ATHABASCA UNIVERSITY

ANALYSIS OF PARALLEL PRE-PROCESSING OF MALWARE DATA FOR MACHINE LEARNING IN PYTHON

BY

NELS LARSEN

A THESIS/DISSERTATION

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE IN INFORMATION SYSTEMS

SCHOOL OF COMPUTING AND INFORMATION SYSTEMS

ATHABASCA, ALBERTA

JULY 2023

Athabasca University
Faculty of Graduate Studies

Approval of Thesis

The undersigned certify that they have read the thesis entitled

ANALYSIS OF PARALLEL PRE-PROCESSING OF MALWARE DATA FOR MACHINE LEARNING IN PYTHON

Submitted by

Nels Larsen

In partial fulfillment of the requirements for the degree of

Master of Science in Information Systems

The thesis examination committee certifies that the thesis and the oral examination is approved

Supervisor:
Dr. Qing Tan
Athabasca University

Committee Member:
Dr. Harris Wang
Athabasca University

External Examiner:
Dr. Ali Dewan
Athabasca University

August 9, 2023

Abstract

Machine learning-driven malware detection and identification helps protect computers. This thesis initially aimed to develop a machine-learning malware detection solution by utilizing intermediate languages so that malware studies can ignore specific hardware and operating systems while effectively using machine learning to detect and classify malware. Reduction of instruction set size lowers computing costs when using machine learning. Malware must be processed for machine learning before it can be studied. The evolution of tools and diversity of intermediate languages requires that binaries must be processed in a way that order to explore opportunities to optimize based on diverse types of opcode generation from binaries.

Since Python programming language is a popular choice for machine learning and much more machine-learning computation has taken place on the cloud, this research aimed to improve existing Python code to detect malware on the cloud platform. However, data preparation is an essential step for all machine-learning processing. This research focused on better malware processing by examining previous research's code. Optimization recommendations of this research can be generalized to applications outside of malware research but are focused on what a malware database would require for processing and would be unsuitable for broad applications without limitations. The first part of the research identified how the data preparation can delay research and incur costs when using cloud infrastructure through the analysis of measurement of performance in different environments.

The second part of the thesis research has created a dictionary system that includes multi-process and coroutines to address the idea of reducing computing costs. This system functions as a single dictionary to optimize time to gain efficiency and save the machine learning pre-processing computation costs. Understanding how to alter parallel processing allowed for the reduction in costs and time for both academic research and commercial practices. The implementation of this dictionary system addresses how to process datasets specific to malware research. The dictionary system demonstrates further knowledge in dealing with parallel processing associated with the high processing requirements and high size of this dataset.

A standard cost model for cloud infrastructure is to charge by time. This thesis research shows that a single-threaded dictionary takes more time but fewer resources than a parallel processing solution used with dictionaries. As Python's internal multithreading mechanisms can slow execution time, this research found that some wasteful types of concurrent processing can save time. Because virtual machine (VM) costs are primarily time-based, this research has proven that a multi-process dictionary can save time for processing machine-learning data. However, the time saving is not linear with increasing a VM's resources.

Keywords: Python Optimization, Effective Cloud Computing, Virtual Machines, Concurrent Processing Dictionary

Table of Contents

Abstract ... 2
Table of Contents ... 4
List of Tables ... 6
List of Figures and Illustrations ... 7
List of Abbreviations ... 8
Chapter 1 Introduction ... 9
1.1 Background and Motivation ... 11
1.2 Research Purposes ... 13
1.3 Research Objectives and Research Problem ... 13
1.4 Research Findings and Contributions ... 17
1.5 Research Limitations and Delimitations ... 18
1.6 Definition of Terms ... 19
Chapter 2. Literature Review ... 21
2.1 Introduction to Malware ... 21
2.2 Datasets ... 24
2.3 Malware Detection ... 25
2.3.1 Static Analysis ... 26
2.3.2 Dynamic Analysis ... 32
2.3.3 Image Analysis ... 34
2.4 Machine Learning ... 35
2.5 Intermediate Languages ... 39
2.6 Evasion Techniques ... 40
2.7 Pre-Processing methods ... 41
Chapter 3. Malware Detection and Classification ... 43
3.1 Opcode Techniques for Detection and Classification ... 43
3.2 Augmentation and Collection of Data Inputs ... 44
3.3 Limitations of the current pre-processing ... 47
Chapter 4 Methodology ... 48
4.1. Machine-learning Data Processing Foundations ... 48
4.1.1 Pre-Processing Specifics for Malware Machine Learning ... 48
4.1.2 Replication Process and Results ... 50

4.1.3 Code Changes 51
4.1.4 Resolution of topic speed issues 52
4.1.5 Standardization for Future Machine-Learning Research Involving Malware . 53
4.1.6 Task Selection 54
4.2 Systems Measurement 54
4.2.1 Goals of Environmental Measurement 54
4.2.2 Python Implementations of parallel processing 55
4.2.3 Evaluation Tools 60
4.2.4 Description of Task 61
4.2.5 Research Design 68
Chapter 5. Environment Measurements and Evaluation 76
5.1 Results from Testing Environments 76
5.2 CPU Comparison 81
5.2.1 GCP Compute Variable Speed 82
Chapter 6. Parallel Processing Results and Recommendations 84
6.1 Implementation of Parallel Processing Method's Code 85
6.2 Data Size Effects on Execution Times 86
6.3 Explanation of the mk13 Algorithm 88
6.4 Time Results for Pre-Processing Data 91
6.5 Hard Drive Effects on parallel pre-processing 94
Chapter 7. Conclusion 97
References 100

List of Tables

Table 1 WinPE File Format 27
Table 2 Dictionary Creation Times Varied by Queue Size 68
Table 3 Dataset Characteristics 68
Table 4 HDParm Hard Drive Speed Results 78
Table 5 Fio Read Speed Results 79
Table 6 Hard drive read comparison 81
Table 7 CPU Speeds 82
Table 8 Virtual Server Dataset Size Runtimes 87
Table 9 Multi-processing Queue Message Size Comparison 88
Table 10 Single Vs Multi-Thread Execution Times 92
Table 11 Multi-Process Vs mk13 Execution Times 94

List of Figures and Illustrations

Figure 1 Generalized Machine Learning Malware Analysis Workflow 16
Figure 2 Citation Timeline 17
Figure 3 Assembly Example 29
Figure 4 Extracted F-Droid Archive 30
Figure 5 Dictionary Update Using Full Dictionary Locking 57
Figure 6 Dictionary Update Using Single Value Locking 58
Figure 7 File Header of file 125y4VsArzkCNOGZfu6o.bytes.. 62
Figure 8 Scalene Output for a control run for 20 files. 62
Figure 9 Pseudocode for entire execution for multi-processing without a queue............ 66
Figure 10 Output of lstopo command for first system 73
Figure 11 Output of lstopo command for virtual system 74
Figure 12 Execution Times Using Slower Hard Drives 95
Figure 13 Execution Times Using Faster Hard Drives 96

List of Abbreviations

Abbreviations	Full Meaning
MAIL	Malware Analysis Intermediate Language
QEMU	Quick Emulator – This technology allows for virtualization
GIL	Global Interpreter Lock
GCP	Google Cloud Platform

Chapter 1 Introduction

Malware detection, or commonly Anti-virus detection, ideally prevents business and personal loss on computer systems. Malware research detection has changed over the last couple of decades. The research requires improvement as malware authors have adapted to malware research to improve their methods of compromising systems. Malware production has become a business. Ransomware-focused criminals have established service desks to help victims to make Bitcoin payments. These service desks will educate the ransomware victims on how to buy cryptocurrency to pay the malware authors or sponsors of the malware.

Malware authors can pursue financial and non-financial goals, although non-financial goals can be preconditions for making more money later. Compromising a computer system can allow an attacker to restrict control of the system, gain access to the system's data or cause detriment to the owners of the system. An attacker's goals can shift as a malware campaign continues. When a malware author threatens a business or individual, the response to the author is not consistent. Many government agencies discourage paying malware authors and stress that a backup and recovery plan is safer than getting financially involved with a cyber threat actor [1].

Ransomware is a popular malicious method of extorting currency from companies and individuals. Ransomware will lock the users of a computer out of the system. Using ransomware, criminals can demand money to restore access to the system. Refusal to pay the attacker can result in data transfer to the attacker. The attacker may transfer the data regardless of the ransom payment status. The data can be sold to other criminals, especially for website compromises to social media platforms. Criminals monetize this information by making it available to perform future social engineering attacks on the current victims. Depending on the type of stolen data, criminals can use the information to steal from banks, credit cards and trade illegally on the stock market. Threatening to release the data is another technique that ransomware authors can use. The DC police were victims of a ransomware attack [2]. The criminals threatened to release data about undercover police and other police matters should the ransom not be paid.

This thesis research initially targeted qualification of the effects of instruction set size in intermediate languages. This target requires information synthesis as the first step. Current intermediate language papers provide solutions using standard machine-learning techniques. For example, many papers have successfully detected classifying malware using random forest algorithms and distributed neural networks (DNN). However, the current works don't significantly overlap while using the same intermediate language or describing intermediate languages that are still supported. To be able to process immediate languages and the original methods of analysis the data must be able to process from the malware executables to organized data in a reasonable timeframe. This research identified the pre-processing speed as a core problem to implementation and switched focus from processing intermediate languages to solving issues with pre-processing intermediate languages and binary input. This research focuses on processing this data specifically and does not attempt to provide a general best form for using Python dictionaries.

Resources needed for machine learning can be demanding. Large datasets used to perform training for machine learning can take hours or days. The shift to using cloud resources encourages machine learning practitioners to use powerful virtual machines, which have high costs that change based on the time the machine is operational. The second part of this thesis research shows analyzes the data processing execution time for this dataset. The processing can be improved by better hardware, and this research provides specific recommendations for hardware improvement. This research demonstrates how a dictionary that accepts multiple concurrent inputs can improve the time taken for data processing tasks. Standardization of the time taken for a task to run can allow for better budgeting decisions and encourages documentation and process changes that allows future researchers better outcomes.

Better outcomes can be achieved by using available resources. This paper discusses that machines needed for machine learning tend to be powerful. To fully use these resources, code must employ methods to run tasks in parallel. Tasks run in parallel or parallel processing can be achieved in Python using multi-processing, multi-threading or coroutines. The parallel processing in Python is discussed further in section 5.2. The algorithm that was written as part of this research was to improve the speed of pre-processing, employing a combination of coroutines and multi-processing.

1.1 Background and Motivation

The initial tasks of this thesis research are to explore and qualify the effects of instruction set size in intermediate languages through literature review and hands-on implementation of machine learning algorithms in malware detection to understand malware algorithms and dataset usage. It strives to use intermediate languages and machine learning techniques, such as random forest and DNN, to detect and classify malware. However, with the progression of the research, there are limitations and barriers to using intermediate language and machine-learning algorithms for malware detection during the first part of the research implementation. We identified the computational issues in machine-learning processing, which this research can contribute towards.

As machines gain additional capabilities, users will want to use CPUs and threads to push computing limits. Having data structures capable of responding to concurrent requests allows larger datasets to be processed. Dictionaries are a standard method to organize datasets in many machine learning applications, especially for frequency analysis. Python built-in data structures are considered thread-safe. However, the overhead for thread safety poses a challenge to implementing concurrent processing for data processing with standardized timing outcomes.

Resources needed for machine learning can be demanding. Performing the data processing required for datasets needed for training can take hours or days. The shift to cloud resources encourages machine-learning practitioners to use powerful, expensive virtual machines. Many cloud providers charge clients based on rate multiplied by usage—the rate increases as the number of cores used, memory available, and increased hard drive size. The usage based on the time the virtual machine is powered on is standard. Machine-learning activities can be broken into parts that utilize several virtual machines but would incur setup costs. A professional using several virtual machines would need to transfer datasets and may need to re-initialize environments needed for machine learning.

A paper produced for the Microsoft Big 2015 challenge uses a huge dataset the researchers stated could complete in two days. This research followed the paper and tried to reproduce the results. As part of this process, the hardware was purchased that could meet the requirements of the task. In 2015 a Google compute instance had a speed of 2.6 GHz [3]. The purchased hardware could not perform the task of data processing in an equal amount of time. The execution of the Python code in this paper was investigated further to achieve the results of the original topic.

1.2 Research Purposes

In this thesis research, two focuses of research have been completed. The first part of the research is to better understand the pre-processing of the malware data that could be used for intermediate languages and machine-learning algorithms in malware detection. The second part of the research focuses on the computational issue of data pre-processing of the machine-learning algorithm. Optimization of computation time will allow future research to start closer to changing the machine-learning process. Data preparation for machine learning requires that data sources are in a form suitable for processing. Dictionaries can be used to organize information when moving from raw data to processed data. However, the powerful computing resources used for machine learning will only be used during data preprocessing if multi-processing, multi-threading or coroutines are used. This research shows the feasibility of a Python dictionary that can handle multiple concurrent inputs. Dictionaries can be used to organize information when moving from raw data to processed data. However, the powerful computing resources used for machine-learning resources will only be used in data pre-processing if a parallel computing method such as multi-processing, multi-threading or coroutines are used. This research shows the feasibility of a Python dictionary that can handle multiple concurrent inputs.

1.3 Research Objectives and Research Problem

Through the first part of the research, we find the limitations to implementing malware detection further using an intermediate language instruction set when we use the machine-learning technique. The computational issue observed is part of the critical path to using machine-learning techniques for malware detection.

In the second part of the research, we focus on improving computation efficiency to enhance the Machine-learning algorithm processing. The speed of Python execution continues to be a topic for improvement. Guido van Rossum, the creator of Python, continues to focus on improving the speed of Python [4]. As a language that requires a byte-code interpreter to improve the speed can, there are limited options to improve execution time in the language. Many features are missing in this high-level language that would allow programmers to perform more optimizations. For example, many of the implementations of machine-learning libraries and data structures in Python are written in C. The Python dictionary is implemented as a hash table [5]. The Python language does not have a hash table implementation. There is no way to create pointers for a hash tables in Python. The lack of language structures limits the possibilities for optimizing Python code.

Building a dictionary or changing the current dictionary will yield poor results with the data structures available in Python. This research focuses on how to load data into the dictionaries better and explore how the current dictionary could be adapted to simultaneously accept multiple inputs. The code written during this research demonstrates the results of a system that allows multiple inputs to a dictionary-like object. The new code functions as a dictionary but uses parallel processing and smaller dictionaries to increase execution time in some environments. Analysis of the underlying available parallel processing methods across different hardware was required to show that this problem is complex and that Python code does not react deterministically when the hardware is changed.

This research implemented a dictionary that can accept parallel input, referred to as mk13. This implementation dictionary was written entirely in Python and attempted to minimize calls to underlying C or C++ functions. These calls can not be eliminated as they are required for system activities like reading a file. The algorithm is novel as it targets the malware dataset used in the original topic and attempts to optimize execution time at the cost of using more CPU resources.

As Python uses byte-code translation, many efforts have been made to optimize machine-learning libraries. Many machine-learning libraries have been written in a compiled language such as C or C++ that interface with Python. The C and C++ are large portions of popular machine-learning libraries code such as PyTorch, NumPy and TensorFlow use compiled languages such as C and C++, representing 51.3%, 36.3% and 63.1% of the code repositories, respectively [6], [7], [8]. Using libraries causes a transfer of data structures between the Python byte-code and the libraries. This transfer penalty can also occur when Python internally transfers data structures to C. In situations where the data structures must remain whole, this can be costly for execution time.

Data preparation for machine learning requires that data sources are in a form suitable for processing. For example, dictionaries can be used to organize information when moving from raw data to processed data. However, the powerful computing resources typical for machine learning were not utilized during the original topic. This research uses previously unused resources to expand on the data preparation topic.

Challenges running older code from previous papers are primarily due to age and hardware. Software becomes more challenging to obtain and execute. Python 2 support has ended. The Python community has been diligent in cleaning older libraries, making it difficult to duplicate software. Additionally, the Python repository requires encrypted connections, which forces installation outside established Python toolchains.

The original work from the BIG 2015 winners [9] was duplicated as part of this research proposal. Finding the older libraries and compiling from the source was necessary because the encryption needed to access the repositories was not in the required Python version. In addition, some of the older versions were unavailable, and implementation forced minor version upgrades, which caused issues with prerequisite library versions. The literature review found that standard modern research follows a consistent flow during malware analysis shown in Figure 1. As issues became clearer while upgrading the code to modern standards, it was observed that this research needed to shift from the target of comparing the intermediate languages to how to load information into data structures better so future research on this topic could flourish.

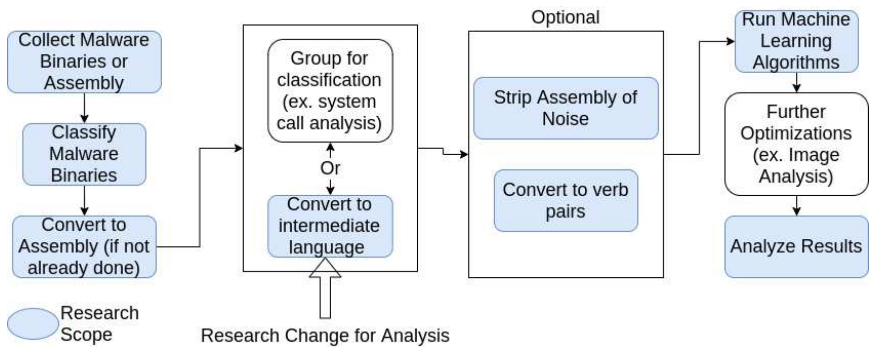

Collect Malware Binaries or Assembly
Classify Malware Binaries
Convert to Assembly (if not already done)
Research Scope
Group for classification (ex. system call analysis)
Or
Convert to intermediate language
Research Change for Analysis
Optional
Strip Assembly of Noise
Convert to verb pairs
Run Machine Learning Algorithms
Further Optimizations (ex. Image Analysis)
Analyze Results

Figure 1 *Generalized Machine-learning Malware Analysis Workflow*

The research review used primarily peer-reviewed sources and conference papers. Twenty-eight percent of the sources were journal articles, and fifty percent were conference papers. Of the 49 sources used, 43 were published after 2015, with the earliest and latest publication dates of 2009 and 2021. The years and papers used in the literary review are shown in Figure 2. Removing the papers with zero citations and the two with hundreds of citations, the average number of citations in the literature review was nineteen. The popular tags for the literature review are feature extraction, support vector machines, training, static analysis, variations on machine learning and security.

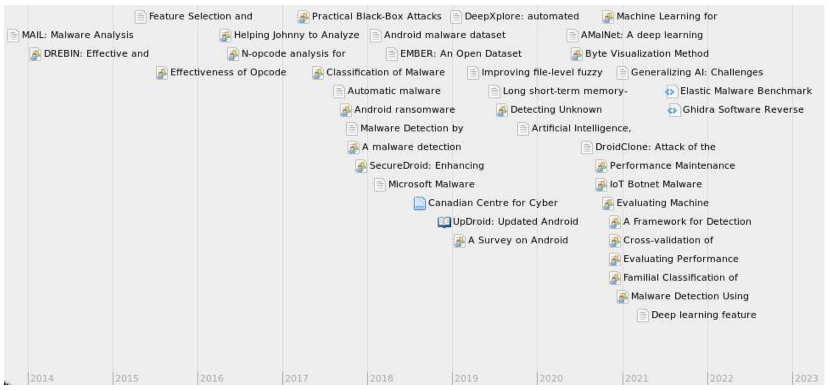


Figure 2 *Citation Timeline*

1.4 Research Findings and Contributions

By conducting the literary review, it became understood that the accuracy of known algorithms would also change as the size changes. Due to some common machine-learning issues, such as overfitting, it was expected that the size of the intermediate language might not have a direct correlation to accuracy. Smaller sizes of instruction sets may be more accurate sets, but tiny instruction sets may be the least accurate. The efficiency of the training and implementation of the algorithm also was measured. Some machine-learning techniques, such as DNN, require massive computational power. Lowering the cost for implementation with no loss in accuracy would be beneficial. There is a trend in more recent research to strip and alter the input. Altering input is done to raise efficiency, raise accuracy and lower false positives. This step is based on researcher selection and requires the researcher's time and knowledge to perform. When using an intermediate language, the file input must be processed into an existing structure, but the knowledge and effort decreases due to the framework an intermediate language provides. As the instruction set size falls, opcodes that benefit and do not benefit the process will be reduced. Therefore, the need for the researcher to make decisions or run machine learning to eliminate opcodes will be lowered. Research that uses intermediate languages can use common structures and patterns regardless of the operating system. Intermediate languages will convert malware from the Windows operating system to the same code as the Android operating system. Whatever the target operating system for the malware, the test files must be processed into a usable form in a reasonable time for the algorithm to be tested and improved. The results are expected to be platform-independent, although any research should track the target operating system for the malware.

