Spring 3-2020

Network Traffic Analysis Framework For Cyber Threat Detection

Meshesha K. Cherie
Dakota State University

NETWORK TRAFFIC ANALYSIS FRAMEWORK FOR CYBER THREAT DETECTION

A dissertation submitted to Dakota State University in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Cyber Operations

March 2020

By

Meshesha K. Cherie

Dissertation Committee:

Dr. Wayne E. Pauli
Dr. Michael Ham
Dr. Tom Halverson
Dr. David Bishop

DAKOTA STATE UNIVERSITY

DISSERTATION APPROVAL FORM

This dissertation is approved as a credible and independent investigation by a candidate for the Doctor of Philosophy degree and is acceptable for meeting the dissertation requirements for this degree. Acceptance of this dissertation does not imply that the conclusions reached by the candidate are necessarily the conclusions of the major department or university.

Student Name: Meshesha Cherie

Dissertation Title: Network Traffic Analysis Framework for Cyber Threat Detection

Dissertation Chair/Co-Chair: Wayne Pauli   Date: April 17, 2020
Name: Wayne Pauli

Dissertation Chair/Co-Chair: Date:
Name:

Committee member: Date: April 17, 2020
Name: Tom Halverson

Committee member: Michael Ham   Date: April 17, 2020
Name: Michael Ham

Committee member: David Bishop   Date: April 17, 2020
Name: David Bishop

Committee member: Date:
Name:

Original to Office of Graduate Studies and Research
Acid-free copies with written reports to library

ACKNOWLEDGMENT

A very special gratitude goes out to Dr. Wayne E. Pauli, my advisor and dissertation committee chair. Thank you for your advice, guidance, support, and encouragement throughout the program. I am also grateful to the dissertation committee members, Dr. Michael Ham, Dr. Tom Halverson, and Dr. David Bishop for their valuable comments, suggestions, and encouragement. Finally, I would like to thank my family, relatives, and friends for supporting and encouraging me throughout my life. Thank you all!

ABSTRACT

The growing sophistication of attacks and newly emerging cyber threats requires advanced cyber threat detection systems. Although there are several cyber threat detection tools in use, cyber threats and data breaches continue to rise. This research is intended to improve the cyber threat detection approach by developing a cyber threat detection framework using two complementary technologies, search engine and machine learning, combining artificial intelligence and classical technologies.

In this design science research, several artifacts such as a custom search engine library, a machine learning-based engine and different algorithms have been developed to build a new cyber threat detection framework based on self-learning search and machine learning engines. Apache Lucene.Net search engine library was customized in order to function as a cyber threat detector, and Microsoft ML.NET was used to work with and train the customized search engine.

This research proves that a custom search engine can function as a cyber threat detection system. Using both search and machine learning engines in the newly developed framework provides improved cyber threat detection capabilities such as self-learning and predicting attack details. When the two engines run together, the search engine is continuously trained by the machine learning engine and grow smarter to predict yet unknown threats with greater accuracy. While customizing the search engine to function as a cyber threat detector, this research also identified and proved the best algorithms for the search engine based cyber threat detection model. For example, the best scoring algorithm was found to be the Manhattan distance. The validation case study also shows that not every network traffic feature makes an equal contribution to determine the status of the traffic, and thus the variable-dimension Vector Space Model (VSM) achieves better detection accuracy than n-dimensional VSM.

Although the use of different technologies and approaches improved detection results, this research is primarily focused on developing techniques rather than building a complete threat detection system. Additional components such as those that can track and investigate the impact of network traffic on the destination devices make the newly developed framework robust enough to build a comprehensive cyber threat detection appliance.

DECLARATION

I hereby certify that this dissertation constitutes my own product, that where the language of others is set forth, quotation marks so indicate, and that appropriate credit is given where I have used the language, ideas, expressions or writings of another.

I declare that the dissertation describes original work that has not previously been presented for the award of any other degree of any institution.

Signed,

Meshesha K. Cherie
____

Meshesha K. Cherie

TABLE OF CONTENTS

DISSERTATION APPROVAL FORM .... **II**

ACKNOWLEDGMENT .... **III**

ABSTRACT .... **IV**

DECLARATION .... **V**

TABLE OF CONTENTS .... **VI**

LIST OF TABLES .... **IX**

LIST OF FIGURES .... **X**

LIST OF CODE LISTINGS .... **XI**

INTRODUCTION **1**

1.1 BACKGROUND 1

1.2 DESIGN RESEARCH PROBLEMS 2

1.3 DESIGN THEORIES AND HYPOTHESES 3

1.4 RESEARCH GOALS AND SCOPE 5

1.5 STRUCTURE OF THE RESEARCH PAPER 5

LITERATURE REVIEW **7**

2.1 INTRUSION DETECTION SYSTEMS 8

2.1.1 Signature-Based Detection Systems *8*

2.1.2 Anomaly-Based Detection Systems *10*

2.2 MACHINE LEARNING ALGORITHMS IN INTRUSION DETECTION SYSTEMS 11

2.2.1 Machine Learning *11*

2.2.2 Machine Learning Tools *12*

2.2.3 Classification Algorithms *13*

2.2.4 Performance Analysis of Classification Algorithms *14*

2.3 SEARCH ENGINE AND VECTOR SPACE MODEL 16

2.3.1 Search Engine Libraries *18*

2.4 NETWORK TRAFFIC ANALYSIS TOOLS 19

2.4.1 Microsoft Message Analyzer *19*

2.4.2 Wireshark *20*

2.4.3 Other Tools *20*

2.5 LITERATURE REVIEW SUMMARY 21

2.5.1 Basic Design Specifications of the Proposed Framework *21*

SYSTEM DESIGN **23**

3.1 RESEARCH METHODOLOGY 23
3.2 TREATMENT EVALUATION AND SOLUTION VALIDATION 24
3.3 ARTIFACT DESIGN AND IMPLEMENTATION 25
3.3.1 Training Phase *26*
3.3.2 Detection Phase *27*
3.3.3 Development Tools *29*
3.3.4 Datasets *30*
3.3.5 Similarity Measurements *30*
3.4 PROTOTYPE PROJECT 32
3.5 DEPLOYMENT AND CONTINUOUS TRAINING 35
3.6 SUMMARY 36

SYSTEM DEVELOPMENT **38**

4.1 DATA PROCESSOR 38
4.1.1 Feature Engineering *39*
4.1.2 Data Balancer *40*
4.1.3 Data Partitioning *41*
4.2 CUSTOM SEARCH ENGINE DATA PREPARATION 45
4.2.1 Feature Name Representation *47*
4.3 CUSTOM SEARCH ENGINE 48
4.3.1 Feature Analyzer *49*
4.3.2 Custom Indexer *51*
4.3.3 Custom Similarity *52*
4.3.4 Custom Searcher *56*
4.4 BINARY CLASSIFIER ENGINE 58
4.4.1 Model Trainer *58*
4.4.2 Predictor *61*
4.5 DECISION ENGINE 63
4.5.1 Detection Engine *63*
4.5.2 Decision Algorithms *65*
4.6 ALERT SERVICE 66
4.7 MONITOR APPLICATION 69
4.8 SUMMARY 72

CASE STUDY **73**

5.1. INITIAL EVALUATION 73
5.1.1. Evaluation Metrics *73*

5.1.2. Model Evaluation 77

5.2. OPTIMIZATION 78

5.2.1 Feature Reduction 78

5.2.2 Euclidean Distance versus Manhattan Distance 81

5.2.3 Changing the Number of Nearest Vectors in KNN Algorithm 82

5.2.4 Variable Vector Dimension 83

5.2.5 ML.NET Binary Classifier Optimization 83

5.2.6 Class Balancing 84

5.3 FINAL EVALUATION 85

5.4 SYSTEM TESTING 86

5.4.1 Continuous Training 87

5.4.2 Enterprise Scale Testing 89

5.5 SUMMARY 94

CONCLUSIONS 95

6.1 CONTRIBUTIONS 95

6.2 LIMITATIONS AND FUTURE RESEARCH 98

6.3 SUMMARY 99

REFERENCES 101

APPENDIX A: CLASS DIAGRAMS 108

DATA PROCESSOR 108

FEATURE GENERATOR 108

ENGINE CONSTANTS 108

CORE ENGINE 109

ML ENGINE 109

SEARCH ENGINE 110

DETECTION ENGINE 110

ALERT SERVICE 111

UTILITY 111

APPENDIX B: LIST OF FEATURES 112

LIST OF TABLES

Table 1. Summary of Snort, Suricata, and Bro 10
Table 2. Comparison between Signature-Based and Anomaly-Based Intrusion Detection Techniques 11
Table 3. Classification Algorithms and Models 13
Table 4. Performance of Classification Algorithms 16
Table 5. Term Frequency Table of a Sample Text 45
Table 6. Feature Value Table of Sample Traffic Data Features 45
Table 7. Initial Model Parameters 77
Table 8. Initial Evaluation of Classification Models 78
Table 9. Model Evaluation Metrics Using Important Features 81
Table 10. Euclidean Distance versus Manhattan Distance 82
Table 11. Number of Nearest Vectors versus Accuracy 82
Table 12. Performance of Variable versus Fixed Dimension Vectors 83
Table 13. Optimized ML.NET Binary Classifier Model 84
Table 14. Detection Performance with 2:3 Attack-Benign Ratio Training Dataset 84
Table 15. Optimized Model Parameters 85
Table 16. Final Model Evaluation Result 85
Table 17. Load Testing Performance Comparison Table 90
Table 18. Performance of the Detection Engine Running as a Host-Based Detection System 94

