University of Cincinnati

Date: 6/27/2022

I, Joshua Owens, hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Science.

It is entitled:

Towards a Malware Language for Use with BERT Transformer—An Approach Using API Call Sequences

Student's name: **Joshua Owens**

This work and its defense approved by:

Committee chair: Carla Purdy, Ph.D.

Committee member: John Gallagher, Ph.D.

Committee member: Anca Ralescu, Ph.D.

43029

Towards a Malware Language for Use with BERT Transformer—An Approach Using API Call Sequences

A thesis submitted to the

Graduate School

of the University of Cincinnati

in partial fulfillment of the

requirements for the degree of

Master of Science

In the Department of Electrical Engineering and Computer Science

Of the College of Engineering and Applied Sciences

By

Joshua Owens

B.A. University of Cincinnati

June 2012

Thesis Advisor & Committee Chair: Carla Purdy, Ph.D.

// Abstract

Google's BERT (Bidirectional Encoder Representations from Transformers) algorithm is a neural network based method for processing natural language. In this exploratory study we have used API call sequences to generate a language for use with BERT to perform malware classification with Support Vector Machines. Detecting malware using sequences of API calls has been shown to be a promising area for malware detection, especially when used in conjunction with other features such as opcodes and system calls. The increase in detection accuracy and efficiency achieved through the use of BERT is a desired outcome as malware authors develop more sophisticated techniques for obfuscating their behavior. We have used an open-source dataset that contains sequences of API calls from both known malware and from non-malware and have performed analysis using Support Vector Machines (SVM) for classification, a common method used in previous work on detecting malicious API-based attacks, while using BERT as a preprocessor.

Acknowledgments

I first need to thank my initial advisor and friend Dr. George Purdy. I greatly enjoyed the time spent with him and am thankful for all that he was willing to teach me.

My current advisor and committee chair, Dr. Carla Purdy, who was willing to step in after George's sudden death and to never give up on me even during stretches where I struggled to continue.

I would also like to thank my committee members Dr. Anca Ralescu and Dr. John Gallagher who were so gracious to put up with a lot of last-minute planning.

None of this would be possible without the love and support of my wife Becky and our children. The road has been long, difficult and at times seemingly never-ending, and through it all you were always encouraging me to continue.

Additionally, I wish to thank my good friends Elise Simonsen and Sarah Simon who helped transform the person I was into the person I have become. Leading by example and showing me that it was ok to show up and be seen as my authentic self.

Contents

Abstract i

Acknowledgments iii

List of Tables vi

Chapter 1 Introduction 1

1.1 Background 1

1.2 Motivation 2

1.2.1 Innate Immune Malware Detection System 2

1.2.2 Survey of malware features 4

1.2.2.1 Static Analysis Features 4

1.2.2.2 N-Grams 4

1.2.2.3 Opcodes 4

1.2.2.4 Dynamic Analysis Features 5

1.2.2.5 API calls 5

1.2.2.6 Hooking 6

1.2.3 Dataset selection 7

1.2.4 Malware detection implementation 8

1.3 Research Objectives 10

1.4 Overview of the Thesis 11

Chapter 2 Background 12

2.1 Literature Review 12

Chapter 3 Proposed Approach 16

3.1 Recurrent Neural Network 16

3.2 BERT 17

3.3 Dataset Description 18

Chapter 4 Experiments and Results 20

4.1 Experiments 20

4.1.1 Data Preprocessing 20

4.1.2 Environmental Setup 22

4.2 Evaluation Metrics 22

4.3 Results 23

Chapter 5 Conclusions and Future Work 25

Bibliography 28

List of Tables

Table 1 Most frequent API calls in dataset 20

Table 2 Least frequent API calls in dataset 21

Table 3 Desktop PC Hardware Specifications 22

Table 4 Comparison of classification accuracy 24

Figure 1 API call freqency analysis in dataset 21

Chapter 1 Introduction

1.1 Background

An email from a colleague arrives in your inbox asking you to review a word document for submission. Seconds after opening the document, malware has scanned your outlook contacts and sends a similar message with an attachment purporting to be from you. The document is empty and seems suspicious but nothing bad happens, so you delete the email and continue working. A trojan has been placed on your system and the system of everyone else who opens the word document. Bad actors now have the ability to access and control your system remotely and can sell access to the botnet it has created.

You are using a workplace pc and suddenly the screen locks up with a message that all your files have been encrypted and will be deleted permanently unless you pay a ransom. You and your organization have followed rigorous security protocols, but a vulnerability in the SMB port of the operating system has allowed malware to penetrate system defenses. Through no actions of your own, your workstation has transferred an infected file through the vulnerable port to workstations throughout your company, bringing production to a halt and costing thousands of dollars in profits.

Both of these situations are real and are recent examples of the increasing destructiveness and sophistication of malware. The first example was the Emotet trojan from 2018 and the second was Wannacry ransomware in 2017. The need to defend systems against attacks is not limited to computer workstations. Every device with a computer chip is susceptible to malicious attacks

and more and more products are being developed that use computer technology, from industrial systems all the way down to personal products like toothbrushes and toasters.

Just as malware authors continue to develop more advanced techniques to defeat malware detection, researchers must continue to develop new techniques in order to protect systems and data from attack.

1.2 Motivation

This research is motivated by a problem proposed by the Air Force Research Lab at Wright Patterson Air Force Base in Dayton, OH. That work was motivated by the desire to create a malware detection system based upon the human innate immune system.