Knowing this research is important, this research focuses on allowing researchers to process this data in a timely manner that makes future research possible. The testing of multiple intermediate languages was necessary. Translating from hex to another input would be provided by the intermediate language. With slow execution times for data organization, it would be difficult to complete the following steps of machine learning analysis. Counting occurrences of artifacts is essential to the original topic and future research. Without an increase in the pre-processing research into intermediate languages could not proceed in a reasonable manner.

1.5 Research Limitations and Delimitations

By conducting the first part of the research, we understood that the accuracy of known algorithms would also change as the size changes. Due to some common machine learning issues, such as overfitting, it was expected that the size of the intermediate language might not have a direct correlation to accuracy. Smaller sizes of instruction sets may be more accurate sets, but tiny instruction sets may be the least accurate. The efficiency of the training and implementation of the algorithm also was measured. Some machine learning techniques, such as DNN, require massive computational power. Lowering the cost for implementation with no loss in accuracy would be beneficial. There is a trend in more recent research to strip and alter the Input. Altering input is done to raise efficiency, raise accuracy and lower false positives. This step is based on researcher selection and requires the researcher's time and knowledge to perform. When using an intermediate language, the input must still be processed to fit into an existing methodology, but the effort required will decrease. As the instruction set size falls, opcodes that benefit and do not benefit the process will be reduced. Therefore, the need for the researcher to make decisions or run machine learning to eliminate opcodes will be lower. Using multiple operating systems is one goal of this research. Malware from the Windows and Android operating system will convert to the same output through conversion to an intermediate language. Whatever the target operating system for the malware, the test files must be processed into a usable form in a reasonable time for the algorithm to be tested and improved. The results are expected to be platform-independent, although any research should track the target operating system for the malware.

1.6 Definition of Terms

The “*Global Interpreter Lock,”* called the GIL, is the mechanism the CPython interpreter uses to ensure that only one thread executes Python byte code at a time. The GIL simplifies the CPython implementation by making the object model (including critical built-in types such as “dict”) safe against concurrent access [10].

Parallel Processing is a method that an application can compute or read/write data in an application. Specifically, to this research and Python, multi-processing is an application's ability to use different processors for parallel tasks. Multi-Threading in Python refers to an application's ability to use different threads on the same processor to perform tasks in parallel.

A library is a compiled piece of code that is used. Python can have expanded functionality using packages such as NumPy. The lack of distinction between packages and libraries emphasizes that this research focuses away from operating system-level calls. This emphasis looked at the Python code that makes calls to action. The system library calls are assumed to be consistent across different operating systems and are not tracked. This research focuses on discovery and optimization by assuming that the underlying call to the operating system could differ but would not change the research outcome. *Pickling* is a storage format used to write data to the hard drive, which involves some processing as it is not a raw data format. It is reasonable to conclude that a package using a system call would have a consistent runtime using the same hardware independent of the operating system.

Pickling is **the process whereby a Python object hierarchy is converted into a byte stream**, and "unpickling" is the inverse operation whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy.

Chapter 2. Literature Review

In this chapter, we present the literature review focused on the goals of processing byte-code into intermediate languages. The research outcome was modified as stabilization of the process to produce the intermediate languages became necessary. This research outcome supports future research goals of measuring the difference between the 4-gram and intermediate language processing of malware.

2.1 Introduction to Malware

A computer application or script that performs actions against the best interest of computer users and computer systems is known as malicious software. Applications may be classified as malicious differently by different professionals. For example, software that bypasses security controls and forces a computer to stop functioning is malicious software. A baseline of what is acceptable for software must be understood to detect and classify malware accurately.

A security researcher created Mimikatz to explore Windows security. Many anti-virus solutions flag Mimikatz as malicious. Microsoft's anti-malware solution Windows Defender will remove the software immediately [11]. Microsoft's code hosting platform GitHub hosts the code for Mimikatz (https://github.com/gentilkiwi/mimikatz). The repository asks that the users "please respect its philosophy and not use it for bad things!" Starting as a learning tool for the author Mimikatz is used to increase the security communities' knowledge of Windows operating system internals. Some cyber-attacks use Mimikatz as part of post-exploitation. Security tokens and passwords harvested with Mimikatz can be used for creating better footholds in an organization or exploiting a new target. The difference in usage is about intent and why the word virus is used instead of malware. The defender community has adopted the stance that malware or malicious applications are independent of intent.

As malware became increasingly widespread, software was written to stop the malware from functioning as intended. The owner of an anti-malware solution would have an employee classify an application as malware. After classification, key identifiers are extracted from the application. These identifiers would be added to a virus definition file and distributed to clients. This process has several gaps to overcome for anti-malware solutions. Before the process can start, the malicious software must be obtained for analysis. The malicious software should be analyzed professionally, not through user submission or voting. Identifiers that would distinguish malicious software from software that is not malicious must be recorded for distribution. Distributed anti-malware systems can rely on user submissions who have already run the malware to collect and determine if something is malicious. Vendors of anti-malware solutions have agents on the customer's computer submitting potential malware for analysis by the anti-malware staff.

A file hash or part of a file is a common way to detect malware. A file hash is a cryptographic output of an input file that will appear random of a set of characters, varying by length depending on the algorithm used. An MD5 algorithm was used in early implementations of malware detection and is still commonly used in malware databases and anti-malware products. Calculating an MD5 of a file is simple, making it attractive to anti-malware implementers and untrained users. However, the increases in the computational power caused the MD5 to fall out of favour. When two hash outputs of two files match, it is called a collision. The birthday paradox illustrated that finding a hash collision between two files is possible when an unguided brute force method fails.

Implementation of the birthday paradox showed that MD5 hashes of two binary files could be reasonably created within the currently available computing power. Malware authors could match MD5 hashes of known good software and fool anti-malware software that relied on hashes. Applications could exist that bypass anti-malware checking because of a known good hash. Adversely, creating malicious applications to damage the reputation of a hash could cause anti-malware mechanisms to block good software. Anti-malware implementers moved on to more robust hash algorithms such as SHA1 and SHA256. The increased complexity of hash algorithms increased security and computing requirements. Some anti-malware implementations are still setting default limits on scan time. An attacker can use huge files to bypass some anti-malware products. Malware detection incorporated file size, attributes and metadata to increase detection. Malware variants defeat traditional cryptographic hash signatures as a change in a single bit of the file produces a very different hash result [12]. Efforts have been made to overcome this limitation through fuzzy hashing research. Fuzzy hashing is a technique to create hashes equal for files that are the same and significantly similar. A driving factor for using hashes is that it is easy to implement, relativity cheaper in computational power, and easily reproduced and documented clearly. Detecting malware using this method is still used, but additional analysis techniques have been developed.

Executables and some classifications of malware must communicate to the user. A simple malware detection method is checking an executable for a specific string. In the case of ransomware, checking an executable if the strings “bitcoin” or “must pay” exist is an excellent malicious indicator. Malware authors can encrypt or obfuscate their strings to avoid detection. If they use a specific network server with a named domain name, it would be advantageous to attackers that defenders can show strings with simple efforts. Natural Language Processing (NLP) has been used to detect malware through string analysis. NLP has been used on emails to indicate that a message was a phishing attempt or contained malware. More recently, research has targeted executables with NLP with successful results [13]. Researchers could use NLP on unknown extracted strings from an executable to detect unseen malware.

2.2 Datasets

Research requires testing data, and machine learning requires training data. Malware research datasets are now available because of efforts made by academia and industry. VirusTotal is a commercial entity that warehouses viruses. The website (https://www.virustotal.com/) offers to check your file or hash at no financial cost. VirusTotal will keep submitted files and hashes as their property to build their database. Users can also search for viruses using names and other information. VirusTotal has an academic program that researchers have utilized [14]. For-profit organizations can download viruses for a fee. The Ember dataset was created to aid in research. The associated paper cited three challenges for creating malware datasets: legal restrictions, labelling challenges, and security liability and precautions [6]. Copyright issues with malware are seldom an issue. Datasets for misclassified applications and applications that are not malware, sometimes called goodware, have potential copyright issues. For example, testing the strength of a malware detection algorithm requires the dataset to have malware and goodware to determine accuracy during testing. Obtaining goodware becomes an issue as goodware can be subject to copyrighting and challenging to obtain without purchase. With the availability of goodware being an issue, unbalanced datasets may be a problem for researchers. Researchers have found that an imbalanced dataset can shift the accuracy of machine-learning algorithms. A random forest implementation was recreated from previous works, and researchers found that an imbalanced dataset could change the accuracy from 90.38 to 98.94 [7]. While an increase in accuracy is good, the change may indicate overfitting of the data. If overfitting occurs, the machine learning performance will suffer as the real-world malware shifts away from the malware found in the training dataset.

Finding maintained datasets and maintaining datasets is an issue for researchers. The Android Malware Genome Project collected Android malware and goodware. Many researchers have used it are part of their datasets [8, 14, 15, 16]. The genome project managed access and required potential users to prove their intentions with the files. For example, the maintainers required proof of association with an academic institution. The maintainers stopped distributing the dataset in 2015, stating that limited resources were an issue [11]. The project has been moved to open cloud storage [17], but not officially by the original maintainers. The Ember dataset is published on GitHub without download protections and has been updated with changes as recently as April 7, 2021[18].

2.3 Malware Detection

The current body of research has separated malware detection and classification into three different techniques. The static analysis focuses on determining if an executable is malicious without running the executable. Not executing malware is the safest and arguably simplest form of malware analysis. Without the need to execute malware, there is less need for sandboxing of the malware and less risk to the researcher’s equipment. A sandbox contains and ideally records actions taken in the sandbox. When testing malicious software, the sandbox runs an executable file and detects the malware by checking its behaviour [13]. The act of running and observing is called dynamic analysis. In a sandbox escape, the application escapes the sandbox and infects the host of the sandbox. Capturing the information can also be difficult since the sandbox used to capture the information may need to allow for many different capabilities for capture, such as monitoring memory, network and disk activity. Hybrid Analysis is the combination of both techniques.

Malware detection is less complex if the malware performs actions and changes settings that leave artifacts for the researcher. For a piece of malware to function, sometimes called an infection of a device, the malware must obtain access to a device. After this initial compromise, the device may relinquish access to allow the malware to execute further steps. The malware will change settings on that device until it has enough access to perform its functions. Part of this elevation of privileges on the device may involve a human using the malware to access the device and perform actions to gain more privileges or move to other systems in the environment. With few exceptions, malware will establish permanence on the device. Without permanence, a device reboot will erase the malware from memory limiting its effectiveness, especially for ransomware infections.

2.3.1 Static Analysis

The proposed research aimed to be operating system agnostic but will only investigate malware on the Windows and Android platforms. The Windows operating system uses PE files, and Android apk files are used on Android systems. This research aims to be platform-independent and targets the top two platforms that receive malware.

Most of the research for malware machine learning requires that the file is minimally dissected into parts. The dissection increases the opportunity for more inspection of features. The binaries' structure differs enough that Android applications will not run natively on Windows and vice versa.

The two formats have some commonalities to them. Both formats have header information that defines how the executable will work. Operation code or opcode is present in both file types. The benefits of examining the opcode sequence lie in the fact that it can be analyzed using raw data [19]. Opcodes are the bulk of the instructions the application can use to run on an operating system. The compiler transforms source code into opcodes during complication and then writes it to the targeted operating system file format.

The Windows operating system runs applications that use the Portable Executable File format (WinPE). The file contains size limits for header information. The size of the header is constant unless an optional header is used. The WinPE file format is shown in Table 1.

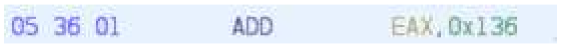
Table 1

WinPE File Format [20]

Section	Size	Purpose
DOS MZ header	64 bytes	Check whether the file is a valid PE file or not
DOS Stub	128 bytes	To prompt a warning message if the PE file is

		not compatible with the operating system
PE File Signature	24 bytes	Defines an executable file as Portable Executable
PE optional header	224 for 32-bit	Contrary to its name, it is essential. Contains information about file such as machine compatibility, number of sections etc.
Data Directories	240 for 64–bit	Provides information about where to find significant portions of file such as export, import, debug, etc.
Section table	8 bytes	Contains information about the sections present in PE files such as virtual size, size of raw data, pointer to raw data, characteristics

The header information contains information about the structure of the file and information about the file's compilation. The first 216 bytes of a PE file will be very similar to any other PE file. The optional header gives information about the executable, such as if it is 32-bit or 64-bit and the locations of code pieces to run. The Section table includes .text and .rsrc sections. The .rsrc section can contain resources such as images and information about the application, such as Copyright information, application name, and product version. The .text section contains the opcodes that will be used to execute the application. The opcode is written in assembly and coded to disk in hex. For example, shown below is a random line from the decompilation of calc.exe from Windows XP. The assembly example shown in Figure 3 is the "ADD EAX,0x136" assembly instruction written in HEX as "05 36 01".

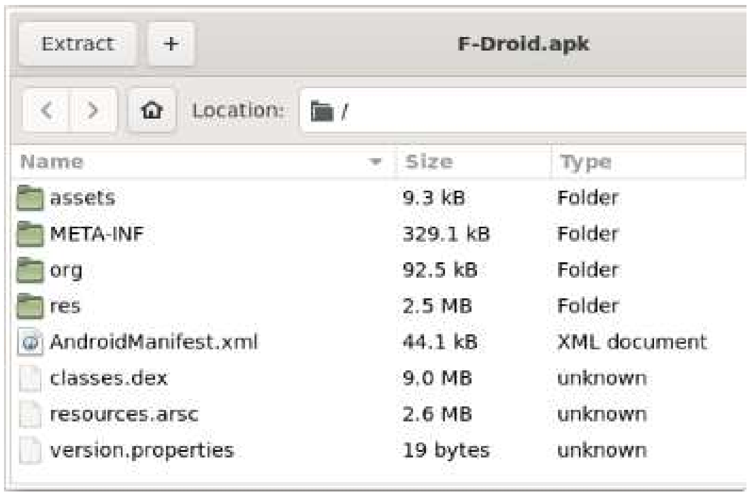


```
05 36 01        ADD        EAX,0x136
```

Figure 3 *Assembly Example*

An Android application is executed from a compressible archive file called an Android Package (APK). The APK contains several manifest files, a Dalvid executable (dex) file, versioning information and assets files, as shown in Figure 4.

The dex file contains the byte code executed when an application runs. The file is based on Java, but the compiled result differs from Java bytecode. The collection of all the opcodes is the bytecode that the application will use during execution. The assets directory contains resources for the application to use during execution and can contain images for display. Finally, the AndroidManifest.xml contains information about the executable and how the executable should run.



Name	Size	Type
assets	9.3 kB	Folder
META-INF	329.1 kB	Folder
org	92.5 kB	Folder
res	2.5 MB	Folder
AndroidManifest.xml	44.1 kB	XML document
classes.dex	9.0 MB	unknown
resources.arsc	2.6 MB	unknown
version.properties	19 bytes	unknown

Figure 4 *Extracted F-Droid Archive [21]*

A common feature that Android machine-learning papers use is permissions [11, 12, 22, 23, 24], and other research stated that incorporating permissions in future work would be worthwhile [25]. Permissions that the application follows are explicitly stated in the AndroidManifest.xml file. User prompts are required to accept the permissions for the application. The Drebin paper states that a great percentage of current malware sends premium SMS messages and thus requests SEND SMS permission [12]. A user may know that an application should not have that permission and not install the malware. In addition to permissions, the apk contains information on the hardware components that the application will use. Malware establishes network connections to infect and propagate. After a device is compromised, the device usually needs to communicate to proceed with lateral movement in the network, download dynamic code or exchange encryption keys in the case of ransomware.

Intents are also contained in the AndroidMainifest.xml file. Intents are part of a messaging system that communicates to other app components. There are two types of intents, explicit intents, which will specify which application will satisfy the intent by supplying either the target app's package name or a fully-qualified component class name and implicit intents, which do not name a specific component, but instead declare a general action to perform, which allows a component from another app to handle it [26]. Most malware runs as a background service. Thus, monitoring intents would be logical, as background services need intents to achieve the application's purpose [15].

Using deconstructed files form will allow machine learning to determine the maliciousness of an application. Application attributes can include information contained in the application and attributes generated from a translation of internal components of the application. Application attributes can also include analysis performed externally on the file, such as generating a hash of the file and noting the current filename of the file and created date. Part of the gathering process for malware datasets contains the elimination of duplicates. Creating a file hash of an application is a popular method. It allows for mostly unique identification and can verify if a file is malicious through services such as VirusTotal. The signature method is reliable and efficient in detecting known malware, but the major limitation of this approach is that it cannot identify new malware (Zero-day) [11]. Some research has combined application attributes with opcode analysis. Other research has also used machine learning to process characteristics of the opcode without analysis of the opcode tuples.

Malware authors adapt to research and anti-malware technologies. The paper Performance Maintenance Over Time of Random Forest-based Malware Detection Models [27] reviewed known algorithms from other papers. They found that machine learning algorithms' performance decreased significantly over subsequent test subsets, representing the performance of the models trained on 2012-2013 data. The study continued to find decreased performance on decisions between malware and goodware for executables in subsequent periods until the end of the training in 2018.

Extracting human-readable strings text information is a common first task in reverse engineering. Many machine-learning techniques do the same thing to establish the features they will use in addition to the application attributes already discussed; API calls can be extracted from the compiled binary. AMailNet, a deep learning framework, identified 32 Android activities relevant to identifying malware [28]. These features include permissions, hardware functionality and system utilities such as timers and network information.

The research developed the idea of using a control flow graph to determine if an application is malicious. The executable's assemble portion is processed and fed into machine-learning algorithms. Malware Analysis Intermediate Language (MAIL) is an intermediate language that attempts to simplify control flow graphs and help malware researchers determine the properties of an application.

2.3.2 Dynamic Analysis

Dynamic analysis requires the research to run the malware and observe and record the behaviour of the executable. The researcher's equipment is at risk since the malware may fully activate and escape control. Researchers use sandboxes to run malware to protect their environments. A sandbox is a technology that will limit access to the system running the sandbox. The sandbox can be an entire operating system reset to a safe state or an application running on top of the base operating system. Virtual machines emulate an entire operating system. The malware analyst will create a snapshot of a virtual machine. The snapshot contains the information to restore the machine to the time that the snapshot was taken. Restoring to a previous state will remove the malware and allow researchers to perform additional tests. Malware authors have reacted to sandboxes by performing checks to confirm internet connectivity and reading of hardware devices. If the checks fail, the malware will not activate to slow researchers' observations.

