into

CNN input data, which is in Incremental Design Exchange IDX format, which is an XML-based messaging format. They then generated a 28*23-byte image for each type of traffic, using CNN image detection to detect the various types of traffic. In their work, they were able to achieve an impressive accuracy of 99.41%. However, the process of converting pcap files into image formats is quite slow, preventing detection from being done in real-time.

Deep Forward Neural Networks are machine learning structures that are designed such that the connections between the various nodes do not form a loop. In [35], the author trained DFNNs on 4.5 Million pcap files and tested them on a distinct set of 2.5 Million malware and benign sample. This study is one of the largest studies that has been performed to date. Features were extracted by malware experts, with domain knowledge in the field and the number of these features gradually reduced from 50,000 and consolidated down to 500 and then trained on the larger subset of samples. A binary classification error rate of 0.357% was achieved on the smaller independent subset of tested samples. However, a very large number of malware features needed to be extracted by domain experts and the authors, Shabtai et al [35] acknowledge that their resulting classification error rate is also too high to be put to practical use.

Hardy et al. [36] monitored Windows Application Programming Interface (API) calls to detect malware. These calls were extracted from Portable Executables PE Files. They studied how a deep learning architecture using the Stack Auto Encoder SAE models could be designed for intelligent malware detection. The SAE models were used as greedy layer-wise training operations for unsupervised learning, which was then followed by supervised parameter fine-tuning. This resulted in 0.986 accuracy. The main drawback with their work is that malware not using a Windows API cannot be detected.

From a survey of the literature, it seems that little or no research has been done to generate signatures for malware detection systems, using already written signatures as the dataset. An important goal for our research is to investigate the area of Generating new Snort Signatures using old Snort rules for specific malware families. With the objective of having the new rules detect more malware than the existing one. We set out to combine NLP techniques with machine learning algorithms to create signatures for use in IDS and IPS systems.

3.2 Motivation for the application of AI and NLP to signature-based detection

Snort resembles a 'language', albeit a very structured and rigid one. Based on this starting assumption, even though the language is 'artificial' to a large degree, we investigated the use of Natural Language processing techniques to generate malware-detecting rules. Similar approaches have been used with other forms of “natural language” with more structure. An example of such work is by Lee et al. [89]. In their work they used the GPT-2 transformer to fine-tune patent claims. Patent language is very limited, specific and restrictive in structure. As with spoken languages, however, there is still more than one way to write a Snort rule. A Snort rule is acceptable for real-time malware detection purposes, if it follows the required syntax and correctly raises an alert, when tested with the requisite malware (i.e., flags 'true positives') and does not trigger in the presence of normal non-malware traffic (i.e., no ‘false positive’ triggering). ‘True positives’ are situations when the model correctly predicts the presence of malicious samples. For 'false Positives’, the model incorrectly predicts a benign sample as being a malicious one. A ‘false negative’ situation arises when the model predicts a malicious sample as being benign. A True negative case is when the model correctly predicts a sample as being benign. We want true positives and true negatives, but not false positives and false negatives.

In practical systems, Snort rules that trigger on non-malware traffic are re-written to be more specific to the malware that they have been designed to detect. This reduces the number of false positives. The total number of rules must also be minimized, in such systems, to allow them to be used for real-time detection on hardware routers and monitoring systems with limited memory and processing time. This improves the speed and performance of the system, when in operation. Rules that no longer trigger must be culled and replaced with ones that provide more malware-detection capability. This must be done on an ongoing and continuous basis, as malware signatures evolve.

Applying NLP to IPS Signature generation assumes that current IPS signatures can be viewed as rules written in a language by humans, even though this language is in fact more artificial than true “natural” languages. However, jut like other natural or artificial languages, IPS signatures must follow conventions for grammar and structure and have “parts of speech”, etc.

In truth, IPS signatures are a form of Artificial Language. They are quite rigid in their use and need to be highly specific and they have limited words that can be used. Nonetheless, we attempt to apply NLP to provide good algorithms for text generation. This generated text will obey the rules of this specific language and hopefully still be useful for intrusion prevention systems.

4 Malware Classification Using NLP on Packet Header Information

In this section, we discuss the task of classifying malware using NLP techniques, operating on data samples from network traffic. We classify malware using NLP and using just the TCP header information available on a packet. Since deep packet inspection is very costly in terms of time and processing power to any IDS or IPS, we hoped that using the header information alone would be feasible. Another benefit of only using header information is that the header is unencrypted even if the packet payload is encrypted.

Network-based malware analysis utilizes the network traffic generated by a given executing malware sample to characterize its behaviour. Network traffic can be captured by executing the malware in a sandboxed environment and by using a packet capture tool (e.g., like Wireshark) to analyze outgoing and incoming network traffic, which is analyzed after capture. Offline processing is usually preferred since it does not require as much power as real-time processing. Such an offline method is widely used in industrial settings for malware analysis for these reasons. Sophisticated network-based malware analysis intrusion detection software exists to capture network packet data and then analyze it to make offline intrusion detection decisions (e.g., SNORT [26]).

Packet inspection tools typically use deep packet inspection to observe patterns in the packet payload. However, only looking at network flow-level information in packet headers might yield acceptable performance for faster processing [38]. A flow is a stream of continuous packets being transmitted between a source and destination, which is our focus here. Flow-level header information could include source address, source port, destination address, destination port and transport layer protocol. Using network flow information like this can reduce the storage and processing costs when analyzing network traffic. Flow-level network packet analysis has become more relevant as the network bandwidth has gone up and the processing costs of deep packet inspection have increased. [39]

4.1 NLP Terminologies

Def^{n} 1: Word

A word can be defined as an encoded version of a packet utilizing various packet header attributes.

Def^{n} 2: Sentence

A sentence can be defined as an encoded version of a stream of packets or, more simply, as a sequence of words.

Def^{n} 3: Context

Context can be defined as the correlation between a group of consecutive words appearing in a sentence.

N-Grams: A sequence of n words is known as an n-gram. These are used in natural language processing and emphasize both the different meanings of words, but also their context, which can affect the meaning too.

Vectorization: Vectorization of obtained text (or of an obtained corpus, or body of text) is an important step in recognizing patterns in natural language. Vectorization is used to compress the size of the corpus being analyzed, for both training and classification purposes. Different techniques used to vectorize words are listed below:

- **Term frequency:** Inverse Document Frequency - a measure of how important a word is in the document. It considers whether a term is common or rare in a document.
- **Bag of Words:** in this model, a sentence is represented as an unordered set of words. The grammar and context are not considered, but the word frequency is noted.
- **Neural Word Embeddings:** is a technique involving the use of a neural network to map each word into a multi-dimensional vector space.

4.2 Related work

Natural language processing techniques have been utilized in the malware classification task in the past. Wang et al. [40] performed a word analysis of HTTP traffic and found that malicious and benign traffic differs in their term frequency, allowing detection of malicious

HTTP traffic. Nagano et al. [41] and Tran et al. [42] utilized the API calls made by an executable to form a characteristic paragraph vector which they use to characterize samples as malicious. The limitations of their approach are that API calls are now normally encrypted, rendering their technique useless. Their work relates to our work in that they are also trying to use metadata about the malware samples to detect whether samples are benign or malicious.

NLP software techniques can be utilized in other parallel fields. An excellent example of this would be the extension of Word2Vec, well-known NLP software, into Gene2Vec and Like2Vec. A significant amount of work has been done in the field of natural language processing for malware analysis [43] [44] [40] [4] [42] [41]. A more thorough discussion of these work will follow next.

Chatter [43] proposes the use of system event ordering to classify malware families. The authors utilize network events and encode them into documents which are used as characteristic classification templates for a malware family. They report over 75% accuracy and precision using three different classification methods: SVMs, k-NN and decision trees. This work relates to our work in that they also use metadata, which is high-level system events and the order in which they occur to detect malware samples.

MalClassifier [44] proposes the use of network packet traffic to classify malware families. The authors claim that this can be achieved at wire-speed rates (i.e., fast enough to work in real-time). Their work builds on previous work by Metty et al. [45] that separates traffic into distinct network traffic "flows" using independent component analysis. Their work also relates to our work with their focus on using metadata, which in their case is network flow sequence behaviour.

4.3 Methodology

Like other network-based malware analysis methods, we analyze sequences of network packets from pcap files. This does not preclude wire-speed inline processing as follow-on work. As already stated, because we are only using header data, the amount of data that needs to be stored and processed is reduced by a factor of 14 times down to a total requirement to process only 96 bits which is the TCP header size.

Since our technique utilizes 'text mining' techniques (which is the process of deriving information from a collection of text), the first step in analyzing a stream of packets is to create an encoding to convert raw packet data into *words* and s*entences*, as already defined. A stream of n packets is encoded into a *sentence*, giving a ‘context’ to a collection of packets. For malware classification of network traffic, a *sentence* can be defined in different ways. These *sentences* are then analyzed using text mining approaches defined by other researchers.

We varied the definition and scope of the context used. We then analyzed which methods classify better and the reason for any relative performance advantages. We also examined whether the identification of flows, which are unique source/destination address and port and protocol instances, results in performance improvements or not.

4.4 Dataset

We utilize a subset of the Deep Traffic dataset [46] for our application. This is a public data set, containing network traffic from malware and non-malware sources. The dataset contains four different traffic sources, of which three sources were benign (i.e. BitTorrent, MySQL and Gmail), and another source was malicious (i.e., Zeus malware). Our selection for these types of packet capture traffic was based on the ubiquity of both types of traffic. As an example, BitTorrent traffic is known to constitute about 40-70% of the current internet traffic [47].

4.5 Design Prototype

The available header data is analyzed and pre-processed using Wireshark to create pcap files. A custom script was then used to convert each of these pcap files into encoded-words (or classification vectors) and subsequently into n-grams. Different sizes of n-gram (i.e. n=1 through n=9) were analyzed to determine which would give better accuracy. An example encoding for n=3 can be seen in the figure below. This encoding allows the text mining software to be utilized, but care must be taken in interpretation, given our definition of a 'sentence' for malware classification purposes in section 3.1. In our design we consider the value of n to be a concatenation of 4 header fields, Port, Protocol, Packet Size and Time.

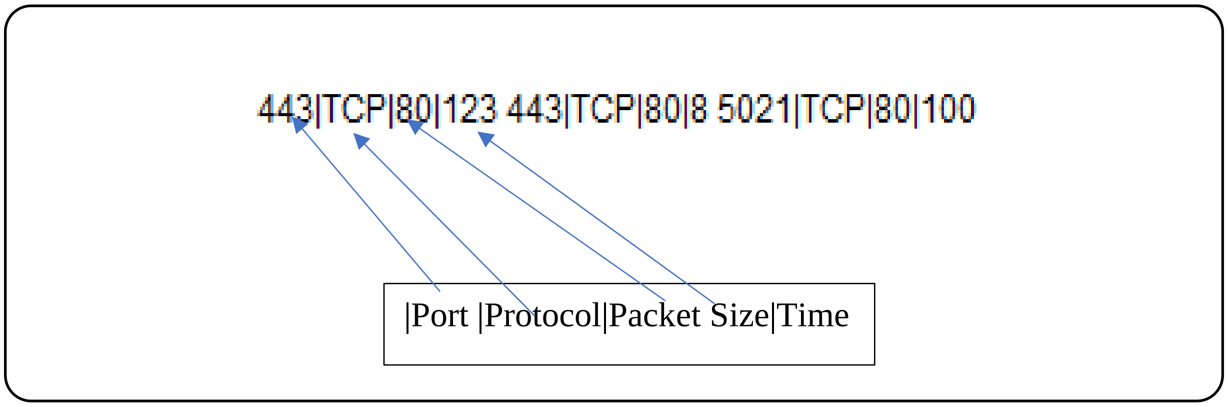

443|TCP|80|123 443|TCP|80|8 5021|TCP|80|100
|Port |Protocol|Packet Size|Time

Figure 1: Example encoding for n=3

The n-grams were fed into a Doc2Vec neural network for vectorization [48]. The Doc2Vec network converts each n-gram into a vector of floating-point values between 0 and 1. These vectors allow packet data size to be reduced, for training and testing/classification purposes. Reduction of the packet data size improves the training speed and of the machine learning models.

The derived vectors were then randomly split into a training set and a test/classification set and labelled as being either malicious or benign, based on the specific file source. The training set was used to train a Support Vector Machine (SVM) classifier. Once established, this SVM classifier is used to make predictions and evaluate the data in the test set. This whole process was then repeated, with different random sampling to produce the classification datasets multiple times, and the results from each such trial averaged to derive a final performance figure for classification accuracy and precision.

4.6 Design Evaluation

The prototype design was evaluated using standard machine learning statistical parameters. Specifically, the following metrics were used: Precision, Recall, and F1 Score. Figure 2 shows how each of these metrics is calculated. These metrics are all defined in terms of: True Positive (TP) and True Negative (TN) as well as False Positive (FP) and False Negative (FN).

A True Positive classification is when malware packets are correctly classified as malware. Conversely, a True Negative classification arises when benign packets are classified as such.

$$Recall=\frac{TP}{TP+FN}$$

$$Precision=\frac{TP}{TP+FP}$$

$$F1\ Score=\frac{TP}{TP+\frac{1}{2}(FP+FN)}$$

LEGEND

Recall: ratio of true predictions to the total true predictions

Precision: ratio of correct true predications to the total true predictions

F1 Score: weighted average of the precision and the recall

Figure 2 Formulas to calculate the evaluation metrics

4.7 Results and Analysis

The results for the different metrics are summarized in Table 1 This data has been plotted graphically, for visualization purposes. Accuracy dips momentarily at n=3 and then continues to increase up to n=7.