1.2.1 Innate Immune Malware Detection System

The innate immune system in humans is the most primitive form of immune system in biological organisms. The innate immune response is an immediate reaction detection of non-specific foreign pathogens with the intention of preventing the spread of infection and activating the adaptive immune response to target the specific threat.

This research focused on the viability of an innate immune response system for detection of malware. The premise was that there are identifying features of malware that are not present in benign software that would allow for the construction of a detection system that would be able to quickly identify malicious behavior.

Our research has not produced a set of features present only in malicious software. One major difficulty in coming up with such a feature set is the term malicious is dependent on the context in which it is used, i.e., an action might be malicious on one type of system and not on another type of system. If malicious behavior of software is non-deterministic, the underlying patterns of

actions that promote this behavior will also be non-deterministic. There will necessarily always be some form of probabilistic determination involved with the detection of malicious behaviors.

The theory that all malicious software exhibits certain features not present in benign software is similar to the linguistical theory of a Universal Grammar, which theorizes that at the most primitive level, human languages all share certain characteristics and that the human brain has built in mechanisms to allow language acquisition [1]. What we wish to have is a set of properties, the universal grammar, of malicious software. Since the linguistical theory has been studied for decades and no universal grammar has been identified it was highly unlikely that we could devise one for malicious software in the short time available for this research, this work will continue as this will be further researched as a part of my dissertation, and I suspect for many years after.

The best approximation that can be made for a universal grammar of malicious features in short order is a hybrid approach that uses several different malicious software behaviors in conjunction with multiple forms of classifiers. The cells of the innate immune system have receptors that detect specific proteins that are present on foreign pathogens. Each receptor is specific to the type of protein that is detected which corresponds to the classification technique suggested in our computer innate immune system.

Our system gains several benefits from considering the classifier as the receptors and the features as proteins. Each feature may be best detected with a different classifier, which directly resembles the situation with protein detection in the innate immune system. Since each classifier runs independently of any other we can offload the computation tasks and all classifiers can be run simultaneously. The output of each classifier will be some weighted probability function that will be combined with all the other probability functions in order to give a result.

1.2.2 Survey of malware features

1.2.2.1 Static Analysis Features

Static analysis of malware involves analyzing the source code or reverse engineering of executable files so that malicious behaviors can be identified without running the malicious programs. A summary of some of the most used static analysis features follow.

1.2.2.2 N-Grams

In a search for a universal grammar of malware we started with a bottom up approach, thinking of the smallest parts of software that we could consider as the building blocks of the grammar. Individual bits do not provide any insight into maliciousness, so we considered byte N-grams and assembly N-grams as potential sources of this grammar.

Several methods of N-grams have been used in the literature for malware detection. Byte N-grams and assembly N-grams have shown some usefulness with static analysis, but the information is highly correlated with the header of the binary file [2].

Taking sequences of bytes, or lines of assembly code we can construct N-grams to be used for malicious behavior detection.

1.2.2.3 Opcodes

We define an opcode to be the assembly instruction (jmp, mov, add, etc.) and not the binary representation of the binary instruction. We choose not to use the binary representation of the instruction as the definition because there is no guarantee that each instruction maps to the same binary code every time.

Executable files can be disassembled and opcode lists can be created. One approach that we considered building upon was to count the frequency of opcode appearance in the opcode list to

create weights and then sequences of opcodes can be generated and weighted for relevance and classified using machine learning techniques [3] or frequent subsequences can be generated for uses with the classifiers [4].

In general, the literature tends to agree that opcode analysis for malware detection will struggle with the classification of packed malware, there does exist some disagreement and there exists the possibility that opcodes can successfully be used to classify unknown packed malware [5].

Malware is typically built using modern programming languages but we felt that looking at the assembly instructions would provide enough information to use in malware detection and would assist us in building a universal grammar of malware. This is an open question that requires more study as our research shifted towards behavioral malware analysis.

1.2.2.4 Dynamic Analysis Features

Dynamic analysis of malware involves running malware on host systems, typically sandbox environments, and monitoring the system for changes to memory or system processes that are caused by the behaviors of the malware.

1.2.2.5 API calls

Most operating systems use an API that allows software developers easy access to system functions. Certain API calls can be used by malware authors to perform malicious functions. While individual API calls may be frequently used by malware, they are also commonly used by legitimate software and thus provide little information for accurate malware classification. However, sequences of API calls have shown promising results in the literature, and there is an available dataset that was constructed from 7107 different types of malware [6].

The Windows API has over 500 API calls and generating sequences of these for thousands of malware and benign applications makes the dataset incredibly large. To reduce the size of the dataset an approach that was considered was to categorize the API calls by generic function [7]. For example, all API calls that open files are grouped into a category, all calls that manipulate the network are another category, and so on. We theorized that instead of categorizing the calls in this way, if we count the frequency of API calls in our malware training data and creating weights, that we can generate weighted sequences of API calls that will provide more information towards detecting malicious behavior. This area still needs to be explored as the API call research was redirected towards other methods of dynamic analysis as another project had done extensive research in this area for the ARFL and we wished to avoid duplication of work.

1.2.2.6 Hooking

Since the defining characteristic of a trojan is the ability for the trojan to remain hidden, we decided to look for the methods they use to hide. All trojans that were studied during this project used some form of hooking behavior to either hide or to carry out their planned attack.

A simple explanation of a hooking event used for malicious purposes would be a trojan that implants the address to one of its own functions in the place of a function from benign software. When the benign software executes it will end up calling the malicious function in its place. From there, depending on the function, the malware has the potential to do anything.