Malicious applications look for human actions, lab settings and the passage of time to avoid detection. If the malware detects that it is being researched, it will typically not activate. Dynamic analysis can take longer since a researcher must wait for the malware's timer to activate a malicious action. A researcher may run malware for 15 minutes as part of their process to give the malware time to activate [24]. If the malware is in an environment where hardware information can be retrieved, it can check specifics about the hardware. Some virtual machines will emulate the CPU of a computer. The emulated CPU will report as a virtual CPU, not an Intel or AMD CPU. Malware may not run if it detects a virtual CPU. Some malware will not check for this since it creates an artifact for malware researchers can check. Most legitimate applications do not concern themselves with the CPU type since it does not affect their application.

Network connectivity is essential for modern computing systems. Malware may check for network connectivity. The check could be to see if running in a sandbox and then not execute if it is not connected. Network connectivity could be required for the malware to function. Dynamic code loading occurs when an application retrieves code by decompression, decryption or network download, then uses the code as part of application execution. The malware detection system named DroidClone stated that dynamic code loading would prevent correct analysis [25]. This limitation would extend to most malware detection systems since analysis is complicated on code that cannot be analyzed as part of the initial file.

Hybrid analysis improves static and dynamic analysis results by fusing the two methods. Zhang used features such as files accessed during runtime and determining the API calls during static analysis to produce successful results [22]. The process can cover gaps by relying on tools that collect information the researcher may need to learn is valuable. For example, the Android dynamic analysis tool DroidBox does not record receivers at runtime [24]. The researchers wanted to use record receivers to monitor changes to the registry. The automated static analysis could perform discovery registry reads and compile changes. Hybrid analysis can trade effort and resources for higher accuracy. The worthiness of the effort is something that researchers and implementers must weigh when using analysis techniques.

2.3.3 Image Analysis

Visualization for classifying malware started in 2004 when researchers used unsupervised neural networks, specifically Self-Organizing Maps (SOM), to identify malware [29]. The paper Byte Visualization Method for Malware Classification converts the binary into an image. The researchers then use Jaccard distances to classify the malware[30]. A Jaccard distance is the coefficient used to gauge similarity between sets. In 2017 another paper improved image visualization to identify malware with neural networks [31]. However, the 2017 paper does not require the execution nor the disassembly of the program to work.

Creating better ways to understand malware has encouraged advancements in intermediate languages for malware visualization and analysis. Visualization is another tool that can help researchers. Dynamic analysis can fail when malware detects the anti-malware software. Pausing the execution of the malware payload is another method of evasion. VERA (Visualization of Executables for Reversing and Analysis) architecture allows researchers to view the malware execution at different times during the infection process [20]. The researchers noted that the Mebroot trojan prevents itself from being analyzed by entering a busy loop for approximately 45 minutes. Image visualizations illustrated the changes in actions taken by the executable after the idle waiting time.

2.4 Machine Learning

Using machine learning for malware detection has been successful, especially in edge cases. Researchers using an Internet of Things dataset achieved 99.99% and 100% accuracy for malware detection using Scikit-learn and Weka [32]. DeepXplore is a white box Deep Learning testing system [33]. They demonstrated the need for further research on how machine-learning algorithms are used. Drebin is an anti-malware application that uses machine learning, specifically linear Support Vector Machines [16].

Machine learning has three areas: supervised, unsupervised and reinforcement learning. Supervised learning uses a dataset to train the specific algorithm to achieve a result. Ideally, data from outside the dataset should produce correct results after training. The dataset fed to the supervised learning algorithm is labelled or classified. In the context of malware detection, the dataset used for training supervised learning will state whether a file is malicious. Algorithms used for malware detection for supervised learning are K-Nearest Neighbor, random forest, decision tree, linear SVM, logistic regression and Gaussian Naive Bayes [19]. Unsupervised learning uses unclassified data and aims to allow the algorithm to interpret the results it generates. Classification of malware is a typical unsupervised learning technique. Reinforcement machine learning is an area that focuses on allowing the algorithm or agent that the algorithm controls to adapt and react to an environment. The training of reinforcement machine-learning algorithms allows the algorithm to find solutions, and then those solutions are given scores based on criteria. Most of the existing malware research that uses reinforcement machine learning is targeting methods to avoid anti-malware detection. Adversarial machine learning research aims to help researchers set up better safeguards against malware. Practical Black-Box attacks against machine learning suggested that defenders can increase the attacker's cost by training models with higher input dimensionality or modelling complexity [19]. Experimental results indicate that these two factors increase the number of queries required to train substitutes.

Different machine-learning algorithms have different strengths when using them to identify malware. Anti-malware applications in home and enterprise environments will want discrete decision-making when determining if a file is malicious. The anti-malware application is tasked with quarantining, blocking or deleting applications it thinks are malicious. Overfitting will reduce the effectiveness of machine learning outside of the research space. Machine learning will train to a dataset and produce results based on the same dataset. An issue called overfitting arises when the trained dataset and training are specific to the dataset. The results are less accurate when testing on new or different datasets. The DeepXplore team stated that overfitting is a cause of poor performance of machine learning in edge cases [33]. Checking a hash of a file to determine if a file is malicious is not machine-learning; it does demonstrate that having a single feature to check is only valid if the dataset contains all current and future malware. Feature selection is vital to reduce overfitting. Increasing the number of features to check is computationally more expensive. Chih-Ta Lin et al. proposed a methodology to reduce feature sets. They found that reducing to 100-1000 selected features was sufficient to maintain equivalent micro-precision. Reducing the feature dimension to less than 1% would allow time cost savings of 99% in high-dimensional feature spaces [34]. Classification of malware into families or types is an exercise to group malware. However, the impact of inconsistent classification naming conventions lessens as the machine-learning algorithms will separate applications into groups and not generate classifications' names.

The order of commands in an executable can determine if an executable is malicious. Early static analysis methods would check for a particular sequence of opcodes to determine maliciousness. Malware authors adapted and reordered the executable so a simple scan would no longer flag an executable. Packing an executable is a method to compress and can be used to obfuscate an executable. Attackers packing and compressing their application requires that the researcher unpack the executable before most static analysis techniques are executed. The prevalence of packing in malicious executables leads to many models learning a direct (but unhelpful) equivalence between "packed" and "malicious." [34] Malware authors must choose between packing and not packing an executable. Suppose an executable is packed, requiring another file or memory to store the unpacked code. Having extra file writes increases the chances of detection and gives analysts a better starting point during analysis.

Control flow graph (CFG) techniques for malware identification and classification are of interest as intermediate languages. Techniques described as part of SAIL and MAIL [35] use opcodes to create CFGs. Converting to an intermediate language will not change the order of function calls. For example, MAIL uses control instructions such as JMP, RET and LOOP to preserve how the malware is executed. The application flow is converted to graphs which are used to determine patterns that represent malware. The researchers who created MAIL could produce false positives using minimal graphs, and large graphs can produce false positives. The number of opcodes to group is a common topic in malware opcode research.

One method of preparing data to be used for machine learning is to arrange opcodes into sets. If a set is kept in the same order, it is called an n-gram. The n in n-gram can also be replaced with the set size to be more descriptive. Adjusting the n-gram size from a 2-gram to a 5-gram is computationally more expensive during the machine-learning process. Research from 2014 found that the best results were obtained from a 5-ngram [14]. However, other papers with different feature reductions found that the four-gram model demonstrated optimal performance regarding the final accuracy compared with the five-gram model and high gram [36]. This impact can be lessened if the feature size is reduced. Removing opcodes of little value is a technique to reduce the feature size. The removed opcodes could be codes that do not occur very rarely or codes that do not change the result. The NOP or no operation opcode could be a candidate for removal if the machine-learning algorithm concerns itself with only the control flow of the executable. The growth of unique opcodes tuples in Android software is manageable. As the number of n-grams grows, the number of tuples increases linearly at first and gradually becomes stable [37]. Opcode n-gram research is not mutually exclusive with other methods. Researchers have found that by using decision tree machine learning and then applying the 3-gram, its accuracy is 16.7% higher [36].

Opcode research has branched from detecting malware that is new to detecting malware from the same family. Malware is said to be from the same family if the code from previous malware is reused and modified to avoid anti-malware detection. As the marketization of malware continues, researchers have moved some research to focus on detecting if two executables are similar. Matching tuples between new executables and known malware has successfully detected new malware. As a counter-detection method, simple forms of polymorphic attacks (i.e., malware that mutates at each infection) targeting Android platforms have already been seen in the wild [38].

2.5 Intermediate Languages

An intermediate language is another representation of the assembly or bytecode of the executable. Intermediate languages vary in size but do not vary by platform. Intermediate language research for malware splits into two groups. One group of researchers created intermediate languages to help further the analysts' abilities and learning. The Dream++ Framework [39] was proven to aid individuals unfamiliar with reverse engineering. An intermediate language aims to convert a binary closer to the source code that created it.

The other primary motivator for intermediate languages is to use them to perform a specific task. REIL [40] and MAIL [35] are intermediate languages to support reverse engineering efforts. MAIL was explicitly created for malware analysis. Malwise is a malware variant detection system. The intermediate language Wire was used to produce Malwise [41]. MDIL (Malware Detection using Intermediate Language) is another intermediate language that aims at discovering malware. MDIL expands on REIL to extra fine-grained features. The intermediate language includes a built-in evaluation system that aids in detecting suspicious malware and classifying it [42]. The target focus for intermediate languages can change as well. Bugerya designed an intermediate language that follows the flow of data [43].

2.6 Evasion Techniques

Malware source code is rarely available for researchers. Final production malware is often stripped of extras that could make decompilation easier. If malware authors left debugging symbols in the application, they write it would only aid in reverse engineering their malware. Faster reverse engineering would increase the chances of discovery and aid anti-malware implementations to detect their malware.

The more difficult malware is to detect and analyze, the longer the malware will be effective. File hash techniques are easily circumvented but are very effective in detecting known malware. Once the malware file hash has been released with major anti-malware solutions, the malware will not effectively compromise new systems. Galen and Steel found that various decision tree-based models showed relatively superior performance maintenance over time [44]. Not all machine-learning models were as successful. Their evaluation of the nearest neighbour algorithm dropped over fifteen percent compared to the result from twelve months earlier. Adapting to the new research, malware authors have tried increasing their capabilities for evasion. Malware authors have transitioned to reusing code again and adapting code from other malware authors. This has caused researchers to start grouping malware into families. Families of malware can have hundreds of malware samples in each family. The DREBIN paper sourced 925 examples from the Android Malware Genome Project [16].

Permissions are used as a part of machine learning to detect malware. Malware authors have pivoted and adjusted their manifests and code to make the applications behave differently to avoid detection. For example, to evade the detection, attackers may manipulate the Android Trojan “net.Mwkek” by injecting the permission of BATTERY_STATS, which is frequently used in benign apps in the manifest file, instead of removing suspicious permission of “SEND_SMS” [45].

2.7 Pre-Processing methods

The Python language is considered friendly and accessible by researchers and programmers due to the limited syntax requirements, lack of boilerplate code and complex installs needed to execute Python code. This accessibility allows everyone to write their libraries. This research was driven by the requirements to run machine learning. There is “no such library which consists of every data pre-processing required function in it,” according to the authors of PrePy [46]. This research aimed to avoid external libraries to avoid crossing coding boundaries that would be time-consuming for processing this type of data.

If the speed decreases for the crossing coding boundaries are ignored, generalized libraries exist but are aimed at other tasks that prevent efficient usage. For example, the DataSist [47] library uses a Jupiter Notebook as part of the library. Jupiter Notebooks are not optimized for opening massive amounts of files. According to the official documentation (https://jupyterlab.readthedocs.io/en/stable/user/file_formats.html) documents can be opened using the GUI or rely on Python coding to process the file into a Jupiter notebook. Tools such as Eisen [48] will group dataset access connectors. These tools still tend only to support formally structured data. Like this research, some papers also have a deep focus on specificity.

Papers can focus on topics machine learning topics such as outlier analysis and missing value imputation [49] or subject matter topics such as bioinformatics [50]. The structure of the data in the malware database prevented using existing research that could be adapted. For example, the paper on bioinformatics lists data cleaning, data integration, data transformation and data reduction as part of the steps for data pre-processing. As the analysis aims at the malware binaries, data cleaning and integration concepts can not be revised to translate to this research. The binary representation of malware is perfect, and combining the files into a larger binary is not logical.

Chapter 3. Malware Detection and Classification

This chapter also presents the focus topic at the beginning of the current research, malware detection and classification. By exploring the original topic, a flow for future work can be completed. Information must be processed for machine learning consumption. This chapter explores the advantages of re-focusing the original algorithms' scope. The pre-processing costs for binary analysis are high, and the original methods did not focus on optimizing repetitive runs needed for future research and reducing the costs for pre-processing. Using non-parallel pre-processing increases costs for duplicating the original topic and limits future research opportunities, such as increasing the size of n-grams.

3.1 Opcode Techniques for Detection and Classification

Using opcodes for machine learning is a specific implementation of static analysis that utilizes machine-learning techniques to classify and detect malware. Previous research has been successfully changing the representation of the opcode input to machine-learning algorithms. For example, by converting the opcodes to a gray-scale image and then reducing the image, researchers have increased accuracy and the number of features needed to perform analysis of mobile and IoT malware [51]. The paper, Long short-term memory-based (LSTM) Malware classification method for information security, used natural language processing techniques called word2vec to perform their research. They stated that the limitations of the proposed method in this paper require high computing resources since using all opcodes and API function names [52]. Their future investigations will classify malware by selecting meaningful malware opcodes and using the same methodology in their previous works.

3.2 Augmentation and Collection of Data Inputs

Augmented analytics makes machine learning and the data involved in machine learning more accessible. Ridhawi et al. stated that in addition to the processing and memory issues facing today's AI solutions, more important is the excessive amount of time needed to select a learning technique, test it, and then reconfigure different learning parameters [53]. Augmented analytics aims to reduce the workload for humans by engaging AI solutions. The malware datasets vary in what they provide and can contain only executables with only the guarantee that the executable is malicious in nature. Augmented analytics builds upon existing investments in analytics, which more likely than not fall into one of four categories: description, explanation, prediction, and prescription [54].

This research aimed to evaluate the size of the intermediate language and measure the correlation of the size change to the accuracy of existing work. Intermediate languages can remove barriers to machine learning. The size of the intermediate language will directly affect the size of the feature set. A reduced feature set size will open opportunities for less computing power during malware detection and classification. Since the feature set size will be reduced, this will eliminate the need for human choices for filtering or using an algorithm such as PCA (Principal component analysis) to reduce the feature set. In addition, possibilities for evasion will are reduced when using intermediate languages. The conversion to an intermediate language will filter out evasion techniques such as adding NOP opcodes.

The ideal length n-gram of opcode machine-learning may increase when the instruction set size is reduced. Part of malware research must monitor if overfitting is occurring. Changing the n-gram from opcodes to intermediate language codes increases the information in an n-gram. Overfitting causes machine-learning algorithms to perform poorly on new data. As the instruction set size decreases, there may be a point where overfitting will cause decreased accuracy.

The availability of datasets has caused some researchers to collect their malware and create their datasets. The paper "Familial Classification of Android Malware using Hybrid Analysis" used three repositories for their research [23]. The researchers combined information from Android Malware Genome Project (http://www.malgenomeproject.org/)and related research [15, 23, 24]. Gathering malware using honeypots is also an option for gathering malware. Researchers will create a resource that looks attractive to attackers and allow the resource to be exploited. The damage of the exploit will be limited by using a sandbox or another technique that keeps the attacker’s malware confined. Datasets created by these methods must be classified as malware or not malware, and classifications must be universal for a single piece of research. Third-party services such as VirusTotal could be used to classify and identify malware.

Some researchers have elected to publish their datasets in a processed form. The maintainers of the Ember dataset state that a wide breadth of benign samples is difficult to include in malware machine-learning datasets. Authors of legitimate software can be approached to donate their software to a malware research project. Donations reduce dataset creation costs but will increase time commitments to the donators when ensuring responsible dataset distribution. Purchasing software is possible for these datasets, but that will increase costs and require the publicly available dataset to modify the dataset to prevent the software from violating copyright issues or piracy. In addition, the Ember dataset does not offer the binaries or the assembly generated from the assembly. This processing would stop this research from moving forward with the Ember dataset.

In situations where the original malware dataset is unavailable or the dataset aged significantly, several other large datasets could be used. For example, the Microsoft Malware Classification Challenge was announced in 2015 and published a vast dataset of nearly 0.5 terabytes, consisting of the disassembly and bytecode of more than twenty thousand malware samples [9]. An additional option provided by the paper Malware Detection Using Gradient Boosting Decision Trees with Customized Log Loss Function used the FFRI Dataset from MWS [55]. The 2019 paper used the current dataset, but MWS has continued to produce datasets for the consumption of researchers.

Using available datasets will reduce the work required to research in malware analysis. Honeypot collection will favour collecting known viruses, and targeting specific malware will be impossible. Adapting a known dataset convention will allow better comparison of results and avoid problems other researchers face. There is no accepted standard for malware classification. Therefore, the first challenge is to identify malware as malicious. Performance Maintenance Over Time of Random Forest-based Malware Detection Models research uses the Brazilian malware dataset. (https://github.com/fabriciojoc/brazilian-malware-dataset) The Updroid [24] paper stated that “ ’Anti-Virus’ have different standards for naming malware and malware families,” the researchers collected 342 family names from the internet. During this research, a data warehouse must maintain the differences in the dataset information.

3.3 Limitations of the current pre-processing

File-targeted malware analysis aided by machine learning requires a vast number of files. A dataset cannot change attributes such as the size of the files to accommodate better methods. For the malware to be used in machine learning, a binary must be processed with other binaries and placed into a dictionary, array or vector. This process involves the combination of hard drive access and CPU processing. It was found that the relationship between hard drive access and CPU processing is not tightly coupled. A faster hard drive may not decrease the processing time consistently. The original topic stated that the entire process took two days. As this research required similar processing, two days were at the edge of reasonability as troubleshooting and testing would take time for the new research. There are only a few intermediate languages to use. Recreating the original topic and additional processing needs of a few intermediate languages required a two-day processing time. At this time, it appeared unachievable using the current hardware. Being unable to run the original code in a timely matter focused this research on optimizations to overcome these limitations and aid future research.

Chapter 4 Methodology

This research aimed to expand knowledge for pre-processing data using Python for malware datasets. The measure of success for this research varied across environments. Data processing foundations require discussing the code used in this research and the code solutions that would typically solve this problem. Despite Python's virtual machine abstraction, implementations of parallel processing dictionaries run differently using different hardware. Section 4.2 reviews the systems used to compare the environments accurately.

4.1. Machine-learning Data Processing Foundations

In order to process the malware data using the original topic's method, the malware must be manipulated from a large set of files to objects representing the data. This research focused on a section of code that counts occurrences of 4-gram hex strings that the original topic generated from a binary representation of malware in the dataset. The goal of this process is to count every sequential combination of 4-grams that exists in the malware files.

4.1.1 Pre-Processing Specifics for Malware Machine Learning

The code from the previous research is representative of the dataset that is used in most of the previous research on this topic. Many of pre-processing goals is to get a data structure that can report quickly on number of occurrences of an item. This research shifted to solve this problem as it was apparent that the time taken for pre-processing would hinder current research efforts. The source topic used dictionaries to count occurrences of items. There are many data structures in Python, but dictionaries are well suited to this task. A dictionary has a key and a value. There can only be one occurrence of any key. The lookup time on keys is O(1) and performs very fast in Python. Following the previous research, a large amount of time for the pre-processing was spent reading files and putting the information in dictionaries. The time taken is more important when looking at how one counts items.

4.1.1.1 Array Implementation

Initial efforts to increase the efficiency of pre-processing involved using arrays. Binaries have limited scope and processing binaries could have reasonable methods of adaptation of techniques to increase timing. Counting occurrences of a 4-gram occurrence was focused on as it seemed important to solve the issues with slow execution in general. A 4-gram in hex can be converted to an integer. Arrays in Python are O(1) for access. Using an index the value in the array can be accessed. Creating an array with a size of 4294967295 was explored as an option. This solution proved to be a slower method to count the occurrences of the n-grams. Significant time slowdowns during array creation and mapping. The data structure was just too large to be effective.