Ngrams	Recall	Precision	F1 Score
2	0.8827	0.8519	0.8398
3	0.8929	0.8762	0.6937
4	0.7638	0.9259	0.7865

5	0.7635	0.9409	0.8555
6	0.7638	0.9395	0.876
7	0.7683	0.9464	0.8964
Baseline1			
Baseline1-2	0.98667	0.99418	0.85538
Baseline1-3	0.87914	0.97148	0.91627
Baseline1-4	0.71811	0.99724	0.98728
Baseline1-5	0.87429	0.97672	0.92406
Baseline1-6	0.73832	0.91304	0.95325
Baseline1-7	0.85993	0.89966	0.94093
Baseline2			
Baseline2-2	0.96734	0.80977	0.87593
Baseline2-3	0.85314	0.92763	0.98292
Baseline2-4	0.9837	0.80175	0.94396
Baseline2-5	0.99201	0.89264	0.81642
Baseline2-6	0.85742	0.81278	0.81266
Baseline2-7	0.81266	0.80947	0.84751
Baseline3			
Baseline3-2	0.9956	0.81112	0.892

Baseline3-3	0.97947	0.81571	0.81733
Baseline3-4	0.94573	0.87846	0.87449
Baseline3-5	0.94436	0.98377	0.99721
Baseline3-6	0.83858	0.85028	0.994
Baseline3-7	0.88694	0.90093	0.92537
Baseline4-			
Baseline4-2	0.98308	0.90922	0.97676
Baseline4-3	0.82462	0.83459	0.9333
Baseline4-4	0.96838	0.95173	0.91388
Baseline4-5	0.9529	0.94319	0.95387
Baseline4-6	0.85831	0.98944	0.99282
Baseline4-7	0.87353	0.87303	0.99231

Table 1: Recall, Precision and F1 Score (n=2 through7)

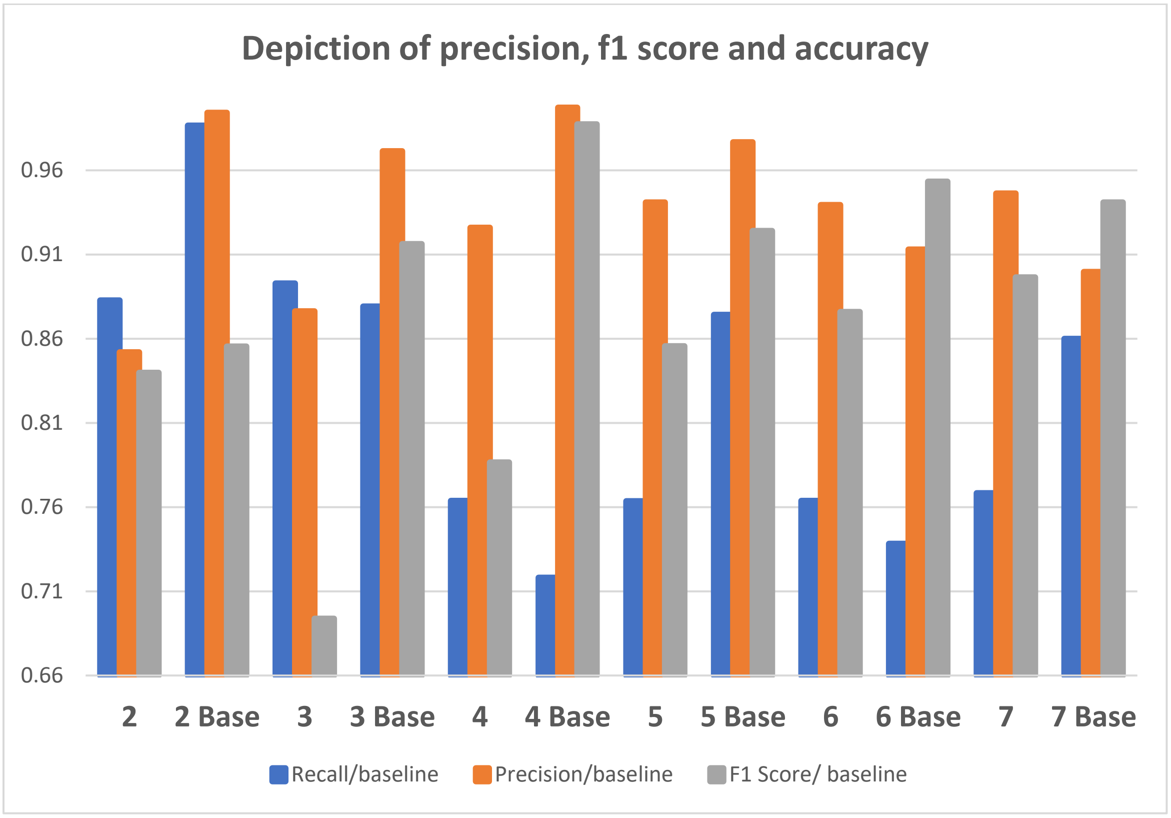


Figure 3: Depiction of precision, f1 score and recall

Both **recall** and **f1 score** follow a similar pattern, peaking initially at n=2, then dipping at n=3 and rising for the subsequent values. We might expect both recall and f1 score to rise for values greater than 3, because greater n values will carry more 'context'. However, it is less clear why the values dip at *n*=3.

One plausible explanation might be that *n*=2 corresponds to cases of simple send/response messages between the sender and receiver (i.e., maximum possible and full context) while *n*=3 packets contain a mix of both of this type of short transactions but also partial samples from longer conversations. This results in 'extra' packets from different flows occurring together, resulting in distorted results.

recall also never exceeds 89% for all tests and the reason for this is also unclear. 11% of malicious n-grams could be exceptionally similar to benign n-grams hence resulting in their misclassification. Equally, malicious traffic may behave identically with non-malicious activity at certain parts of a conversation flow.

When our results are compared to baseline scores it can be observed that baseline scores are better than our results, in general. This may be because “distinctive” malware communication might occur in packets that are spread out over non-consecutive time intervals. Therefore, randomizing the packets allows related communication packets from a “flow” to be closer to each other in the pcap (packet capture) file. This could be why the randomization of the packets leads to better scores across board. The ordering of the packet header information for our experiment does not have to be unique. We plotted our results with only one set of baseline results since other baseline results follow the same pattern.

We considered base rate fallacy in our work. In base rate fallacy, people tend to ignore or overlook the base rate of incidences of an event because they tend to be very low [49]. It is known that just a very small amount of network traffic is actually malicious research shows that the percentage could be as low as 7% [50]. Despite being that low, there is a tendency to focus on false positives instead of true positives Since true positives are the packets that cause harm to the network.

4.8 Conclusion

We have demonstrated a lightweight method to detect malware using network traffic, although our method is improved by packet randomization. This seems to indicate that temporal "locality" in a pcap file is interfering with detection somehow. This would seem to be dependent on network traffic statistics.

A detection accuracy of 89% is not enough for commercial environments, but our work is comparable with other malware detection systems [51]. As already mentioned, the advantage of anomaly-based mechanisms is that new malware types with different signatures from previous malware types might also be detected, although the new type of malware must differ in some way from "normal" traffic types.

In Chapter 6, rather than investigating and analyzing large sets of pcap files from different networks, we explain a completely different approach to malware detection, which should be less dependent on the network characteristics that affect pcap temporal locality of network "flows" in such files. We study the automation of the process for writing Snort rules to detect malware, since these rules can be used directly in many/most IDS and IPS systems. We assume that the writers of "successful" instances of such rules have been forced to optimize them to minimize false positives and false negatives, while maximizing true positives for a wider variety of networks and network conditions, as captured in network packet capture files. In this way, we avoid the need to evaluate large numbers and varieties of pcap files, ourself.

5 IPS Snort Rule Generation for Malware Detection using NLP

This chapter discusses how Natural Language Processing techniques can be applied to the generation of rules that can trigger an IPS. These rules are written in a rules definition language called Snort [52]. The architecture of the Snort engine is described, along with the anatomy of Snort rules An explanation is given as to how such rules are written. The second section in this chapter compares Natural and Artificial Languages. The chapter concludes with a discussion of text-generation techniques, applicable for artificial languages like SNORT.

5.1 Snort Intrusion Protection System

Snort is a leading industry tool that is used in intrusion detection and prevention systems for detecting specific patterns of network traffic based on a rule. It is open-source and was created by Martin Roesch [26]. In most industries, Snort is considered to be the de-facto standard for capturing the rules that are used in both IDS and IPS. The language is highly customizable and operates very efficiently (i.e., allowing real time detection of specific network traffic patterns). Because Snort is an open-source tool and widely available, a large number of rules for detecting different types of malicious software are also available in an open-source format, making it attractive for research purposes.

The Snort organisation provides two kinds of rules: Registered and Subscriber rulesets [53]. The Subscriber ruleset is paid, and they receive rulesets in real-time as they are released. It also has the complete set of known rules, along with rules for newly-identified and as-yet-unpublished (i.e., zero-day) threats. The registered ruleset is freely available for individuals and businesses and is released thirty days after the Subscriber ruleset is released.

Users can also contribute rules for detecting malware, as they create them. These community rules are not vetted by the Snort organization, unlike their own published Subscriber rules. The Snort language is flexible/customizable and relatively simple to read and understand. As already mentioned, it is designed to be efficient enough for practical use in real-time scanning for malware, making it a reasonably 'terse' language.

Indeed, the system resources required to run Snort are minimal and it can be run on small IoT devices, like a raspberry pi [54]. The Snort rule detection engine is lightweight, with excellent throughput for deep packet inspection, but execution time depends on both the number and complexity of the rules being used on network packets. Snort systems have evolved over the years, and most proprietary IDS/IPS systems resemble Snort, in terms of their architecture and the structure of their rules. A comparison of some modern IPS/IDS systems is given in Table 1, along with the key features of each one. Snort is the simplest of the three languages and is designed for single-threaded implementations, with minimal extensibility and offering no support for hardware acceleration. We chose Snort because it is the leading Opensource IPS tool on the market [52], has a very vibrant rule-writing community and most of the rules published are freely available.

Functionality	Snort	Suricata	Bro (Zeek)
Year Created	1998	2009	1998
Number of Users	600,000	-	-
High-Speed Performance	Yes	Yes	Yes
Multithreaded	Single	Multi	Multi
Modular Design	Yes	Yes	Yes
Deep Packet Inspection	Yes	Yes	No
Multiplatform Support	Windows, Linux Mac OS	Windows, Linux Mac OS	Windows, Linux Mac OS
Hardware Acceleration	No	Yes	No

Table 2: Comparison of Snort and other IDS/IPS [53]

Snort can perform protocol analysis and content searching, which is based on keyword searching. This is done using regular expressions that are built into the Snort engine. Snort can detect various attacking probe methods, like port scanning, OS fingerprinting, buffer overflow, etc. Snort syntax and structure are relatively simple with only a few keywords to be learned, making configuration, and both the reading and writing of rules relatively easy, when compared to other alternatives like Suricata and Bro. This simplicity, combined with the open-source organization, has made Snort very popular. Many rule sets have been written and published for different types of malicious software. Because of this popularity, even Suricata also accepts rules formatted in Snort format as a valid input.

Once a new exploit or type of malware is detected, researchers can study the specific behaviours and so write rules to detect them in the future. The rule base is a separate text file that the Snort detection engine parses. Therefore, rules can be added "on the fly," without needing to re-compile anything. Rules can be written that capture a specific string or pattern in the payload, although packets cannot be encrypted in this case. In cases where malicious packets *are* encrypted. For encrypted payloads, however, signature-based rules are generally less effective, although rules can be written to detect abnormal patterns or keywords in the unencrypted portion (e.g., using the header information as we demonstrated in the previous section) of a packet. Alternatively, other packet characteristics like size or timing of packets can be used [54]. The main disadvantage of these approaches are they generally lead to many false positives, when header patterns are identical to other non-malicious packets, which we have already seen to be reasonably common.

5.1.1 Snort Operation Fundamentals

In this section, we look at the various parts of Snort and how they each function. We discuss the performance of Snort in relation to other similar IPS/IDS systems. The various parts of a Snort system are shown in (figure 4).

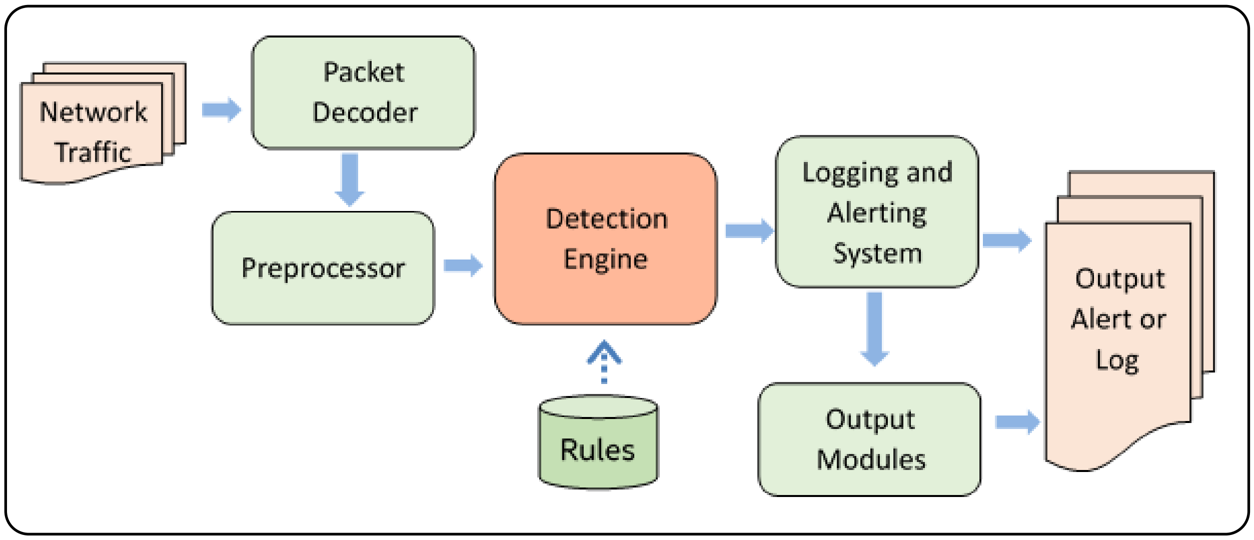


Figure 4 The Architecture of Snort[55]

Referring to the figure, we note the following, which has been adapted from [55]:

Network Traffic: The network traffic is usually captured on a network interface that is set up to be promiscuous (i.e., to detect all traffic and not just the traffic destined for the host node). Network traffic is captured via this interface and then passed on to the packet decoder.

Packet Decoder: The packet decoder determines which underlying protocols are being used in the packet (i.e., Ethernet, TCP, UDP) and saves this information, along with the size of the payload, to be used by the preprocessor.

Preprocessor: As the name suggests, the preprocessor pre-processes all received packets, which includes normalizing the packets that are being received into a format that the Snort signature processor can recognize and detecting network abnormalities that may be precursors to an intrusion.

Detection Engine: The detection engine loads the various rulesets that are specified in the Snort configuration file. The engine performs two main tasks: rule parsing and signature detection. It builds attack signatures by parsing rules. These rules are read, one line at a time, and placed into an internal data structure, to increase the overall speed of the detection process. For signature detection, the traffic is then analyzed against this internal data structure to verify if there is a match to any of the information that is held in the data structure. If so, an alert is raised; otherwise, another packet is loaded, and the process continues.

Rules: A Snort rule is a formatted set of conditions and subsequent actions that are to be performed when a packet meets the predefined criteria that is specified in those rules. Rules are written manually and are specific to a particular set of malicious behaviours and stored in a database, to be parsed by the detection engine, as already explained. Snort rules have mandatory and optional components but must be tested rigorously to ensure that they do not trigger on non-malicious traffic (i.e., no "false positives").

Logging and Alert systems and Output modules: The logging and alert systems, combined with the output modules, allow more flexible formatting for user output files. The user can specify the logging facility and the priority or logging level (e.g., critical, minor, information, etc.) of information being logged to the file. Snort logs are saved in printable ASCII format.

5.1.2 Anatomy of a Snort Rule

The Snort rule consists of two main parts: the rule header and rule options. The rule header contains actions, protocols, source and destination IP address and netmask, and the source and destination ports [56]. The rule options contain alert messages and a specification of which part of the packet should be inspected when deciding if the rule action is to be taken [56].