LIST OF FIGURES

Figure 1. Conceptual design of the proposed framework. 24
Figure 2. High level model of the framework. 26
Figure 3. Training process. 26
Figure 4. Detection process. 28
Figure 5. Decision engine activity diagram. 28
Figure 6. Vectors in two-dimensional space. 31
Figure 7. Process flow design. 37
Figure 8. Data processor process flow diagram. 39
Figure 9. Lucene customization basic architecture. 49
Figure 10. Process flow diagram of the monitor application. 70
Figure 11. ROC curve. 75
Figure 12. Detection framework dashboard. 87
Figure 13. Project publish settings. 91
Figure 14. CICFlowMeter extracting features from live traffic. 93
Figure 15. Detection engine processing live traffic features. 93

LIST OF CODE LISTINGS

Listing 1. Partial implementation of VSM. 32

Listing 2. Partial implementation of VSM evaluation. 33

Listing 3. Implementation of similarity measures. 34

Listing 4. Partial implementation of prediction model and KNN algorithm. 35

Listing 5. Feature Extractor method implementation. 40

Listing 6. Implementation of the data preparation method. 43

Listing 7. Implementation of the data split method. 44

Listing 8. Custom search engine-compatible data preparation method. 47

Listing 9. Partial implementation of Feature Analyzer. 50

Listing 10. Implementation of the Feature Value Token Filter. 51

Listing 11. Assigning Feature Analyzer to the Index Writer. 51

Listing 12. Extending Lucene Similarity Base. 53

Listing 13. Implementation of Score method in Euclidean distance similarity. 55

Listing 14. Implementation of the Score method in Manhattan distance similarity.....56

Listing 15. Partial implementation of Searcher returning K nearest vectors. 57

Listing 16. ML Model Trainer implementation...59

Listing 17. Implementation of model evaluation...61

Listing 18. Feature Data class...61

Listing 19. Implementation of ML Model Predictor. 62

Listing 20. Partial implementation of the detection engine. 64

Listing 21. Implementation of threat detection method. 66

Listing 22. Log4net logger instance configuration...67

Listing 23. Custom MemoryAppender implementation. 68

Listing 24. Implementation of AlertLogWatcher class. 69

Listing 25. Partial implementation of SignalR Service Hub. 71

Listing 26. Partial implementation of client-side service..71

Listing 27. Implementation of evaluation metrics. 76

Listing 28. Partial implementation of Permutation Feature Importance (PFI)............80

Listing 29. Partial implementation of threat detection task. 88

Listing 30. Implementation of search engine retraining. 89
Listing 31. Live traffic feature file watcher implementation. 92

CHAPTER 1

INTRODUCTION

1.1 Background

This dissertation examines existing cyber threat detection approaches and introduces a new approach to cyber threat detection framework development using hybrid technologies such as search engines and machine learning. Although the primary purpose of search engines is for text mining, this research shows that they can serve as cyber threat detection frameworks as well.

When people and organizations connect to the Internet, there is a corresponding growth in risk as they increase their exposure to cyber threats. New threats are emerging, and the number of previously unseen malware and attack techniques is growing (Statista, 2019). Threat detection systems provide the frontline defense against cyber attacks. Using machine learning-based threat detection techniques enhances accuracy by detecting previously unseen attacks (Lin et al., 2018). Furthermore, using hybrid detection techniques increases the accuracy of detection (Samrin & Vasumathi, 2017).

Currently no cyber threat detection system combines search engine and machine learning techniques. This research focuses on the design and development of a hybrid cyber threat detection framework using a customized search engine model that functions as a cyber threat detection engine and a machine learning model. Search engines process a huge amount of data related to search queries every day. Search engines often generate results that match the search query, and the results are generally ranked based on relevancy to the search query. According to Yahoo research (Yin et al., 2016), ranking relevance is the most critical problem in search engines; search problems can be treated as filtering relevant search results from less relevant ones. Using machine learning classification algorithms such as decision trees on search engines can filter out bad search results and increases relevance (Yin et al., 2016). This research investigates the capabilities of a search engine used as a cyber threat detection system. The purpose of threat detection systems is to filter out threat activities from normal activities or to detect anomalies from regular activities. Anomaly-based threat detection systems use machine learning classification algorithms to classify threats and normal events. In addition to the custom search engine, the newly developed framework uses a machine learning-based cyber threat detection model to work together with the search engine. The main purpose of the machine learning model is to reinforce and continuously train the search engine.

The new framework analyzes network traffic behaviors to detect cyber threats. The framework is intended to be used by security analysts and security tool developers. Design science research methodology was used to design, develop, and evaluate the proposed framework. Each component of the framework has been developed using superior cutting-edge technologies such as .NET Core to target multiple platforms, multithreading for robust performance, the machine learning framework used in security tools, and WebSocket for real-time communication for alerting service. Mathematical analysis has been performed to design efficient algorithms when necessary. A case study has been employed to evaluate the framework and compare the customized search engine model with the machine learning model.

1.2 Design Research Problems

As a solution-oriented technical research project, this research designs artifacts that improve existing threat detection approaches and conducts experiments to answer empirical questions about the efficiency, accuracy, and performance of the designed artifacts. The following are a list of Knowledge Problems (KP) and Design Problems (DP) that are investigated and answered in the research:

A. KP: How do the signature-based and anomaly-based detections systems function?
    - What specific techniques are used?

B. KP: What are the advantages and disadvantages of signature-based and anomaly-based detection techniques?

C. DP: How can the threat detection framework be designed using search engine and machine learning techniques?

- KP: What are the benefits of using a search engine for a threat detection framework?

D. DP: Which system requirements and specifications need to be included in the proposed framework?

E. DP: Which design makes for the best cyber threat detection framework using search engine and machine learning?
    - What is the training data selection strategy?

F. KP: Are there standard evaluation criteria for intrusion detection techniques?
    - DP: What comparison criteria can be designed if they are not already available?

G. KP: How should the evaluation dataset be chosen?
    - DP: What is the evaluation data selection strategy?

H. KP: What is the performance of the proposed threat detection technique and the framework as a solution?
    - How fast are the algorithms?
    - How efficient are the components at handling large datasets?
    - How accurate is the detection system?
    - How broad is the training data coverage?

1.3 Design Theories and Hypotheses

A design science project uses prior knowledge, which includes design specifications, useful facts, and practical knowledge, to produce additional knowledge, called posterior knowledge (Wieringa, 2014). A design theory contains generalizations (treatment) about the effects of the interaction between an artifact and its context. The design theory describes the effect of an artifact in the context. Artifacts include devices, software, techniques, notations, and so on that are designed for the purpose of contributing to stakeholder goals (Wieringa, Daneva, & Condori-Fernandez, 2011). In this research, the primary artifacts include two engines—a custom search engine and a machine learning model—as well as algorithms and techniques such as indexing, searching, training, prediction, and binary classification within the context of cyber threat detection. The developed cyber threat detection framework is intended to be used primarily by security analysts and security tool developers (stakeholders). Wieringa's (2014) effect generalization states that "(an artifact designed like this) interacting with (a context satisfying these assumptions) produces (effects like these)" (p. 96). This can be expressed as (specifications of artifact) X (assumption about context) → effects. The effect is a generalization over a class of similar artifacts and a class of similar contexts. The following design theories using effect generalization provide the basis for this research and are thoroughly investigated within it. These serve as warrants and arguments used to prove the main research hypothesis, which is that the use of a search engine as a cyber threat detection system with machine learning techniques provides an improved cyber threat detection approach:

1. Search engines use similarity measures between indexed documents in a Vector Space Model (VSM) to search for matching documents for a given query (Turney & Pantel, 2010).
2. A search engine can be tuned up to efficiently process data within a specific context (for example, Google image search, Google Scholar). Therefore, a search engine can be customized to process network traffic data in cyber threat detection contexts.
3. As a search engine is primarily used for text data mining, machine learning can be applied to it for the best result (Chauhan et al., 2015). Therefore, machine learning can also be applied to the custom search engine specially designed to process network traffic data.
4. Network traffic data can be mapped to a special form of dataset that uniquely specifies the traffic, which is called feature extraction. The extracted features set can be arranged in a way that makes it suitable for indexing in a search engine. Based on the training data, the set of features data represents a single event and can be classified as safe and unsafe while indexing. Each feature set can be represented as a vector within the search engine VSM.
5. The search engine has a searcher component that can retrieve the best matches for a given query from the index based on configured similarity algorithms. Input traffic data can be a query for the searcher to return the top matching results. Similarity measures can be used to compare the similarity between documents. When VSM is used for classification, the nearest-neighbor algorithm can use similarity measures

(Turney & Pantel, 2010). Therefore, the matching results likely determine the type of the input traffic—that is, whether it is safe or a threat.

6. The search engine can be reinforced by machine learning techniques to increase its accuracy (Mitra et al., 2017; Yin et al., 2016) and to expand its capability to detect yet unknown malicious traffic.