There are multiple types of hooks and hooking behaviors and current tools that can be used to scan for hooks run into problems as the malware evolves and uses more sophisticated methods of hooking [8]. This is a problem that every malware feature will have unless a true universal grammar of malware is found.

1.2.3 Dataset selection

One of the biggest problems malware researchers have is the lack of a consensus dataset [9]. When most of the research is conducted using wildly different datasets there is no easy way to compare one method to the next. Additionally, it can take significant amounts of time to build a useful dataset from the ground up. The closest thing to a consensus dataset for malware research is the Microsoft Malware Classification Challenge (BIG 2015) dataset hosted on Kaggle [10]. The Microsoft dataset was designed as a challenge to the research community to develop a technique that could identify which family a virus comes from.

The problem of building a better dataset for malware research is a significant problem on its own. It is our opinion that there can be no consensus dataset in this area without the assistance of a large corporation or organization, many of which will be unwilling to provide enough access to their systems and data as this could present loss of proprietary knowledge, data privacy issues and creates vulnerabilities to hacking.

Malware classification based upon behavioral analysis requires a single virus to be run in an isolated environment and be monitored, recording any relevant information. This is achievable using an environment known as the Cuckoo sandbox [11]. This process must then be repeated for each additional virus that is to be added to the dataset. Most classification algorithms will require large datasets for training and testing, making this a very time intensive proposition. To have a large enough dataset to satisfy the training and testing of classification methods a correspondingly large number of live viruses would be needed. Again, this is almost impossible to do without organizational assistance. A hundred or so live viruses can be found easily on the internet but finding each additional virus file requires a significant amount of time.

The last option is to create a synthetic dataset by hand. For example, we have been exploring creating small programs that just create a hooking event in the system. However, every one of these small programs needs to present a different hooking method and even then, because they are small programs, they miss the complexity needed to properly train a classifier for detecting sophisticated malware. Alternatively, we could write synthetic viruses and run those to generate features for our dataset. This is problematic because the sophistication needed to write many different types of malware that would use many different malicious techniques is immense for a single individual and time constraints made this an unusable method.

Given these options we propose using the Microsoft dataset [10] where possible and analyzing as many live viruses we can find in the wild using the Cuckoo sandbox.

1.2.4 Malware detection implementation

In the late 1990s Jeffrey O. Kephart realized that computers were becoming more connected, due in large part to the Internet, and that computer viruses were going to be written much more frequently. He devised a blueprint for a system in computers that functioned similarly to the human immune system [12].

Kephart distilled the human immune system into four basic components: recognizing that there is a virus, eliminating the virus threat, the ability to recognize and eliminate unknown viruses and remembering how to do this when it encounters a known virus again and the ability to self-replicate and self-proliferate in order to eliminate the infection [13][14].

Kephart's computer immune system can really be thought of as a local area network immune system. If we view each computer system in an organization as a cell that can be infected by a virus, Kephart's aim is to prevent the infection from spreading to the other cells. He proposes to

do this by having a dedicated computer system set up that will be purposely infected with any new viruses so that they can be scanned and added to the antivirus software database. This machine runs multiple different types of programs that act as traps to entice the virus to infect and then the machine can capture it and add it to the database. A signature of the virus is created from the trapped code and the signature is then tested against known uninfected code to eliminate the possibility of false positive recognitions. If a virus in the database is then detected on any of the other computers in the network, a kill signal would then be sent to disconnect infected machines from the network to prevent the spread of infection.

Our system, while based on the same biologically inspired immune system, takes the view that the individual computer itself is the organism and each system has its own (possibly identical) immune system.

In its abstract form, our malware detection system uses multiple features. In the case of research currently underway we are using API call sequences, hooking in memory, and a third undecided feature. The system is modular, so at any time we can create new features and training data and plug them into the model, either expanding the number of features used or swapping out an existing feature.

Almost all malware methods proposed in the literature use only a single feature for detection. The main theory of our system is that we can gain more information about malicious behavior by using several different types of malware behavior and also use classification methods optimized for those specific features. A system set to use three features will also be running three, separate and distinct classifiers.

Performance was not a primary consideration; however, the system has been designed in a way that a large portion of the computational tasks can be run in parallel and can be offloaded to a central server or to multiple servers, combining the results at the end to reach a probabilistic determination of maliciousness.

The Cuckoo sandbox environment is currently being used to run live viruses found in the wild and storing complete memory dumps on an external hard drive for further analysis of hooks in memory. An equal number of benign software memory dumps will be taken in order to be used with test data. The API call dataset [6] is being used as an additional feature for comparison.

In order to think about a detection system based on biological methods, it becomes necessary to think about the parts of software programs in terms of cells and inflammation processes.

Thinking about software and malware at a smaller, component level scale leads to the question, what is the smallest part of software that could be used to differentiate between malware and benign software and how could a collection of that type of data be used to improve malware classification.

A method proposed in [15] attempted a somewhat similar approach using the BERT language model for classifying different types of glucose molecules. If a suitable feature set could be found for malware it might be possible to create a new language as the building block of a detection system and allow BERT to determine semantic and contextual meanings which would allow a classifier to detect malware with improved performance over traditional methods.

1.3 Research Objectives

This is exploratory research to attempt to determine if API calls are a suitable feature to create a language for the BERT model to use as a preprocessor for classification methods to improve

malware detection results. The primary objective is to explore the research area and collect evidence that would further inform future researchers.