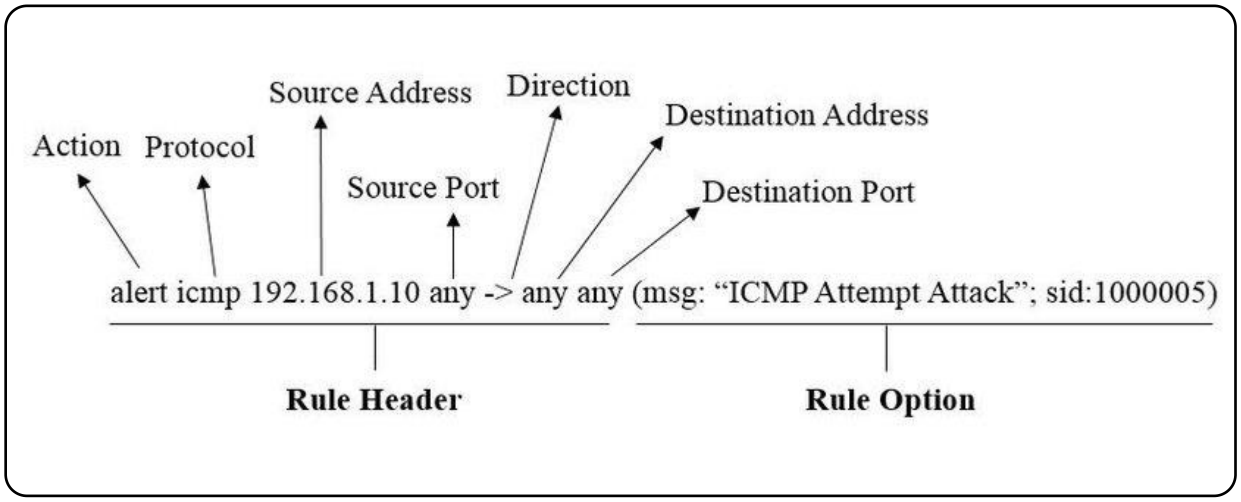


Figure 5 the structure of a snort rule

5.1.3 How Snort Rules are Written

Snort rules are created in stages. First, the purpose of each Snort rule must be defined. Rules are written to detect the defined malicious activity occurring on a network. The outlier behaviour(s) that are to be detected must be defined carefully and characterized to determine a reliable method for detection. This characterization is usually done inside a test or captive network. A rule must be created to detect the outlier behaviour(s), but this rule should not "detect" other types of traffic, especially benign or non-malicious types of traffic, since this would lead to false positives.

An example of a rule and its explanation is given below:

alert tcp any any -> 192.168.1.0/24 111 (content:"|00 01 86 a5|"; msg: "mounted access";)

The rule above indicates that: TCP traffic (from any source) is heading to the specified IP address with the corresponding subnet mask; the 111 denotes the port number. The payload section should be inspected to see if it has the specific content: "|00 01 86 a5|". If such a string is detected, an entry in the Snort log file should be created, with the corresponding message 'mounted access'.

5.1.4 How Snort rules are tested.

Once a rule is written, it needs to be tested with pcap or network files that contain the malicious traffic targeted by the rule and then tested with other types of traffic, including non-malicious traffic. As already stated, besides triggering on the desired types of malicious traffic, the objective is to reduce the number of false positives that may occur when the rule is deployed in a production environment. Real networks contain a variety of malicious and non-malicious traffic. Therefore, testing for triggering proceeds as follows: verify whether Snort is loading the rule properly, check if Snort can trigger an alert and, finally, confirm if alerts are being logged correctly [54]. Finally, the rule should be tested with a large enough representative sample of benign traffic, to make sure that it does not trigger. In real systems, the consequences of false positives can range from log overflows to impaired detection or prevention.

There are two techniques for a user to test Snort rules. The most common technique is to obtain pcap files from a trusted source, some of which are known to contain the malicious traffic type that is being targeted by the rule in question. This file can then be parsed with the designed Snort rule. This method allows rule efficacy to be verified on a representative set of samples in a safe “offline” manner without the need to gather the data from a network with “live” malware executing.

Alternatively, Snort rules can be tested on different types of “live” systems, allowing the malware in question to run (e.g., a stealth scan or a buffer overflow exploit). Obviously, such “live” systems need to be isolated from other networks and other computers very carefully (e.g., “air-gapped”), which may change the behaviour of the malware in question (i.e., if the malware is able to detect the use of these kinds of "sandbox" systems and is programmed to be dormant, in such environments). Indeed, this second technique is often used to create the files that are used with the first technique, anyway. Therefore, the difference is mostly in the captive test equipment that is required by the Snort rule writer.

5.2 Natural and Artificial Language Processing and Deep Learning

In Linguistics, Natural Language (NL) is any language that has evolved naturally in humans through repeated use and without conscious planning [57]. This definition includes all the various languages that are spoken by people during communication. An artificial Language (AL) is any language that has been specifically devised and usually by a single creator [58]. AL is normally used in computing environments. Some examples of AL include programming languages and markup languages. Since both languages have their technical requirements for forming words and sentences, they can be used in various linguistic tasks, related to computing activities [59]. These tasks include: Lexical Analysis, Statistical Analysis, Semantic Analysis, Question and Answering, Text Generation etc.

5.2.1 Artificial Language Processing

The earliest research work in AL is from Manna et al. [60] . They used a theorem-proving approach to automate the task of program synthesis. They proposed the use of mathematical induction to construct a simple program with loops or recursion.

Simon et al. [61] performed experiments to construct a compiler that makes use of heuristic problem-solving techniques. They concluded that there was no practical approach to the construction of compilers for heuristic programming. However, their interest was to understand what these tasks can teach about the nature of the programming task.

More recently, Husain et al [62] looked at the task of retrieving relevant code by using natural language queries. They sought to bridge the gap between natural language and the highly technical language that is used when creating code. This led to the creation of CODESEARCHNET CORPUS [62], which contains query-like natural language for over 2 million functions. Their objective was to simplify the process of using natural language to search for code. For example, entering '*find a function that adds two numbers*' into the code search engine should retrieve specific code examples that satisfy the query, no matter what programming language is used for the code in question.

Clement et al. [63] translated natural language and python code using transformers, that map between the two. A python method for text-to-text transformation, PYMT5, was created. This transformer can predict a single method from python documentation, which has been captured in NL.

5.2.2 Natural Language Processing

Natural Language Processing (NLP) is a subfield of linguistics, computer science and artificial intelligence (AI). NLP relates to interactions between computers and human language. at the focus is on how humans can communicate with computers in their natural language and how the computers can process the text, to make sense or meaning out of it and so proceed to respond accordingly. A very popular current use of NLP is in the recent suite of smart assistants such as Apple Siri, Google Assistant and Amazon Alexa [64].

NLP uses statistical methods and often Neural Networks to implement different tasks. Popular NLP tasks include: text to speech processing, Text generation, Question-Answering systems etc., and a growing number of others. Some common usage applications of NLP include: information retrieval, named-entity recognition and Parts of Speech tagging [65].In information retrieval, documents are found that satisfy some specific, required information within a large corpus of text. Named Entity Recognition allows identification and location, based on specific predefined categories (e.g. percentages, a person’s name, locations, medical codes, monetary values, etc.). In Parts of Speech tagging, various words in a document are classified or tagged with the corresponding part of speech (e.g., verbs, nouns, etc.). Many of these are “front-end” functions, serving to enable subsequent processing in different computer applications.

5.2.3 Using Deep Learning for NLP

Deep Learning (DL) is a subfield in machine learning that relates to algorithms inspired by the structure and function of the brain. They are often called artificial neural networks for this reason. Artificial neural networks are machine representations of human brain processes and models (i.e., based on the firing of connections between neurons).

There are several types of artificial neural networks. The popular ones include Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), Gated Recurrent Neural Networks (GRUs), Long Short Term Memory (LSTM), and Transformers, but there are others [66]. These types of neural networks currently represent the State of the Art for performing various tasks like: Image Recognition, Financial Forecasting, Text Generation, and a growing list of others.

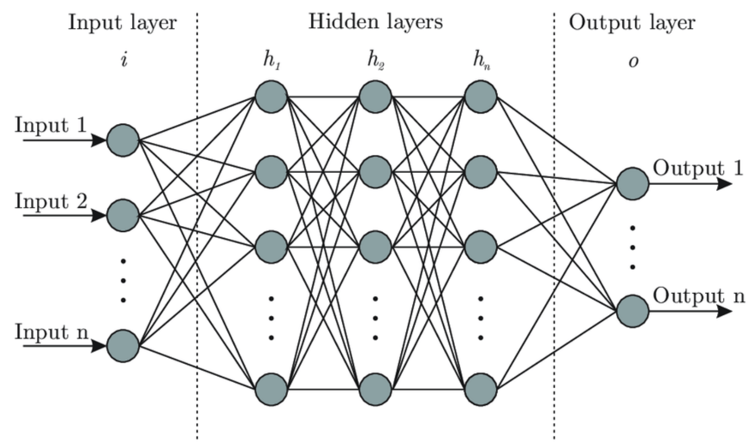


Figure 6 Structure of a Neural Network [82]

A typical Deep Neural Network (NN) consists of three basic layers (input, hidden and output layers). Each layer has a specified number of neurons. Neurons are used to compute the weighted average of their inputs, and this weighted sum is then passed through a non-linear function called the activation function. Some activation functions include the sigmoid function, this is an "S-shaped" curve that maps all inputs smoothly and in way that is continuously differentiable to output values and ensures that the output always falls between 0 and 1 [67]. Neurons are interconnected in layers, as shown in Figure 6, with neurons in a specific layer being unconnected to each other, but often being fully interconnected with previous or subsequent layers.

5.3 NLP Text Generation Techniques

Text generation, also called Natural Language Generation NLG, is a very popular application of NLP and is the closest application to our own. NLG is the process of training a system to generate coherent and logical text. This generated text might be used for various tasks, such as: writing newspaper articles, summarization of complex documents, etc.

In each case, the basic idea is to create new text that is coherent and logical. Training a neural network to generate coherent and logical text for natural languages, like English, has proved to be a very challenging task. This is due to the complexity of the English language and the fact that the system must understand the context, sentiment and other language nuances. In this way, long sentences or documents can be generated, that are both coherent and meaningful.

Recurrent Neural Networks RNN are the main class of Artificial Neural Networks DNN that are used for NLG. RNN are a type of DNN where the connection between the nodes forms a directed graph. This allows temporal dynamic behaviour to be shown [68]. RNNs can use their memory to process input sequences of variable length. This makes them suitable for text generation tasks.

DNN's are also known to be difficult to train because they are classified as Feed-Forward Networks. In feedforward networks, input signals move in one direction, from an input layer onto hidden layers and then to the output layer. In contrast, RNN's can move signals both forward and backward. Allowing feedback like this means that RNNs can even contain loops in the network. In NLP, the neural network should have the capability of 'remembering' things, because it needs to take inputs into context. The neural network should be able to ‘remember’ because it needs to have the ability to take several inputs and see how they relate to each other. Fundamentally, this “memory” property provides the ‘context’ for different inputs.

Certain real-world applications of text generation are more constrained and less "natural". An example of this would be the text generation for complicated "legalese" in legal documents, converting legal terms into more understandable forms for non-lawyers, without changing the meaning or implications of such contracts. The objective is to make documents that are easier to comprehend by those who do not have any legal training, but that are still true to their more rigorous source information. Text generation models can also be used to tailor articles or stories based on a reader’s reading preference. For our work, these types of NLG applications are the most promising, in that they resemble an AL like Snort more closely than techniques optimized for natural languages.

5.3.1 LSTM

Long Short Term Memory, LSTM, is a type of RNN. They are popular for the task of language modeling because they are capable of learning long-term dependencies. Figure 7 shows an LSTM unit with feedforward elements. There are several of these that are connected together to form layers.

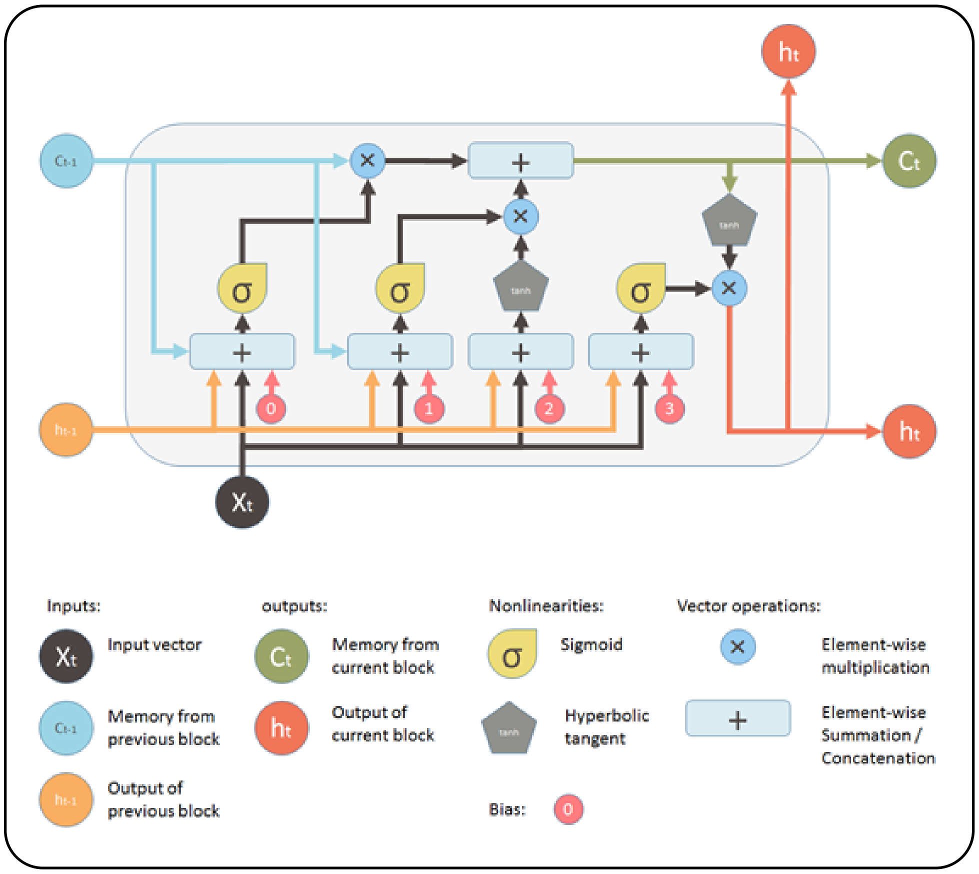


Figure 7: LSTM Unit [69]

Looking at just the inputs and outputs of the architecture shown in the figure, we observe that the network takes three inputs on the left-hand side of the diagram: X_t, (current time input), h_{t-1} (output from the previous LSTM unit) and C_{t-1} (memory input from the previous unit). This single unit makes decisions, generating its own output, h_t, and memory, C_t, by considering the current input, the previous output and the previous memory [69].

5.3.2 Transformers

Transformers are a novel architecture designed to solve the sequence-to-sequence tasks while at the same time handling long-range dependencies. Long-range dependencies refer to the ability of a neural network to remember the context in a given sequence [70]. Transformers use context (i.e., surrounding words that are related to each other) to eliminate superfluous content. Text generation techniques often use context (i.e., surrounding words in a larger group like a sentence) to improve the quality of the generated text. They can also make the derived text more concise or meaningful. A sequence-to-sequence task is a task given to a neural network to convert a text sequence into another text sequence. The goal is to understand the input sequence and create a smaller dimensional representation of that sequence. In turn, this reduced-dimension representation is then used to generate the output sequence [71]. Context is very important in NLG, mainly because a context of what is being said or what is to be understood is a basic requirement when forming new sentences that are non-cumbersome. A transformer looks at earlier and later positions in the input sequence for clues that can help it to better encode a particular word. The process of doing this is called self-attention. The self-attention module works by comparing every word in the input sequence to every other word in the input sequence, including the word itself. The word embeddings of each word are re-weighted to include contextual relevance as well as meaning.