1.4 Research Goals and Scope

The purpose of this research project is to design and develop a network traffic analysis framework for cyber threat detection primarily using search engine and machine learning techniques. The popular opensource Intrusion Detection Systems (IDS), Snort, Suricata, and Bro (Hu et al., 2017; Thongkanchorn et al., 2013), as well as machine learning-based detection techniques, are investigated to identify the basic design specifications for the proposed framework. This research is intended to improve cyber threat detection approaches by using hybrid technologies, such as a search engine that is specifically customized to classify traffic data, and a machine learning model in order for security analysts to analyze network traffic activities for the purpose of cyber threat detection.

The scope of this research comprises customizing search engine components such as the analyzer, tokenizer, indexer, and searcher, developing a machine learning based threat detection engine, and evaluating and validating these components. The research is limited to the development of a threat detection framework that can be used in security tools or can grow to a standalone cyber threat detection application and does not produce a complete end-to-end cyber threat detection appliance.

1.5 Structure of the Research Paper

This research paper has been divided into six chapters. As it involves the development of software, contents are organized to show the details of each major software engineering phase, such as requirement analysis, system design, system development, testing, and deployment. Chapters are organized as follows:

Chapter 1. Introduction: This chapter discusses the background of the research, research problems, theories and hypothesis, and research goals and scope.

Chapter 2. Literature Review: The literature review chapter provides a comparative analysis of existing cyber threat detection tools, intrusion detection approaches, network traffic analysis tools, and detection algorithms. This chapter also discusses existing machine learning and search engine technologies. Finally, this chapter lays out the basic specifications for the new cyber threat detection framework.

Chapter 3. System Design: This chapter incorporates research methodology as well as the new framework design from a software engineering perspective. Several architectural diagrams, algorithm designs, and use cases are discussed in this chapter. A prototype project analysis is included in this chapter to make design decisions on the core algorithms.

Chapter 4. System Development: This chapter discusses implementation of the new framework and its components as designed in Chapter 3. This chapter involves mathematical analysis and the development of algorithms and components.

Chapter 5. Case Study: This chapter discusses setting up evaluation metrics, evaluating of algorithms and components, optimizing techniques, and validating and deploying the new framework. Empirical analysis and performance results of the framework are discussed in this chapter.

Chapter 6. Conclusion: This chapter concludes with the overall outcome of the research, its contributions and limitations, and recommendations for future development.

CHAPTER 2

LITERATURE REVIEW

The National Institute of Standards and Technology (NIST) defines the term intrusion as the “unauthorized act of bypassing the security mechanisms of a system” (p. 104). The institute defines the term “threat” as “any circumstance or event with the potential to adversely impact organizational operations (including mission, functions, image, or reputation), organizational assets, individuals, other organizations, or the Nation through an information system via unauthorized access, destruction, disclosure, modification of information, and/or denial of service” (Kissel, 2013, p. 202). Intrusion detection is the formal process of detecting intrusions, generally characterized by gathering and analyzing information about abnormal usage patterns and processes in order to determine if a security breach or violation has occurred.

Before the development of Intrusion Detection (ID) tools, system administrators used to check user and device activities by sitting in front of screens. Then in the late 1970s and early ’80s, administrators were able to print activity logs although the manual log analysis was time consuming and inefficient at catching attacks in progress. In the early ’90s real-time intrusion detection systems that could detect attacks as they occurred emerged; in some cases, this allowed real-time responses to the attacks (Kemmerer & Vigna, 2002).

The first idea of ID was published in 1980 by James P. Anderson who outlined computer security problems within United States Air Force operations. The first model of a real-time Intrusion Detection System (IDS), the Intrusion Detection Expert System (IDES), was developed between 1984 and 1986 by Dorothy Denning and Peter Neumann. The tool was a rule-based system that detected known malicious activities. Then in the 1980s and ’90s, research about IDS was begun and funded by the US government, and IDS tools such as Discovery, Haystack, Multics Intrusion Detection and Alerting System (MIDAS), and Network Audit Directory and Intrusion Reporter (NADIR) were developed. In 1997 the first commercial real-time attack detection called RealSecure was developed for Windows NT 4.0. But as networks expanded and got faster, carrying out network-based attack detection became difficult. Therefore, to solve this problem, host-based intrusion detections such as TCP Wrappers, Tripwire, and Snort were introduced. Snort was introduced in 1998 for UNIX systems, and then was ported to Windows systems in 2000 (Bruneau, 2001).

2.1 Intrusion Detection Systems

Based on the scope of the detection, there are two types of IDS: Network Intrusion Detection Systems (NIDS) and Host-Based Intrusion Detection Systems (HIDS) (Samrin & Vasumathi, 2017). Based on the detection techniques, intrusion detection systems can be classified into two groups: signature-based detection and anomaly-based detection (Bello et al., 2015; García-Teodoro et al., 2009). In some research (Bello et al., 2015; Samrin & Vasumathi, 2017) signature-based detections are called misuse detections.

2.1.1 Signature-Based Detection Systems

Signature-based detection mechanisms search for predefined signatures or patterns within the analyzed data to determine an attack (García-Teodoro et al., 2009). There are several IDS in the cybersecurity industry, but the popular opensource IDS solutions are Snort, Suricata, and Bro (Hu et al., 2017; Thongkanchorn et al., 2013).

Snort: Snort is an opensource intrusion detection and prevention system originally developed in 1998. Snort is capable of real-time traffic analysis and packet logging with minimal disruption to running operations. Snort uses rule-based pattern-matching techniques to detect intrusions. Snort rules specify unique characteristics of the network traffic and trigger an alert when conditions met. In Snort, rules are divided into two parts: the header and the options. The rule header contains the action, protocol, IP addresses, and port numbers of the rule. The rule option part contains alert messages and information on which parts of the packet should be inspected (Cisco, 2019). As Snort is a signature-based detection system, it cannot detect previously unseen attacks. Developing a Snort rule, therefore, requires a core understanding of how the vulnerability works. Because Snort is widely known in the industry, extension works have also been created to optimize it. For instance, (Gómez et al., 2009) extended Snort by adding a statistical anomaly-based detection feature. The training model is stored in a MySQL database. The performance of this approach depends on the database operations and the statistical model.

Suricata: Introduced in 2009, Suricata is another popular signature-based detection system. Like Snort, it is an opensource threat detection engine using extensive rules and signature language. Suricata supports standard input and output formats like YAML Ain’t Markup Language (YAML) and JavaScript Object Notation (JSON), and it integrates with external systems like Logstash, Kibana, and different databases. Suricata signatures/rules consist of three parts: the action that determines when the rule matches; the header that defines the protocol, IP addresses, and direction of rules; and the rule options that define specific properties of the rules. Suricata offers compatibility with Snort rules (Suricata, 2019), and can handle larger volumes of traffic than Snort with similar detection capability (Albin, 2011). As Suricata is a multithreaded system, its detection capability is more accurate in multicore environments; however, Suricata has higher system overhead than Snort and is less accurate in stressed environments (Day & Burns, 2011). Suricata uses Lua scripting language, which provides the flexibility to create dynamic rules that would be difficult to create with Snort.

Bro: Bro, recently renamed Zeek, is a network security monitor system used to detect suspicious activity, measure performance, and troubleshoot. Bro is an opensource project that has been in development since 1995, but the project has been widely supported since 2010. Unlike Snort and Suricata, Bro focuses extensively on network analysis. Bro has the capability to log and store network activities in high-level terms (ASCII forms) for several application-layer protocols, such as DNS, FTP, HTTP, IRC, SMTP, SSH, and SSL. Bro uses standard log file formats, which is suitable for postprocessing with external log searchers and databases, such as Elasticsearch. Bro supports pattern-based intrusion detection and uses an event-based programming model for anomaly detection. Bro is a single-threaded application and runs on a single core; however, it supports clustered deployment so that multiple workers can process the traffic streams (The Zeek Project Revision, 2019). Table 1 summarizes the three popular intrusion detection systems.

Table 1. Summary of Snort, Suricata, and Bro

Snort	Suricata	Bro
Signature-based IDS.	Signature-based IDS.	Comprehensive network logging tool and Signature-based IDS.
Supports multithreading and multicore systems.	Supports multithreading and multicore systems.	Uses single thread and supports single core system.
Rule-based detection.	Rule-based detection. Uses Lua scripting for custom detection.	Rule-based detection. Uses Bro scripting for intrusion detection.
Rules are easy to write but challenging to adapt to complex threats with high-speed networks.	Has flexibility to write dynamic rule and handles complex threats.	Capable of recording detailed network behaviors. Deep-packet inspection is resource intensive.
Supports Linux, FreeBSD, OpenBSD.	Supports Linux, FreeBSD, OpenBSD, MacOS, Windows.	Supports Linux, FreeBSD, MacOS X.

2.1.2 Anomaly-Based Detection Systems

Anomaly-based detections identify a suspicious event from a security perspective by analyzing its behavior. Anomaly-based detection techniques can be classified into three groups based on the nature of the behavioral model processing: statistical-based, knowledge-based, and machine learning-based. Statistical-based detection is based on the metrics of the network traffic, such as traffic rate, number of packets per protocol, and number of IP addresses. The detection system compares the current traffic behavior with the previously trained statistical profile. Knowledge-based systems are expert systems intended to identify classes from the audit data according to a set of rules, parameters, or procedures. Machine learning-based techniques are based on models that enable the patterns analyzed to be categorized (García-Teodoro et al., 2009).