1.4 Overview of the Thesis

In chapter 2 we review historical and recent literature that gives the background on malware detection methods and on the recent use of the BERT algorithm for areas outside of natural language processing. Chapter 3 presents our proposed approach and gives an overview of the data on which our experiments will be conducted. Chapter 4 covers our experimental results and the analysis of those results. Chapter 5 provides a discussion of our research and results and possible future directions.

Chapter 2 Background

2.1 Literature Review

We have several options to keep computer systems safe from viruses, but the only solution that provides perfect protection also provides imperfect usability, which is to isolate the computer system entirely from all other networks and users. We will lay the groundwork through historical literature showing why there are no perfect solutions, why we must rely on probabilistic results and thus why AI techniques are a logical choice for virus detection.

If we are unwilling to isolate our system from network connectivity and other users, the first choice to solve this problem would be a virus detection system that is able to detect all viruses. By constructing a simple virus, Cohen [16] was able to show that such a detection system is not possible by creating a decision structure inside the source code of a host program that infected a victim program only if it was determined that the host program was not a virus, but by executing this code the host program becomes a virus. Thus Cohen concludes that it is not possible to detect a virus by appearance alone.

Cohen further showed that detection of computer viruses by behavior is undecidable. Determining behaviors a computer program exhibits that are malicious or legitimate is dependent on the specific system. What is malicious on one system might be legitimate on another. A compiler can be considered a virus because it creates and modifies code on a system, but since the compiler must be activated by some other process, we can only recognize it as a virus by appearance, a method which Cohen previously showed was not possible. Formal proofs were presented in Cohen [17]

Further complicating the situation, Chess and White [18] showed that there are computer viruses that are undetectable by every virus detection system. The practical results of this are that even if you have a copy of such a virus it would be impossible to build a detection algorithm that detects this virus deterministically. A probabilistic detection system is the best approach we can take. The uncertainty in a probabilistic approach creates situations where we may falsely detect a benign program as a virus, or even worse, we may fail to detect the virus at all.

Having these facts established we have no choice but to accept uncertainty in our detection algorithms. Most recently this has been done using machine learning or artificial intelligence algorithms.

Moser, et al. [19] showed that static analysis of malware is insufficient as the sole method of detection by developing several obfuscation tactics that can evade malware detection systems. By manipulating the control flow or hiding the location of used data, rewriting the binary files so that no debugging information can be found, they show that there are countless methods to avoid any detection method that relies on static analysis alone.

Zhu et al. [20] use a Deep Belief Network (DBN) to detect malware on Android systems by analyzing the flow of data in applications. Control flow graphs are made and attached to a dummy main function. This novel approach requires training data generated by creating control flow graphs for thousands of benign and malicious software examples while unsupervised classification using the DBN results in 96% accuracy for detecting unknown malicious applications.

Zhong and Gu [21] explored a system that uses multiple deep learning algorithms for malware detection. Parallel K-means clustering partitions the dataset and agglomerative hierarchical

clustering is used to generate multiple cluster subtrees. Training uses a decision function to evaluate the performance of deep learning algorithms on clusters. The best performing algorithms for each cluster of the tree are determined, which therefore allows for classification using a decision tree. Their results show this method performs generally better than decision tree and SVM and found a 3 and 4 percent increase over deep learning methods.

Yaun et al. [22] proposed a two headed neural network to increase anomaly detection performance over standard machine learning techniques such as support vector machines as malware evolves over time. The performance of machine learning algorithms and various other detection schemes becomes less accurate over time, which is possibly due to malware authors developing more sophisticated methods to avoid detection by evolving the malware. Results from this work showed slight improvements detecting evolved malware and show the importance of continuing to develop more sophisticated detection methods as malware evolves.

Wadakar et al. [23] suggested that a standard support vector machine method using static analysis of Windows portable executable file data could be used to detect malware evolution. Their method classified malware into families (trojan, worm, etc.) and used a support vector machine on sliding one-year windows for viruses created during that window and used a X^2 statistic to compute the deviation of the computed value from the expected value. Each spike in X^2 computation corresponds to a significant code change within that malware family.

Finder et al. [24] collected time stamp data from API calls that were called by malware run in a virtual environment to create temporal patterns. The temporal patterns were then mined using the Temporal Probabilistic proFile [25] Using this method resulted in a 99.6% classification accuracy for detecting unknown malware and 97.65% accuracy for detecting which family malware belongs to.

Applications of BERT outside of natural language processing (NLP) are starting to be published. Ali Shah et al. [15] used BERT to classify glucose transporter families with classification results greater than 90%. Their method involved transforming glucose protein sequences into n-grams and then using BERT to generate feature vectors that were then classified using Support Vector Machines.

The ability to extend the functionality of BERT has been shown by Levine et al. [26]. By using external lexical sources, BERT was expanded to be able to add word sense to the pre-training with limited supervision. This method allows the meaning of missing words from an analyzed text to be determined.

The above research shows that using BERT could be effective in improving the classification of malware by the novel method of considering capturable behavioral characteristics of malware as a language and this work provides a starting point for showing that this could bring promising results.

Chapter 3 Proposed Approach

Our proposed approach is based on the methods in [15]. The authors used a dataset comprised of information collected from the National Center for Biotechnology that included three families of glucose transporters. Their dataset included a total of 15749 protein sequences from which they removed any protein sequence that was too similar to any other sequence, in other words, protein sequences that had a 40% similarity were removed and they used an 80:20 ratio for testing and training. Their sequences were constructed with an n-gram of size one using a single amino acid as the n-gram and all sequences were shortened to a maximum length of 510. Their research was attempting to classify families of glucose transporters.