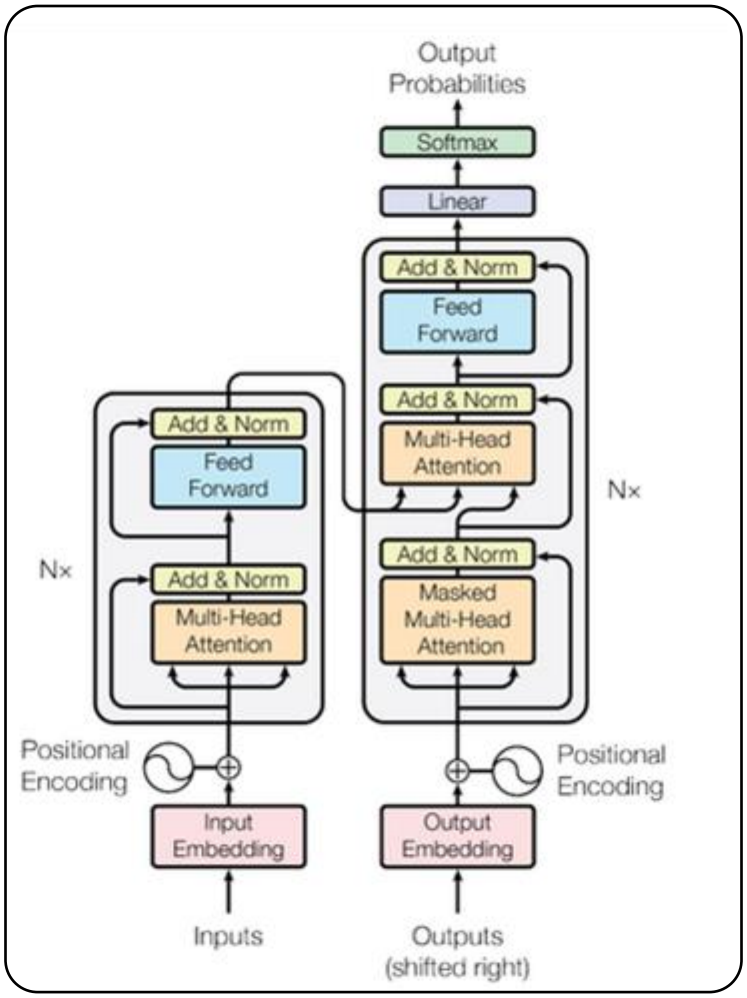


Figure 8: Basic Transformer Architecture [72]

Transformers are networks with no recurrence or loops that use only attention. Attention here is a technique that tries to mimic cognitive attention. The attention goal is to enhance or focus on the important part of the data and diminish the importance of the rest. This means that more processing power should be spent on the important parts of the data.

Recurrent Neural Networks, and transformers specifically, were hard to implement in parallel processor architectures, historically, and were less effective in learning text with long-range dependencies in the input and output sequence. Implementation was complicated by a requirement for computations that required significant memory. The term “Long-range dependencies” refers to the ability of the neural network to remember or be able to train on the context of words. Long-range dependencies imply that it takes a large part or the entirety of a sentence and then tries to make meaning or understanding out of it. The transformer can improve overall text-generation performance with judicious use of context.

Because of this emphasis on context, a transformer can also model long-term dependencies nicely. Transformers use multiple heads (i.e., neurons, with multiple attention distributions) and multiple outputs to create a single input. Multiple attention distributions have multiple attention heads running simultaneously, allowing them to attend to different parts of the input sequence in parallel. This greatly increases the speed at which these models can learn, making them more practical for many new applications, including our own.

5.3.3 Architecture of Transformers

The transformer architecture was introduced by Ashish et al. [73]. The transformer is the first model that relies entirely on self-attention. As already explained, self-attention is a technique that is used by transformers to confirm the 'understanding' using surrounding words to provide a context [72].

A transformer consists of two major parts: the encoder and decoder, both denoted by Nx, with the encoder shown on the left-hand side. The encoder has one layer of Multi-

Head attention, which is immediately followed by another layer of Feed Forward Neural Network. The decoder, on the other hand, has the same structure, just with an addition of an extra Masked Multi-Head Attention. Masking is done to make the training more parallel, thereby allowing the model to train faster.

Various neural network designs can be used in NLG tasks [74]. We have used the Generative Pretrained Transformer 2 (GPT2) model from OpenAI [21] in our work. This model is a successor to the GPT model, released to the public in 2018 [75]. The GPT-2 architecture is not entirely new and is like a decoder-only transformer. The main difference is that it is trained using a very large dataset of about 8 million web pages. When the transformer was created, no previous model had been trained on anything like as much data. Different sizes of the GPT-2 model are available. In our work, we use the 124 million parameter model size. Training speed is faster than with larger model sizes, but a smaller transformer model is more manageable, in terms of required system memory, for fine-tuning purposes with our dataset. Therefore, our selection of this model size is a trade-off between these two factors.

5.3.4 Text Generation Techniques with Context (GPT and GPT-2)

GPT is a very large transformer-based language model, with the largest model having over 1.5 Billion parameters [75]. It was trained on a dataset of 8 million web pages with the object of predicting the next, given a sequence of preceding words within different texts [75]. GPT-2 is a scaled-up version of this GPT model, with more than ten times the number of parameters of GPT, needing to be trained on more than ten times the quantity of data [75].

Lots of research has been done using GPT-2 to generate different text. Lee et al. use the transformer to generate patent claims [89]. In their work, they focused on fine-tuning the GPT-2 Model on their datasets to be able to generate patent claims. As already stated, their task is quite similar to our task, mainly due to the unique and constrained language structure often used in patent claims [89]. As with Snort, the authors were able to generate coherent patents claims using this technique. They believe that augmented inventing may be viable in the near future. In June 2020, Google researchers from OpenAI released the GPT-3 transformer [76] although it was not widely available in time for our own work. This transformer has achieved text generation at near-human levels [76]. GPT-3 is the first General Purpose AI to be popularized. It is the largest ANN ever created, with over 175 billion parameters. OpenAI researchers also showed that GPT-3 was able to generate samples of news articles, which human evaluators were not able to distinguish from those that had been written by humans. In several datasets on which GPT-3 was tested, it consistently outperformed several State of the Art Models (SOTA) models, by a margin of at least 20% Relevant SOTA models include: BERT [77], XLNET [78] and ALBERT [79]. Other researchers have also obtained some good results using similar methods [80], [81], [82], [83], [84].

5.4 Justification of the use of SNORT and NLP with Deep Learning

Despite it being an AL, we believe that Snort can still be a good fit for NLP. Snort rules follow the specific requirements of a natural language, in terms of having a prescribed structure, even though that structure is constrained by more severe grammatical rules than might be “natural”. However, not all NLP tasks would be suitable to use with Snort, which does not fit the linguistic definition of a natural language. For example, techniques that rely on subtle nuances that are present in natural languages would probably be less appropriate. These include Lemmatization & Stemming, Semantic Analysis, Sentiment Analysis etc. However, techniques like tokenization and Syntactic Analysis could be quite suitable.

Importantly, we believe that the use of context is important. A major objective of malware creators is to hide or disguise the operation of their code. Specifically, these attackers seek to avoid detection by computerized defences and adjust the behaviour of their malware, adaptively, until they are successful. By using context, illegal sequences of legitimate-looking individual packets can still be detected, increasing the effectiveness of NLP-based malware detection methods.

6 Data Collection, Experimental Results and Analysis

In this chapter, we discuss how we obtained our data. We also outline the process that we used to prepare or clean the data prior to training the model. We then discuss and analyze our results.

6.1 Data Collection

We used several sources to create the Snort Rules that are used in our work. The initial source was from the official Snort rules repository [52], which provides only the ten most recent rulesets. To get a larger and more comprehensive dataset of rules, an archived Snort ruleset from a GitHub repository was used [85]. These files contained rules for many different types of malware, but we chose to generate rules using the ruleset with the largest number of rules. This gives our Neural Network enough training data and provides a greater opportunity of improving the performance of the GPT-2 transformer. Based on this criterion, the Trojan malware ruleset was selected for our training process, containing over 7600 rules, each created to identify this specific type of malware.

6.2 Data Cleaning and Preparation

As with any Machine Learning project, the first step is to clean and prepare the data. Our data is formatted as per the requirements of Snort rules. Because these rules are quite specific, we must be careful not to remove any portions of the rule that are critical to rule operation. However, some fields of a Snort rule do not directly impact its performance, and these are discussed in [54]

Stage	Description of Task	Original Rule	Sample entries removed/ Added
0	*Initial Structure of the Snort Rule*	alert tcp $EXTERNAL_NET 443 -> $HOME_NET any (msg:"ET TROJAN Linux/Tsunami Connection Tunnel to C&C"; flow:established,to_client; content:"ET TROJAN Linux/Tsunami Connection Tunnel to C&C"; byte_test:1,&,0x80,0,relative; content:"|10 00 00 FF FF FF FF FF FF FF FF ff ff|"; depth:18; metadata: former_category TROJAN; reference:url,blogs.mcafee.com/mcafee-labs/linux-tsunami-cryptocurrency-stealer-lure/; reference:url,github.com/dwzw935/malwryc/blob/20/Linux/Tsunami-Malwryc1.7.dll; reference:url,md5,24f3c4b935de1d24f1ccc6dae5d7ee6; classtype:trojan-activity; sid:2018658; rev:1; metadata:created_at 2014_03_29, updated_at 2014_03_29;)	
1	*Removing metadata & revision entries*	alert tcp $EXTERNAL_NET 443 -> $HOME_NET any (msg:"ET TROJAN Linux/Tsunami Connection Tunnel to C&C"; flow:established,to_client; content:"ET TROJAN Linux/Tsunami Connection Tunnel to C&C"; byte_test:1,&,0x80,0,relative; content:"|10 00 00 FF FF FF FF FF FF FF FF ff ff|"; depth:18; metadata: former_category TROJAN; reference:url,blogs.mcafee.com/mcafee-labs/linux-tsunami-cryptocurrency-stealer-lure/; reference:url,github.com/dwzw935/malwryc/blob/20/Linux/Tsunami-Malwryc1.7.dll; reference:url,md5,24f3c4b935de1d24f1ccc6dae5d7ee6; classtype:trojan-activity; sid:2018658;)	metadata:created_at 2010_07_30, updated_at 2010_07_30;)
	Removing category entries:	alert tcp $EXTERNAL_NET 443 -> $HOME_NET any (msg:"ET TROJAN Linux/Tsunami Connection Tunnel to C&C"; flow:established,to_client; content:"ET TROJAN Linux/Tsunami Connection Tunnel to C&C"; byte_test:1,&,0x80,0,relative; content:"|10 00 00 FF FF FF FF FF FF FF FF ff ff|"; depth:18; reference:url,blogs.mcafee.com/mcafee-labs/linux-tsunami-cryptocurrency-stealer-lure/; reference:url,github.com/dwzw935/malwryc/blob/20/Linux/Tsunami-Malwryc1.7.dll; reference:url,md5,24f3c4b935de1d24f1ccc6dae5d7ee6; sid:2018658;)	metadata: former_category MALWARE;
	Removing reference URL entries	alert tcp $EXTERNAL_NET 443 -> $HOME_NET any (msg:"ET TROJAN Linux/Tsunami Connection Tunnel to C&C"; flow:established,to_client; content:"ET TROJAN Linux/Tsunami Connection Tunnel to C&C"; byte_test:1,&,0x80,0,relative; content:"|10 00 00 FF FF FF FF FF FF FF FF ff ff|"; depth:18; sid:2018658;)	reference:url,doc.emergingthreats.net/bin/view/Main/2008562; classtype:unknown
2	*Adding BOL and EOL descriptors to cleaned rules*	<|startoftext|> alert tcp $EXTERNAL_NET 443 -> $HOME_NET any (msg:"ET TROJAN Linux/Tsunami Connection Tunnel to C&C"; flow:established,to_client; content:"ET TROJAN Linux/Tsunami Connection Tunnel to C&C"; byte_test:1,&,0x80,0,relative; content:"|10 00 00 FF FF FF FF FF FF FF FF ff ff|"; depth:18; sid:2018658;) <|endoftext|>	<|startoftext|>, <|endoftext|>

Table 3: Stages of Snort rule cleaning

Revision entries are just informational entries that show when a rule was created and when it was last updated. Category entries and URL entries are also informational fields in Snort.

These link to documentation for the specific malware for which the rules are expected to trigger (e.g. a category of X or a URL of Zeus Trojan).

This chapter discusses the various experiments that have been performed. First, the environment and the various techniques and strategies taken to get our desired results are described. Next, the results of the rule generation process are documented. Finally, these results are analyzed using appropriate metrics.

6.3 Development and Testing Environments

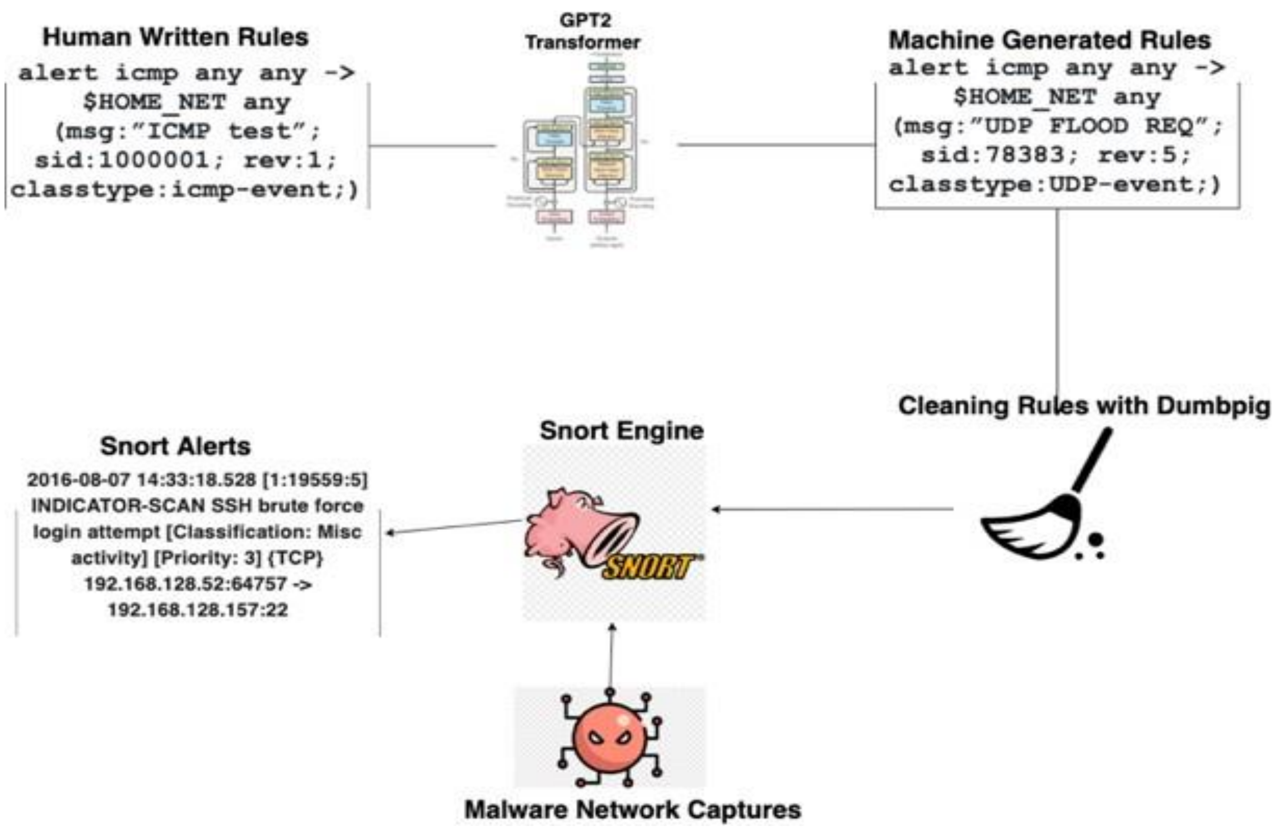
Google Colaboratory (Colab) is a free, online, cloud-based Jupyter notebook-like environment that allows any user to execute code. It is a very popular platform for developing machine learning models. This popularity is due to the availability of free Central Processing Units (CPU), Graphical Processing Units (GPU) and Tensor Processing Units (TPU) which are usually required for faster training of the machine learning models. Table 4 below gives the Hardware Specifications for Google Colab [86]. Google, Colab allows twelve hours of continuous execution [87], after which the virtual machine is reset.