In anomaly-based detection systems, there are two phases: the learning phase and the detection phase. The detection system builds a system profile in the learning phase and compares the current system parameters with the one stored in the learning phase; if there is a deviation, an alert is reported. Therefore, anomaly-based detection systems can detect yet unknown attacks (Al-Jarrah & Arafat, 2015). Machine learning detection techniques such as

Neural Networks classify different forms of network attacks (Mowla et al., 2017). Using intrusion detection techniques based on Convolutional Neural Networks (CNN) improves the accuracy of detection system since they can extract enhanced behavior features (Lin et al., 2018). Table 2 summarizes the pros and cons of signature-based and anomaly-based intrusion detection systems.

Table 2. Comparison between Signature-Based and Anomaly-Based Intrusion Detection Techniques

Detection Technique	Advantages	Disadvantages
Signature-based	Provides good detection result for known attacks.	Not capable of detecting new attacks.
Anomaly-based	Capable of detecting new intrusion events.	Less accurate for known attacks. Resource intensive.
	Statistical-based: Prior knowledge of normal traffic activity is not required.	Can be misled or easily trained by an attacker.
	Knowledge-based: Flexible and scalable.	Depends on high-quality data, which is not easily available and time consuming.
	Machine learning: Flexible and scalable.	Lack of descriptive model. Resource intensive.

2.2 Machine Learning Algorithms in Intrusion Detection Systems

2.2.1 Machine Learning

There are several machine learning algorithms such as classification, logistic regression, etc. In machine learning the intrusion detection lies in the classification problem category. There are two machine learning techniques for data classification and clustering: supervised and unsupervised learning. The supervised learning method uses labeled datasets to learn the classification, whereas unsupervised learning finds similar groups within the training dataset (Kong et al., 2018).

2.2.2 Machine Learning Tools

Machine learning tools vary by the algorithms and techniques they support, programming languages they use, ease of implementation, and prediction accuracy of trained models. Machine learning tools share trained models. For instance, a TensorFlow trained model can be used by ML.NET (Microsoft, 2019a). Some of the machine learning tools that support classification algorithms are listed as follows:

Weka: Weka (Waikato Environment for Knowledge Analysis) is collection of data mining algorithms. It was first developed in C by the University of Waikato, New Zealand in 1997, then later rewritten in Java. It is composed of several data processing and classification algorithms such as classification, clustering, and regression (Choudhury & Bhowal, 2015; Hall et al., 2009).

TensorFlow: A machine learning tool developed by Google that focuses on deep neural networks, TensorFlow uses unified dataflow graphs to represent the computation in an algorithm, a shared state, and the operations that mutate that state. Several Google services use TensorFlow (Abadi et al., 2016).

Microsoft Cognitive Toolkit (CNTK): An opensource artificial intelligence toolkit developed by Microsoft, (Microsoft, 2019) claims that the CNTK toolkit is commercial-grade quality and compatible with many programming languages and algorithms. The CNTK is scalable for efficiency and capable of running on CPU, GPU, and distributed environments.

ML.NET: ML.NET framework is an opensource and cross-platform machine learning framework used to develop machine learning models using C# or F# languages. The ML.NET framework incorporates data loading, transformation, model training, and evaluation. Data transformation, normalization, and different machine learning algorithms, such as SymSGD, SDCA, FastTree, LightGBM, K-Means, SVM, and Averaged Perceptron, are supported by the framework. ML.NET works with other machine learning tools such as TensorFlow, Accord.NET, and CNTK (Microsoft, 2019a). Although ML.NET was initially released as recently as 2018 as an opensource machine learning component, the underlying machine learning components have been used for over a decade in Microsoft products such as Microsoft Defender Advanced Threat Protection (ATP), Windows Defender, and Anomaly Detection in Azure Stream Analytics (Microsoft, 2019d, 2020).

2.2.3 Classification Algorithms

In (Choudhury & Bhowal, 2015; Garg & Khurana, 2014) machine learning classifier techniques are grouped into several types based on their functionality and the machine learning algorithms they are using. The common classification groups used in both Choudhury and Bhowal, and Garg and Khurana include Bayes Classifier, Functional Classifier, Lazy Classifier, Meta Classifier, Multi-Instance Classifier, Rules Classifier, and Decision Tree-based classification. Table 3 shows a list of classification algorithms in recent research.

Table 3. Classification Algorithms and Models

Classification Algorithm	**Model**
SVM	Vector Space
K-Means	Euclidean Space Distance
C4.5	Decision Tree
J.48	Decision Tree
Random Forest	Decision Tree
Bayes Net	Graph model

SVM: Support Vector Machine (SVM) is a supervised learning algorithm that represents features data in the form of n-dimensional vectors and gives a specific score to each piece of feature data as the basis of evaluation (Qi et al., 2017). SVM was first introduced in 1963 but became widely used in deep neural network-based learning in the 1990s (Qi et al., 2017). The SVM algorithm uses hyperplanes to classify linear datasets. However, in real cases the dataset may be nonlinear. Therefore, SVM uses different kernels to transform initial features to higher dimensional space in order to address nonlinear classifying techniques. Type of kernels include Linear kernel, Polynomial, RBF (Radial basis function) kernel, and Sigmoid kernel (Kong et al., 2018).

K-Means: K-Means is an unsupervised algorithm to cluster data with similar properties using the Euclidean space distance measurement by finding the centroid of the clusters (Qi et al., 2017).

C4.5 Algorithm: Also called Statistical Classifier, this machine learning algorithm builds decision trees from the training data for classification (Singh & Agrawal, 2011).

J.48: This type of decision tree depends on the variables (features in the training data). The dependent variables (labels) are decided by the value of the connected nodes, which represent the independent variables (features). The root node contains the feature with the highest information gain to build the decision tree (Mehmood & Rais, 2016).

Random Forest: This decision tree-based algorithm uses several methods to obtain better prediction performance (Choudhury & Bhowal, 2015).

Bayes Net: This is a probabilistic graphical model based on the Bayes theorem to form a Bayesian network with each node representing a random variable. The edges between the nodes represent probabilistic dependencies among the random variables. Statistical methods are often used to compute the probability of the nodes (Choudhury & Bhowal, 2015; Singh & Agrawal, 2011).

2.2.4 Performance Analysis of Classification Algorithms

In a comparative analysis of machine learning algorithms (Singh & Agrawal, 2011), the C4.5 and Bayes Net showed better accuracy in IP traffic classification. The experiment used two different datasets with 2,800 data samples, out of which 300 samples were used for testing. Traffic classification using the C4.5 and Bayes Net algorithms resulted in about 94% accuracy. The experiment used Weka as a machine learning tool.

In another study (Choudhury & Bhowal, 2015), several algorithms were used to classify network traffic to detect anomalies using Weka. The training dataset consisted of 1,166 records with 42 features and the testing dataset consisted of 7,456 records. Analysis results show that Random Forest and Bayes Net algorithms yielded greater accuracy with 91% and 90%, respectively.

In an experiment using Weka and the NSL-KDD dataset with 94,000 training data instances and 48,000 instances for testing, the Random Forest classification algorithm was among the top-performing algorithms along with Rotation Forest, Random Tree, and Random Committee (Garg & Khurana, 2014). Each instance in both sets was composed of 41 features. The dataset contained four types of attacks: DoS (Denial of Service), R2L (Remote to Local), U2R (Unauthorized access to local superuser), and Probing (surveillance or others).

In another study (Sewak et al., 2018), Random Forest classification algorithm showed a greater accuracy than Deep Neural Network (DNN) algorithms for malware classification. The experiment used 11,308 malicious files and 2,819 benign files. The Adaptive Synthetic (ADASYN) technique was used to balance the training dataset classes. The analysis was performed on the opcodes of each instances using the Linux objdump utility. The training-testing dataset was randomly split to the ratio of 2:1. The Python libraries Sci-Kit and Keras with TensorFlow were used for Random Forest and DNN, respectively. Results showed that Random Forest, with an accuracy of 99.78%, slightly outperformed DNN, which had an accuracy of 99.21%.

A different study (Narudin et al., 2016) investigated 1,000 malware families with 49 different families out of the 1,260 MalGenome project dataset, and the top 20 free apps from Google Play store as benign apps. Eleven TCP/IP-based features were extracted from the network traffic for training and testing. The research results showed 99% malware detection accuracy with Random Forest classifier algorithm.

In a study performed on traffic data to classify attacks and normal traffic using supervised and unsupervised learning, K-Means classification was used for unsupervised learning, and SVM was used for supervised learning (Kong et al., 2018). The KDD’99 dataset, a widely used dataset in anomaly-based detection experiments, was used with 42 features extracted based on traffic flow (Mehmood & Rais, 2016). The experimental dataset consisted of four types of attacks: DoS, R2L, U2R, and Probing. Further, the result showed 91% accuracy with SVM and 83% accuracy using K-Means. However, SVM was slower to predict specific attacks in the dataset than the K-Means algorithm.

In a particular study of intrusion-detection research (Mehmood & Rais, 2016), SVM, Naïve Bayes classifier, J.48 decision tree, and decision table algorithms were used on the KDD99 dataset. The analysis result showed that each algorithm had different results in the classification of each of the attack classes (DoS, R2L, U2R, Probe) and normal traffic. No algorithm outperformed the others with a high true positive rate (TPR). The J.48 decision tree algorithm was found to have the highest accuracy and minimum classification error.