4.1.1.2 Dictionary Properties

Dictionaries allow quick access to a key and slow access to values. Like a paper dictionary there is a key or word that will lead to a value or definition. When a dictionary is used to count occurrences of an item finding the highest one, two or more items involves going through every entry in the dictionary. The advantage of using a dictionary is during the creation of the summarized data. When reading data, the occurrences are likely to be in a random order and avoiding searching through a data structure is preferred. Generally, arrays and linked lists are not used as finding an item involves in the worst case a search of the entire data structure.

4.1.1.3 Malware Machine Learning Specifics

This research was focused on improving the execution of the pre-processing based on previous research. Previous research and other research that includes malware will have certain characteristics. Binary analysis is based on decomposition of binaries. Binaries tend to be smaller files when compared to processing traditional datasets. Malware databases grow by adding more binaries rather than adding a row to a file or database. This creates a specificity to optimize for this type of research. Processing binaries switches between CPU and hard drive reads. This structure also prevents traditional methods of improving processing speeds. When loading from a database, table or large text file a large set or all the data can be loaded into memory. The original research's malware database size exceeded possibilities for this as it is 500GB.

4.1.2 Replication Process and Results

A baseline must be set between using opcodes for the machine learning processing and using an intermediate language. The original topic must be reproduced and measured as part of developing a baseline. As this research was to discover the effects of changing from opcodes to an intermediate language, a new baseline must be established without the additional steps. To get the success rate for malware detection, the researchers used image analysis to tune their detection method further. A measurement before this step would be more helpful for this research since a bitmap representation of a file is static, and the concepts of opcodes and intermediate languages would not change the outcomes.

Analysis of the processing of the files into usable data that can be fed into a machine-learning algorithm requires that the input and output be known—the input in the original topic consisted of two files for one piece of malware. The asm file contains a representation of the assembly code. The bytes file contained a HEX representation of the file malware, and its processing was the bottleneck for processing in the current research. Creating temporary files can store pre-processing steps. Temporary files allow the code to check for their existence and skip processing if the files exist. Future runs can skip ahead to the machine-learning part of the code to allow for quick execution.

The reproduction of the original topic faced issues around the discontinuation of Python 2, the isolation of the research environment, the unknown properties of the cloud server used and the lack of metrics included in the research write-up. The hardware used for the local server reasonably matched the CPU specifications at the time of publishing for the original topic [56]. The hardware configuration is listed as a Google compute engine with 104GB of memory with 16 CPUs. The hard drive space was listed as 1TB, but the speed was not measured. The server used during this research is a 24-CPU machine with 72GB of memory. The hard drive space consists of hardware RAID 5 and RAID 1 with 4TB and 1TB, respectively. The server could have expanded the memory, but during all testing, the memory was not fully utilized, and the swap file was not engaged. The paper listed Debian as the operating system used and continued to state that any operating system could be used as long as the libraries could be supported.

The run time of the full version was listed as two days, with most of the time spent on feature engineering. As that is a long time for a single process to run, steps were taken to obtain checkpoints in the code to resume a previous run and verify that the results of the run were successful. The code changes were made, and it was established that this hardware took significantly longer to get the modelling part than two days. The code was run to completion using the best single model. The result was uploaded to Kaggle for verification. Kaggle truncates scores to prevent some competitive advantages. A lower score is better, and the paper used as the source for this research scored 0.00283. The reproduction of the original topic yielded results of 0.003 and 0.00675 for a small and large test set dataset. The run was successful, but the results took eight days to complete. The source of the slow results needed to be addressed; otherwise, testing of new methods would be significantly hampered.

4.1.3 Code Changes

To standardize the research results, the stability needed to be improved. The original code was written in Python 2.7. A Python virtual environment could easily handle this requirement. The packages in the environment caused issues. The packages were not backwards compatible and had dependency problems. Challenges around the upgrades appeared as upgraded packages would not be compatible with Python 2.7. The upgrade tool pip would not support the language's age and refused to function for security reasons. The public repository for pip refused all requests due to the older environment. A local server was able to solve the security issues, but the change introduced issues with versioning, making version compatibility a manual process. An environment was built that could support the original topic's code, but it was a poor fit for changing the code. If results were produced using packages past the end of life would allow for speculation on if the libraries were the issues for the results. Upgrading the code might also provide relief for the execution timing issues.

There were many versions of a dictionary that were capable of parallel processing written as part of this research. As mentioned in section 4.1.1.1 an array implementation was tested. Overhead consistently created issues for decreases in time to complete the process. Section 7.2 shows that the number of messages changes the time to complete. This optimization was used to complete previous versions of the algorithm and the algorithm that is described in this paper. The early iterations of the algorithm exposed that adding more multi-processing or multi-threading complexity will add overhead, including management overhead that will drive times for processing completion higher. Most iterations of using multi-processing on top of multi-threading or vise-versa resulted in times slower than the single-threaded method and were deemed uninteresting to furthering this research. With the other iterations improvement was not significantly different from the single-threaded method to be interesting for research. Using coroutines and then multi-processing allowed for more successful results.

4.1.4 Resolution of topic speed issues

The long code execution time prompted checks against the original topic's equipment against the local equipment. The local equipment has equivalent specifications to the google cloud platform at the time. The google cloud platform at the time of the foundational topic ran at 2.67 GHz [56]. The Google Clould Platform(GCP) will use technologies to help code that runs single-threaded by facilitating the CPU to run faster than the advertised speed [57]. The higher speed is utilized when an application stresses only one CPU and several are available and marketed as "turbo". The base frequency is the frequency at which the CPU runs. This frequency is what the operating system reports. The newest C3 machines show a base frequency of 1.9 GHz compared to a single-core max turbo frequency of 3.3 GHz. It is reasonable to conclude that without the turbo functionality, there would be a significant change in the time it takes to complete a process. The test results include information to confirm whether this technology changes the results.

4.1.5 Standardization for Future Machine-Learning Research Involving Malware

This research moved focus to provide foundational support for further research on optimizing machine learning focused on malware. This research could be generalized to any research with a dataset with a massive amount of small I/O inputs mixed with high CPU usage between the inputs. To advance research for malware-based machine learning, examples of measurements are taken of a core piece of the data transformation process. These measurements could be used to evaluate the option to optimize code. The measurements taken were expanded upon and showed the execution time difference between multi-threading, multi-processing and the combination of coroutines and multi-processing.

Statuses given by the operating system are used as reference points but can not be trusted. The conditions for activating a turbo CPU core can not be safely assumed. Measuring the difference between the CPU speeds is difficult as profilers are also part of the ecosystem and may cause conditions that stop single CPUs from activating the turbo. Chapter 4 explains the measurements used to describe the systems and Google Cloud Platform's turbo. Chapter 4 continues to describe the measured differences between the systems. Chapter 7 demonstrates the disconnect between the expected results from system to system and make recommendations for parallel processing, including reasons to exclude results that represent the best that a system can complete a task.

4.1.6 Task Selection

The pre-processing is necessary to feed into the random forest algorithm in the source topic and vary on what was to be generated. The source topic used an image-based input to the algorithm and frequency analysis. The code selection to monitor and optimize was based upon the high-CPU usage of this section of code and the need to flip between hard drive activity and CPU activities. It would be possible for any data pre-processing to combine data into a large file or database and then process the larger file at once. The code selection aimed to avoid monolithic data processing as it reduces the opportunity for optimization and changes the focus from processing data to data warehousing.

4.2 Systems Measurement

4.2.1 Goals of Environmental Measurement

The time to complete a task is the success metric used for this optimization. The system test used has incorporated several tasks that would cause difficulty for a simple solution. Using code from the winner of the Microsoft Malware Classification Challenge (BIG 2015) [9] and the source data [58] from Kaggle, who hosted the machine-learning competition. The code and the dataset are ideal for this testing since it exposes issues with using Python to complete tasks concurrently. As part of data preparation, the code reads from files, performs a CPU-intensive operation and combines the result into a single object.

A complex process was desired to test the dictionary processing solution to introduce pitfalls that would occur when machine-learning practitioners used it. Many machine-learning preprocessing tasks will involve a data reading step, data processing and then combination into a final data structure for future usage. This activity will require I/O from a data source and CPU-intensive operations. Multi-threading and multi-processing have different strengths and weaknesses in Python. Multi-threading is better suited for I/O, and multi-processing is better suited for resource-intensive operations.

During data preparation for a machine learning task, counting occurrences are common. In many machine learning applications, the processed data may not be defined until further into the research process. Many malware machine learning researchers will filter low occurrence data but only know what that data is once the data is processed into data structures. A Python dictionary takes two inputs, a key and a value. The underlying code for a Python dictionary uses a hash table [59]. Dictionaries are tuned to retrieve keys efficiently. Finding a specific value in a dictionary may require checking every key. The keys of a dictionary are unique. This data structure allows for efficient totaling of random objects in a random order but relatively high costs to find the highest total.

4.2.2 Python Implementations of parallel processing

Python implements three different kinds of parallel processing. The two types of parallel processing match the capabilities of CPUs. Modern CPUs have multiple cores and multiple threads inside each core. Cloud infrastructure can create machines with multiple cores and modify the number of threads per core. Decisions on how to structure the execution of the code in Python need to address these different configurations. By measuring multi-threading and multi-processing under different situations, this research demonstrates the differences in execution time to provide the pre-processing needed for efficient execution.

4.2.2.1 Muti-threading

Multi-threading uses one CPU and allows for several executions to be completed at the same time. The limit on threads is total CPU power minus the cost of task switches and synchronization overhead [60]. As requirements for more complex CPU-bound applications increase, Python threading becomes more suited for specific tasks with delays that are hardware based. User interfaces such as keyboards and hard drive reading are often encouraged when opportunities for threading are being evaluated. Tasks in different threads are scheduled. Scheduling allows processing to continue when unknown delays in execution, such as waiting for a hard drive arm to spin to the correct location.

All processes running in parallel using multi-threading must be delayed or put to sleep at some point since a single CPU can only execute one instruction at a time. Python will choose the thread that is running through cooperative and preemptive scheduling. Cooperative scheduling occurs if a thread knows it will be waiting or idle and signals to the scheduler to give another thread a chance to execute. Preemptive scheduling occurs when the scheduler forces a thread to sleep. Periodically Python will force a thread to allow for another thread to run [61]. As Python does support cooperative scheduling, the implementation shown in this paper uses threading for socket connections for its parallel operations.

Threading allows for shared access to instances of objects. An instance of a dictionary in Python has challenges that a concurrent dictionary has in any language and challenges associated with the Python language. A concurrent dictionary must ensure that locking prevents adding keys until the dictionary can report that the key has been created. Counting occurrences of a string in the files is the goal of the testing methodology. The update action can use mutexes to avoid issues of lost data. High-level languages tend not to allow the locking of data structures by row or key. A custom data structure would be needed to ensure thread safety, or the language would handle the locking, and the transaction would be transparent to the programmer. Assuming a dictionary contains a key of "A" and a value of 2, Figure 5 shows the process for a safe update. The code using mutex would check for a value in the dictionary and lock all changes that involve the key "A." After locking the value, a thread would be allowed to process the update to the dictionary. Upon completion, the thread would signal to the mutex mechanism that it is done, and then the update on thread two could occur. An update becomes an addition if the key is not already present.

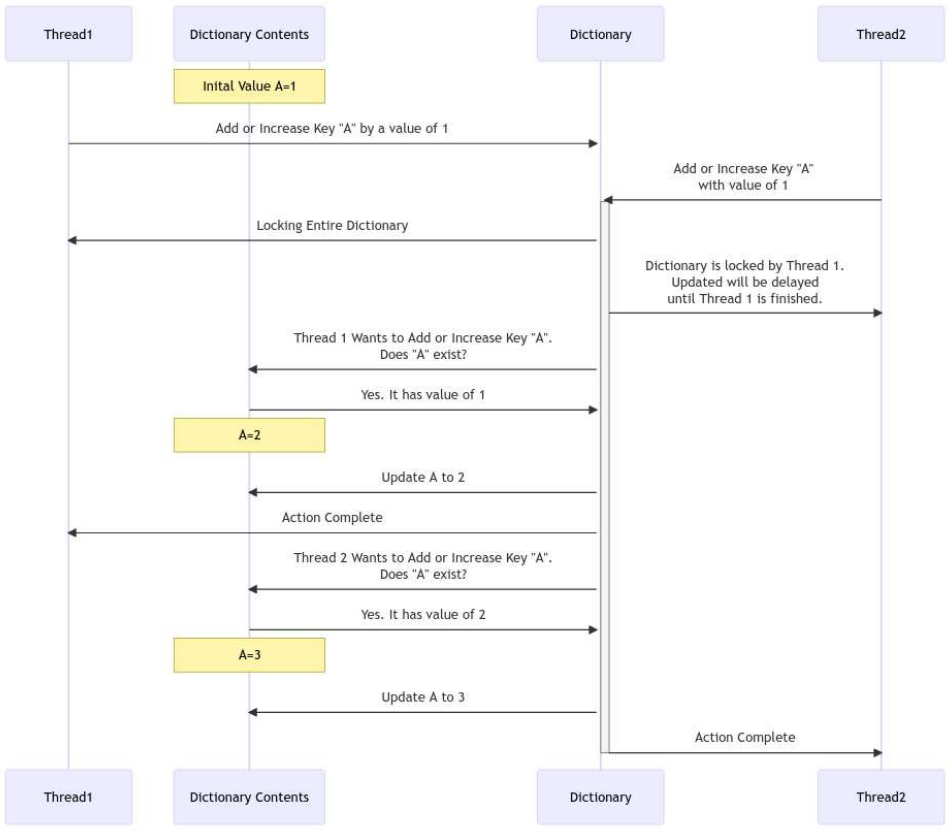

Thread1
Dictionary Contents
Dictionary
Thread2
Inital Value A=1
Add or Increase Key "A" by a value of 1
Add or Increase Key "A" with value of 1
Locking Entire Dictionary
Dictionary is locked by Thread 1. Updated will be delayed until Thread 1 is finished.
Thread 1 Wants to Add or Increase Key "A". Does "A" exist?
Yes. It has value of 1
A=2
Update A to 2
Action Complete
Thread 2 Wants to Add or Increase Key "A". Does "A" exist?
Yes. It has value of 2
A=3
Update A to 3
Action Complete

Figure 5 *Dictionary Update Using Full Dictionary Locking*

Adding a new dictionary key will force a lock for the entire dictionary without an outside data structure. In the example in Figure **5**, the key already existed. If the key does not exist, then values can be corrupted in the dictionary depending on the mutex implementation. If the access is not concurrent, only one value can be checked at a time. Having only one value available effectively locks the entire dictionary. Storing the active key in the locking mechanism will lock the entire dictionary. The Big O notation for key retrieval in a Python dictionary is O(1). Checking for a value in the locking mechanism could not beat the O notation of checking for the value in the dictionary itself. There are some real-world issues with dictionaries becoming too large. The target dictionary will be slower than keeping a separate list, but it is unlikely to become significant. If the mutex is not used, the dictionary determines if an add or update should occur, as shown in Figure 6.

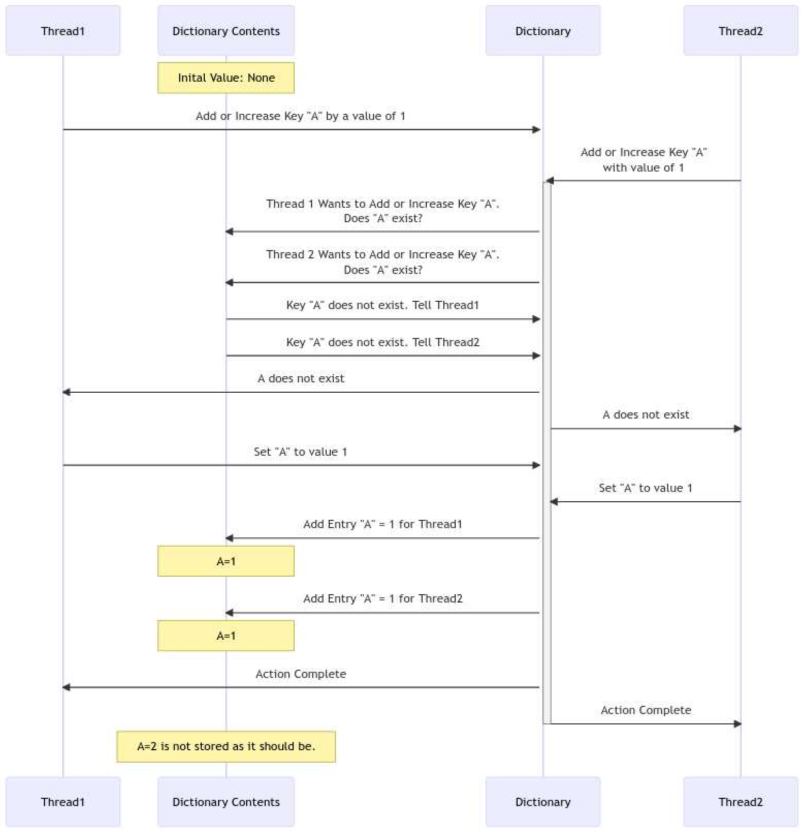

Thread1
Dictionary Contents
Dictionary
Thread2
Inital Value: None
Add or Increase Key "A" by a value of 1
Add or Increase Key "A" with value of 1
Thread 1 Wants to Add or Increase Key "A". Does "A" exist?
Thread 2 Wants to Add or Increase Key "A". Does "A" exist?
Key "A" does not exist. Tell Thread1
Key "A" does not exist. Tell Thread2
A does not exist
A does not exist
Set "A" to value 1
Set "A" to value 1
Add Entry "A" = 1 for Thread1
A=1
Add Entry "A" = 1 for Thread2
A=1
Action Complete
Action Complete
A=2 is not stored as it should be.
Thread1
Dictionary Contents
Dictionary
Thread2

Figure 6 *Dictionary Update Using Single Value Locking*

Python has locking mechanisms and mutex for programmers to implement their concurrency solutions. The mechanisms are unable to lock part of an object. If a dictionary wishes to lock a key, then the entire dictionary must be locked. Other data structures, such as arrays, have the same requirement. The mechanism to ensure thread safety in Python is called the Global Interpreter Lock (GIL).

The GIL locks on all threads can quickly cause significant runtime increases compared to a version that does not have multi-threading. Python the Global Interpreter Lock (GIL) to ensure that only one thread at a time makes forward progress [62]. The GIL does not activate when multi-processing is used. The GIL locks on threads are limited to Python code. There are countless activities that Python code initiates that would not run in Python. System calls in Python get changed in the byte code to C. These calls continue independently from the GIL. For example, reading from a file and other I/O activities do well in threads since each thread can start an I/O operation and continue until completion, independent of GIL locking.

The GIL has been a discussion topic for Python developers due to the speed issues observed in some implementations. Jesse Jiryu Davis has demonstrated that the GIL's pre-emptive locking is not perfectly thread-safe without correct coding [63]. Removing the GIL from Python has been attempted several times. Larry Hastings, a core Python developer, proposed removing the GIL in 2015 with his project Gilectomy [64]. This project does not have recent updates. Larry stated that he needed to explore new ideas in 2018. There continues to be attempts to remove the GIL. The nogil project first started in 2021 [64]. The project has recent updates but has not been merged into the core Python development stream [65].

4.2.2.2 Muti-Processing

Multi-processing is another method of execution concurrency in Python. Mulit-processing will execute independently of other processes and the main Python thread of an application. Being independent, the newly created processes will not cause activation and issues related to the GIL. If the newly dependent process starts multi-threading, the GIL will activate because of the multithreading. Since processes can not share objects between processes, there is no risk of collisions due to multi-processing. If data needs to be returned from a process, a queue must be used.