Hardware	Details
CPU	Intel Xeon Processor 2.32Ghz, 2 Cores 2 Threads
GPU	Tesla P100, 16GB RAM, 3584, CUDA Cores
RAM	16GB
Storage	120GB

Table 4: Hardware Specs of Google Colab Pro

Figure 9 demonstrates the various stages of our rule generation process. Our initial Snort rule dataset was written manually by experts. These rules were cleaned to remove information redundant for Snort. This redundant information is useful to a human rule writer or reader, but was not used in our work, even though we are using techniques adapted from NLP research.

The next stage was to train GPT-2 on the dataset. After some training epochs (distinct periods of time, where training is iterated over the entire training dataset), we then proceeded to generate sample rules using the trained model. The resulting rules were cleaned using the dumbpig tool [88] and then fed into Snort. Pcap files with malware and ‘goodware’ were also used as inputs into Snort.



```
alert icmp any any ->
$HOME_NET any
(msg:"ICMP test";
sid:1000001; rev:1;
classtype:icmp-event;)
```

```
alert icmp any any ->
$HOME_NET any
(msg:"UDP FLOOD REQ";
sid:78383; rev:5;
classtype:UDP-event;)
```

```
2016-08-07 14:33:18.528 [1:19559:5]
INDICATOR-SCAN SSH brute force
login attempt [Classification: Misc
activity] [Priority: 3] {TCP}
192.168.128.52:64757 ->
192.168.128.157:22
```

Figure 9: Snort-Based Rule Generation Process using GP2

6.4 Training of the Neural Network

Small (124M), Medium (345M) and Large (774M)) GPT-2 models are possible. As already stated, because of persistent insufficient memory issues on Google Colab Pro, we experimented with the medium and large model sizes we only used the small 124M Parameter model.

The key parameters for the architecture of the model are:

- **Vocabulary Size:** This is parameter is represented by *vocab_Size*: and it defines the number of different tokens that can be represented by the *inputs_ids*. We used 50257, which is the default. [89].
- **Dimensionality of the causal mask:** This is represented by *n_ctx*. The value used is 1024, which is the default. [89]
- **Dimensionality of the embeddings and hidden states:** This is represented by *n_embd*. The default value of 768 was used [89].
- **Number of Attention heads:** We used a value of 12 for the number of heads, which is the default value and is denoted by *n_head* [89]
- **Hidden Layers:** We used the default value of 12 for the number of *hidden* layers in our network. This is denoted by *n_layer* [89]

If non-default parameters were used, the output would probably be less suitable for use in detecting malware samples. For our purposes, the default values seem “reasonable”. This is because the though the GPT-2 transformer is pretrained on natural language. Since Snort rules have a similar structure to English language the default values worked well for our task.

6.4.1 GPT2-small (124M-parameter) model

The small size of the model makes it faster to train. The figures below show the statistics of our model during training over 2000 epochs (i.e., optimization phases).

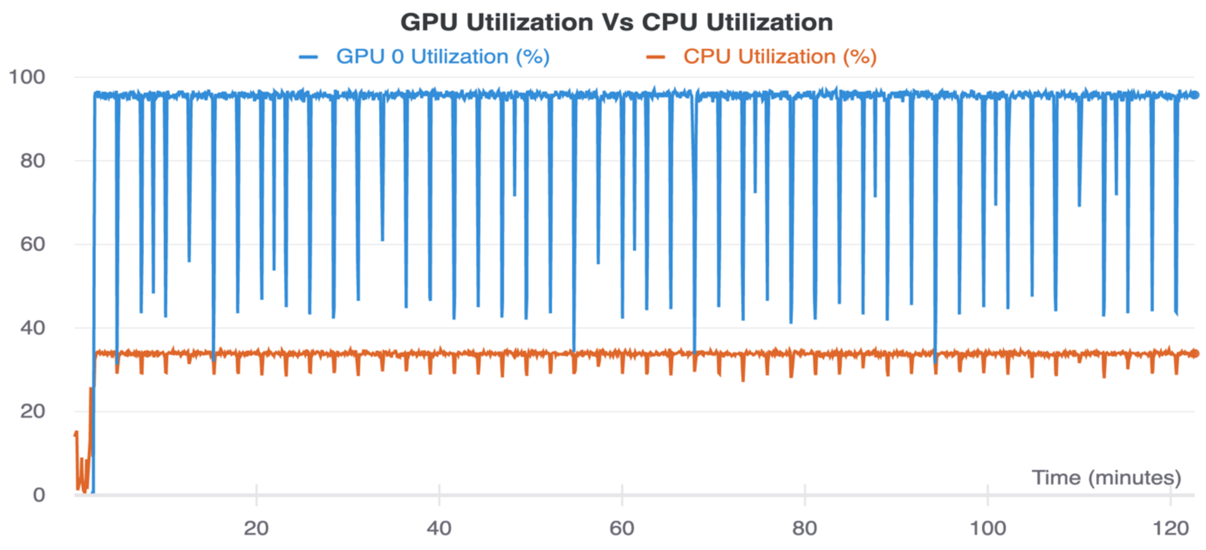


Figure 10: GPU Vs CPU utilization when training

Figure 10 shows the GPU vs CPU Utilization. This clearly shows that the neural network uses the GPU more heavily than the CPU when training.

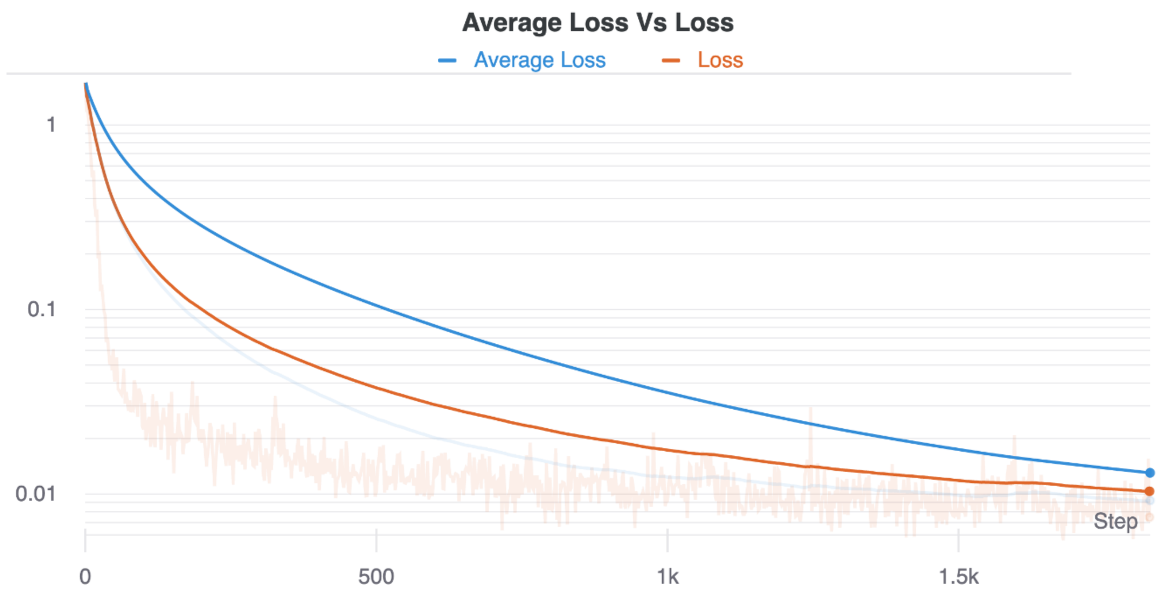


Figure 11 Average Loss Vs Loss when training

(Figure 11) shows the average loss over the epochs, compared with the actual loss over the epochs. Eventually, the neural network converges at around 1700 epochs, which gives acceptable performance, in terms of computation time.

Figure 12 shows a comparison of the system memory compared to the GPU memory utilization during the training period. The graph is intuitive, showing that a lot more GPU memory than system memory is required for training the neural network to accommodate our large Snort ruleset of over 7,600 rules.

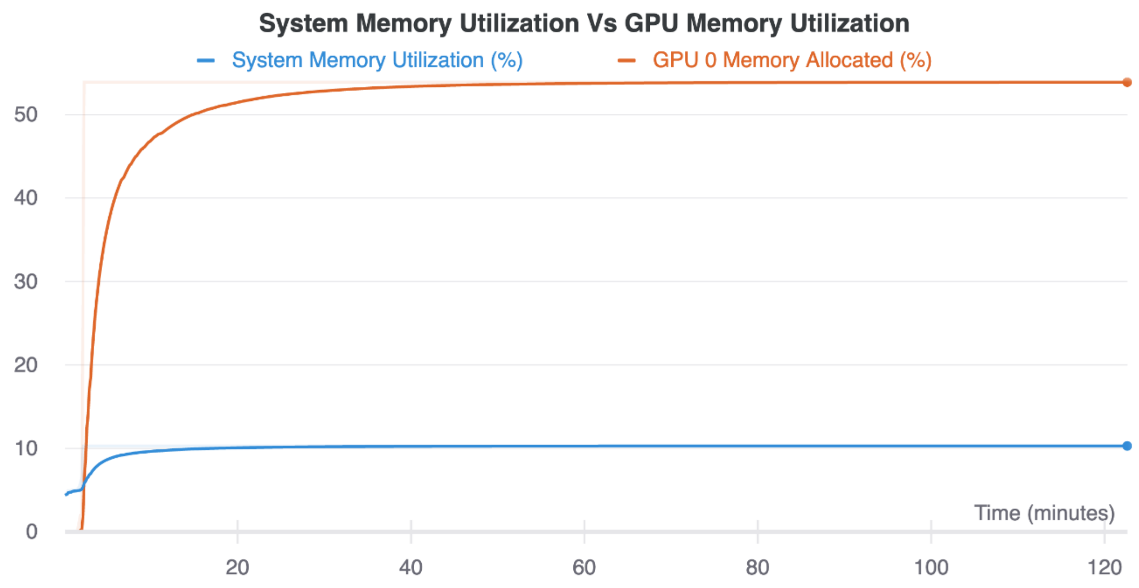


Figure 12: System memory Vs GPU Memory use during training

6.4.2 Rule Generation

During the training stage, the ANN generated periodic printed samples of the rules at every 200 epochs. This enables us to monitor the progression of the rules being generated, as the number of epochs increased. As the training process converges, these rules stabilize to constant values with very subtle changes being made as rules generation progresses.

GPT-2 has several parameters that can be adjusted, when requesting the model to generate rules. One of them is referred to as temperature. The Temperature parameter ranges between *0.1* and *1.0*, and it defines how 'hot' or 'cold generated texts would be. The higher or hotter the value, the more random the output will be. This means that there is more deviation from the original dataset, which may also deviate from the structure of a Snort rule.

6.4.3 Cleaning Up Generated Rules

Cleaning of generated rules happens in three distinct stages. In the first stage, the Notepad++ text editor is used, along with a plugin called ‘compare’. This allows the two files to be compared for basic differences, manually. One file contained the generated rules, and the other file contained the dataset that was used to train the neural network. Any generated rules that matched exactly with the dataset were commented out, manually, as being trivially-generated instances.

Next, the derived rules were cleaned up, to make them both syntactically correct and also forcing them to meet the basic syntax requirements of a Snort rule. *Dumbpig* was used to do this checking [88]. *Dumbpig* is a Perl tool developed by Leon Ward to check for 'dumb' or obvious Snort rule errors. Some of the checks performed by the script include checking if each rule has revision numbers, class type, SID number etc. The script also checks to ensure that a rule does not do deep packet inspection and a host of other best practices for rule-writing.

Using *Dumbpig*, we were able to trim down our generated set of rules from 400 rules generated by the neural network, down to 187. In the final stage of cleaning, Snort was then used to sanitize the rules yet further. Snort has an in-built pre-preprocessor engine that also checks the syntax of rules and ensures that they meet the standards.

Using the Snort engine, we were able to detect rules with the same Snort ID (SID). Whenever we noticed a repeated SID, we would first check to see if the two rules were syntactically the same. If they were, we would comment out the duplicate. When they were not identical, we would give the new rule a different ID. After this stage of cleaning was complete, a total of 92 valid rules remained.

6.5 Evaluation Framework

Using an artificial language like Snort, we need to formulate various ways of evaluating the effectiveness of our rules. Since normal NLP techniques are known not to work with such artificial languages fully, we decided to evaluate our work using the following criteria. Based on our discussions with working security professionals, Snort rules written by humans are often evaluated using similar criteria in industry. These criteria are:

Functional Correctness: Whether rules are indeed able to perform the tasks they were designed to do (i.e., to detect malware)

Human Evaluation: We had initially planned to ask experts in the field to review our rules and give us their professional insight into them (i.e., do the generated rules look: mundane, interesting, 'weird', wrong etc.) Unfortunately, we were not able to get enough experts to spend the time required to do this, so we compared our rules to proprietary, documented, Snort rule-writing best practices, instead. Unfortunately, we have not been given permission to publish these practices, making this criterion a difficult one to present, but not decreasing its importance.

False Positive Evaluation: Our limited set of human rule reviewers all highlighted that false positives are a major practical issue. Therefore, we evaluated our rules using sample 'goodware,' i.e., known malware-free traffic, to quantify the number of false positives for any specific Snort rule. In truth, the difficulty of testing against all possible types of goodware' is similar to or greater than the complexity of getting good test coverage over all types of malware.

NLP Metrics: We used two NLP metrics: BLEU and ROUGE to measure the quality of our rules. We explain the challenges of using these metrics in the section 0 and section 6.5.4, respectively.

6.5.1 Functional Correctness (True Positive evaluation)

A basic requirement is that the generated rules must be able to detect malware properly. At this stage, our task was to test if any of the newly generated rules could detect malware samples. We had about 980 varied malware samples provided to us by the Nokia Threat Intelligence Labs [90] sing data from their in-house captive testing system and also from a smaller private version of that same lab.

We placed the new rules in the location where all other rules are tested. We then proceeded to add the entry to Snort.conf and then commented out all other rules located in that configuration file. Using the command *Snort -r foo.pcap,* we listed all the 980 pcap files and allowed Snort to parse each pcap file using each of the rules specified in the rules file. The -r flag indicates the pcap filenames you would want Snort to parse. The results of some of the malware that triggered detection are shown in the table below. [1]

Snort RuleID	Name of Malware Detected
2020789	TROJAN Zeus Bot GET to Google checking Internet connectivity, TROJAN W32/SpeedingUpMyPC.Rootkit Install CnC Beacon, MALWARE W32/PullUpdate.Adware CnC Beacon, TROJAN Linux/Onimiki DNS trojan-activity long format (Outbound), TROJAN Fareit/Pony Downloader Checkin 3, TROJAN Zeus Bot GET to Google checking Internet connectivity
2020789, 50064, 49772	92MALWARE Win32/BrowseFox.H Checkin 2
45093	MOBILE_MALWARE Android.Walkinwat Sending Data to CnC Server
45093, 45092	MOBILE_MALWARE AndroidOS.Simplocker Checkin

Table 5: Sample Malware triggered by generated rules

As can be seen from the table, a comparatively small number of malware files were triggered, and only a few of the rules triggered them. We believe this to be the case because of the limitations of our sample size (i.e., GPT-2 results were based on millions of training samples, rather than thousands). Our initial requirement when obtaining the rules and malware pcap files was to have a one-to-one/one-to-many efficiency mapping, where each pcap file triggered at least 1 rule. We reached out to several Snort and malware repositories,

[1] In this part of our experiment, we performed deep packet inspection. Hence, we used the complete information contained in a network packet capture

such as Emerging Threats and Snort User and Signature mailing lists to try to obtain such data but we were not successful. Virus Total granted us an educational license to their malware repository, but this license did not allow us to request specific malware samples. Instead, the samples were randomly allocated. Only paying customers can request specific samples. Because of their policy, all provided files were zipped executable files, rather than being in the desired pcap file format.