Table 4. Performance of Classification Algorithms

Algorithm	Training Data Count	Testing Data Count	Overall Accuracy
C4.5	2800	300	94%
Bayes Net	2800	300	94%
Random Forest	1166	7456	91%
Bayes Net	1166	7456	90%
Random Forest	14127	4700	99.78%
DNN	14127	4700	99.21%

As summarized in Table 4, decision tree-based classification algorithms provide good results in the context of network traffic analysis. Based on results compiled from different studies, Random Forest classification shows better accuracy over the other decision tree-based classification algorithms and DNN in network classification for intrusion detection.

2.3 Search Engine and Vector Space Model

Search engines are capable of efficiently processing large volumes of data and are known for fast information retrieval. Search engine processes have at least two major phases: indexing and searching. Data structure models such as VSM are used in search engines for efficient data processing and document matching. In search engines, the concept of VSM is used to represent each document mathematically as a vector in space. Document similarity is based on the notion that vectors that are closer to each other are semantically similar and vectors that are far apart are semantically less similar (The Apache Software Foundation, 2019).

VSM was first developed in 1975. The idea of VSM was to represent documents with vectors in a document space for efficient document retrieval (Salton et al., 1975). Vectors are common in Artificial Intelligence and cognitive sciences (Turney & Pantel, 2010). The VSM can be used in classification problems such as spam Short Message Service (SMS) filters in the telecommunication industry (Li & Zeng, 2016). In VSM, terms in the documents are represented by vectors, which consist of term weights to signify the semantic value of the term in the document. The commonly used weighting algorithms are Term Frequency (TF) and Inverse Document Frequency (IDF) (Li & Zeng, 2016). The TF-IDF weight is a statistical measure used to evaluate the importance of a term to the document in the collection. The Term Frequency (TF) is the count of terms in the document. IDF measures the uniqueness of the term with respect to the entire corpus (Pathak & Lal, 2017). IDF is computed from the total documents count and the count of documents that contain at least one occurrence of the term.

$$TF_i = log_2\left(tf_{ij}\right)$$

Where *tf_{ij}* represents the frequency of term *i* in document *j*.

$$IDF_i = log_2\left(\frac{N}{n_j}\right) + I = log_2(N) - log_2\left(n_j\right) + 1$$

Where *N* is the number of total documents and n_j is the count of documents that contain at least one occurrence of term *i*.

The measure of the distance between two vectors is the measure of similarity between them. The most popular distance measures for vectors includes Euclidean distance, Manhattan distance, Cosine similarity, Dice, and Jaccard Coefficient. Research shows that the Cosine similarity is the best formula for similarity measurement in search engines. With the Cosine formula, the length of the vectors is less relevant; what is important is the angle between them (Turney & Pantel, 2010). The Cosine of angle θ between vectors *A* and *B* can be expressed as:

$$\cos(\theta) = \frac{\mathrm{A} \cdot \mathrm{B}}{\| \mathrm{A} \| \| \mathrm{B} \|}$$

As noted in (Yin et al., 2016), retrieving the most relevant result is the core problem of commercial search engines. The problem of relevance in search engines is beyond text matching. The search problem can be treated as a binary problem to filter out irrelevant results from relevant ones. The research work that has been tested and deployed in the Yahoo commercial search engine shows that adding machine learning algorithms increases relevance. Decision tree algorithms such as Gradient Boosting Decision Tree (GBDT) with logistic loss can find a decision boundary that can classify bad URLs as opposed to relevant ones (Yin et al., 2016).

Search engines can be reinforced by machine learning for accurate search results (Chauhan et al., 2015). There are many machine learning algorithms that can work with VSM. If the machine learning algorithm uses real-valued vectors, it can use vectors from VSM. Machine learning algorithms can handle vector comparison, and vectors from VSM may also be used in semi-supervised learning (Ando & Zhang, 2005). In some studies, network intrusion detection systems using neural networks showed better detection results than known intrusion detection tools like Snort (Al-Jarrah & Arafat, 2015). Neural networks also achieved better results over other vector-based models (Hung et al., 2017).

2.3.1 Search Engine Libraries

There are several commercial and opensource search engines. Google, Bing, and Yahoo are among the top commercial search engines. Opensource search engines include Lucene, Indri, Terrier, Sphinx, and Xapian. Lucene, Indri, and Terrier are the popular search engine libraries that are tuned up for fast execution time (Mitra et al., 2017) and support multiple language interfaces.

Lucene: Lucene is an opensource fully featured search engine library with advanced analysis capabilities. First started in 1997 by Doug Cutting, it was later donated to the Apache Software Foundation in 2001 (Białecki et al., 2012). Offering VSM and efficient indexing, storage, searching, and ranking functionalities, Lucene can be customized. It has a scalable and high-performance indexing library. Its memory requirement ranges from 1MB through systems with several cores. Lucene also supports concurrent searches (The Apache Software Foundation, 2019). Lucene is used by AOL, Comcast, Disney, Wikipedia, Twitter, Netflix, Instagram, and other search engines (Białecki et al., 2012; Turney & Pantel, 2010).

Indri: An opensource search engine library that can provide text searches and structure query language, Indri supports multithreading and distributed search, and can run on a cluster of nodes for faster indexing and searching. It allows fine-grained control of searching and low-level access to document repositories (Van Gysel et al., 2017).

Terrier: An opensource Java-based search engine developed at the University of Glasgow, Terrier supports term dependence proximity models and supervised ranking models via learning to rank. Learning to rank in the Terrier search engine means using multiple calculated features in a uniform way during ranking, and learning an appropriate method to combine those features. The calculated features can be query-dependent or independent (University of Glasgow, 2019).

In a study that compared the two search engines, Lucene and Indri, Indri results were better than Lucene for short queries; however, Lucene is faster than Indri for long queries. The two search engines also yield different documents, especially for short queries (Turtle et al., 2012).

2.4 Network Traffic Analysis Tools

Network traffic analysis tools involve packet capturing and analysis. Packet capturing is a process of collecting packets as they travel over a network. Packet capturing takes place in kernel space, but analysis tools run in the user space (Gracia, 2008). There are several network traffic analysis tools; however, this section mainly reviews the popular network traffic analysis tools that support packet capturing, filtering, and analysis: Microsoft Message Analyzer, Wireshark, NetworkMiner, Fiddler, and OpenNMS.

2.4.1 Microsoft Message Analyzer

The Microsoft Message Analyzer is the successor to Microsoft Network Monitor 3.4, used for capturing, displaying, and analyzing network traffic, events, and application log messages in network and other diagnostic scenarios. It can capture local or remote traffic and live or archived data from multiple data sources simultaneously. Highlighted features include automated data capturing, session filters, flexible user interface, etc. Microsoft Message Analyzer functionalities are enabled for PowerShell scripting environments. This includes stopping a trace session, saving trace session data without stopping the session, injecting a trace filter into a trace session at specific time, and miscellaneous scripting for remote traffic capturing. During a live trace session, data selection can be performed by applying session filters to isolated trace results for analysis. There are several session filter techniques such as fast filter, which operates at the kernel level; keyword filter; and Windows Filtering Platform (WFP) Layer set filter, which consists of kernel-mode TCP/IP stack filters that operate at the Transport layer. These session filter techniques allow selectively enabling or disabling of inbound or outbound packets at the Transport layer when capturing IPv4 or IPv6 messages. Other filter options include HTTP filters, which are enabled to isolate traffic based on the hostname or port value. Keyword and Level filters can be used when configuring a live trace session that uses a system Event Tracing for a Windows (ETW) Provider (Microsoft, 2016).

2.4.2 Wireshark

Wireshark was first started by Gerald Combs in late 1997 with its original name, Ethereal. In 2006 the Ethereal project was renamed Wireshark, and version 1.0 was released in 2008. Wireshark is an interactive network protocol analyzer and capture tool that provides detailed inspection of several protocols and runs on multiple platforms. An opensource software, Wireshark can decrypt protocols such as SSL/TSL, WEP, WPA/WPA2, Kerberos, and more. Wireshark can capture traffic from many different network media types such as Ethernet, Wireless LAN, Bluetooth, USB, and more. Wireshark has a terminal-based version called TShark. TShark captures data from live traffic or from a previously saved capture file, prints a decoded form of the packets to the standard output, or writes packets to a file. TShark has several options and parameters to perform packet analysis. Without any options set, TShark works like Tcpdump (Wireshark, 2019). The Tcpdump tool prints out a description of contents of packets on a network or saves the packet data to a file. Windump is a clone of Tcpdump for Windows operating systems (Tcpdump, 2019). Wireshark uses Libpcap library to capture packets directly from the network card. Libpcap provides a high-level interface to network packet capturing. It was first developed by McCanne, Leres, and Jacobson in 1994 (Gracia, 2008). Libpcap supports packet filtering at the kernel level for systems that support Berkeley Packet Filter (BPF).

2.4.3 Other Tools

These tools provide specific traffic analysis functionalities such as analyzing specific protocols, extract files, analyzing network status, etc.

Telerik Fiddler: A web application debugging tool that can capture HTTP traffic between the client and the server, Telerik Fiddler allows users to monitor and modify HTTP responses and requests in transition. Fiddler provides detailed information about HTTP traffic and can be used for performance testing and debugging for web applications. Fiddler can also decrypt HTTPS traffic (Telerik, 2019).