3.1 Recurrent Neural Network

Bert [27] is an evolution of the Recurrent Neural Network (RNN) model and it simplifies the understanding of BERT if we briefly explain the basics of RNN operation.

Encoder and decoder models work by sending an item from the input data into the encoder, one item at a time, in sequence, then outputting a context vector that is then sent to the decoder which produces the output, sequentially in the same order as data was input into the encoder.

The encoding and decoding is typically done through the use of Recurrent Neural Networks (RNN). To begin the encoding and decoding process, each item from our input data is converted to a vector of floating point numbers using word embedding algorithms [28]. An initial hidden state input is created and the encoder RNN is sent the initial input vector and initial hidden state. The RNN updates the hidden state and includes this with the next input item vector. Once the input list has been exhausted, the final hidden state from the encoder, which is typically referred

to as the context, is sent to the decoder RNN, which follows the same process, in the same sequence, and outputs the decoded data.

3.2 BERT

BERT, Bideirectional Encoder Respresentations from Transformers, is a language representation model developed by Google for natural language processing [27]. BERT is an improvement over the RNN approach as it uses an attention mechanism which allows BERT to improve upon the contextual approach of the tradition RNN. Instead of the encoder sending the final hidden state to the decoder, all hidden states generated from each step of encoding are sent to the decoder. This gives BERT the ability to determine more information about each word based on all the other words in the sentence.

BERT is customizable, but is typically pre-trained with massive amounts of unlabeled text data. Once training is complete, labeled data is given to BERT, which has incredible flexibility to be fine-tuned to produce better results for the specific problem area, by allowing each individual task to use its own specific fine-tuned parameters.

A tokenizer is created from the unlabeled data using WordPiece embeddings [29]. The WordPiece method used by BERT is a greedy method that generates the vocabulary. Vocabulary is generated by scanning the provided test data and referring to the pretrained language corpus and creating a file which contains individual characters, whole words, subwords that are at the beginning of a word, subwords that are not at the beginning of the word, and special tokens that are used to tell BERT the beginning and end of sentences. The WordPiece embeddings included in the pre-trained English version of BERT include a 30,000 token library [27].

With the vocabulary and text generated, BERT is a bidirectional model that reads the entire sequence of words at a single time, instead of a single word at a time, which allows BERT to use the surrounding words to learn the meaning of a word.

A vector with the series of tokens is sent to the neural network and each token produces its own vector as an output from the neural network. The base version of BERT is a 12 input layer neural network with 768 hidden layers. There are other versions of BERT that use more layers than the base version of BERT.

The output of BERT can then be used for a variety of tasks, with the most common being NLP tasks such as sentiment classification, sentence prediction, answering questions, and language translation.

3.3 Dataset Description

One of the biggest issues in the literature is the lack of standard datasets for malware detection problems. Collecting and constructing a dataset of malware consumes a significant amount of time and many authors have no choice but to undertake this as an initial step of their research before moving on to their actual proposed contributions to the literature.

There are several significant drawbacks to everyone constructing their own malware datasets. Many people do not post their dataset for public retrieval and if they do those places are often temporary, as an example, a postdoc might host on the University website and then that host gets disabled when he or she moves on to another position. Comparing techniques on totally different data doesn’t give a true comparison of performance. For an accurate comparison the data should be identical. For these reasons we will be using a publicly available dataset of malware API calls.

We are using the API call sequences dataset hosted by IEEE (Malware API Call Dataset | IEEE DataPort (ieee-dataport.org)) [6]. This dataset was constructed using the Cuckoo Sandbox environment and contains the first 100 non-repeated consecutive API calls associated with each different piece of malware. A major factor in the decision to use this dataset for our work is the inclusion of API call sequences from benign software. There are 1,079 API call sequences from non-malware and 42,797 API call sequences from malware. There are a total of 307 individual API calls represented in the API call sequences in this dataset.

Chapter 4 Experiments and Results

4.1 Experiments

4.1.1 Data Preprocessing

API calls as presented in the dataset were mapped to integer values. These were converted back to text for language processing necessary for BERT. There are a total of 309 API calls in this dataset but after frequency analysis was performed it is shown in Table 1 that there are a few API calls that are considerably more frequent than most of the others and Table 2 shows that there are many that never appear in any of the API call sequences. The frequency distribution shown in Figure 1 illustrates that this dataset is very unbalanced. A total of 43 API calls are not represented in this dataset at all which leaves us only 264 API calls to use for language construction.

API Call	Frequency
ControlService	737275
NtSetContextThread	414282
NtMapViewOfSection	316939
GetSystemWindowsDirectoryA	264575
CoUninitialize	225345
CreateThread	192361
GetVolumePathNameW	191028
SizeofResource	188369
CreateServiceA	187091
Thread32Next	177892
NetGetJoinInformation	102525
CoGetClassObject	102151
NtQueryInformationFile	85667
LoadResource	73127
CryptAcquireContextA	53475

Table 1 Most frequent API calls in dataset

API Call	Frequency
RegEnumValueW	0
Getaddrinfo	0
CryptDecodeObjectEx	0
EncryptMessage	0
LdrLoadDll	0
InternetGetConnectedStateExA	0
RegCreateKeyExA	0
GetBestInterfaceEx	0
NtReadFile	0
CryptProtectMemory	0
FindFirstFileExA	0
NtQueryAttributesFile	0
HttpSendRequestW	0
SetFilePointerEx	0
__exception__	0