However, based on the work that we have been able to do in analyzing the network behaviour of malware families like Trojans, we are very hopeful, but cannot show, that malware samples would trigger some of the other rules that we have generated, if only we had more of them.

6.5.2 Human Evaluation—Hand-written Evaluation Set

We compared our generated rules with the human-written rules to see if there was any novelty in our rules. To do this, we shared the generated rules with a Snort researcher on the Snort mailing list called Andrew Williams, who works actively with Snort at Cisco and his feedback was: "Taking a quick look at the rules, most of the individual pieces (e.g.: content sections) seem fine, but it is really difficult to say whether it would detect malicious traffic and/or whether some of the more generic rules would lead to false positives in the field. We face this challenge a lot when creating rules as well - how to make them specific enough to not FP but also have them match on a wide range of malware-generated traffic"

His comments indicate that our rules are “fine”, to a first order, at least. However, the problem of “false positives” was raised, while still obtaining high malware “true positive” coverage. The industry approach to measuring this is to assess rules (generated by humans or otherwise) in a captive testing lab, as has already been discussed.

In spite of the proprietary nature of Snort rules writing at Cisco, Nokia and other large network vendors, here are a few documented, public best practices that can be followed when writing such rules [91]. Two of these are now discussed:

Targeting the vulnerability and not the exploit kit: This advises the rule writer to focus on writing a rule that targets the vulnerability that the malware is exploiting, rather than writing the rule to detect the exploit itself. There are numerous exploit kits available. To write a rule to detect a particular kit would mean that the moment the malware designer only needs to make slight changes or ‘dither’ their attack to bypass the rule [91]. Our rules were obtained from the approved Snort rules repository and so are vetted to be in line with this best practice, before being approved for release to the broader community that use the rules. Therefore, our generated rules are in line with this best practice, since the training data was defined in exactly this way.

Detect the “Oddities” of the Protocol: This best practice focuses on ensuring that the rule matches the various options or oddities that a protocol permits (i.e. all of the legal variations of a particular ‘signature’). A typical example would be the FTP protocol which allows different ways for the username to be written like user<space>root user<tab>root [92]. Snort rules should be written to match *all* of the possibilities allowed by a protocol. In most cases, for the rule to handle all such possibilities, a Perl-Compatible Regular Expression (PCRE) keyword is used. This allows the user to write a regular expression to match all the possibilities. It was observed that most of the rules that had specific protocols nested in them also used the PCRE keywords.

6.5.3 Nuisance-Triggering Avoidance ("False Positive" Evaluation)

Our public and private human rule reviewers emphasized the importance of avoiding false positives, which can quickly cause overflows in log files and can reduce real-time detection times for malware. We decided to evaluate our rules using sample 'goodware', (i.e., known malware-free traffic) to see if we would get any false positives. This is, by no means, an exhaustive test in a proper captive testing facility, but is a basic type of "Go/No Go" test.

To obtain representative samples of goodware, we downloaded goodware pcaps from [https://www.stratosphereips.org/datasets-normal] as shown in (Table 6). The CTU dataset was selected, since it is a widely-used source of both malware and goodware datasets [93]. Other 'goodware' datasets contain either encrypted malware or were not considered to be representative of everyday browsing activities. As already indicated, testing against all possible types of benign traffic is a formidable task. Even in industry applications, we were told, some "false positive" problems are only detected in the field and not in the pre-testing that is done in a captive network.

Name of Capture File	Description	Size (MB)
CTU-Normal-7:	P2P traffic from a user with a Linux laptop.	398
CTU-Normal-12:	P2P traffic running on a normal Linux notebook (with webpage access)	809
CTU-Normal-13:	Computer access to this website https://www.us.hsbc.com	494k
CTU-Normal-20	Traffic collected with users browsing the internet and using different desktop applications	296M
CTU-Normal-21:	Traffic collected with users browsing the internet and using different desktop applications	297M

Table 6: 'Goodware' dataset used for testing false positives

As a very basic sanity check, we were able to achieve zero false positives with all of our generated rules. Direct access to live systems with 'wild' malware and 'goodware' samples would be required for more reliable estimates of functional correctness and nuisance-triggering avoidance. However, this still might not be adequate, in terms of adequate testing coverage.

In this section, Natural Language Processing metrics are used to analyze the rules generated by the neural network. To measure the quality of our generated rules for text generation, the generated text coherence is measured, for a large body of text.

As already stated, a basic way to measure rule quality is to determine whether the generated rule is indeed effective at detecting malware or not. To do this, we used the following metrics: **Bil**ingual **E**valuation **U**nderstudy Score (BLEU), **R**ecall **O**riented **U**nderstudy for

Gisting **E**valuation (ROUGE) score. These scores are described in the next two sections and the results are shown in tables for each section.

BLEU: Bilingual Evaluation Understudy Scores

BLEU-n is a metric that calculates an n-gram (i.e., recall that this was n "words", each separated by spaces) overlap between the reference text and the generated text. This n-gram overlap is a measure of precision or "similarity" between the generated text and correctly generated text and is independent of the order of the n-grams. However, the BLEU-n calculation process also includes a stage where ubiquitous words or constructs are identified and included as being mandatory, rather than discretionary. There is also a brevity penalty in BLEU, which is applied when the generated text is too small compared to the target text.

The ideal behaviour for a BLEU score would be that, as the n-gram size is increased, the score monotonically reduces and gradually levels out between a score of 0.6 -0.7 [39]. With the addition of more layers, the neural network converges faster but might also tend towards overfitting. In our experiment results (Figure 13 & Table 6), the 12-layer neural network is the closest to the desired performance and the 16-layer neural network BLEU scores for various n-grams deviates the most. We believe that this is indeed because of overfitting types of effects. It can also be observed that the baseline performances are far worse than for our test case. This is because Snort rules are required to follow specific formats in terms of keywords used and where these keywords are to be situated within the text.

Transformer Layers	**BLEU-1**	**BLEU-2**	**BLEU-3**	**BLEU-4**
8L	0.887	0.697	0.482	0.438
12L	0.937	0.799	0.601	0.414
16L	0.65	0.645	0.569	0.523
8L-baseline1	0.398	0.143	0.123	0.353
12L-baseline1	0.326	0.312	0.183	0.122
16L-baseline1	0.11	0.113	0.186	0.16
8L-baseline2	0.181	0.138	0.186	0.174
12L-baseline2	0.166	0.16	0.117	0.221
16L baseline2	0.276	0.214	0.154	0.166
8L-baseline3	0.133	0.22	0.216	0.133
12L baseline3	0.288	0.265	0.278	0.23
16L baseline3	0.177	0.131	0.166	0.249
8L baseline4	0.279	0.128	0.159	0.255
12L baseline4	0.232	0.208	0.121	0.251
16L baseline4	0.114	0.266	0.238	0.231

Table 7: BLEU Scores

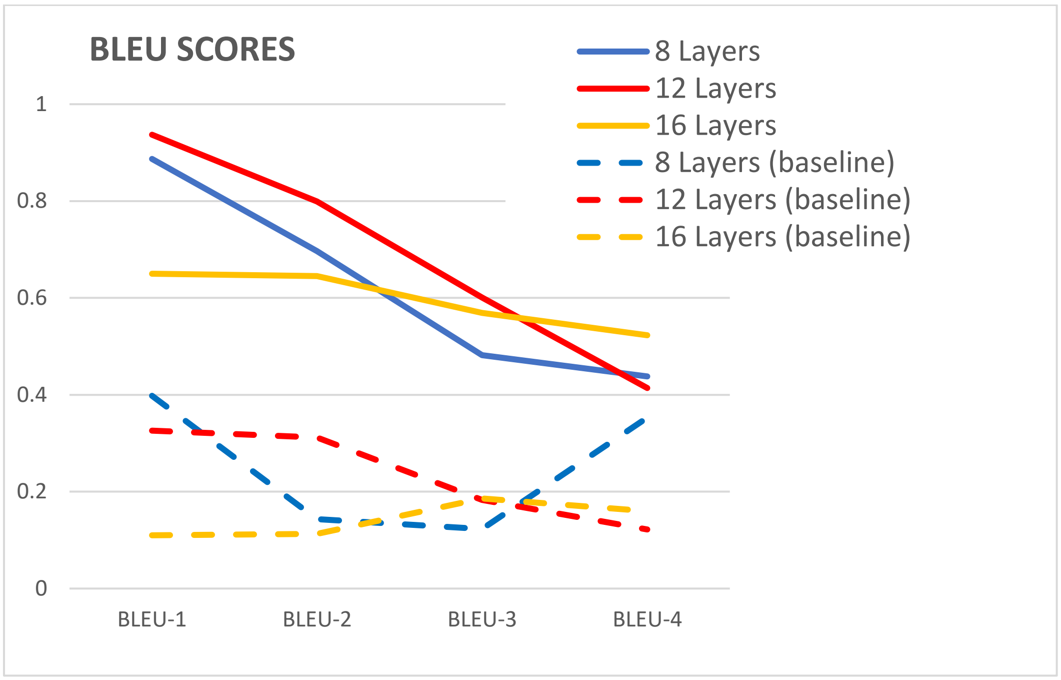


Figure 13: A plot of BLEU 1-4 scores

6.5.4 ROUGE: Recall Oriented Understudy for Gisting Evaluation

A ROUGE metric is often reported along with BLEU scores for standard tasks. While similar to the BLEU definition, ROUGE is recall-focused, rather than precision-focused (see previous definitions for these terms). The ROUGE scores count the number of overlapping n-grams that are found in both the model's output and the reference text and then divides this by the total number of n-grams in the reference text.

There are three types of ROUGE score. The most common is ROUGE-n, which has n-gram overlap, where n can be 1 or higher. L-ROUGE checks for the Longest Common subsequence, rather than for n-gram overlap. The third type of ROUGE score is s-ROUGE, which focuses on the number of skip grams, where a skip-gram is a contiguous set of s words that are skipped or ignored. In our work we used ROUGE-n , which seemed best at capturing all of the relevant information contained in the reference text.

The expected behaviour for a ROUGE-n score would be that values increase to 1.0 as n increases, indicating that the model is capturing all relevant information in the dataset. Different ROUGE-n scores are plotted for our method (Figure 14 & Table 7). The 12-layer model approaches 1.0 at the fastest rate. As with the BLEU score, the 12-layer model is optimal for training with our dataset and provides more relevant machine-generated rules.

Transformer	ROUGE-1	ROUGE-2	ROUGE-3	ROUGE-4
8L	0.683	0.699	0.717	0.818
12L	0.721	0.812	0.894	0.983
16L	0.521	0.583	0.618	0.759
8L-baseline1	0.226	0.187	0.112	0.242
12L-baseline1	0.382	0.238	0.125	0.358
16L-baseline1	0.194	0.249	0.101	0.137
8L-baseline2	0.054	0.269	0.245	0.394
12L-baseline2	0.201	0.017	0.043	0.291
16L baseline2	0.291	0.013	0.232	0.066
8L-baseline3	0.258	0.185	0.118	0.216
12L baseline3	0.022	0.28	0.067	0.334
16L baseline3	0.127	0.309	0.106	0.314
8L baseline4	0.247	0.351	0.099	0.095
12L baseline4	0.316	0.082	0.324	0.169
16L baseline4	0.234	0.22	0.289	0.241

Table 8: ROUGE Scores 1-4

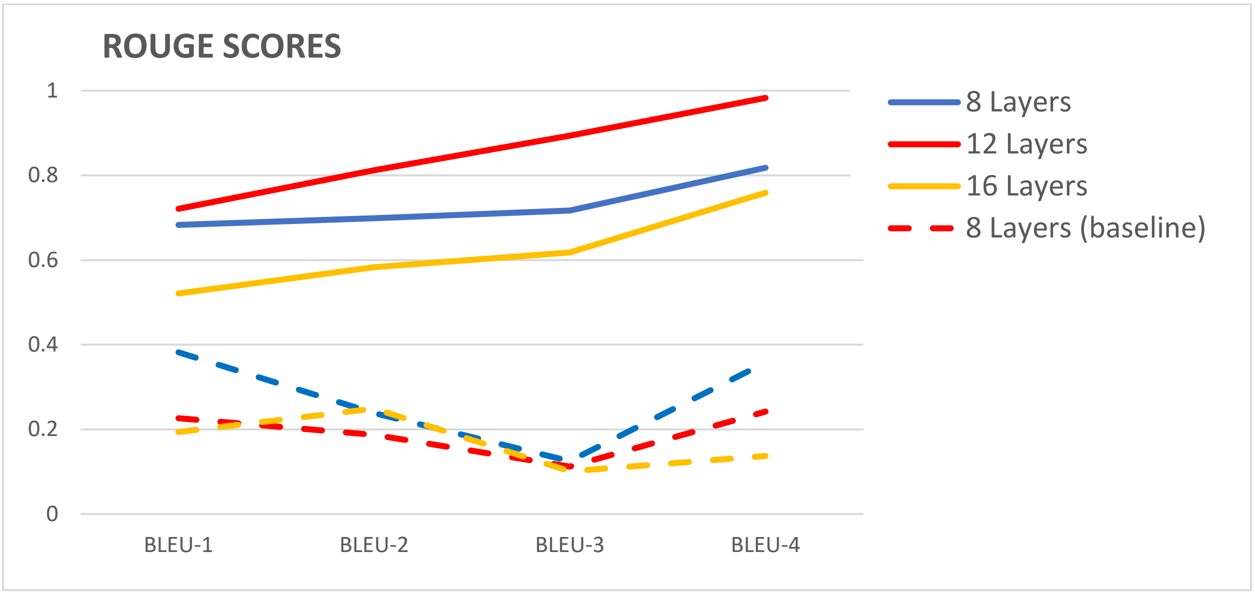


Figure 14: A plot of ROUGE scores 1-4

In summary, the increasing ROGUE-n scores imply that our model is capturing the relevant information contained in the dataset and the decreasing BLEU-n scores indicate that our model is not merely regurgitating rules form the original dataset. These two factors imply that the quality of our model is high. Also, since the main task of our model is to generate text for a language that is artificial and the structure and syntax are not varying a lot, the higher scores also show that our model is not generating syntactically errored text. It can also be observed that all the baseline performances are far worse than for our test cases. This is because Snort rules are required to follow specific formats in terms of keywords used and where these keywords are to be situated within the text therefore having them rearranged randomly would not lead to them being similar to the base text.

The quality of the text generated is measured more practically by applying these generated rules to real malware (in a real-world network or simulator) and then evaluating the detectable types of malware. Indeed, other researchers [94] have also noted that these NL metrics are less suited for Artificial Language (AL) tasks and suggest that AL tasks are measured better using functional correctness metrics instead (e.g., in “live” or simulated systems). This would have been our preference too, given access to such resources.