NetworkMiner: NetworkMiner captures and parses network packets to extract files, images, and other artifacts to reconstruct events that a user has received on the network.

Classified as a Network Forensic Analysis Tool (NFAT), NetworkMiner can process archived Packet Capture (PCAP) files (Netresec, 2019).

OpenNMS: An opensource network management application that offers event and notification management, OpenNMS has a framework that logically groups related faults (alarms) into higher level objects (situations). The framework supports unsupervised machine learning and deep learning algorithms. OpenNMS has a web-based user interface to display any outages, alarms, or notifications in the network infrastructure (OpenNMS, 2019).

CICFlowMeter: An opensource bidirectional network traffic flow generator. It can generate statistical features from PCAP files and live traffic. (Canadian Institute for Cybersecurity, 2019). CICFlowMeter will be discussed more later in the next chapters.

2.5 Literature Review Summary

In this section, details of signature-based and anomaly-based threat detection systems and widely used opensource intrusion detection systems have been discussed. In general, the literature review shows that anomaly-based intrusion detection is the most recent development in intrusion detection technologies. It is also likely the future of intrusion detection technologies. Specific aspects of machine learning techniques, and search engine internals have also been discussed. Decision tree-based classification algorithms show better attack classification than anomaly-based intrusion detection systems. The design of the new framework, which involves a custom search engine and machine learning techniques, will be discussed in the next chapter.

2.5.1 Basic Design Specifications of the Proposed Framework

In order to improve the existing traffic analysis approach for threat detection, which is the purpose of the research, the following specifications need to be incorporated in the proposed framework based on the literature review:

1. The proposed network traffic analysis framework should support anomaly-based intrusion detection because anomaly-based detection approach has better detection capability than signature-based detection approaches.

2. A decision tree-based classification algorithm should be used as a primary classifier for machine learning because decision tree-based algorithms achieve greater accuracy than other algorithms.
3. For comprehensive analysis, the proposed framework needs to use the Microsoft ML.NET machine learning framework because it is opensource and actively supported by Microsoft. It also supports the trained models of other machine learning tools such as TensorFlow and CNTK. Microsoft ML.NET is used by security tools like Windows Defender, is well documented, and supports several classification algorithms including decision tree-based algorithms.
4. Lucene needs to be used as a search engine library because Lucene supports several functionalities, is widely used in large-scale search applications like Wikipedia, and is easily extensible.
5. The proposed framework should be evaluated against a large dataset to perform empirical analysis on its performance and accuracy. Most studies reviewed in this literature review used fewer than 20,000 data instances to evaluate intrusion detection approaches.

Chapter 2 illustrated the comparative analysis of existing cyber threat detection tools, intrusion detection approaches, network traffic analysis tools, and detection algorithms. This chapter also discussed existing machine learning and search engine technologies. Finally, this chapter laid out the basic specifications for the new cyber threat detection framework.

CHAPTER 3

SYSTEM DESIGN

This chapter incorporates research methodology as well as the new framework design from a software engineering perspective. Several architectural diagrams, algorithm designs, and use cases are discussed in this chapter. A prototype project analysis is included in this chapter in order to make design decisions on the core algorithms.

3.1 Research Methodology

There are three factors to consider when selecting a research approach: the research problem, the personal experience of the researcher, and the audience of the research. For example, if the problem is to identify factors that influence an outcome, then a quantitative approach is best. If a concept or phenomenon needs to be understood, a qualitative research approach is preferred (Creswell, 2014).

This research is solution-oriented with the primary goal of improving threat detection system by designing new artifacts. Design science has a problem-solving paradigm that seeks to create new and innovative artifacts (Hevner et al., 2004). According to (Wieringa, 2014), design science is the design and investigation of artifacts in a specified context. Artifacts are designed to improve something in that context. In this research, several techniques (artifacts) are used to improve cyber threat detection (context) approaches. Wieringa (2014) defines Technical Action Research (TAR) as a way to validate the artifact in the field. TAR is artifact-driven and part of the validation of an experimental artifact. In TAR, the researcher plays three roles: (a) as a technical researcher, the researcher designs a treatment intended to solve a class of problems; (b) as an empirical researcher, the researcher answers some validation questions about the treatment; (c) as a helper, the researcher applies a client-specific version of the treatment to help a client. In this research, the researcher designs a new approach to improve existing threat detection mechanisms, provides empirical evidence on the validity of the new approach, and provides a means of applicability of the new approach in the field of cyber threat detection by designing and developing the proposed framework.

The initial survey of (Santos & Travassos, 2009) indicates that there is an increasing tendency to use TAR in software engineering addressing different research topics. Wieringa (2014) stated that single-case mechanism experiments are useful for implementation evaluation in TAR because they can provide insight into the behavior of artifacts and problematic phenomena in the real world. Single-case experiments can be done in the lab or in the field. This research employs lab research for evaluation because the researcher can control the lab environment within the scope of this research.

This research comprises three major phases: problem analysis, solution design, and evaluation and validation. The problem analysis was conducted through a literature review. In the literature review, existing cyber threat detection approaches such as signature-based and anomaly-based detection techniques were investigated. The problems of the investigated approaches were clearly identified. Based on the problem analysis, requirements for the new approach were derived. As shown in Figure 1, the new framework solution must be designed with respect to the identified specifications and proposed techniques. Finally, each artifact implementation is evaluated, and the designed solution as a whole is validated.

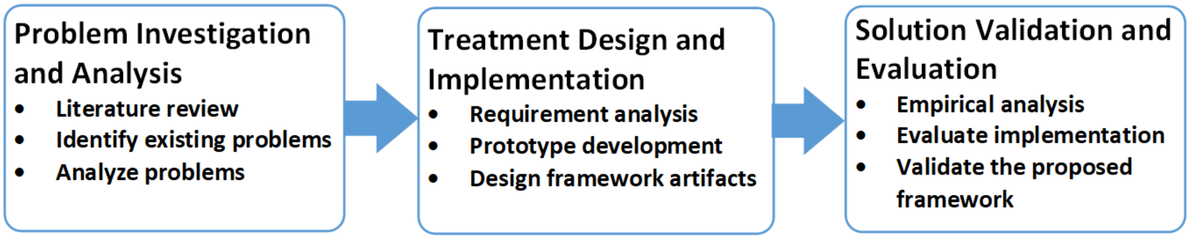

Problem Investigation and Analysis
• Literature review
• Identify existing problems
• Analyze problems
Treatment Design and Implementation
• Requirement analysis
• Prototype development
• Design framework artifacts
Solution Validation and Evaluation
• Empirical analysis
• Evaluate implementation
• Validate the proposed framework

Figure 1. Conceptual design of the proposed framework.

3.2 Treatment Evaluation and Solution Validation

Evaluation in design science research consists of evaluation of design science theories and artifacts (Pries-Heje et al., 2008). In this research the evaluation phase focuses on the newly developed artifacts such as custom search engine components and machine learning components. Evaluation requires researchers to demonstrate the utility, quality, and efficiency of artifacts using rigorous evaluation methods (Hevner et al., 2004). According to (Olan, 2003), validation is a process designed to ensure confidence that the program functions as intended. As a piece of software engineering research, software engineering phases like unit testing and system testing are employed in the evaluation process. Generally, validation involves testing. Careful testing greatly increases the confidence that a program satisfies its specifications. Therefore, treatment evaluation is supported by unit testing. Unit testing validates the correctness of program components. (Koomen & Pol, 1999). Unlike an observational case study, unit testing requires a unit tester to intervene in each use of the artifact to see “what happens” when using software development tools like debuggers. Once bugs are identified through unit testing, artifacts are modified for improvement until the unit test passes in the defined framework. System testing is applied to validate the interaction of artifacts and optimize efficiency of the system.

3.3 Artifact Design and Implementation

The proposed framework utilizes search engine capabilities such as indexing, searching, and VSM. The indexer and searcher components of the search engine are customized in order to use the VSM feature of the search engine for intrusion detection. In addition to the search engine, the framework uses a binary classifier engine to reinforce the search engine capability because a survey in the related literature shows that a single detection technique is not able to provide accurate detection rate and, therefore, a hybrid technique is suggested to increase detection accuracy (Samrin & Vasumathi, 2017). The framework has two major implementation phases: the training and detection phases. A high-level model of the framework is shown in Figure 2 (with the technical scope of the research marked in the dashed border). Although both the training and detection phases use a feature extractor component, designing a new feature extractor from live traffic is not within the scope of this research. Conceptual framework designs and architectural structures are provided for each major component of the framework in the next sections.