Queues are used to store and transfer data in Python. Python has a specific implementation of a multiprocessing queue that allows information to flow from one process to another. The queues are bi-directional. In a situation where there are three processes with a shared queue, two of the processes can read from the queue at the same time. If the remaining process sends something through the queue, there is no guarantee that which process will grab the information. If the remaining process decides to read from the queue, no error will be detected. In a system that pushes information back and forth on the queue, there is no method to ensure that the same information isn't taken off the queue again by the same process that put the information on the queue.

4.2.2.3 Coroutines

Using coroutines is another method of concurrent execution available in Python. It is a relatively new feature that was introduced in Python 3.5. Coroutines are considered a separate technology independent of multi-processing or multi-threading. The GIL would typically protect an object from multiple unsafe accesses to an object. Coroutines are not supervised by the GIL and are considered not thread-safe.

4.2.3 Evaluation Tools

The core measure of success for this paper is the time to complete the creation of the dictionary and return it to the calling function. The timing is measured using the timeit library available by default in Python. When the timer is started, the loader function runs the specific code package, which returns the dictionary, and then the timer is stopped. The times reported by this method can differ significantly from those reported by the profiling tools. The profiling times were often double the recorded execution time. The profiling tools showing the percentage of the time taken allowed prototypes to be quickly discarded as the effects of the changes were clearly shown. As a check, the sections marked with high execution time were timed using the timeit library. The percentage of time taken from the timeit trials was found to be similar to the percentage taken from the profiling tools. To summarize, the profiling tools would increase the execution time uniformly. The profiling tools could accurately show the issues with the code, which aided the development of the multi-threaded dictionary.

Scalene was created to profile Python code. The profiler includes functionality to perform accounting for Python and C separately. Scalene introduced a metric called copy volume. The copy volume metric allows analysts to identify inadvertent copying, including silent coercion or crossing the Python/library boundary [66]. During Berger's talk at Strange Loop, he discussed the issues between transferring between Python code and native code [67]. When Python code moves data between Python and C code the data will be forcefully recreated. The recreation can get expensive for loops that have high amounts of iterations.

4.2.4 Description of Task

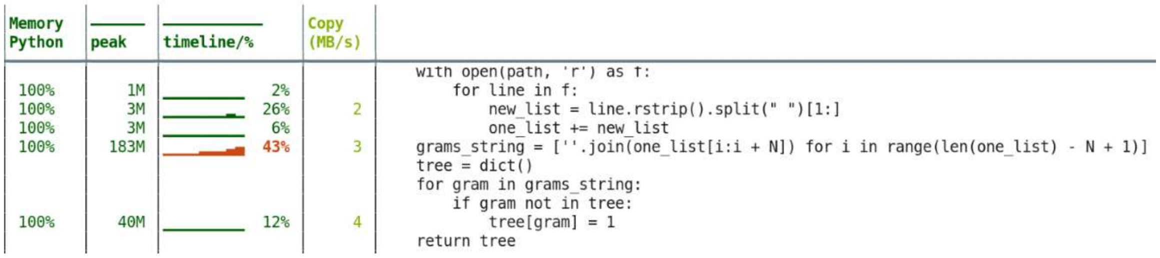
The single-threaded process to gather a dictionary for this code has three core steps. Load specific pieces of a file, create a sliding window of the data from the file and then create a dictionary based on that file. The original code referred to sets of files, and each set would create a different dictionary. The dictionaries would be added to a list and then returned.

The ASCII file used for the data source requires processing. The file is a HEX representation of malware. The files in the dataset vary in length and have the same structure. The file header shown in Figure **7** shows the first five lines of a sample file. The first numeric portion is the addressing information and is removed from future processing. The rest of the line is stripped of whitespaces and used for processing.

```
00401000 33 C0 C2 10 00 CC CC CC CC CC CC CC CC CC CC CC
00401010 8B 4C 24 04 B8 27 EE 07 3F F7 E1 C1 EA 1D 53 8B
00401020 DA B8 7B A3 80 0E F7 E1 8B C1 2B C2 D1 E8 03 C2
00401030 56 C1 E8 1F 23 D8 57 8B D1 F7 D2 8D 79 FF 23 FA
00401040 B8 25 F6 84 6D F7 E1 8B F1 81 F6 B6 01 87 F8 C1
```

Figure 7 *File Header of file 125y4VsArzkCNOGZfu6o.bytes*

The sliding window operations that the original code causes high CPU usage and significant time. The code generates all sequential combinations of four grams by iterating over the file set and then iterating over the file using two loops. Using the file in Figure 7, the first four values entered into the dictionary are 33C0C210, C0C21000, C21000CC and 1000CCCC. Figure **8** shows the scalene output for the original code from the BIG 2015 winner performing this task.



```
Memory                                  Copy
Python   peak      timeline/%           (MB/s)
                                                 with open(path, 'r') as f:
 100%       1M                      2%               for line in f:
 100%       3M                     26%         2         new_list = line.rstrip().split(" ")[1:]
 100%       3M                      6%                   one_list += new_list
 100%     183M                     43%         3     grams_string = [''.join(one_list[i:i + N]) for i in range(len(one_list) - N + 1)]
                                                 tree = dict()
                                                 for gram in grams_string:
                                                     if gram not in tree:
 100%      40M                     12%         4             tree[gram] = 1
                                                 return tree
```

Figure 8 *Scalene Output for a control run for 20 files*

Building the dictionary using the single-threaded method has 89% of the execution reading the file and performing the sliding window to generate values. Outside of this section is the combination of the smaller dictionary generated here into a larger dictionary. The combination of the dictionaries, not shown in Figure **8**, takes 10% of the total time.

There are several summary operations as part of this process. The provided dataset was divided into nine different classes. Each class holds a different number of files. The classes correspond to a specific malware family. As part of the inputs to the machine learning, the 4-grams are added to a container that represents all 4-grams in a file. The container is added to a dictionary that will total the occurrences of all 4-grams across a class. Each dictionary is added to a dictionary that contains all class information. This research compared the output of the original code to the new code to confirm that the same results were achieved.

4.2.4.1 Research Adaptations of Single-Threaded Process

The single-threaded version of the original topic performed a frequency analysis. This research replaces the mechanism to add or update a dictionary by implementing tuned optimizations for a certain data set. The pseudocode shown in Figure **9** shows the entire process. Line 21 and 22 have been optimized for Python and was not part of the original topic. In Python, strings are immutable. Any operations that require adding to a string in a loop will often be more efficient to use an array or avoid using a data structure that requires recreation. This research aims to show optimizations to create summarized data that requires high CPU and file reading. The original topic handled this processing using a single-threaded process which is inefficient, especially when considering the sunk cost of the machines that are typically used during machine learning. Mutli-threading handles high CPU workloads well. Multi-processing handles high I/O tasks well. When a process requires both high CPU and I/O, the choice to mix multi-processing and multi-threading is available.

The code in the parallel and non-parallel implementations contain the same core concepts and do not differ when focusing on how the file is read. In this implementation processing one file is atomic and does not focus on better file processing techniques. The file reading process was optimized and then assumed to be ideal. The differences in execution focus on the summarization that occurs after the file is read. It is easy to use parallel execution to read multiple files simultaneously. As discussed in section 5.2, parallel access to the dictionary is difficult. Figure 9 shows the execution of the single-threaded process. As an attempt to reduce boilerplate code, the figure does not show the addition of the collective dictionary into the larger final dictionary.

```
# create dictionary to hold all dictionaries
# Sets 1 - 9 are available.
# This research tests on size varied files sets from malware
# labeled 3 and 4
# loop through each file subset in file sets

# Return all occurrences of a sequential 4-gram HEX code for a file
def grams_dict(file_path, N=4):
    hex_2bytes = []
    # create a list of all sequential HEX codes from a file, while stripping
    # undesired file artifacts.
    with open(path, 'r') as f:
        # Loop through file line by line
        for line in f:
            # read line, clear whitespace and remove first entry
            new_list = line.rstrip().split(" ")[1:]
            # Add to list
            hex_2bytes += new_list
    # Go through list and perform sliding window on each 2byte HEX code
    # to create a list of 4-grams.  For example: AABBCCDD or AA1122FF
    grams = [''.join(hex_2bytes[i:i + N]) for i in \
                                        range(len(hex_2bytes) - N + 1)]
    # Initialize a dictionary to return a dictionary full of unique 4grams
    tree = dict()
    for gram in grams:   # Loop through each 4gram
        if gram not in tree:  # Only Add 4gram if it is unqiue for the file
            tree[gram] = 1
        # Multiple occurances of a HEX code are not totaled per file
    return single_file_dict


# Take incoming dictionary and add to the collective dictionary
def create_ngram_dictionary(files, collective_dictionary):
    # Cycle through each file in the file_set
    for file_name in files:
                # Find if a combination of sequential 4-gram HEX code occurs
        d = grams_dict(file_name)
        # Total 4-gram HEX codes into larger dictionary.  Incoming dictionary
        # existing entries by 1
        for k, v in d.items():
            if k in collective_dictionary:
                collective_dictionary[k] += v
            else:
                collective_dictionary[k] = v
    return result

# Initialize the final dictionary.
ngrams_dictionary = {}
for files in file_subset:
  # create dictionary for this file_set and add to the final dictionary
  create_ngram_dictionary(files, ngrams_dictionary)
```

Figure 9 *Core code example for entire execution for single threading*

Using multi-processing will result in the highest optimization for this specific dataset. The issue is that this method is not universally transferable to other applications. Figure ***10*** shows the pseudocode for the multi-processing version of the code. The “arg_list” variable is used to split up the files into separate tasks and allows the code to be run in parallel. The time to create the dictionary is generally faster than anything found in this research. This version does limit options when looking to optimize when using dictionaries across other datasets. The code processes the files in parallel and then sequentially enters all data into the dictionary. Only the file read is parallel using this code. This method will not be viable if there are requirements to enter values into the dictionary during the file processing. This requirement may be present during file processing, where information is streamed to the client, such as processing a live video stream. This method may also fall short should the entries into the dictionary become large enough that the small nature of the entries does not offer significant gains in time.

```
from multiprocessing import Pool
# from the example above function grams_dict remains the same
# create dictionary to hold all dictionaries

# Take incoming dictionary and add to the collective dictionary
def create_ncollective_dictionaryh(files, collective_dictionary):
  split = 4
  worker_count = 4
  dictionaries = []
  # The functional code has helper functions that feed build_arg_list
  # For this illustration build_arg_list splits the file lists into
  # roughly same number of split chucks.  The return of this function
  # returns an list of lists that have the parameters for
  # function file_loader
  arg_list = build_arg_list(split, files)
  # p.map will run the input function in a sepearte process (CPU) up
  # to the number of worker_count.  A new process will be spawned
  # should there be remaining
  # parameters in the arg_list.
  p = Pool(worker_count)
  collective_dictionary = {}
  for grams_result in p.map(grams_dict, arg_list):
    # process_dict combines incoming results with the
    # collective_dictionary this function is equivalent
    # to create_ngram_dictionary except there is no
    # needs to process files
    collective_dictionary = process_dict(grams_result,
                                         collective_dictionary)
    dictionaries.append(collective_dictionary)
```

Figure 10 *Pseudocode for entire execution for multi-processing without a queue*

Using multi-threading would allow the dictionary to be added to at any time without needing a queue but would activate the GIL if it was attempted. The code from the multi-processing example was re-executed using the multi-processing dummy import. This import will map all multi-processing calls to multi-threading calls. This change allowed the code and the Python interpreter to access the function with shared resources simultaneously. This code change demonstrated the effects of the GIL. Some testing scenarios showed an increase in execution time for the multi-threaded version, while other environments showed marginal decreases in execution time.

Queues are a way for multi-processing to access a shared resource. Data can be put in the queue and then processed as it arrives. Using queues can eliminate the issues around isolation around multi-processing; unfortunately there is a cost to use a queue. The cost of initialization and serialization of the queue are significant enough that care must be taken to avoid code that will slow down the process rather than help it. Initial testing using a queue was performed during this research. The prototype failed integrity checks against the single-threaded method that were not investigated as the time to complete was not going to exceed the times for the single-threaded method. The error rate was less than 1% over more than nine hundred thousand dictionary entries. Altering the decreasing object's size that will be put in the queue, there is a decrease in time to complete. The chart in Table 2 shows the results for a run using ten files. The dictionary produced 2112591 entries. This prototype aimed to explore the effects of putting large amounts of data through the queue compared to many small amounts of data. The queue serialization is based on Python pickling (https://docs.python.org/3/library/pickle.html). The prototype would load a variable number of values in the dictionary, send the values through the queue and then reassemble the values on the other end of the queue. Using only ten values at a time involved a significant initialization and showed that sending large pieces of data through a queue was not optimal. By sending ten large items through a queue, the time to complete the same process was increased almost thirty times over an equal amount of total data transfer. The times were still dependent on locking the dictionary as the core process must reassemble the dictionary from the queue into the in-memory dictionary. Using a simple queue system for parallel processing only adds overhead if the target dictionary can only handle one change at the same time. The difference between passing ten thousand entries and one hundred thousand entries is assumed to be due to environmental variance, and the high number of entries shows a plateau for timing increases.

Table 2
Dictionary Creation Times Varied by Queue Size

		Size of Array In Queue				
	Single-Threaded	10	100	1000	10000	100000
Time to Complete in Seconds	6.19	182.20	65.03	20.85	14.66	14.84

4.2.5 Research Design

4.2.5.1 Data Input

The time taken for the dictionary to process entries is related to the input that the dictionary receives. The testing results are generated using the same file set for each type of method used. A test run is defined by size and references the number of files processed when assigning data to a dictionary set. For example, a run of "5" will create two dictionaries which will process ten files. Table 3 lists the statistics for the file sets and the dictionaries it produces. The statistics for the dataset used for the BIG 2015 challenge are included for a measure of scope but were only processed once in this research due to resource constraints. The entire dataset is 35 times larger than the largest dataset fully tested.

Table 3
Dataset Characteristics

Dataset Designation (Files Used per dictionary)	**5**	**10**	**20**	**25**	**50**	**100**	**200**	**Total Dataset**
Total Line Count (In Thousands)	478	1217	3092	3475	6330	11631	24714	876680
Total Byte Count (in MegaBytes)	27	68	172	196	351	644	1433	50847
Dictionary 1 Size (In Thousands)	928	2022	3678	5346	8320	1475	27709	■
Dictionary 2 Size (In Thousands)	1184	2034	2660	4285	6241	11415	17696	■

4.2.5.2 Environment

This research for a multi-threaded dictionary was performed in a virtual machine using QEMU on a dedicated server. The server is a Dell PowerEdge R710 Server. The CPU runs at a speed of 2.67 GHz. The virtual machine used in this research has been assigned 21 cores broken apart using a configuration of seven threads and three cores. Memory for the virtual machine is set to 64 gigabytes out of a possible 72 gigabytes. Memory was monitored in all testing and found not to be a factor in execution time. As protection, the server was put behind an internal firewall with allow listing by IP address for services needed for the server. The server is a security risk as it contains thousands of pieces of malware and does not have anti-malware protection that would automatically remove malware. As a further precaution the libraries used by Python were sourced internally. Python will gain new functionality by pulling in libraries using the pip command. Configuration changes changed the target of pip downloads to another server internally. The other internal server running devpi (https://github.com/devpi/devpi) would act as a proxy and allow for downloads that were otherwise blocked.

To better understand the speed differences between the original topic and the current implementation of the single-threaded model, the equipment and environments additional environments were used. Three machines were used to form nine environments to test upon. A virtual machine was deployed in the Google Cloud and modified in three different ways. The CPU was changed between the Google Machine standard and compute family. The original topic cites using the Google Compute family. Currently the CPUs of standard and compute deployment are 2.2 GHz and 3.1 GHz. At the time of the original research the google compute speed was advertised as 2.6GHz according to archived GCP documentation (https://web.archive.org/web/20150114113051/https://cloud.google.com/compute/docs/machine-types#highcpu) Two different hard drive types were used for testing. The base operating system uses the balanced persistent disk (pd-balanced). The base operating system disk was not changed during any time during this research. The second drive containing the test data was swapped during the different testing. For the test data a standard persistent disk was used (pd-standard) and the performance (SSD) persistent disk (pd-ssd) were created and attached to the cloud virtual machine as needed. The testing methodology recommended by google (https://cloud.google.com/compute/docs/disks/benchmarking-pd-performance) calls for sixteen ten gigabyte files which exceeds the limits of the provided for the base operating system. To match the original environment as much as possible Ubuntu was used. The kernel in use was version 5.15.0.

The third consumer environment was added for testing to allow for the inclusion of a SSD. The code and data are for malware analysis and contain malware. As discussed above the datasets for the tests only included ascii files which avoids the risk of infection. The consumer system has a CPU speed of 3.475GHz. The operating system is Gentoo and runs a custom kernel version 5.15.80. The consumer system has a Seagate BarraCuda drive running at 7200 RPM. The SSD in this system is a with an advertised sequential read of 560 MB/s. The first environment's, the Dell server, primary drive is a RAID 5 and uses a RAID 1 for the test data. The server is running using PERC6/I with all drives being 7200 RPM Seagate drives.

For consistency the systems have had common names assigned to them during this research. The Dell Server contains the host operating system and the virtualized operating system. The two environments will be referred to as Server Host machine and Server Virtual machine. The GCP cloud environments had different configurations. The primary operating system was placed on a HDD. This drive is common throughout all the GCP configurations. The CPU (and memory) were changed as well as the secondary drive. While it is the same system this equates to four different environments. The name convention for these environments is

"Google Cloud" or "GCP" followed by the CPU package used, followed by the hard drive used. If the CPU or hard drive is not important in the current topic it was removed from the name. For example, Google Cloud standard hard drive does not include the standard or compute CPU in the hard drive testing since CPU type does not change the speed a hard drive reads.

The operating system for the server class host machine uses Arch Linux and the guest machine uses Ubuntu. These operating systems use Linux kernel version 5.18.15 and 5.13.0-44 respectively. Overhead for the virtual machine and host were lowered as much as possible. The host is headless and has no running X server. The machines are network isolated in testing primarily due to the concerns about using this dataset with a secondary side effect of updates and other automatic processes not being able to activate during a test cycle. Just before the cloud server testing the network connection is removed from the server and restored after the testing.

The Python version from the BIG 2015 source code was 2.6 and was upgraded to Python version 3.8.10 used in the final testing. Minor functional modifications to the code were made to allow the code to run. The final code required that the range command be used instead of the legacy Python 2 equivalent. For uniformity the original Python virtual environment was created and then copied and then sourced to the other environments.

4.2.5.2 Virtualization using QEMU

QEMU options for the virtual machine include optimizations were made to increase the execution of the code. Intel processors can utilize Non-Uniform Memory Access (NUMA) while running virtual machines. NUMA is a share memory architecture that describes the placement of main memory modules with respect to processors in a multiprocessor system. This allows all memory requests to have nominally the same latency [68]. Tests were completed to compare results using NUMA by using the numactl as part of the startup command for QEMU. There were no changes to results when using NUMA and not using NUMA. After testing with sysbench and a forty-file file dataset for dictionary operations there was no difference between the times produced. All the final results were completed without the NUMA option.

CPU pinning was implemented and reversed to gain increased execution time. Figure **11** shows the output of the lstopo command on the Server System. The output shows that the CPUs are in different sockets. Online recommendations to pin the socket to the virtual machine were completed. The pinned socket would reduce management needed by the kernel. The pinned socket in the virtual machine showed no significant change in execution times. As the system would lock the entire socket to the virtual machine it limited the number of CPUs available. The host system required at least one CPU. This would leave only one socket available to the guest and halved the number of CPUs. Virtual sockets were also tested with no increase in performance found. Brief operating system performance slowdowns were observed using configurations with high socket counts, such as 7 sockets and 3 CPU cores, but this was not observed during testing. As the socket pinning for the virtual machine did not increase performance a single socket with many CPUs was used in the virtual machine. The final configuration of the virtual machine is shown in Figure **12**.