Table 2 Least frequent API calls in dataset

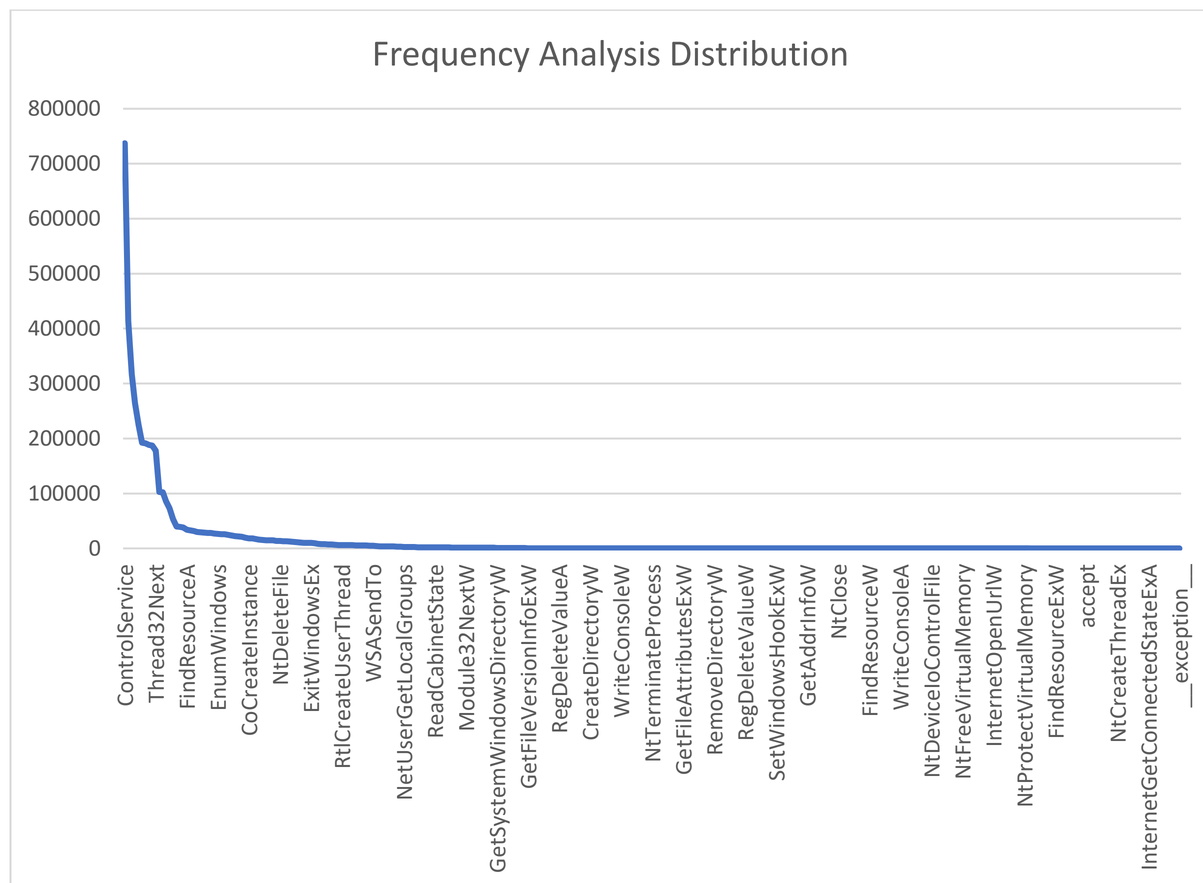

Figure 1 API call freqency analysis in dataset

4.1.2 Environmental Setup

All experiments have been run on a desktop system. This is a reasonable choice for this preliminary work but training the BERT model for alternate language use and using a large dataset will require a significantly more powerful, multi-processor system with as much RAM as possible.

Processor	**Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz 3.70 GHz**
RAM	**32.0 GB**
Operating System	**Windows 11 Pro 64-bit**

Table 3 Desktop PC Hardware Specifications

4.2 Evaluation Metrics

We will use the standard evaluation metrics used in machine learning. Precision allows us to see the proportion of correct positive identifications and recall is the fraction of actual positives that were correctly identified. The F-score gives us a single measure of our approach against our given dataset. Accuracy is the fraction of identifications that we got correct out of the total number of identifications.

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

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

$$F_1 = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Where TP is the number of true positives, FP is the number of false positives, FN is the number of false negatives and TN is the number of true negatives.

4.3 Results

The main goal of this research was to show that it would be possible to use API call sequences to generate a custom language for use with BERT as a preprocessor for malware classification and obtain positive results. Progress was made in this direction, but the proposed approach will not provide improved results without additional work.

The BERT transformer obtains better results when it is trained with massive amounts of data and malware data is just not available at the size needed to fully train BERT. BERT is fully customizable and able to be trained with any language desired, but unlike foreign languages which have massive amounts of data readily available on the Internet, the difficulty in obtaining even a fraction of malware data makes the scope of solving this problem enormous. This is a critical piece of the hypothesis, without which the hypothesis is unable to be verified.

The formatted data was stored in a csv file in which 70% was used as training data and the remaining data used for testing. The data was classified without BERT preprocessing with a Support Vector Machine using a Radial Basis Function kernel.

Using this approach, we have Precision = 0.972, Recall = 0.886, F_1 = 0.921, and Accuracy = 0.886.