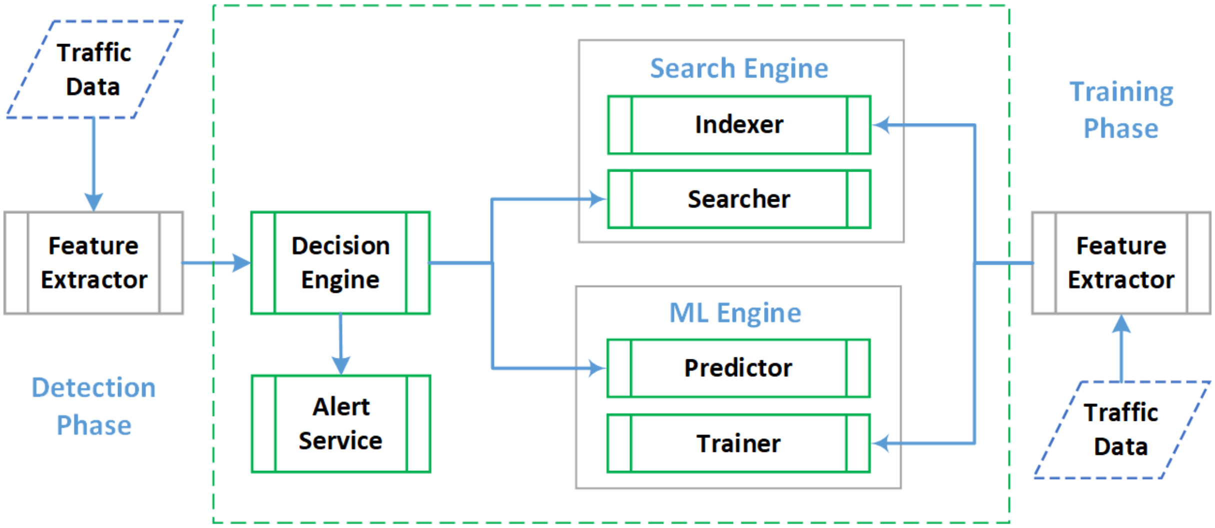


Figure 2. High level model of the framework.

3.3.1 Training Phase

As shown in the process design in Figure 3, the training phase involves the search engine, Machine Learning (ML)-based binary classifier, feature extractor, and indexer components. Since the framework is based on search engine technology that is capable of processing parallel tasks, a training task can be running during the detection phase. This facilitates the self-learning capability of the engine.

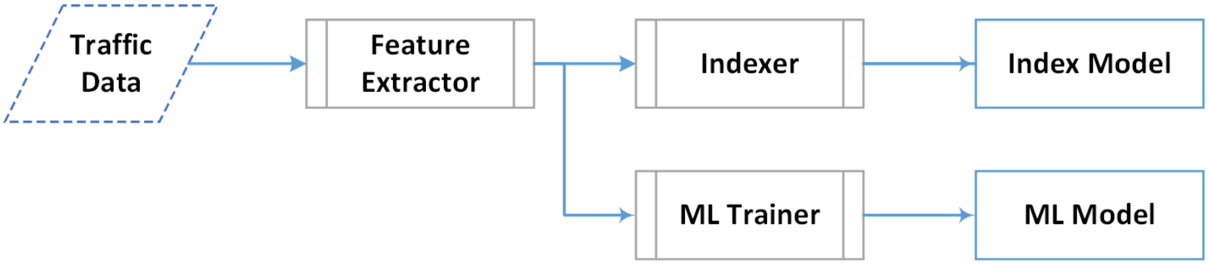


Figure 3. Training process.

Feature Extractor: The feature extractor parses the network traffic raw data and extracts features. The extracted feature dataset is used to train the custom search engine and the binary classifier. Feature extractor tools such as CICFlowMeter (Canadian Institute for Cybersecurity, 2019) can extract features from raw traffic data. CICFlowMeter is capable of extracting several features that can be used for anomaly-based detection techniques (Habibi Lashkari et al., 2017). Feature engineering is not a trivial task; generating a reliable and compressive feature set is critical for anomaly-based intrusion detection systems (Chio & Freeman, 2018).

Search Engine: A fully featured data retrieval engine such as Lucene is used to retrieve data from large datasets. Major features of the search engine include indexing, VSM, ranking, scoring, data storage, parallelism, and distributed deployment support.

Indexer: The indexer is a search engine component that analyzes the classified set of features and systematically adds them to the index storage. Since the search engine is designed to process text files, the features dataset is converted to indexable text and treated as set of documents in the search engine. Therefore, the indexer component designed in the proposed framework is a customized indexer that overrides the default properties of the search engine, so that each feature set is stored as a vector in the VSM.

ML Binary Classifier: This component is a machine learning-based binary classifier engine used to reinforce the decision engine that uses the search engine as threat detection. The Binary Classifier has two components: the predictor and trainer. The predictor accepts input feature data to determine the traffic state during detection phase. The trainer generates a model during the training phase. The trained model can be used to retrain the search engine for new data or undetermined traffic. During the training phase, both the search engine and the binary classifier train with the same features dataset. But during the detection phase, the binary classifier serves as a decision engine to retrain the search engine when the search engine fails to determine the state of the input traffic with high confidence.

3.3.2 Detection Phase

During the detection phase, the same feature extractor component that is used for training is used to extract features from the input traffic data. Once the feature set is extracted, it goes through the decision engine, as shown in the detection process design diagram in Figure 4. Then the decision engine uses the search engine to determine the status of the input. In cases when the search engine detection confidence is low, the decision engine is retrained by the binary classifier.

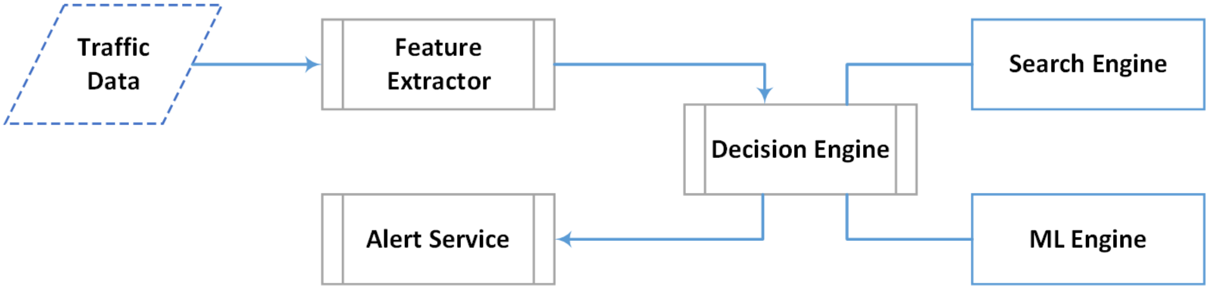


Figure 4. Detection process.

Searcher: The searcher component takes the input feature data as a query, searches for the best matches, and provides the result to the decision engine. The searcher uses similarity algorithms discussed later in the Similarity Measurements section.

Decision Engine: This component determines the nature of the input traffic based on the search result and decision algorithms. As indicated in Figure 5, the activity diagram of the decision engine, the engine accepts the features set as input and uses the searcher component to search for the top matches. If a threat match is found, it sends a notification to the alert service. If the search yields a low confidence score, the features set goes through the binary classifier. Depending on the binary classifier result, the input data is analyzed to retrain the search engine. This way the search engine learns by itself with the help of the Machine Learning (ML) Engine.

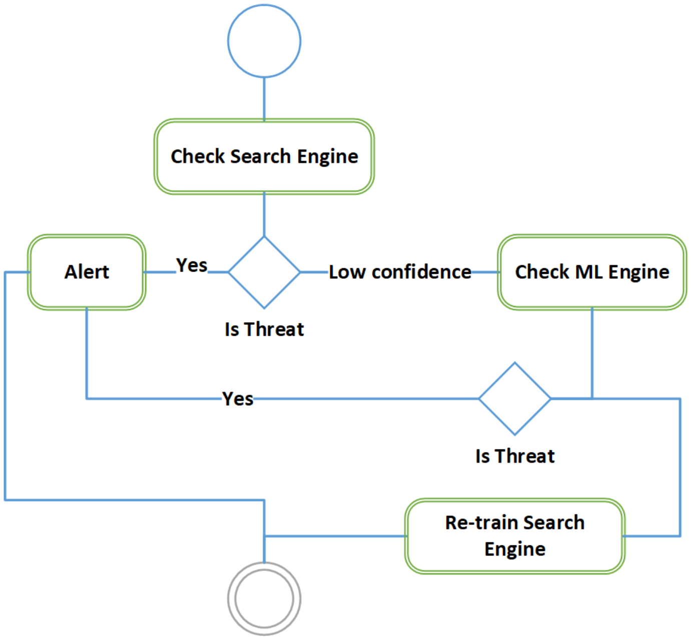


Figure 5. Decision engine activity diagram.

Alert Service: This component is a notifier service used when the system detects a threat. This service receives an alert flag from the decision engine logs and triggers alerts.

3.3.3 Development Tools

Development tools and platform selection for this research are based on the accessibility, adaptability, and performance of the tool, and researcher's expertise.

Visual Studio: A software development environment that supports multiple programming languages and development tools, Visual Studio is a complete package for software development, testing, and deployment. The Visual Studio Community Edition is free for researchers and small teams, and supports building software using different third-party components, including search engine libraries, Lucene, and the machine learning framework ML.NET (Microsoft, 2019).

Platforms: Windows is the primary development and testing platform. C-Sharp (C#) is used as a primary programming language to develop, test, and validate the framework. PowerShell, Python, and other scripting languages may be used in the evaluation and development phases. Microsoft .NET Core is used as the development platform for the proposed framework, which means the proposed framework is intended to run on Windows, macOS, and Linux as .NET Core runs on these environments.

Search Engine Library: The popular opensource search engine library, Lucene is used as a core search engine. As discussed in the literature review, Lucene has several features, and is well tested, is extensible, and used in large-scale applications.