6.6 Broader Impact

This technique, if further investigated, has the potential to be useful in a variety of ways:

Speed up the generation of Snort rules: Currently, Snort rule creation requires an extensive amount of domain knowledge and malware behaviour knowledge. In our discussions with Rule writing experts at the Nokia Threat intelligence lab, at least one co-op term (i.e. about four months) is required to get students to the point where they can write efficient and effective rules. In addition to this, extra time is also required for students to understand new types of malware behaviour and then analyze a malware sample to identify the unique string or behaviour that is going to be used in a generated rule.

Enable non-rule writers to generate usable rules: With our process, people who have almost zero knowledge on how to create Snort rules or, potentially, little knowledge of the operation or behaviour of the malware, can still generate working rules quickly. Malware researchers can use this tool to generate rules and test malware “in the wild”. If the generated rules detect the malware, the core part of the rule can be analyzed to see if it matches the unique behaviour of specific, desired malware being analyzed. Unfortunately, we have not been able to do this testing ourselves.

The implicit assumption for usefulness here is that new malware is somehow predictable from older types of malware or, more specifically, to the rules that were created to detect that malware at that time. The NLP techniques we propose here are expected to partially represent or aggregate some of the skills of the experts who generated these rules.

7 Conclusions and Future Work

This chapter gives a summary of our findings, the main contribution of our research work challenges that we encountered and suggestions for improving and extending the work.

7.1 7. 1 Summary of Contributions

We have examined the process of automating the task of Snort rule generation using neural networks. The task has always been done by humans, and it is quite a tedious and time-consuming one. It is also a task that requires skill to understand the specific behaviour of the malware before it is possible to create an effective and efficient corresponding detection rule. Our solution has the benefits of requiring less time and reducing the required amount of expert knowledge about virus behaviour or rule-writing.

- We demonstrate a method in which we use a selection of a few elements in an IP header to give context and then proceed to classify traffic flowing in a network as malware (or not) although performance seems to vary with the "temporal locality" of data packets in the file. This variation was not investigated further.
- We demonstrate a novel method of training a neural network to generate malware rules for the Snort IDS that seem to be "reasonably effective".
- We show that machine-generated Snort rules can be used in tandem with the Snort engine to detect malware flowing through a network.

- We demonstrate that our generated rules do not trigger any false positives when tested with everyday malware-free (i.e., 'goodware') traffic, albeit using a limited subset of such traffic and limited amounts of testing.
- We describe various methods by which machine-generated rules can be sanitized to ensure they meet the strict requirements of the Snort language.
- We show that a neural network can be trained on old Snort rules and generate new rules that can detect more recent types of malware, although this requires further testing over time (i.e. with future types of malware) in “live” networks before stronger statement can be made about practicality.

7.2 Challenges

Access to a wider set of pcap files, generated in different types of networks would have helped to refine our method using only the packet headers. Alternatively, techniques for sorting pcap files to isolate flows (e.g. based on the source and destination address) might have helped to create a more robust detection method, when compared with a randomize baseline of packets. The difficulty of obtaining a “diverse” set of pcap files, with unique temporal distributions of packets for both malicious and non-malicious software was somewhat side-stepped by the main approach described in this work (i.e. by using rules generated to fire only for specific malware types).

As mentioned earlier in the work, for this main approach, the previous difficulty of getting a diverse set of pcap files is replaced by the task of getting a one-to-one/one-to-many mapping of Snort rules and their corresponding malware is also a major challenge. Such a dataset would simplify the training and detection phases, allowing us to assume that a set of rules were representative *before* starting the analysis and synthesis work. With active adversaries, malware can be expected to continue to evolve so that it becomes undetectable by existing defences, so this would need to be an ongoing process. It would have been ideal to have been able to study the performance of generated rules over a longer duration in a captive or “live” network, running representative levels of traffic, as malware was adapted in this way.

Besides having access to our own captive network, it would have been very useful to have had access to a wider set of Snort experts and their generated rules. We believe that this would have enriched the training process. Initially, assistance and expert opinions were received from a contact person at the Nokia Threat Intelligence Lab (who we’d like to thank, anonymously!). However, with recent global pandemic developments and resulting changes in the security landscape, this same person was not available for consultation and nor was their lab, nor were other possible diverse sources of ruleset creators and testers.

7.3 Conclusions

We believe that this work provides a basis for further research into using GPT-2 (and its successor, GPT-3) to train more artificial languages, including rule specification languages like Snort. In the field of rule writing for Snort, this work can be used to explore the various possibilities that exist in rule writing. To the extent that previous versions of malware are predictive of future ones (i.e., “zero day” attacks that are variants of previous malware types) we have shown that our method has good value in being used to create more rules for various families of malware. The possibilities seem endless since code can be considered as an Artificial Language. This work also shows that the future of malware signature generation has a lot of potential using upcoming language models being designed.

7.4 Future work

For future work, we recommend the use of the largest size of the GPT2 model (i.e. the 1.5 Billion parameter model [89, p. 2] or, better still the OPT Model[95] which is a new language model which is open access. Recent variants of GPT-3, especially the Codex model, which is being used for code generation, are also exciting developments that might be useful tools. Unfortunately, because of its big parameter size and our limited processing and memory, these were not feasible in our testing. With a more powerful platform, these tools could be trained with the same dataset that we have used. However, an even larger number of parameters might allow more data points to be generated from the initial rule set and thereby improve yield.

Rather than using our offline analysis approach alone, it would be useful to correlate classification performance with results that were obtained in “live” networks and/or captive networks, running with representative traffic environments, in terms of both malware and ‘goodware’. Becoming a paid subscriber to the rules feeds/services or enlisting their co-operation to gain access to the Snort rules that are being used currently might also improve the results. Unfortunately, the proprietary nature of this content, stemming from the hard-won expertise that is needed to write effective and efficient rules that generate true positives only, makes this unlikely. For example, neither of our industrial collaborators were prepared to grant us public access to this material.

Finally, it would be informative to study the “future” effectiveness of the rules we have generated. This would allow us to test more thoroughly the implicit assumption that is made in this work about malware similarities over time (i.e. that new types of malware are derived from previous malware versions, in a way that is detectable at the network level). It would also be informative to measure the effects of new types of “goodware” and how changes in benign traffic patterns over time affect our generated rules.

8 References

[1] J. A. P. Marpaung, M. Sain, and H.-J. Lee, “Survey on malware evasion techniques: State of the art and challenges,” in 2012 14th International Conference on Advanced Communication Technology (ICACT), Feb. 2012, pp. 744–749.

[2] A. Somayaji, “How to win an evolutionary arms race,” IEEE Secur. Priv., vol. 2, no. 6, pp. 70–72, Nov. 2004, doi: 10.1109/MSP.2004.100.

[3] N. Milošević, “History of malware,” ArXiv13025392 Cs, Jan. 2014, Accessed: Aug. 16, 2021. [Online]. Available: http://arxiv.org/abs/1302.5392

[4] Q. Chen and R. A. Bridges, “Automated Behavioral Analysis of Malware: A Case Study of WannaCry Ransomware,” in 2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA), Dec. 2017, pp. 454–460. doi: 10.1109/ICMLA.2017.0-119.

[5] “What is WannaCry ransomware?,” www.kaspersky.com, Apr. 26, 2021. https://www.kaspersky.com/resource-center/threats/ransomware-wannacry (accessed Aug. 16, 2021).

[6] C. Srinivasan, “Hobby hackers to billion-dollar industry: the evolution of ransomware,” Comput. Fraud Secur., vol. 2017, no. 11, pp. 7–9, Nov. 2017, doi: 10.1016/S1361-3723(17)30081-7.

[7] S. Sibi Chakkaravarthy, D. Sangeetha, and V. Vaidehi, “A Survey on malware analysis and mitigation techniques,” Comput. Sci. Rev., vol. 32, pp. 1–23, May 2019, doi: 10.1016/j.cosrev.2019.01.002.

[8] J. Aycock, Computer Viruses and Malware. Springer US, 2006. doi: 10.1007/0-387-34188-9.

[9] H. Orman, “The Morris worm: a fifteen-year perspective,” IEEE Secur. Priv., vol. 1, no. 5, pp. 35–43, Sep. 2003, doi: 10.1109/MSECP.2003.1236233.

[10] J. Milletary, “Citadel Trojan Malware Analysis,” p. 18, 2012.

[11] “MADAM: Effective and Efficient Behavior-based Android Malware Detection and Prevention | IEEE Journals & Magazine | IEEE Xplore.” https://ieeexplore.ieee.org/abstract/document/7422770 (accessed Aug. 17, 2021).

[12] K. Griffin, S. Schneider, X. Hu, and T. Chiueh, “Automatic Generation of String Signatures for Malware Detection,” in Recent Advances in Intrusion Detection, 2009, pp. 101–120.

[13] S. K. Sahay, A. Sharma, and H. Rathore, “Evolution of Malware and Its Detection Techniques,” in Information and Communication Technology for Sustainable Development, Singapore, 2020, pp. 139–150. doi: 10.1007/978-981-13-7166-0_14.

[14] D. Gavriluţ, M. Cimpoeşu, D. Anton, and L. Ciortuz, “Malware detection using machine learning,” in 2009 International Multiconference on Computer Science and Information Technology, Oct. 2009, pp. 735–741. doi: 10.1109/IMCSIT.2009.5352759.

[15] K. Sethi, S. K. Chaudhary, B. K. Tripathy, and P. Bera, “A Novel Malware Analysis Framework for Malware Detection and Classification using Machine Learning

Approach," in Proceedings of the 19th International Conference on Distributed Computing and Networking - ICDCN '18, Varanasi, India, 2018, pp. 1–4. doi: 10.1145/3154273.3154326.

[16] L. Liu, B. Wang, B. Yu, and Q. Zhong, "Automatic malware classification and new malware detection using machine learning," Front. Inf. Technol. Electron. Eng., vol. 18, no. 9, pp. 1336–1347, Sep. 2017, doi: 10.1631/FITEE.1601325.

[17] H. Rathore, S. Agarwal, S. K. Sahay, and M. Sewak, "Malware Detection Using Machine Learning and Deep Learning," in Big Data Analytics, Cham, 2018, pp. 402–411. doi: 10.1007/978-3-030-04780-1_28.

[18] I. Firdausi, C. lim, A. Erwin, and A. S. Nugroho, "Analysis of Machine learning Techniques Used in Behavior-Based Malware Detection," in 2010 Second International Conference on Advances in Computing, Control, and Telecommunication Technologies, Dec. 2010, pp. 201–203. doi: 10.1109/ACT.2010.33.

[19] K. A. Scarfone and P. M. Mell, "Guide to Intrusion Detection and Prevention Systems (IDPS)," National Institute of Standards and Technology, Gaithersburg, MD, NIST SP 800-94, 2007. doi: 10.6028/NIST.SP.800-94.

[20] D. Mudzingwa and R. Agrawal, "A study of methodologies used in intrusion detection and prevention systems (IDPS)," in 2012 Proceedings of IEEE Southeastcon, Mar. 2012, pp. 1–6. doi: 10.1109/SECon.2012.6197080.

[21] S. B.S., N. S., N. Kashyap, and S. D.N., "Providing Cyber Security using Artificial Intelligence – A survey," in 2019 3rd International Conference on Computing

Methodologies and Communication (ICCMC), Mar. 2019, pp. 717–720. doi: 10.1109/ICCMC.2019.8819719.

[22] S. R. Kumbhar, “An Overview on Use of Artificial Intelligence Techniques in Effective Security Management,” vol. 2, no. 9, p. 6, 2007.

[23] K. R. Dalal and M. Rele, “Cyber Security: Threat Detection Model based on Machine learning Algorithm,” in 2018 3rd International Conference on Communication and Electronics Systems (ICCES), Oct. 2018, pp. 239–243. doi: 10.1109/CESYS.2018.8724096.

[24] Y. Ki, E. Kim, and H. K. Kim, “A novel approach to detect malware based on API call sequence analysis,” Int. J. Distrib. Sens. Netw., vol. 2015, p. 4:4, Jan. 2015, doi: 10.1155/2015/659101.

[25] V. S. Sathyanarayan, P. Kohli, and B. Bruhadeshwar, “Signature Generation and Detection of Malware Families,” in Information Security and Privacy, Berlin, Heidelberg, 2008, pp. 336–349. doi: 10.1007/978-3-540-70500-0_25.

[26] M. Roesch, “Snort – Lightweight Intrusion Detection for Networks,” p. 11, 1999.

[27] A. Sagala, “Automatic SNORT IDS rule generation based on honeypot log,” 2015 7th Int. Conf. Inf. Technol. Electr. Eng. ICITEE, 2015, doi: 10.1109/ICITEED.2015.7409013.

[28] N. Fallahi, A. Sami, and M. Tajbakhsh, “Automated flow-based rule generation for network intrusion detection systems,” in 2016 24th Iranian Conference on Electrical

Engineering (ICEE), May 2016, pp. 1948–1953. doi: 10.1109/IranianCEE.2016.7585840.

[29] T. Vollmer, J. Alves-Foss, and M. Manic, “Autonomous rule creation for intrusion detection,” in 2011 IEEE Symposium on Computational Intelligence in Cyber Security (CICS), Apr. 2011, pp. 1–8. doi: 10.1109/CICYBS.2011.5949394.

[30] J. Nivethan and M. Papa, “Dynamic rule generation for SCADA intrusion detection,” in 2016 IEEE Symposium on Technologies for Homeland Security (HST), May 2016, pp. 1–5. doi: 10.1109/THS.2016.7568964.

[31] “Top 10 BEST Intrusion Detection Systems (IDS) [2021 Rankings].” https://www.softwaretestinghelp.com/intrusion-detection-systems/ (accessed Dec. 07, 2021).

[32] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, “‘Andromaly’: a behavioral malware detection framework for android devices,” J. Intell. Inf. Syst., vol. 38, no. 1, pp. 161–190, Feb. 2012, doi: 10.1007/s10844-010-0148-x.

[33] F. A. Narudin, A. Feizollah, N. B. Anuar, and A. Gani, “Evaluation of machine learning classifiers for mobile malware detection,” Soft Comput., vol. 20, no. 1, pp. 343–357, Jan. 2016, doi: 10.1007/s00500-014-1511-6.

[34] W. Wang, M. Zhu, X. Zeng, X. Ye, and Y. Sheng, “Malware traffic classification using convolutional neural network for representation learning,” in 2017 International Conference on Information Networking (ICOIN), Jan. 2017, pp. 712–717. doi: 10.1109/ICOIN.2017.7899588.

[35] W. Huang and J. W. Stokes, “MtNet: A Multi-Task Neural Network for Dynamic Malware Classification,” in Detection of Intrusions and Malware, and Vulnerability Assessment, Cham, 2016, pp. 399–418. doi: 10.1007/978-3-319-40667-1_20.

[36] W. Hardy, L. Chen, S. Hou, Y. Ye, and X. Li, “DL4MD: A Deep Learning Framework for Intelligent Malware Detection,” p. 7.

[37] J.-S. Lee and J. Hsiang, “Patent claim generation by fine-tuning OpenAI GPT-2,” World Pat. Inf., vol. 62, p. 101983, Sep. 2020, doi: 10.1016/j.wpi.2020.101983.

[38] D. Bekerman, B. Shapira, L. Rokach, and A. Bar, “Unknown malware detection using network traffic classification,” in 2015 IEEE Conference on Communications and Network Security (CNS), Sep. 2015, pp. 134–142. doi: 10.1109/CNS.2015.7346821.

[39] R. Sommer and A. Feldmann, “NetFlow: Information loss or win?” 2002.

[40] S. Wang, Q. Yan, Z. Chen, B. Yang, C. Zhao, and M. Conti, “Detecting Android Malware Leveraging Text Semantics of Network Flows,” IEEE Trans. Inf. Forensics Secur., vol. 13, no. 5, pp. 1096–1109, May 2018, doi: 10.1109/TIFS.2017.2771228.