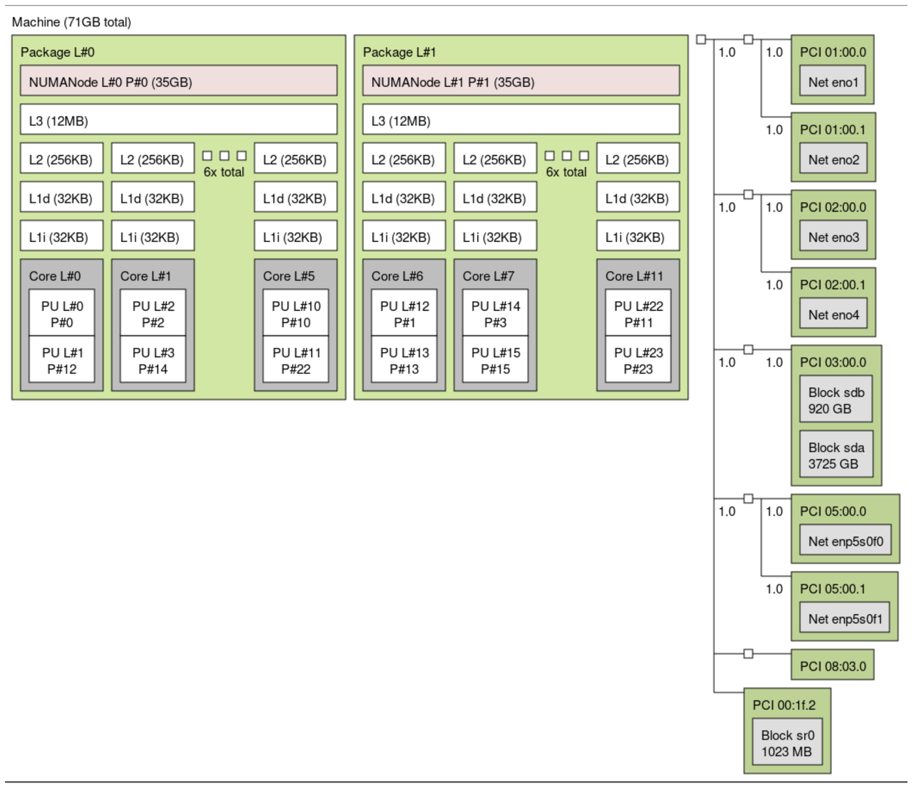


Figure 11 *Output of lstopo command for Server System*

The virtual machine uses the enable-kvm flag with the option to use l3-cache. These switch and flags will allow the virtual machine hypervisor level 1 access to the CPU. The L3 cache variable was set for the virtual machine at different amounts, but the guest system would not recognize the changes. L3 cache flag was tested and found marginally faster than not using the L3 cache flag. Using different amounts for the L3 cache including matching the host L3 cache of 1 did not display any differences in performance during benchmark tests.

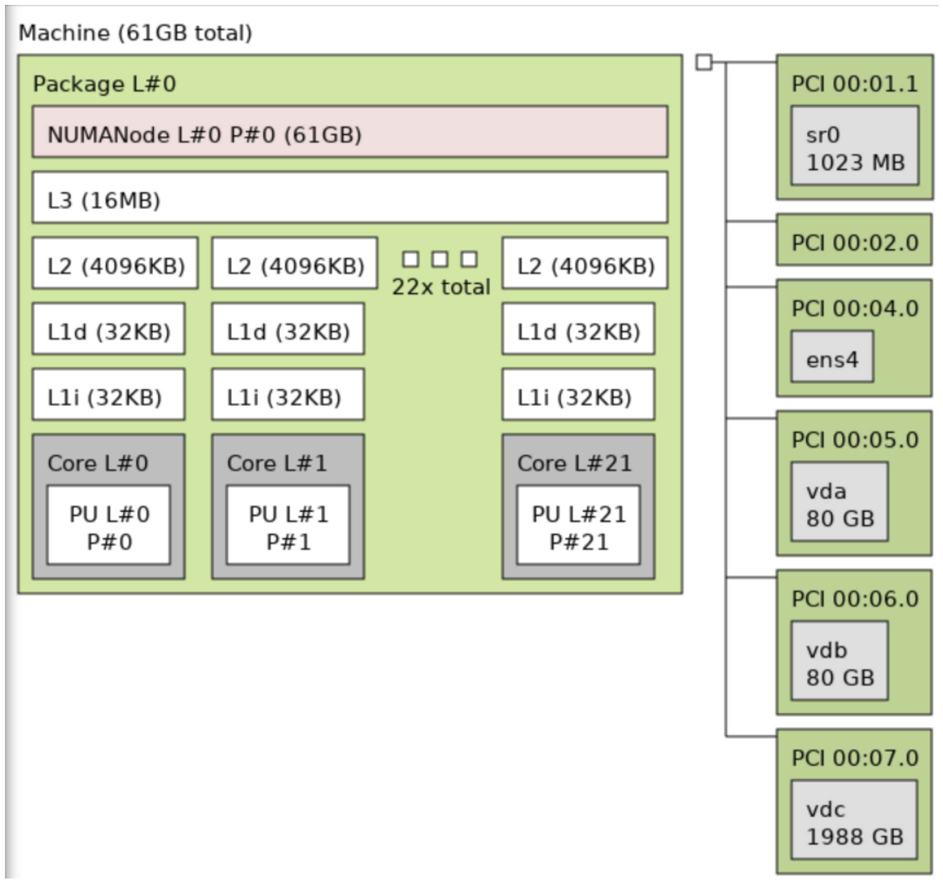


Figure 12 *Output of lstopo command for virtual system*

Linux virtio drivers are used for network, video and hard drive devices. This allows for better performance as it avoids the need to emulate physical devices. (https://wiki.libvirt.org/page/Virtio) Instead of using VNC inside the virtual machine, the spice client is used to offload processing for remote control on the host and not the guest. Hard drives use the flag aio=threads for the attached hard drives. The flag forces host user-mode-based threads. (https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_tuning_and_optimization_guide/sect-virtualization_tuning_optimization_guide-blockio-io_mode) Setup testing found that hard drive performance was higher than using the default “native” method. The larger drive on the virtual machine uses direct mounting from the host system partition. The main drive on the virtual machine uses a QEMU image in the raw format. Snapshots of the drives were avoided to reduce the possibilities for additional time for I/O and testing required. Memory available in the virtual machine was set high enough that the limits for the memory were never exceeded.

4.2.5.3 Monitoring and Gathering Results

Full assessment of machines used for data feature creation must be measured to accurately determine the cause for slowdowns of the running code. There has been developments that aid developers improve efficiency in their existing code. Tools such as PySpy and Scalene were used to analyze a solution for a multi-threaded dictionary. As research is focused on starting a solution and reproducing results, mid-pipeline tools for solving speed issues would not produce the results needed for evaluation. The Linux tools hdparm and fio were used for evaluation of the hard drives. CPU speed was tested using the sysbench tool. For CPU evaluation the tool will calculate primes and measure the execution time.

Chapter 5. Environment Measurements and Evaluation

Since the execution of the complete code from the Big 2015 research was not consistently running in a timely manner, tests were designed to identify the nature of the execution of the code and how to get future code to run consistently and reliably. The original code contains a hard drive read, a high CPU activity and an output. To compare execution fairly, an understanding of the hardware's capabilities must be established. Once the hardware is evaluated, then factors then the code can be evaluated further for further opportunities for faster execution. It was found that timed results from running the code did not contain a linear relationship to the hardware used. Evaluation of these results can allow for more realistic expectations when running future code and reporting on times for completion of execution.

5.1 Results from Testing Environments

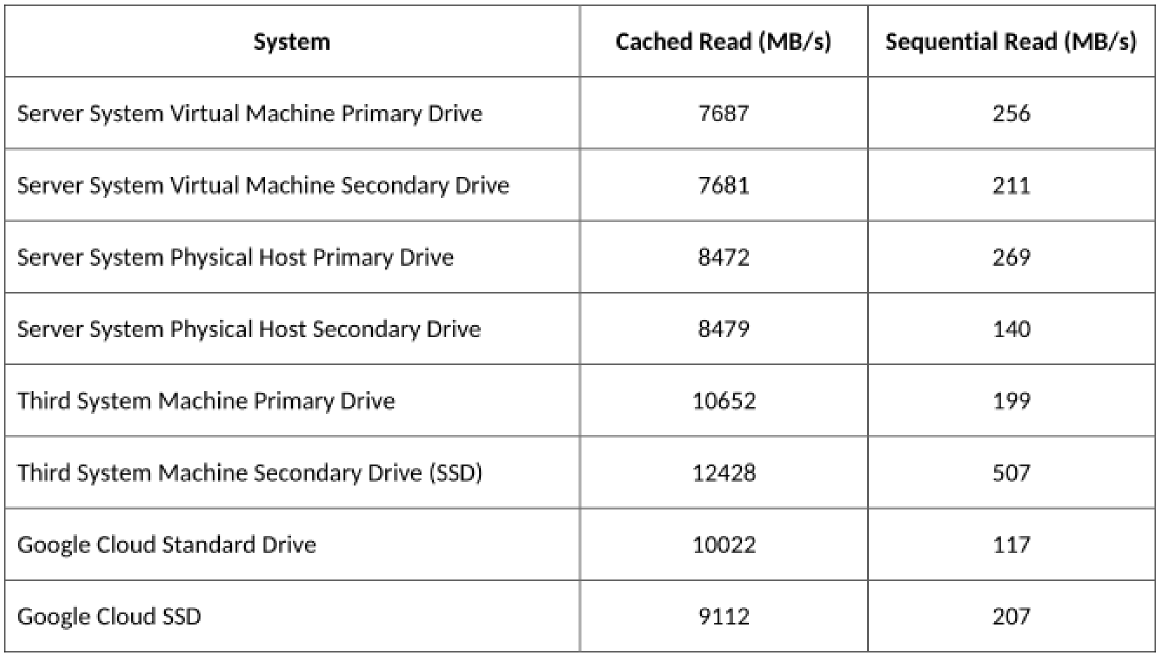
5.1.1 Hard Drive Results

Hard drives were tested using multiple types of read testing using fio (https://fio.readthedocs.io/) and hdparm (https://sourceforge.net/projects/hdparm/). The set of commands was executed in eight different scenarios. Hard drive testing tested the host and virtual machine on the local server. The systems were tested using the hard drive running the operating system and a hard drive that did not run the operating system. The primary drive refers to the partition that holds the operating system. The secondary drive refers to the drive that holds the dataset.

The server class host system is the only system that holds multiple partitions on one drive. A host system test on the operating system hard drive is referred to as the server-class virtual machine primary drive, and a test of the second drive is referred to as the server-class virtual machine secondary drive. The server class host primary drive contained a raw file for the virtual machine. The secondary hard drive is mounted in the virtual machine directly. Testing for the server-class host physical secondary drive requires the shutdown of the virtual machine, as the drive will corrupt if anything is written to the secondary drive from the host while it is mounted in the virtual machine. The remainder of the hard drive tests were performed in the environment native to the environment. The consumer system does not have a virtual machine, resulting in tests completed on the physical hardware. The cloud system does not have physical machine access, so all tests were completed on the virtual machine. The cloud system uses a non-operating system drive in all tests.

The first test was to determine the general properties of hard drive activity. The systems have optimizations for hard drive reads. One common optimization that systems will have is to cache data that has been seen before. The hdparm test was run in the environments to determine the difference between cache speed and the random read speed. If the drive uses caching, the results would not be consistent and lead to unfair hardware comparisons. Most datasets will exceed the limits of hard drive caching. The results from the virtual server class machine were thirty times or faster; this increase prompted the removal from the result set. The elimination of the data was justified as it was clear that more favourable results resulted from caching.

Table 4
HDParm Hard Drive Speed Results



System	Cached Read (MB/s)	Sequential Read (MB/s)
Server System Virtual Machine Primary Drive	7687	256
Server System Virtual Machine Secondary Drive	7681	211
Server System Physical Host Primary Drive	8472	269
Server System Physical Host Secondary Drive	8479	140
Third System Machine Primary Drive	10652	199
Third System Machine Secondary Drive (SSD)	12428	507
Google Cloud Standard Drive	10022	117
Google Cloud SSD	9112	207

5.1.2 Fio IO Engine

The fio tool can utilize the psync engine or the libaio engine. The fio tool can run drive tests measuring sequential reads or random reads. The base command for these tests was:

```
fio --name=read_iops --directory=$TEST_DIR --size=10G  --time_based --runtime=60s --
ramp_time=2s --direct=1 --verify=0 --bs=4K --iodepth=256  --group_reporting=1
```

These settings are based on Google compute's recommendations for testing hard drive speed (https://cloud.google.com/compute/docs/disks/benchmarking-pd-performance). The GCP recommendation uses the libaio engine. The psync engine was run in addition to the libaio engine. During the research, it was important to confirm the effects of the hard drives while using virtualization. Reading using psync will use synchronous reads that will block further reads to the hard drive. Reading using libaio will use asynchronous reads that will not block further reads. The results of these tests are shown in Table 5.

Table 5

Fio Read Speed Results

System		psync sequential read	libaio sequential read	psync random read	libaio random read
Local Virtual Machine Primary Drive	Average (MB/S)	100.9	113	0.596	4.979
	Standard Deviation	1.153	1.732	0.329	0.1605
Local Virtual Machine Secondary Drive	Average (MB/S)	70.87	85.8	0.6107	5.253
	Standard Deviation	3.361	15.51	0.004619	0.2454
Local Physical Host Primary Drive	Average (MB/S)	86.67	79.57	0.668	5.314
	Standard Deviation	13.97	17.13	0.002646	0.371684
Local Physical Host Secondary Drive	Average (MB/S)	69.0	62.1	0.603	2.54
	Standard Deviation	2.3	0.361	0.004	0.151
Third Machine Primary Drive	Average (MB/S)	85.27	119.3	0.51	1.364
	Standard Deviation	12.31	14.22	0.123	0.04464
Third Machine Secondary Drive (SSD)	Average (MB/S)	466	474.7	35.4	224.3
	Standard Deviation	6	1.155	2.816	4.041
Google Cloud Standard Drive	Average (MB/S)	55.85	25.65	0.2285	0.616
	Standard Deviation	48.30	0.3535	0.3359	0
Google Cloud SSD Drive	Average (MB/S)	210	217	7.2525	49.1
	Standard Deviation	0	0	0.3359	0

Asynchronous reads will result in better performance. The changes in performance between physical and virtual are listed in Table 5. The table demonstrates the differences between the systems. The physical server location for GCP machines was chosen based on low usage. Over the two tests on each server, the results for the SSD were more consistent than the standard drive. The standard drive's performance was significantly variable compared with all the machines. Sequential read tests were also more prone to higher variability during the tests. The file format of the virtual machines on the server class machine was the "raw" type provided by the QEMU image command. The host system can not read this format until it is mounted using an offset. The secondary drive was mounted in the virtual system as an entire partition using the aio=threads flag. Mounting the second drive in the virtual machine can be accomplished by creating a local file on the secondary host drive or mounting the partition straight into the underlying virtual machine. The secondary drive in the virtual machine running faster than the secondary drive on the host shows that the aio=threads flag with the direct mounting is the best choice.

Using a secondary drive for the data was based upon more favourable results when using random reads. The data sets used in malware analysis tend to be large numbers of small files rather than processing data from large databases or clean summarized data. Table 6 shows the hard drive read differences between the different drives. The GCP platform does not allow for easy transfer of the operating system partition between the standard drive and the SSD. The systems could use a secondary drive to have the same dataset on two drives. The GCP virtual machine could be shut down, swapped the secondary drive, and rebooted. This process failed to automount the secondary drive. Using a command to mount the drive after reboot was the only change to the system that occurred between the different environments.

Table 6
Hard drive read comparison

Reference Drive	Compare Test Drive	**psync sequential read**	**libaio sequential read**	**psync random read**	**libaio random read**
First System Virtual Primary Drive	Server System Virtual Secondary Drive	42% Slower	31% Slower	2.4% Faster	5.2% Faster
	Server System Host Primary Drive	16% Slower	42% Faster	11% Faster	6.3% Faster
	Server System Host Secondary Drive	46% Slower	82% Slower	1.2% Faster	96% Slower
	Consumer System Primary Drive	18% Slower	5.3% Faster	17% Slower	265% Slower
	Consumer System Secondary Drive	78% Faster	76% Faster	98% Faster	98% Faster
	GCP HDD	81% Slower	340% Slower	160% Slower	708% Slower
	GCP SSD	52% Faster	48% Faster	92% Faster	90% Faster
Server System Virtual Secondary Drive	Server System Host Secondary	2.7% Slower	38% Slower	1.3% Slower	107% Slower
	Consumer System Primary Drive	17% Faster	28% Faster	19% Slower	285% Slower
	Consumer System Secondary Drive	85% Faster	82% Faster	98% Faster	97% Faster
	GCP HDD	27% Slower	234% Slower	167% Slower	752% Slower
	GCP SSD	66% Faster	60% Faster	92% Faster	89% Faster
GCP HDD	Consumer System Primary Drive	35% Faster	79% Faster	55% Faster	55% Faster
	Consumer System Secondary Drive	88% Faster	94% Faster	99% Faster	100% Faster
	GCP SSD	73% Faster	88% Faster	97% Faster	99% Faster

5.2 CPU Comparison

There are four sets of CPU configurations used to produce the results. With the use of QEMU, the server system’s host and virtual CPU speed of the host matched the speed and capabilities of the virtual guests. The speed of the processors is listed in Table 7. The table lists the speeds reported by the operating system. Google Cloud Compute instance states that the reported speed from the operating system may be lower than the actual speed. The operating system does not depend on the reported speed, and incorrect reporting does not affect the machine's functionality.

Table 7
CPU Speeds

Machine	**Speed**	**Percent Difference Compared with Server Class Machine**	**Processors Available**
Server Class Machine	2.67 MHz	0%	21 Virtual/24 Host
Consumer Machine	3.475 Mhz	23% Faster	16
GCP C2 (Compute Instance)	3.1 MHz	14% Faster	4
GCP E2/E4 (Standard Instances)	2.2 MHz	17% Decrease	2/4

5.2.1 GCP Compute Variable Speed

The investigation into the differences in execution time of the original research and this research was prompted due to the significant differences in the execution time of the code provided and the time stated to finish by the original researchers. One goal of the mk13 algorithm is to standardize the execution times of code for research. The GCP Compute overperformed related to the results for the published speed of the server. Google states that virtual machines using the compute instance may increase utilize other processors as they are available. This optimizes non-parallel code to run faster than on a physical machine. Execution times using the same SSD for the single-threaded process were 18.76 and 29.51 seconds. This is a 36% speed increase which is reasonable when the difference between the computing speed is 40%. The times for the single process standard drive are 21.7 and 29.51. This increase of 37%, when matched with the SSD results, shows that the speed increase was not occurring for these machines.

Chapter 6. Parallel Processing Results and Recommendations

Using different methods of concurrency is encouraged for high-volume datasets. The tools provided by Python for multi-processing and multi-threading can be made more time efficient by avoiding queues and creating custom socket functionality. For a solution that requires heavy CPU usage, system calls, and data sharing between objects, choosing a technology to support parallel operations may not be apparent. This research designed an algorithm that performs reliably across large datasets using local resources. The algorithm performed second best in local environments and performed with high variability in cloud environments. The variability produced for best results across all methods tested in two environments. The complexity of a custom solution requires more demanding coding work involving more Python knowledge may deter researchers from attempting optimization. In addition, the added complexity of a concurrent dictionary will increase the number of lines of code significantly. This complexity adds overhead that may cause an increase in runtime for small tasks.