We then took the same data and built a BERT tokenizer to generate a vocabulary. The API call names are not proper English words and are not included in the corpus used by BERT, thus our tokenizer only generated 271 words and 1089 partial word tokens.

The results of the tokenization process already show that this approach will not be successful in proving the desired hypothesis. The English version of BERT has a 30,000 token library and we are only able to generate 4.5% with our dataset. Therefore, it is not possible to get improved results using this approach.

When comparing the results of our API call sequence classification using Support Vector Machines with the approach from [15] the overall accuracy from their best experimental result is 97.3% for SGLT, 92.43% for GLUT, and 92.97% for SWEET, where SGLT, GLUT, and SWEET are glucose transporter families. These results did not come from the use of the same BERT model however, so their approach doesn't have a single method that returns this accuracy, but it does show that using a different pre-trained model of BERT can result in improvements in classification accuracy, thus lending support to the theory that creating a custom language for a non-NLP task is a viable option, because a custom language trained BERT model will have different classification accuracy than the pre-trained English language versions using a non-English vocabulary.

Classification target	Method	Accuracy (%)
API Sequences	SVM	88.6
GLUT transporter	BERT-Large (Uncased) & SVM	92.43
SGLT Transporter	BERT-Base (Uncased) & SVM	97.3
SWEET Transporter	BERT-Large (Uncased) & SVM	92.97

Table 4 Comparison of classification accuracy

Chapter 5 Conclusions and Future Work

This research was an experimental evaluation using techniques that work exceptionally well for natural language processing, in search of a method that might further increase the ability to detect malware. While this research did not produce significant results, there are several ways to improve the detection ability of this method and further research might show significantly better results.

An approach that could obtain better results is to consider the set of all possible API calls as its own language and training BERT to use this language. This was an initial goal of this study but the lack of a massively large open-source dataset to use as a language corpus and lack of research time to collect the data made it an infeasible addition to this work. Collecting any features from malware is a difficult and time-consuming task. Malware is not readily available in the large amounts needed to create enough diversity for classification. Once malware is located, each virus needs to be run in a sandbox environment to prevent host infection, normal system use must be simulated to attempt to trigger the virus to act, and then the desired features must be captured.

BERT performs significantly better when using massive amounts of text data for training. Using BERT for languages other than English presents no problem, with many common foreign language versions being directly available, and the more uncommon languages can manually be trained if necessary, by using massive text datasets, which are readily available or easily collected.

Using BERT for areas outside of NLP is possible if the problem data can be used to construct a language. In our case, each feature is a sequence of API calls where an individual API call is a word and a sequence is a sentence. A language can be constructed using this method, and with a corpus of data large enough BERT can be trained to detect the context of the words within sentences.

API call sequences may not be the best feature to use with this method and a study comparing the use of other malware features and behavioral characteristics, such as opcodes, n-grams or system calls to determine if another feature might be a better choice for language construction. If one wants to pursue this comparison research it would be best to have all features (API calls, opcodes, etc.) from malware captured in the virtual sandbox environment at a single time, otherwise multiple researchers pursuing this comparison would need to spend an incredible amount of time generating the dataset for each feature.

As the language constructed using API calls will contain a small vocabulary, the API call sequences should be increased in length in order to compensate for the lack of vocabulary and help to increase the ability of BERT to capture the context or our language use. The dataset used in this research stopped the sequence length at the first one hundred API calls. This has several drawbacks for our use. This size may not capture the entire behavior of the malware, and this could be why several API calls included in the dataset actually never appeared in the sequences. Without the non-truncated sequence data, it is not possible to say for certain, but it is possible that all API calls that were used in the integer mapping were at one point reflected in the data but disappeared in the truncated sequences.

While the theory of generating a language out of malware features has not been disproven, this research has shown that using API call sequences is not sufficient to generate such a language to

obtain results accurate enough for malware detection. The project scope should be large enough to fulfill the requirements for a PhD program or possibly for a team of students working at the Master's level.

Bibliography

[1] C. Boeckx and E. Leivada, "On the particulars of Universal Grammar: Implications for acquisition," *Language Sciences*, vol. 46, no. PB, pp. 189–198, 2014, doi: 10.1016/j.langsci.2014.03.004.

[2] R. Zak, E. Raff, and C. Nicholas, "What can N-grams learn for malware detection?," *Proceedings of the 2017 12th International Conference on Malicious and Unwanted Software, MALWARE 2017*, vol. 2018-Janua, pp. 109–118, 2018, doi: 10.1109/MALWARE.2017.8323963.

[3] I. Santos, F. Brezo, X. Ugarte-Pedrero, and P. G. Bringas, "Opcode sequences as representation of executables for data-mining-based unknown malware detection," *Information Sciences*, vol. 231, pp. 64–82, 2013, doi: 10.1016/j.ins.2011.08.020.

[4] H. Darabian, A. Dehghantanha, S. Hashemi, S. Homayoun, and K. K. R. Choo, "An opcode-based technique for polymorphic Internet of Things malware detection," *Concurrency Computation* , no. February, 2019, doi: 10.1002/cpe.5173.

[5] A. Shabtai, R. Moskovitch, C. Feher, S. Dolev, and Y. Elovici, "Detecting unknown malicious code by applying classification techniques on OpCode patterns," *Security Informatics*, vol. 1, no. 1, pp. 1–22, 2012, doi: 10.1186/2190-8532-1-1.