[41] Y. Nagano and R. Uda, “Static analysis with paragraph vector for malware detection,” in Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication, Beppu Japan, Jan. 2017, pp. 1–7. doi: 10.1145/3022227.3022306.

[42] T. K. Tran and H. Sato, “NLP-based approaches for malware classification from API sequences,” in 2017 21st Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES), Nov. 2017, pp. 101–105. doi: 10.1109/IESYS.2017.8233569.

[43] A. Mohaisen, A. G. West, A. Mankin, and O. Alrawi, “Chatter: Classifying malware families using system event ordering,” in 2014 IEEE Conference on Communications and Network Security, Oct. 2014, pp. 283–291. doi: 10.1109/CNS.2014.6997496.

[44] B. A. AlAhmadi and I. Martinovic, “MalClassifier: Malware family classification using network flow sequence behaviour,” in 2018 APWG Symposium on Electronic Crime Research (eCrime), May 2018, pp. 1–13. doi: 10.1109/ECRIME.2018.8376209.

[45] H. Mekky, A. Mohaisen, and Z.-L. Zhang, “Separation of benign and malicious network events for accurate malware family classification,” in 2015 IEEE Conference on Communications and Network Security (CNS), Sep. 2015, pp. 125–133. doi: 10.1109/CNS.2015.7346820.

[46] “DeepTraffic/Htbot.7z at master · echowei/DeepTraffic,” GitHub. https://github.com/echowei/DeepTraffic (accessed Aug. 19, 2021).

[47] I. U. Haq, S. Ali, H. Khan, and S. A. Khayam, “What Is the Impact of P2P Traffic on Anomaly Detection?,” p. 17.

[48] Q. V. Le and T. Mikolov, “Distributed Representations of Sentences and Documents,” ArXiv14054053 Cs, May 2014, Accessed: Aug. 18, 2021. [Online]. Available: http://arxiv.org/abs/1405.4053

[49] Computer Security and the Internet. Accessed: Jun. 14, 2022. [Online]. Available: https://link.springer.com/book/10.1007/978-3-030-83411-1

[50] C.-Y. Ho, Y.-C. Lai, I.-W. Chen, F.-Y. Wang, and W.-H. Tai, “Statistical analysis of false positives and false negatives from real traffic with intrusion detection/prevention systems,” IEEE Commun. Mag., vol. 50, no. 3, pp. 146–154, Mar. 2012, doi: 10.1109/MCOM.2012.6163595.

[51] H. Kurniawan, Y. Rosmansyah, and B. Dabarsyah, “Android anomaly detection system using machine learning classification,” in 2015 International Conference on Electrical Engineering and Informatics (ICEEI), Aug. 2015, pp. 288–293. doi: 10.1109/ICEEI.2015.7352512.

[52] “Snort Rules and IDS Software Download.” https://www.snort.org/downloads (accessed Aug. 10, 2021).

[53] “What are the differences in the rule sets?” https://www.snort.org/faq/what-are-the-differences-in-the-rule-sets (accessed Aug. 10, 2021).

[54] Cisco, SNORT\textregistered Users Manual 2.9.16. The Snort Project, 2020.

[55] “Hyperscan and Snort* Integration,” Intel. https://www.intel.com/content/www/us/en/develop/articles/hyperscan-and-snort-integration.html (accessed Aug. 10, 2021).

[56] “Understanding and Configuring Snort Rules | Rapid7 Blog,” Rapid7, Dec. 09, 2016. https://www.rapid7.com/blog/post/2016/12/09/understanding-and-configuring-snort-rules/ (accessed Aug. 10, 2021).

[57] D. T. Langendoen, “Review of Natural Language and Universal Grammar,” Language, vol. 69, no. 4, pp. 825–828, 1993, doi: 10.2307/416893.

[58] “Artificial Languages,” obo. https://www.oxfordbibliographies.com/view/document/obo-9780199772810/obo-9780199772810-0164.xml (accessed Aug. 10, 2021).

[59] E. D. Liddy, “Natural Language Processing,” p. 15.

[60] Z. Manna and R. J. Waldinger, “Toward automatic program synthesis,” Commun. ACM, vol. 14, no. 3, pp. 151–165, Mar. 1971, doi: 10.1145/362566.362568.

[61] H. A. Simon, “Experiments with a Heuristic Compiler,” J. ACM, vol. 10, no. 4, pp. 493–506, Oct. 1963, doi: 10.1145/321186.321192.

[62] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “CodeSearchNet Challenge: Evaluating the State of Semantic Code Search,” ArXiv190909436 Cs Stat, Jun. 2020, Accessed: Aug. 23, 2021. [Online]. Available: http://arxiv.org/abs/1909.09436

[63] C. B. Clement, D. Drain, J. Timcheck, A. Svyatkovskiy, and N. Sundaresan, “PyMT5: multi-mode translation of natural language and Python code with transformers,” ArXiv201003150 Cs, Oct. 2020, Accessed: Aug. 23, 2021. [Online]. Available: http://arxiv.org/abs/2010.03150

[64] “Top NLP-Based Personal Assistant Apps Used In 2019.” https://analyticsindiamag.com/top-nlp-based-personal-assistant-apps-used-in-2019/ (accessed May 04, 2022).

[65] "Natural Language Processing (NLP): What Is It & How Does it Work?," MonkeyLearn. https://monkeylearn.com/natural-language-processing/ (accessed May 03, 2022).

[66] C. C. Aggarwal, Neural Networks and Deep Learning: A Textbook. Cham: Springer International Publishing, 2018. doi: 10.1007/978-3-319-94463-0.

[67] "What are neurons in neural networks / how do they work?," Cross Validated. https://stats.stackexchange.com/questions/241888/what-are-neurons-in-neural-networks-how-do-they-work (accessed Aug. 19, 2021).

[68] L. R. Medsker and L. C. Jain, "American University."

[69] S. Yan, "Understanding LSTM and its diagrams," Medium, Nov. 15, 2017. https://blog.mlreview.com/understanding-lstm-and-its-diagrams-37e2f46f1714 (accessed Aug. 19, 2021).

[70] "Range Dependency - an overview | ScienceDirect Topics." https://www.sciencedirect.com/topics/computer-science/range-dependency (accessed Aug. 19, 2021).

[71] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," ArXiv14093215 Cs, Dec. 2014, Accessed: Aug. 19, 2021. [Online]. Available: http://arxiv.org/abs/1409.3215

[72] J. Alammar, "The Illustrated Transformer." https://jalammar.github.io/illustrated-transformer/ (accessed Aug. 19, 2021).

[73] A. Vaswani et al., “Attention Is All You Need,” ArXiv170603762 Cs, Dec. 2017, Accessed: Aug. 19, 2021. [Online]. Available: http://arxiv.org/abs/1706.03762

[74] Z. Xie, “Neural Text Generation: A Practical Guide,” ArXiv171109534 Cs Stat, Nov. 2017, Accessed: Aug. 19, 2021. [Online]. Available: http://arxiv.org/abs/1711.09534

[75] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language Models are Unsupervised Multitask Learners,” p. 24.

[76] “OpenAI API,” OpenAI, Jun. 11, 2020. https://openai.com/blog/openai-api/ (accessed Aug. 19, 2021).

[77] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” ArXiv181004805 Cs, May 2019, Accessed: May 03, 2022. [Online]. Available: http://arxiv.org/abs/1810.04805

[78] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le, “XLNet: Generalized Autoregressive Pretraining for Language Understanding,” ArXiv190608237 Cs, Jan. 2020, Accessed: May 03, 2022. [Online]. Available: http://arxiv.org/abs/1906.08237

[79] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, “ALBERT: A Lite BERT for Self-supervised Learning of Language Representations,” ArXiv190911942 Cs, Feb. 2020, Accessed: May 03, 2022. [Online]. Available: http://arxiv.org/abs/1909.11942

[80] G.-3 Demo, “10 Thought experiments | GPT-3 Demo.” https://gpt3demo.com/apps/10-thought-experiments (accessed Aug. 19, 2021).

[81] G.-3 Demo, “Blog Idea Generator by Topic | GPT-3 Demo.” https://gpt3demo.com/apps/topic-blog-idea-generator (accessed Aug. 19, 2021).

[82] G.-3 Demo, “Compose.ai | GPT-3 Demo.” https://gpt3demo.com/apps/compose-ai (accessed Aug. 19, 2021).

[83] G.-3 Demo, “Essay Writing by EduRef | GPT-3 Demo.” https://gpt3demo.com/apps/essay-writing-by-eduref (accessed Aug. 19, 2021).

[84] G.-3 Demo, “Ask me anything | GPT-3 Demo.” https://gpt3demo.com/apps/ask-me-anything (accessed Aug. 19, 2021).

[85] “GitHub - codecat007/snort-rules: An UNOFFICIAL Git Repository of Snort Rules(IDS rules) Releases.,” Jun. 28, 2021. https://web.archive.org/web/20210628084100/https://github.com/codecat007/snort-rules/ (accessed Aug. 10, 2021).

[86] “Google Colaboratory.” https://colab.research.google.com/drive/151805XTDg--dgHb3-AXJCpnWaqRhop_2#scrollTo=gsqXZwauphVV (accessed Aug. 09, 2021).

[87] “Colaboratory – Google.” https://research.google.com/colaboratory/faq.html (accessed Aug. 09, 2021).

[88] L. Ward, dumbpig. 2020. Accessed: May 03, 2022. [Online]. Available: https://github.com/leonward/dumbpig

[89] “OpenAI GPT2.” https://huggingface.co/transformers/model_doc/model_doc/gpt2.html (accessed Aug. 10, 2021).

[90] “Threat Intelligence Center,” Nokia. https://www.nokia.com/networks/products/threat-intelligence-center/ (accessed May 03, 2022).

[91] “Writing Effective Snort Rules with Examples [Best Practices],” Coralogix, Oct. 19, 2020. https://coralogix.com/blog/writing-effective-snort-rules-for-the-sta/ (accessed Aug. 23, 2021).

[92] “3.9 Writing Good Rules.” http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node36.html (accessed Aug. 23, 2021).

[93] S. Shamshirband and A. T. Chronopoulos, “A New Malware Detection System Using a High Performance-ELM method,” ArXiv190612198 Cs, Jun. 2019, Accessed: Nov. 24, 2021. [Online]. Available: http://arxiv.org/abs/1906.12198

[94] M. Chen et al., “Evaluating Large Language Models Trained on Code,” ArXiv210703374 Cs, Jul. 2021, Accessed: Aug. 19, 2021. [Online]. Available: http://arxiv.org/abs/2107.03374

[95] S. Zhang et al., “OPT: Open Pre-trained Transformer Language Models,” arXiv, arXiv:2205.01068, May 2022. doi: 10.48550/arXiv.2205.01068.

9 APPENDIX A

Source Code available at https://github.com/afoteygh/SnortRuleGen

Sample Machine Generated Rules which triggered

```
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any (msg:"ET TROJAN
ELF/Emptiness PWS Variant CnC Beacon"; flow:established,from_server; content:"|02
00 00 01 00 00 00 00|"; fast_pattern:only; sid:2020789;)

alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"MALWARE-
CNC User-Agent known malicious user-agent string - Win.Tool.Shindo";
flow:to_server,established; content:"User-Agent|3A| Apache-
HttpClient/UNAVAILABLE"; fast_pattern:only; http_header; metadata:impact_flag red,
policy balanced-ips drop, policy security-ips drop, service http; sid:45093;)

alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"MALWARE-
CNC User-Agent known malicious user-agent string - Win.Tool.Shindo";
flow:to_server,established; content:"User-Agent|3A| Apache-
HttpClient/UNAVAILABLE"; sid:45092;)

alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any (msg:"ET TROJAN
ELF/Emptiness PWS Variant CnC Beacon"; flow:established,from_server; content:"|02
00 00 01 00 00 00 00|"; fast_pattern:only; sid:2020789;)


alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"MALWARE-
CNC Doc.Trojan.Agent variant outbound cnc connection attempt";
flow:to_server,established; content:"WinHttp.WinHttpRequest.5"; fast_pattern:only;
http_header; content:!"Referer"; http_header; metadata:impact_flag red, policy balanced-
ips drop, policy max-detect-ips drop, policy security-ips drop, service http; sid:50064;)

alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"MALWARE-
CNC Doc.Trojan.Agent variant outbound cnc connection attempt";
flow:to_server,established; content:"WinHttp.WinHttpRequest.5"; fast_pattern:only;
http_header; content:!"Referer"; http_header; metadata:impact_flag red, policy balanced-
ips drop, policy max-detect-ips drop, policy security-ips drop, service http; sid:49772;)
```

Sample Machine Generated Rules at various temperatures

<u>0.1</u>

```
alert tcp $HOME_NET any -> $EXTERNAL_NET 25 (msg:"ET TROJAN SC-KeyLog
Keylogger Installed - Sending Log Email Report"; flow:established,to_server;
content:"SC-KeyLog log report"; nocase; content:"See attached file"; nocase;
content:".log"; nocase; reference:url,www.soft-central.net/keylog.php;
reference:url,doc.emergingthreats.net/2008348; classtype:trojan-activity; sid:2008348;
rev:2; metadata:created_at 2010_07_30, updated_at 2010_07_30;)
```

```
alert tcp $HOME_NET any -> $EXTERNAL_NET 1024: (msg:"ET TROJAN Turkojan
C&C Keepalive (BAGLANTI)"; flow:established,to_server; dsize:9;
content:"BAGLANTI?"; metadata: former_category MALWARE;
reference:url,doc.emergingthreats.net/2008026; classtype:trojan-activity; sid:2008026;
rev:3; metadata:created_at 2010_07_30, updated_at 2010_07_30;)
```

<u>0.7</u>

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"ET TROJAN
Banload FakeAV GET Request to .zsh file"; flow:established,to_server; content:"GET";
http_method; content:"/files/DownloadReport.zip"; http_uri; metadata: former_category
MALWARE; reference:url,www.fireeye.com/blog/fireeye-technical-threat-
analysis/2014/07/a-lil-monthly-operation-uplinkocean-to-the-chairs.html;
classtype:trojan-activity; sid:2017972; rev:2; metadata:created_at 2014_07_27,
updated_at 2014_07_27;)
```

<u>0.9</u>

```
:10; classtype:trojan-activity; sid:2029222; rev:1; metadata:affected_product
Windows_XP_Vista_7_8_10_Server_32_64_Bit, created_at 2017_03_10,
performance_impact Low, updated_at 2019_09_28;)<|endoftext|>
```

<u>1.0</u>

```
alert; metadata: former_category MALWARE;
reference:md5,18b182656fda9d6f3f4ebd06d9d1fea10; classtype:trojan-activity;
sid:2018671; rev:2; metadata:created_at 2014_11_07, updated_at
2014_11_07;)<|endoftext|>

<|startoftext|>alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS
(msg:"ET TROJAN x2telegram sending data via SMTP"; flow:to_server,established;
content:"POST"; http_method; content:"POST"; http_client_body; depth:10;
pcre:"/^POST\x2e\x20/U"; content:"POST (User-Agent|3a 20|SMSV1|20 2d 0a|";
http_header; fast_pattern:36,20; pcre:"/^POST\x2d\xff\x20/U"; content:!"Referer|3a
20|"; http_header; metadata: former_category TROJAN;
reference:url,github.com/Dwzw935/malwryc/blob/20/Linux/Tsunami-Malwryc2.7.dll;
reference:url,github.com/dwzw935/malwryc/blob/20/Linux/Tsunami
```