The bottleneck of process-to-process communication must be mitigated to achieve faster times than a comparable non-concurrency method. Splitting the incoming data without overwhelming the new process or thread must be balanced with the workload of splitting the data into smaller pieces. Processing the incoming workload required the payload to split into five different processors. Splitting the payload into less than five pieces would cause queues or socket congestion. The congestion would cause delays causing results higher than the single-threaded method. The congestion was not an issue for the memory in testing, but system monitoring showed a spike in memory and CPU usage.

The size and type of the queue's payload need to be balanced to achieve better execution times. The solution demonstrated splits the incoming files into five different dictionaries and combines the different dictionaries into one dictionary to be returned to the primary process. Passing a large payload to the individual processors resulted in a slower time. Immediately splitting the payload into smaller pieces also caused slower times. The faster time was achieved by processing smaller batches, which were split into different processing processes.

6.1 Implementation of Parallel Processing Method's Code

The code source used in this research has either been altered from the original topic or written. Besides code from the original topic code, some code was from Raymond Hettinger's PyBay 2017 keynote talk (https://pybay.com/site_media/slides/raymond2017-keynote/async_examples.html) that included coroutines. The single-threaded example used and documented in Figure 9 is very close to the original code. Updates for the new version of Python and optimizations for string addition were tested as part of the conversion. The optimizations were not significant when compared to the original code. Despite the changes, this code functions as the original code did.

A loading and testing system was developed to run the tests involved with this research. The loading system includes function tagging that allows imports and related outputs to state the algorithm's version. In all, the code base contains twenty-two different attempts for optimization and nine different versions of code matched with these attempts. Five of these attempts are documented in this research. The four versions are code for single-threading, multi-processing, multi-threading and the method tagged as mk13. The last attempt was to code to document the effects of queue size, and not expected to be successful. The rewrite of the single-threaded method has fifty-three lines. The multi-processing and multi-threading have 83 and 120 lines of code, respectively. The mk13 code has 768 lines of code in the client portion and 1437 in the server portion. The client and server were tested on the same process and, therefore, the same server. The idea behind client and server is one for understanding rather than a statement of infrastructure.

The code base for this project shows 22456 lines of code, while the original code has 5116 lines, and the source from the PyBay talk has 142 lines of code. There was significant work developing the mk13 code, but the single-threading, multi-processing, and multi-threading code goals were aimed to stay as close to the original code as possible. The research wanted a baseline to understand the difference between single-thread and multi-processing or multi-threading methods. With that baseline, the results of the mk13 code could be better analyzed, and understandings with recommendations for the effectiveness of these methods could be created.

6.2 Data Size Effects on Execution Times

While object instantiation makes this solution inappropriate for small datasets, the size of the object created is a more significant factor for the time taken. Socket communication between processes performed poorly when the information passed through the socket was too large or very small. The smaller batches used for processing helped solve bottlenecks but also solved issues with queues processing data slowly. Pickling of information takes time, and a queue can cause time loss when using tiny pieces of information, such as an eight-character string. While results varied, testing using the virtual server showed that the new code was shifting in success direction using twenty files. As the number of files increased, the methods for processing hit a consistent state. Table 8 lists the results of the different methods. Unless stated, all average times of execution were at least 7 test runs. For example, in the table below, a dataset of 10 indicates that the number of files were processed at least 70. The new piece of code named mk13 is slower than the single-threaded method with small amounts of processing but increases in efficiency as the dataset grows. Refer to Table 3 for size information about the processing.

Table 8
Virtual Server Dataset Size Runtimes

Dataset Size	Method	Average Time	Standard Deviation	Time Per File
10	Single Thread	15.5	0.3403	1.55
	Multi Process	7.215	0.09143	0.7215
	Multi Thread	25.18	0.1727	2.518
	Mk13	17.24	0.0923	1.724
20	Single Thread	36.36	0.5064	1.818
	Multi Process	13.01	0.1959	0.6505
	Multi Thread	59.98	0.3048	2.999
	Mk13	29.2	0.2423	1.46
25	Single Thread	47.72	2.301	1.909
	Multi Process	20.39	0.3329	0.8156
	Multi Thread	72.13	1.118	2.885
	Mk13	45.54	3.503	1.822
50	Single Thread	81.22	1.665	1.624
	Multi Process	29.25	0.4459	0.585
	Multi Thread	124.9	0.6744	2.498
	Mk13	67.07	0.5588	1.341
100	Single Thread	164.5	6.923	1.645
	Multi Process	63.46	2.002	0.6346
	Multi Thread	238.8	2.855	2.388
	Mk13	132.9	2.627	1.329
200	Single Thread	356.6	5.758	1.783
	Multi Process	124	2.916	0.62
	Multi Thread	514.3	4.593	2.571
	Mk13	246.2	3.248	1.231

The method using threading performs the worst. The mk13 method starts with threading and then uses parallel execution to perform the hard drive reads and the CPU processing. The data indicates a startup cost for this method that must be overcome before results are faster than single-threaded execution. The queues to communicate back and forth between processes are a source of delay. Tests were run to determine the effects of queue size on this process. Over a five-file run, different queue sizes were tested. The data size stayed the same, but the size of the message across the queue changed. The results shown in Table 9 demonstrate why mk13 prefers reducing the size of messages for communication between processes. As the message size to the internal queue increases, the activity of converting a message to bytes and pickling the data for serialization increases non-linearly.

Table 9

Multi-processing Queue Message Size Comparison

Number of Messages	Time Taken
10	64.81
100	65.03
1000	20.85
10000	14.66
100000	14.84

The code produced in this research optimizes the processing of raw information to feed into larger data structures. The data structures needed must return to the main process for further writing. Utilizing disk writing to avoid inter-process communication was not a viable solution. The mk13 code functions like a server/client model using coroutines and sockets as the initial communication between the concurrent processes.

6.3 Explanation of the mk13 Algorithm

The mk13 algorithm mixes the different types of parallel processing available in Python. The typical implementation favours multi-processing and then multi-threading. Implementing multi-processing can reduce GIL activation as it creates a boundary that the GIL will not cross. In many scenarios, if remediations aren't taken, a shared object could lock the GIL and block the entire application from further actions until the object is available. The results presented in the next section show that multi-threading delays are favourably reduced with processor speed upgrades.

Multi-processing is used to start the file load process in the mk13 algorithm. The multiprocessing can set up queues and attach them to sockets for communications with other processes. Mk13 follows a client/server structure. The server portion of the code set up Linux-based sockets for communication between the client and server processes. The data being transferred is primarily ASCII strings representing HEX. The communication between processes also uses ASCII to control flow. A socket of 256 bytes was be opened between the client and server. The client sends a command string that lists what is being sent, including the size and then wait for a response. The grams are sent as small dictionaries to the server. The sockets are simple and cannot communicate in virtual channels in two directions at once. If two ends of the socket send communications without either side receiving, the socket breaks with no way to recover. The management overhead is on the client as it controls the data flow. The server will indicate that it is done with a request. The client limits the number of requests to the server. The server communicates that tasks are complete, and the client process will send new tasks. This management is done through separate queues and sockets than what is required for the data transfer. This management is a part of the overhead observed when this algorithm uses smaller datasets.

The client process reads the file, obtain the grams from the file and place the grams in a dictionary. At this point, the mk13 diverges from the multi-processing method. The multi-processing method would lock the entire dictionary to add the grams processed by the file read. It is impossible to add to the dictionary simultaneously while reading a file, as that would require a dictionary that can take values without locking. There was a requirement for additional code in the mk13 implementation to limit the client from sending data, as the client could overwhelm the server process. Earlier implementations attempted a simple queue system in which the client read the file and put data in the queue. The server process would add values to the dictionary that it received. The server process became quickly overwhelmed as the amount of data it received wasn't based on slower hard drive reads but also the sliding window generation that quadrupled the data per file. The client would send all the grams to the server and then wait for the finished dictionary. The queues and sockets are quickly shut down and left to garbage collection for cleanup. Flow control was stable on the server class machine and the consumer machine. Stability issues arose on some slower GCP platforms as the code relies on some activities to finish before others. For the GCP E4 standard hard drive test, an average of 47.03 seconds was paired with a standard deviation of 29.18 seconds. This variability caused some lockups during the execution of the algorithm. Data would be put onto a socket, and then execution would loop faster for either the server or client and then the server would read the data off the socket, leaving the client waiting forever.

On startup, the server process communication spawns a connection broker that uses asyncio. Asyncio allows coroutines (https://docs.python.org/3/library/asyncio-task.html) to start separate tasks for each message received. The messages in this part of the code are to start the conversation between the client and server. The server process uses coroutines to wait for messages and then creates sockets for data communication. Since coroutines are being utilized, the server process and receive grams from other client requests. After the data is transferred, the data is placed on an idle queue attached to a process responsible for combining the incoming data into the larger dictionary. This part of the code allows the client to input into the dictionary without locking the process.

Besides the connection broker, the server process starts five collector processes and five sorter queues. These are created for moving grams into the combined dictionary. Five processes are created to wait for incoming data through the collector and sorter queues. There is a dictionary for each process. The data from the communication broker must be cleared as quickly as possible to avoid bottlenecks. A separate process is spawned to take data from the connection broker and then distribute the data. This allows the connection broker to continue accepting more connections. The data is then distributed to the five sorter queues. Each sorter queue receives the entire communication without filtering to avoid bottlenecks in the processing. Each of the sorter queues is processed by a matching process. Each sorter is responsible for one dictionary. The dictionary only contains a set of allowed entries.

Each dictionary was to represent a specific piece of the final dictionary. Since the incoming data was known, each dictionary was assigned a grouping of letters representing one-fifth of the total data. This could be accomplished more generally using hashes, but it was faster to assign number and letter ranges manually to each dictionary. For example, dictionary “a” would contain values starting with 0, 1 or 2. Dictionary “b” would contain values starting with 3,4, or 5. Each sorting process receives a copy of the incoming grams. This results in every incoming gram being processed five times. This is a trade of CPU resources for speed. The multi-processing is wastefully repeating the same task but allows for eliminating bottlenecks that would slow the process.

The server start releases queues after the client sends all the data. When the smaller dictionaries on the server contain all the data, the five dictionaries are combined into one extensive dictionary. The Python implementation of dictionary combinations is extremely fast. Analysis of the algorithm showed that high-cost activities, compared with the multi-processing method, are the serialization of data and related picking for communication. The last communication between the server and client is notable but does not increase rapidly as the dataset size increases. If queue and sockets costs were lower, this algorithm could likely exceed the performance of the multi-processing method.

6.4 Time Results for Pre-Processing Data

Multi-threading was typically a poor performer for this activity. The GIL was activating during the processing, causing locks which delayed the execution. This was expected as threads are suitable for I/O, but the heavy CPU activity would cause blocking. The advantage this method has for I/O did not show any advantages in the results. This indicates that the CPU portion of the processing is higher than the requirements for I/O. The GCP E2 Standard Drive showed a high standard deviation for all processing methods. Only five tests were performed for the single-threaded method. Compared with the single-threaded method, the times are worse or marginally better. Changing the CPU speed or number of processors influences timing. The faster processors can overcome the GIL issues and have the ability to beat the times of the single-threaded method. Hard drive speed appears to be not a significant factor in determining the success of this method. The over 2200% speed difference shown in the libaio random read test is not reflected in the results between the consumer hard drive and the consumer SSD tests.

Table 10
Single Vs Multi-Thread Execution Times

Environment	Single Thread Average	Single Thread Standard Deviation	Multi-Thread Average	Multi-Thread Standard Deviation
Server Class Host	36.36	0.5064	59.98	0.3048
Server Class Virtual	34.36	1.958	56.48	0.9836
Consumer Hard Drive	23.79	0.1774	23.75	0.194
Consumer SSD	24.13	1.343	23.98	0.3841
GCP E2 Standard Drive	26.7	1.102	40.91	14.56
GCP E4 Standard Drive	26.36	0.9693	28.24	0.2134
GCP E4 SSD	29.51	1.269	31.74	0.6108
GCP Compute Standard Drive	21.7	3.13	21.52	1.076
GCP Compute SSD	18.76	0.1403	19.39	0.1135

The mk13 algorithm and the multi-processing method are the most viable options when choosing how to process this type of data. The mk13 shows it can perform better under edge cases in the cloud. This would exploit a lucky condition the cloud platform produces and shouldn’t be relied upon. The average indicates that the mk13 algorithm had issues performing on the cloud platform. The tests from the server class virtual machine indicated that the number of processors did not significantly alter the results of the timing of the resources on the server that remained available. The server class virtual machine was tested with a three CPU architecture with no significant change in performance. There may be a difference between GCP’s implementation of virtual systems and this research’s implementation. While the server class virtual machine had more processors assigned to it, all systems exceeded the number of processors available for the algorithm. The number of parallel processes varies due to the connection broker attempting to run as few processes as possible to increase availability. With the mk13 algorithm’s management, the number of processes required to handle different queue paths, and the multi-processing of the client’s file processing, the number of concurrent activities will exceed most systems. This overallocation could cause poor performance in the cloud and superior performance occasionally seen in the GCP E2 standard drive and GCP E4 standard drive tests. Another reason for the poor performance could be the cloud platform not being tuned to handle local virtual sockets needed for communication between the client and server process. The mk13 algorithm outperformed the single-thread and multi-thread methods outside the cloud environment. The host and virtual machine operating systems variably consume resources. The variance in the mk13 results compared to other environments showed that the algorithm did take advantage of available resources better than other methods.

Table 11

Multi-Process Vs mk13 Execution Times

Environment	Multi-Process	Multi-Process Standard Deviation	mk13 Average	mk13 Standard Deviation	mk13 Best
Server Class Host	13.01	0.1959	29.2	0.2423	28.92
Server Class Virtual	12.55	0.2243	28.27	0.1288	28.04
Consumer Hard Drive	8.828	0.07998	18.84	0.1168	8.717
Consumer SSD	9.034	0.3948	18.89	65.82	18.64
GCP E2 Standard Drive	77.43	65.82	77	36.38	13.73
GCP E4 Standard Drive	18.38	0.1738	47.03	29.18	13.73
GCP E4 SSD	21.55	0.5469	45.83	0.8538	44.3
GCP Compute Standard Drive	14.46	0.5952	29	0.671	28.28
GCP Compute SSD	13.16	0.03502	26.47	0.3397	26.092

6.5 Hard Drive Effects on parallel pre-processing

The data shows that using the same CPU but a different hard drive will not significantly change the time taken for execution. This value stability is shown in the results between the two consumer drive tests and suggested by the GCP compute tests. The speed of the hard drives on the local virtual host was consistent, while the speed on the consumer test system was drastically different. If a speed requirement for CPU speed is met, it will negate the speed difference between a slower hard drive and a faster SSD drive. The line charts shown in Figure 12 and Figure 13 break up the data between the system with slower hard drives and the system with faster hard drives. All methods of executing this task were the best at some point when observing all systems. Notably, in mid to high-range CPU speed increased, multi-threading went from the worst performer to the second-best performer for a faster hard drive, and the results on the slower hard drive showed increased time compared to the other methods. The other methods showed more expected results by matching changes in performance consistently across the different hard drives. In contrast, the times with methods other than multi-threading were changed in scale and not in direction. For example, the multi-processing method changed from performing better to performing worse to then performing better with both the slower and faster hard drive.

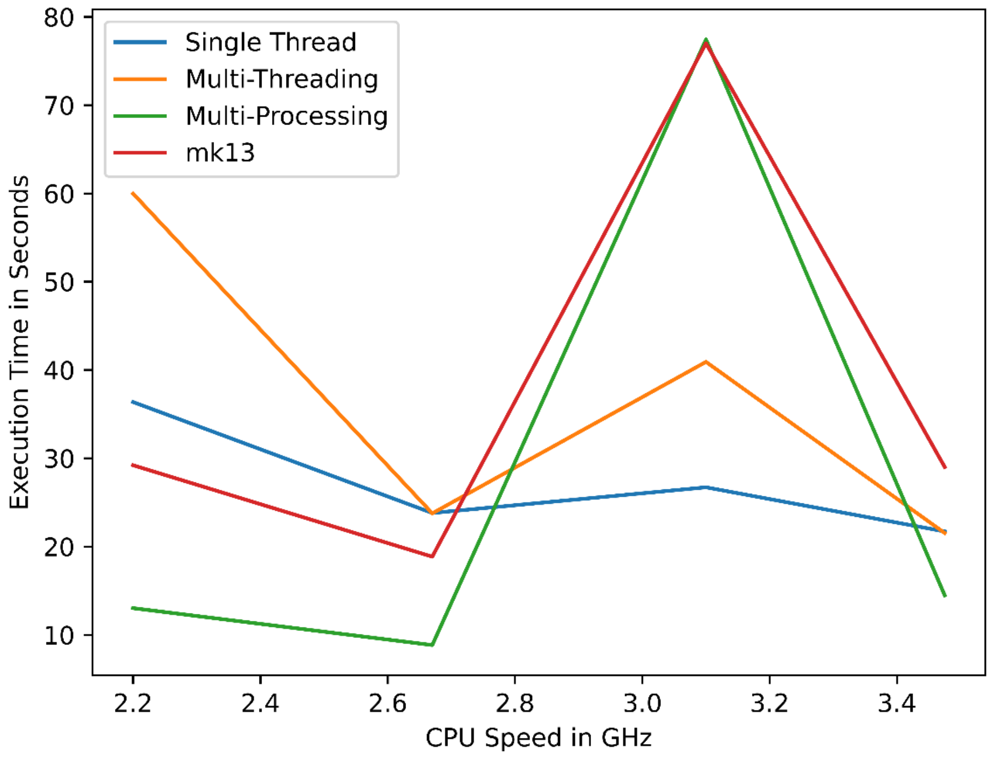


Figure 132 Execution Times Using Slower Hard Drives

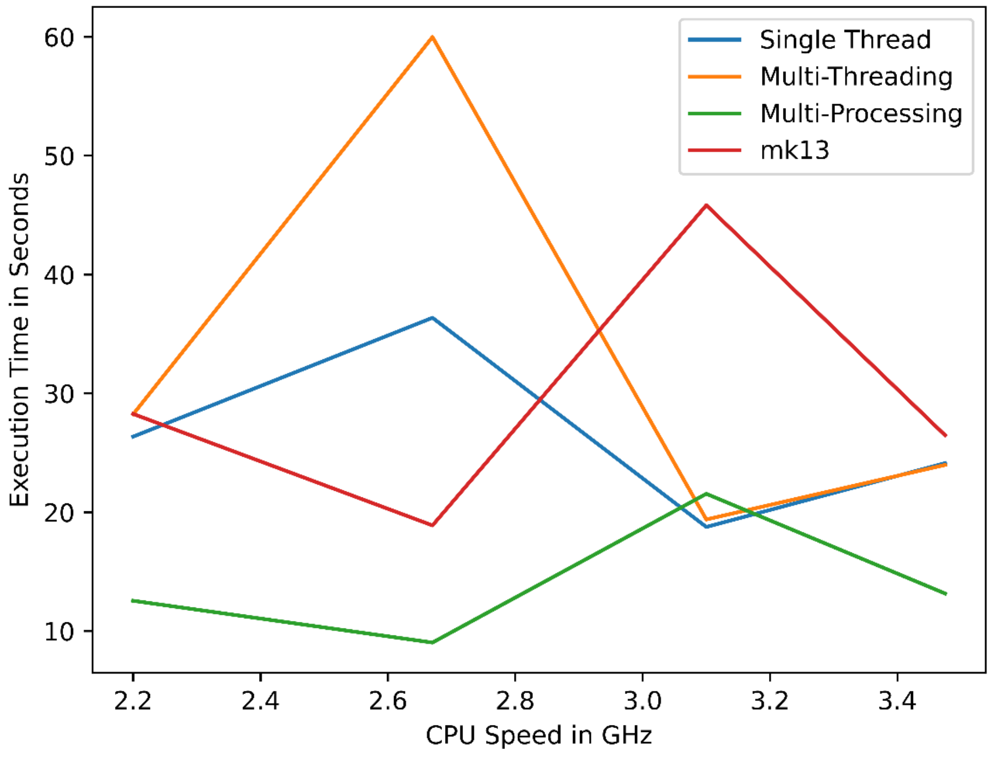


Figure 13 *Execution Times Using Faster Hard Drives*

Chapter 7. Conclusion

The environment to produce results for malware machine learning resources must be specific for optimal results. This research found that virtualization with QEMU using local machines could allow researchers to separate their environments without having significant losses to production. Using a secondary drive does not cause overhead running processes and allows researchers to swap environments quickly. Additionally, the second drive allows machines to take the malware offline while updating. This research used a local server to upgrade the required Python components. Still, any workflow that involves upgrading Python libraries benefits from having the ability to remove malware from a system quickly.