[6] F. O. Catak and A. F. Yazi, "A BENCHMARK API CALL DATASET FOR WINDOWS PE MALWARE CLASSIFICATION A PREPRINT," 2019. Accessed: Aug. 11, 2019. [Online]. Available: https://github.com/ocatak/malware_api_class

[7] S. Gupta, H. Sharma, and S. Kaur, "Malware Characterization Using Windows API Call Sequences," *Journal of Cyber Security and Mobility*, vol. 7, pp. 363–378, 2018, doi: 10.13052/jcsm2245-1439.741.

[8] Z. Liang, H. Yin, and D. Song, "HookFinder: Identifying and understanding malware hooking behaviors," *Proceedings of the Network and Distributed System Security Symposium*, p. 41, 2008.

[9] Z. Bazrafshan, H. Hashemi, S. M. H. Fard, and A. Hamzeh, "A survey on heuristic malware detection techniques," *IKT 2013 - 2013 5th Conference on Information and Knowledge Technology*, pp. 113–120, 2013, doi: 10.1109/IKT.2013.6620049.

[10] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, M. Ahmadi, and M. 2 Crowdstrike, "Microsoft Malware Classification Challenge." 2018. doi: 10.1145/2857705.2857713.

[11] "Cuckoo {Sandbox} - {Automated} {Malware} {Analysis}." Accessed: May 20, 2020. [Online]. Available: https://cuckoosandbox.org/

[12] J. O. Kephart, G. B. Sorkin, D. M. Chess, and S. R. White, "Fighting Computer Viruses," *Scientific American*, vol. 277, no. 5, pp. 88–93, 1997. doi: 10.2307/24996006.

[13] J. O. Kephart, "A Biologically Inspired Immune System for Computers," in *4th International Workshop on the Synthesis and Simulation of Living Systems*, 1994, pp. 130–139.

[14] J. O. Kephart, G. B. Sorkin, and M. Swimmer, "Immune system for cyberspace," *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, vol. 1, pp. 879–884, 1997.

[15] S. M. Ali Shah, S. W. Taju, Q. T. Ho, T. T. D. Nguyen, and Y. Y. Ou, “GT-Finder: Classify the family of glucose transporters with pre-trained BERT language models,” *Computers in Biology and Medicine*, vol. 131, no. February, p. 104259, 2021, doi: 10.1016/j.compbiomed.2021.104259.

[16] F. Cohen, “Computer viruses,” *Computers & Security*, vol. 6, no. 1, pp. 22–35, Feb. 1987, doi: 10.1016/0167-4048(87)90122-2.

[17] F. Cohen, “Computational aspects of computer viruses,” *Computers & Security*, vol. 8, no. 4, pp. 297–298, Jun. 1989, doi: 10.1016/0167-4048(89)90089-8.

[18] D. M. Chess and S. R. White, “An Undetectable Computer Virus,” *Proceedings of Virus Bulletin Conference*, vol. 5, no. 4, pp. 409–422, 2000, [Online]. Available: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:An+Undetectable+Computer+Virus#0

[19] A. Moser, C. Kruegel, and E. Kirda, “Limits of static analysis for malware detection,” in *Proceedings - Annual Computer Security Applications Conference, ACSAC*, 2007, pp. 421–430. doi: 10.1109/ACSAC.2007.21.

[20] D. Zhu, H. Jin, Y. Yang, D. Wu, and W. Chen, “DeepFlow: Deep learning-based malware detection by mining Android application for abnormal usage of sensitive data,” *Proceedings - IEEE Symposium on Computers and Communications*, pp. 438–443, 2017, doi: 10.1109/ISCC.2017.8024568.

[21] W. Zhong and F. Gu, “A multi-level deep learning system for malware detection,” *Expert Systems with Applications*, vol. 133, pp. 151–162, 2019, doi: 10.1016/j.eswa.2019.04.064.

[22] C. Yuan, J. Cai, D. Tian, R. Ma, X. Jia, and W. Liu, “Towards time evolved malware identification using two-head neural network,” *Journal of Information Security and Applications*, vol. 65, no. January, p. 103098, 2022, doi: 10.1016/j.jisa.2021.103098.

[23] M. Wadkar, F. di Troia, and M. Stamp, “Detecting malware evolution using support vector machines,” *Expert Systems with Applications*, vol. 143, no. October, 2020, doi: 10.1016/j.eswa.2019.113022.

[24] I. Finder, E. Sheetrit, and N. Nissim, “Time-interval temporal patterns can beat and explain the malware,” *Knowledge-Based Systems*, vol. 241, p. 108266, 2022, doi: 10.1016/j.knosys.2022.108266.

[25] E. Sheetrit, D. Klimov, N. Nissim, and Y. Shahar, “Temporal probabilistic profiles for sepsis prediction in the ICU,” in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019, pp. 2961–2969. doi: 10.1145/3292500.3330747.

[26] Y. Levine *et al.*, “SenseBERT: Driving Some Sense into BERT,” in *58th Annual Meeting ofthe Association for Computational Linguistics*, 2020, pp. 4656–4667. doi: 10.18653/v1/2020.acl-main.423.

[27] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, vol. 1, no. Mlm, pp. 4171–4186, 2019.

[28] “Word embeddings | Text | TensorFlow.” https://www.tensorflow.org/text/guide/word_embeddings (accessed Jun. 24, 2022).

[29] Y. Wu *et al.*, “Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation,” pp. 1–23, 2016, [Online]. Available: http://arxiv.org/abs/1609.08144