The mk13 algorithm presented superior results in execution time in particular instances, but runs slower in more modern environments. As the pre-processing of the files would not be repeated, and implementers of a system would rather have the average time be superior to the best time. This research shows that system implementers with existing slow code using multi-threading can have significant gains and loses depending on the environment. This could allow implementers to make cost-benefit decisions between hardware and code re-writing. Hard drive speeds were also evaluated and showed the cost-benefits of increasing hard drive speeds may not show returns for pre-processing large amounts of small files that require high CPU loads.

The choice of parallel processing technology must be weighed against the hardware used despite any assumptions based on the isolation provided by the Python virtual machine. The results show that increased CPU and hard drive speed will only sometimes decrease execution times for parallel processing. Python internal scheduling of parallel processes is a prime suspect for causing the drift in expected time results with the timing for all the algorithm executions. The Python core developer team flagged the internal scheduling algorithms for improvement, and there has been some success. As part of this improvement, the process scheduler has code to address the locking after a thread is forced to give up on execution. The complaint targeted at the scheduler was that it would not check if a thread were becoming active just after the thread was forced not to be active.

Scalene was used as part of this research to identify issues with the design of the mk13 algorithm. The tool showed that a viable solution for processing this dataset was focused on the high CPU usage during processing. Unfortunately, it is challenging to measure some Python internals. Hooking into some internal Python structures changes the internal structure and changes the result. When using Scalene, the timing results were much higher than without the tool, which prompted using a simple timer for the results. With the aid of Scalene, the mk13 algorithm beat the single-threaded method for execution. As this research aims to help future research, this is a success.

To better understand the difference in timing results between the original topic statements and the current research, different environments were used to verify the new algorithm, and other methods were used as part of benchmarking. The research's due diligence shows that this dataset and processing can react negatively to increased CPU and hard drive speeds. This discovery can alert future researchers to test their environment as a cheaper cloud environment may perform better depending on the code used by the next researcher. This research aimed to understand how to process data for input into machine-learning algorithms. When processing varied files with high CPU requirements, the results support that a simple multiprocessing method will reliably increase performance. If an external security company has a requirement to stream files, the structures in mk13 could provide the best performance. This research showed that hardware and environment changes could change results significantly enough to justify code changes. With large, small file datasets, the initial urge to prioritize hard drive resources needs to be corrected, and prioritization of CPU resources will show a greater return.

References

[1] C. C. for C. Security, “Canadian Centre for Cyber Security,” *Canadian Centre for Cyber Security*, Aug. 15, 2018. https://cyber.gc.ca/en/guidance/ransomware-how-prevent-and-recover-itsap00099 (accessed Aug. 09, 2021).

[2] “Cybersecurity News and Events | Mass.gov.” https://www.mass.gov/info-details/cybersecurity-news-and-events (accessed Aug. 09, 2021).

[3] “CPU platforms | Compute Engine Documentation,” Google Cloud. https://cloud.google.com/compute/docs/cpu-platforms (accessed May 17, 2023).

[4] D. Cassel, “Guido van Rossum’s Ambitious Plans for Improving Python Performance,” The New Stack, May 23, 2021. https://thenewstack.io/guido-van-rossums-ambitious-plans-for-improving-python-performance/ (accessed Nov. 30, 2022).

[5] “python/Objects/dictobject.c.” Python, Nov. 30, 2022. Accessed: Nov. 30, 2022. [Online]. Available: https://github.com/python/cpython/blob/e0f91deb5930ecb02e7f8ced9bd82609e6889fb0/Objects/dictobject.c

[6] H. S. Anderson and P. Roth, “EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models,” ArXiv180404637 Cs, Apr. 2018, Accessed: Jul. 13, 2021. [Online]. Available: http://arxiv.org/abs/1804.04637

[7] M. Goyal and R. Kumar, “Machine Learning for Malware Detection on Balanced and Imbalanced Datasets,” in 2020 International Conference on Decision Aid Sciences and Application (DASA), Nov. 2020, pp. 867–871. doi: 10.1109/DASA51403.2020.9317206.

[8] W. Aslam, M. M. Fraz, S. K. Rizvi, and S. Saleem, “Cross-validation of machine learning algorithms for malware detection using static features of Windows portable executables: A Comparative Study,” in 2020 IEEE 17th International Conference on Smart Communities: Improving Quality of Life Using ICT, IoT and AI (HONET), Dec. 2020, pp. 73–76. doi: 10.1109/HONET50430.2020.9322809.

[9] X. Wang, J. Liu, and X. Chen, “Microsoft Malware Classification Challenge (BIG 2015): First Place Team: Say No to Overfitting,”

[10] “Glossary — Python 3.11.0 documentation.” https://docs.python.org/3/glossary.html#term-global-interpreter-lock (accessed Dec. 01, 2022).

[11] M. Dhalaria and E. Gandotra, “A Framework for Detection of Android Malware using Static Features,” in 2020 IEEE 17th India Council International Conference (INDICON), Dec. 2020, pp. 1–7. doi: 10.1109/INDICON49873.2020.9342511.

[12] I. Shiel and S. O’Shaughnessy, “Improving file-level fuzzy hashes for malware variant classification,” Digit. Investig., vol. 28, pp. S88–S94, Apr. 2019, doi: 10.1016/j.diin.2019.01.018.

[13] R. Ito and M. Mimura, "Detecting Unknown Malware from ASCII Strings with Natural Language Processing Techniques," in 2019 14th Asia Joint Conference on Information Security (AsiaJCIS), Aug. 2019, pp. 1–8. doi: 10.1109/AsiaJCIS.2019.00-12.

[14] Q. Jerome, K. Allix, R. State, and T. Engel, "Using opcode-sequences to detect malicious Android applications," in 2014 IEEE International Conference on Communications (ICC), Jun. 2014, pp. 914–919. doi: 10.1109/ICC.2014.6883436.

[15] P. Agrawal and B. Trivedi, "A Survey on Android Malware and their Detection Techniques," in 2019 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT), Feb. 2019, pp. 1–6. doi: 10.1109/ICECCT.2019.8868951.

[16] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket," presented at the Symposium on Network and Distributed System Security (NDSS), Feb. 2014. doi: 10.14722/ndss.2014.23247.

[17] "Android malware dataset for machine learning 1." figshare, Feb. 05, 2018. doi: 10.6084/m9.figshare.5854590.v1.

[18] "Elastic Malware Benchmark for Empowering Researchers." elastic, Aug. 01, 2021. Accessed: Aug. 02, 2021. [Online]. Available: https://github.com/elastic/ember

[19] P. Agrawal and B. Trivedi, "Evaluating Machine Learning Classifiers to detect Android Malware," in 2020 IEEE International Conference for Innovation in Technology (INOCON), Nov. 2020, pp. 1–6. doi:10.1109/INOCON50539.2020.9298290.

[20] D. A. Quist and L. M. Liebrock, "Visualizing compiled executables for malware analysis," in 2009 6th International Workshop on Visualization for Cyber Security, Oct. 2009, pp. 27–32. doi: 10.1109/VIZSEC.2009.5375539.

[21] "F-Droid / Client," GitLab. https://gitlab.com/fdroid/fdroidclient (accessed Aug. 01, 2021).

[22] N. Zhang, Y. Tan, C. Yang, and Y. Li, "Deep learning feature exploration for Android malware detection," Appl. Soft Comput., vol. 102, p. 107069, Apr. 2021, doi: 10.1016/j.asoc.2020.107069.

[23] O. F. T. Cavli and S. Sen, "Familial Classification of Android Malware using Hybrid Analysis," in 2020 International Conference on Information Security and Cryptology (ISCTURKEY), Dec. 2020, pp. 62–67. doi: 10.1109/ISCTURKEY51113.2020.9308003.

[24] K. Aktas and S. Sen, "UpDroid: Updated Android Malware and Its Familial Classification," 2018, pp. 352–368. doi: 10.1007/978-3-030-03638-6_22.

[25] S. Alam and I. Sogukpinar, "DroidClone: Attack of the Android Malware Clones - A Step Towards Stopping Them," Comput. Sci. Inf. Syst., vol. 18, no. 1, pp. 67–91, Aug. 2020, doi: 10.2298/CSIS200330035A.

[27] C. Galen and R. Steele, “Performance Maintenance Over Time of Random Forest-based Malware Detection Models,” in 2020 11th IEEE Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON), Oct. 2020, pp. 0536–0541. doi: 10.1109/UEMCON51285.2020.9298068.

[28] X. Pei, L. Yu, and S. Tian, “AMalNet: A deep learning framework based on graph convolutional networks for malware detection,” Comput. Secur., vol. 93, p. 101792, Jun. 2020, doi: 10.1016/j.cose.2020.101792.

[29] I. Yoo, “Visualizing windows executable viruses using self-organizing maps,” in Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security - VizSEC/DMSEC ’04, Washington DC, USA: ACM Press, 2004, p. 82. doi: 10.1145/1029208.1029222.

[30] Z. Ren, G. Chen, and W. Lu, “Byte Visualization Method for Malware Classification,” in Proceedings of the 2020 5th International Conference on Machine Learning Technologies, Beijing China: ACM, Jun. 2020, pp. 136–140. doi: 10.1145/3409073.3409093.

[31] T. M. Kebede, O. Djaneye-Boundjou, B. N. Narayanan, A. Ralescu, and D. Kapp, “Classification of Malware programs using autoencoders based deep learning architecture and its application to the microsoft malware Classification challenge (BIG 2015) dataset,” in 2017 IEEE National Aerospace and Electronics Conference (NAECON), Jun. 2017, pp. 70–75. doi: 10.1109/NAECON.2017.8268747.

[32] Susanto, D. Stiawan, M. A. S. Arifin, M. Y. Idris, and R. Budiarto, “IoT Botnet Malware Classification Using Weka Tool and Scikit-learn Machine Learning,” in 2020 7th International Conference on Electrical Engineering, Computer Sciences and Informatics (EECSI), Oct. 2020, pp. 15–20. doi: 10.23919/EECSI50503.2020.9251304.

[33] Pei Kexin, Cao Yinzhi, Yang Junfeng, and Jana Suman, “DeepXplore: automated whitebox testing of deep learning systems,” GetMobile Mob. Comput. Commun., Jan. 2019, doi: 10.1145/3308755.3308767.

[34] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas, “Malware Detection by Eating a Whole EXE,” ArXiv171009435 Cs Stat, Oct. 2017, Accessed: May 06, 2021. [Online]. Available: http://arxiv.org/abs/1710.09435

[35] S. Alam, R. N. Horspool, and I. Traore, “MAIL: Malware Analysis Intermediate Language - A Step Towards Automating and Optimizing Malware Detection,” Novemb. 2013, p. 9, Nov. 2013.

[36] L. Liu, B. Wang, B. Yu, and Q. Zhong, “Automatic malware classification and new malware detection using machine learning,” Front. Inf. Technol. Electron. Eng.

[37] B. Kang, S. Y. Yerima, K. Mclaughlin, and S. Sezer, “N-opcode analysis for android malware classification and categorization,” in 2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security), Jun. 2016, pp. 1–7. doi: 10.1109/CyberSecPODS.2016.7502343.

[38] G. Canfora, A. De Lorenzo, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Effectiveness of Opcode ngrams for Detection of Multi Family Android Malware," in *2015 10th International Conference on Availability, Reliability and Security*, Aug. 2015, pp. 333–340. doi: 10.1109/ARES.2015.57.

[39] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith, "Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study," in 2016 IEEE Symposium on Security and Privacy (SP), May 2016, pp. 158–177. doi: 10.1109/SP.2016.18.

[40] T. Dullien and S. Porst, "REIL: A platform-independent intermediate representation of disassembled code for static code analysis," Jan. 2009.

[41] S. Cesare and Y. Xiang, "Wire – A Formal Intermediate Language for Binary Analysis," in 2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications, Jun. 2012, pp. 515–524. doi: 10.1109/TrustCom.2012.301.

[42] B. Zhao, J. Han, and X. Meng, "A malware detection system based on intermediate language," in *2017 4th International Conference on Systems and Informatics (ICSAI)*, Nov. 2017, pp. 824–830. doi: 10.1109/ICSAI.2017.8248399.

[43] A. B. Bugerya, I. I. Kulagin, V. A. Padaryan, M. A. Solovev, and A. Y. Tikhonov, "Recovery of High-Level Intermediate Representations of Algorithms from Binary Code," in 2019 Ivannikov Memorial Workshop (IVMEM), Sep. 2019, pp. 57–63. doi: 10.1109/IVMEM.2019.00015.

[44] C. Galen and R. Steele, "Evaluating Performance Maintenance and Deterioration Over Time of Machine Learning-based Malware Detection Models on the EMBER PE Dataset," in 2020 Seventh International Conference on Social Networks Analysis, Management and Security (SNAMS), Dec. 2020, pp. 1–7. doi: 10.1109/SNAMS52053.2020.9336538.

[45] L. Chen, S. Hou, and Y. Ye, "SecureDroid: Enhancing Security of Machine Learning-based Detection against Adversarial Android Malware Attacks," in Proceedings of the 33rd Annual Computer Security Applications Conference, Orlando FL USA: ACM, Dec. 2017, pp. 362–372. doi: 10.1145/3134600.3134636.

[46] Z. Mundargi, S. Bhatti, A. Chandra, A. Kamble, B. Jiby, and R. Arole, "PrePy - A Customize Library for Data Preprocessing in Python," in 2023 International Conference for Advancement in Technology (ICONAT), Jan. 2023, pp. 1–5. doi: 10.1109/ICONAT57137.2023.10080134.

[47] R. O. Odegua and F. O. Ikpotokin, "DataSist: A Python-based library for easy data analysis, visualization and modeling".

[48] F. Mancolo, "Eisen: a python package for solid deep learning." arXiv, Mar. 18, 2020. Accessed: Aug. 03, 2023. [Online]. Available: http://arxiv.org/abs/2004.02747

[49] O. Çelik, M. Hasanbaşoğlu, M. S. Aktaş, O. Kalıpsız, and A. N. Kanli, "Implementation of Data Preprocessing Techniques on Distributed Big Data Platforms," in 2019 4th International Conference on Computer Science and Engineering (UBMK), Sep. 2019, pp. 73–78. doi: 10.1109/UBMK.2019.8907230.

[50] A. Sivakumar and R. Gunasundari, “A Survey on Data Preprocessing Techniques for Bioinformatics and Web Usage Mining”.

[51] A. Karimi and M. H. Moattar, “Android ransomware detection using reduced opcode sequence and image similarity,” in 2017 7th International Conference on Computer and Knowledge Engineering (ICCKE), Oct. 2017, pp. 229–234. doi: 10.1109/ICCKE.2017.8167881.

[52] J. Kang, S. Jang, S. Li, Y.-S. Jeong, and Y. Sung, “Long short-term memory-based Malware classification method for information security,” Comput. Electr. Eng., vol. 77, pp. 366–375, Jul. 2019, doi: 10.1016/j.compeleceng.2019.06.014.

[53] I. A. Ridhawi, S. Otoum, M. Aloqaily, and A. Boukerche, “Generalizing AI: Challenges and Opportunities for Plug and Play AI Solutions,” IEEE Netw., vol. 35, no. 1, pp. 372–379, Jan. 2021, doi: 10.1109/MNET.011.2000371.

[54] S. J. Andriole, “Artificial Intelligence, Machine Learning, and Augmented Analytics [Life in C-Suite],” IT Prof., vol. 21, no. 6, pp. 56–59, Nov. 2019, doi: 10.1109/MITP.2019.2941668.

[55] Y. Gao, H. Hasegawa, Y. Yamaguchi, and H. Shimada, “Malware Detection Using Gradient Boosting Decision Trees with Customized Log Loss Function,” in 2021 International Conference on Information Networking (ICOIN), Jan. 2021, pp. 273–278. doi: 10.1109/ICOIN50884.2021.9333999.

[56] “Machine Types - Google Compute Engine — Google Cloud Platform.” https://web.archive.org/web/20150114113051/https://cloud.google.com/compute/docs/machine-types#highcpu (accessed Apr. 03, 2023).

[57] “CPU platforms | Compute Engine Documentation,” Google Cloud. https://cloud.google.com/compute/docs/cpu-platforms (accessed May 17, 2023).

[58] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi, “Microsoft Malware Classification Challenge.” arXiv, Feb. 22, 2018. doi: 10.48550/arXiv.1802.10135.

[59] “cpython/Objects/dictobject.c.” Python, Jun. 29, 2023. Accessed: Jun. 28, 2023. [Online]. Available: https://github.com/python/cpython/blob/3fb7c608e5764559a718ce8cb81350d7a3df0356/Objects/dictobject.c

[60] “Introduction — PyBay 2017 Keynote documentation.” https://pybay.com/site_media/slides/raymond2017-keynote/intro.html#martelli-model-of-scaleability (accessed Dec. 02, 2022).

[61] “This is Python version 3.12.0 alpha 2 - ceval.c.” Python, Dec. 04, 2022. Accessed: Dec. 04, 2022. [Online]. Available: https://github.com/python/cpython/blob/854a878e4f09cd961ba5135567f7a5b5f86d7be9/Python/ceval.c

[62] T. G. Mattson, T. A. Anderson, and G. Georgakoudis, “PyOMP: Multithreaded Parallel Programming in Python,” Comput. Sci. Eng., vol. 23, no. 6, pp. 77–80, Nov. 2021, doi: 10.1109/MCSE.2021.3128806

[63] A Jesse Jiryu Davis Grok the GIL Write Fast And Thread Safe Python PyCon 2017, (Mar. 18, 2018). Accessed: Dec. 04, 2022. [Online Video]. Available: https://www.youtube.com/watch?v=nUxzopuxWhk

[64] larryhastings, “larryhastings/gilectomy.” Nov. 30, 2022. Accessed: Dec. 04, 2022. [Online]. Available: https://github.com/larryhastings/gilectomy

[65] “Mailman 3 Python multithreading without the GIL - Python-Dev - python.org.” https://mail.python.org/archives/list/python-dev@python.org/thread/ABR2L6BENNA6UPSPKV474HCS4LWT26GY/ (accessed Dec. 04, 2022).

[66] S. Gross, “Python Multithreading without GIL.” Dec. 04, 2022. Accessed: Dec. 04, 2022. [Online]. Available: https://github.com/colesbury/nogil

[67] E. D. Berger, “Scalene: Scripting-Language Aware Profiling for Python.” arXiv, Jul. 25, 2020. doi: 10.48550/arXiv.2006.03879.

[68] “Python Performance Matters” by Emery Berger (Strange Loop 2022), (Oct. 06, 2022). Accessed: Dec. 04, 2022. [Online Video]. Available: https://www.youtube.com/watch?v=vVUnCXKuNOg

[69] B. Jacob, S. W. Ng, and D. T. Wang, “CHAPTER 2 - Logical Organization,” in *Memory Systems*, B. Jacob, S. W. Ng, and D. T. Wang, Eds., San Francisco: Morgan Kaufmann, 2008, pp. 79–115. doi: 10.1016/B978-012379751-3.50004-7.