Machine Learning Framework: The Microsoft machine learning framework ML.NET is used as a machine learning component. As discussed in the literature review, ML.NET is new, supports several other machine learning models, and is used by security tools like Windows Defender.

The proposed cyber threat detection framework that incorporates Lucene, ML.NET, and training and evaluation components is intended to run on different environments but was designed, developed, and tested using Visual Studio on a Windows platform.

3.3.4 Datasets

As discussed in the literature review summary, the proposed framework uses an anomaly-based detection approach. Anomaly-based intrusion detection accuracy depends on the amount of collected behavior or features (Modi et al., 2013). Lack of a sufficient dataset impacts the accuracy of analysis and evaluation in anomaly-based IDS (Sharafaldin et al., 2018). With the proposed approach, training the engines is the crucial step to generate models. The accuracy of the proposed model depends on the quality of the training data. For the purpose of training, evaluating each treatment implementation, and validating the overall solution, a wide variety of publicly available datasets, such as datasets collected by the Canadian Institute for Cybersecurity (Sharafaldin et al., 2018), and the Center for Applied Internet Data Analysis (CAIDA) datasets (CAIDA, 2019) were used. These datasets contain a variety of records. The performance of the Canadian Institute for Cybersecurity datasets in machine learning algorithms has been analyzed and evaluated (Sharafaldin et al., 2018).

3.3.5 Similarity Measurements

Extracted features from the training data are stored in a vector form within the VSM in the search engine. Vectors are stored in a multidimensional space. If the data has *n* features, the vector is represented in *n* dimensional space. In reality, several features can be extracted from single session traffic data. For instance, CICFlowMeter can extract over 80 features (Habibi Lashkari et al., 2017). Just to understand the similarity equation, let us assume we have a two-dimensional space with points representing vectors on a plane, as shown in Figure 6. Suppose each traffic is represented by a vector on the coordinate plane, and suppose vector *A* represents an input traffic.

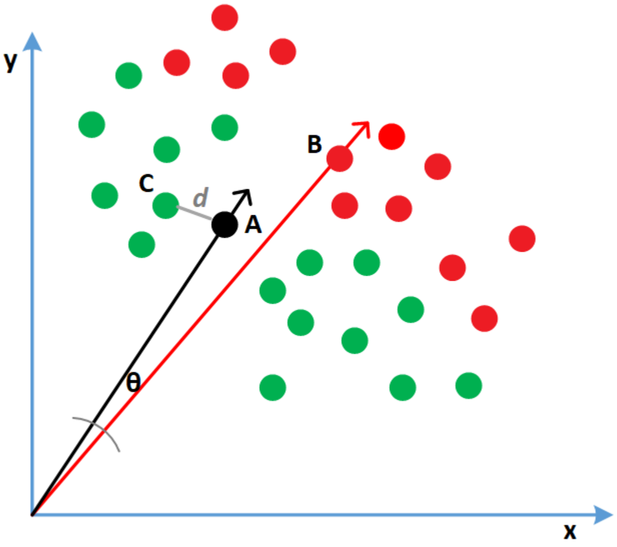


Figure 6. Vectors in two-dimensional space.

The status of the input vector *A* is determined by the nearby vectors using similarity measurements such as Euclidean distance and Cosine similarities. While Euclidean distance similarity measures the distance between the two points *A* and *B* in space, Cosine similarity measures the angle between two vectors.

Cosine similarity between vectors *A* and *B*, each with *n* elements can be expressed as:

$$\text{similarity}(\mathrm{A},\mathrm{B}) = \cos(\theta) = \frac{\mathrm{A} \cdot \mathrm{B}}{\parallel \mathrm{A} \parallel \parallel \mathrm{B} \parallel} = \frac{\sum_{i=1}^{n} A_i \cdot B_i}{\sqrt{\sum_{i=1}^{n} {A_i}^2 \cdot \sum_{i=1}^{n} {B_i}^2}}$$

Euclidean distance *d* between two points in space *A* and *C* can be computed as:

$$d(A, C) = \sqrt{\sum_{i=1}^{n} (A_i - C_i)^2}$$

Similarity measurements produce different results. For instance, in Figure 6, vector *B* is the closest vector to the input vector *A* using Cosine similarity; however, point *C* is the closest point to point *A* using Euclidean distance. Selection of the appropriate similarity measurement is an experimental process. The next section discusses the results of a prototype project developed to determine the appropriate similarity measurement and related algorithms. Similarity measurements are crucial in search engines, so a prototype project was needed to make the right selection.

3.4 Prototype Project

In software engineering, the main purpose of prototyping is for design verification of product development as it demonstrates the feasibility of the proposed design before launching the actual development. The goal of this prototype project was to assess the overall feasibility of the research and select the appropriate similarity algorithm used in the customization of the search engine VSM component. As a prototype project, a miniVSM component with training and prediction features was implemented from scratch. The prototype project was based on the hypothesis of the research, explained in short as the following: the nearby vectors that represent network traffic data determine the status of the input traffic data vector. The prototype project had data preparation, training, prediction, and evaluation procedures. A total of 12,000 traffic instances were involved in the training and evaluation, out of which 2,000 traffic instances represent attacks. The evaluation data were proportionally and randomly selected. Implementation of the VSM prototype involved building matrix of vectors from the features dataset, as shown in Listing 1.

```
public static void BuildVSM(IEnumerable<string> trainingDataSet)
{
    int index = 0;
    featureVectors = new ConcurrentDictionary<int, double[]>
                          (CONCURRENCY_LEVEL, COLLECTION_CAPACITY);
    labelVectors = new ConcurrentDictionary<int, string>
                          (CONCURRENCY_LEVEL, COLLECTION_CAPACITY);

    foreach (string featureSet in trainingDataSet)
    {
        IEnumerable<string> values = featureSet.Split(',');
        double[] vector = values.Take(FEATURES_COUNT)
                                .Select(val => Convert.ToDouble(val)).ToArray();
        featureVectors.TryAdd(index, vector);
        labelVectors.TryAdd(index, values.Last());
        index++;
    }
}
```

Listing 1. Partial implementation of VSM.

The VSM prototype was evaluated with a test dataset. The test dataset contains a total of 2,400 instances of attack and benign traffic features. The evaluation model had a simple prediction accuracy measure that computes the percentage of accurate predictions out of the total predictions, as shown in Listing 2.

```
public static double EvaluateVSM(IEnumerable<string> testingDataSet)
{
    int testingDataCount = testingDataSet.Count();
    List<(string ActualValue, string PredictedValue)> evaluationCheckList =
          new List<(string, string)>(testingDataCount + RANDOM_PRIME_NUMBER);

    foreach (var featureSet in testingDataSet)
    {
        IEnumerable<string> values = featureSet.Split(',');
        double[] vector = values.Take(FEATURES_COUNT)
                                .Select(val => Convert.ToDouble(val)).ToArray();

        string actualValue = values.Last();
        string predictedValue = Predict(vector);

        evaluationCheckList.Add((actualValue, predictedValue));
    }

    int accuratePredictionCount = evaluationCheckList.Where(
                              item => item.ActualValue == item.PredictedValue
                              ).Count();
    double accuracy = (double)accuratePredictionCount * 100 /
                      (double)testingDataCount;
    return accuracy;
}
```

Listing 2. Partial implementation of VSM evaluation.

The prediction model uses both Cosine similarity and Euclidean similarity to measure the nearness value between the vectors. Listing 3 shows implementation of these similarity measures.

```
static double EuclideanDistanceSimilarity(double[] vector1, double[] vector2)
{
    double sum = 0.0;
    int length = vector1.Length;
    for (int i = 0; i < length; ++i)
        sum += Math.Abs((vector1[i] - vector2[i]) * (vector1[i] - vector2[i]));
    return Math.Sqrt(sum);
}
public static double CosineSimilarity(double[] vector1, double[] vector2)
{
    double lengthV1 = ComputeVectorLength(vector1);
    double lengthV2 = ComputeVectorLength(vector2);
    double dotProduct = ComputeDotProduct(vector1, vector2);
    return dotProduct / (lengthV1 * lengthV2);
}
public static double ComputeDotProduct(double[] vector1, double[] vector2)
{
    double product = 0.0;
    if (vector1.Length == vector2.Length)
    {
        for (int i = 0; i < vector1.Length; i++)
        {
            product += vector1[i] * vector2[i];
        }
    }
    return product;
}
public static double ComputeVectorLength(double[] vector)
{
    double length = 0.0;
    for (int i = 0; i < vector.Length; i++)
    {
        length += Math.Pow(vector[i], 2);
    }
    return Math.Sqrt(length);
}
```

Listing 3. Implementation of similarity measures.

Model prediction is based on the K-Nearest Neighbors (KNN) algorithm. The KNN algorithm is the simplest prediction algorithm in machine learning (Chio & Freeman, 2018). In KNN algorithm the prediction is based on the plurality of votes of its neighbors. The input vector is similar to the most common class among its *k* nearest neighbors. Selection of the best value of *k* is heuristic. Listing 4 shows the implementation of the KNN algorithm in the prototype project.

```
public static string Predict(double[] inputVector)
{
    ConcurrentDictionary<int, double> vectorMeasures =
    new ConcurrentDictionary<int, double>(CONCURRENCY_LEVEL, COLLECTION_CAPACITY);

    Parallel.ForEach(featureVectors, (vector) =>
    {
        if (SIMILARITY == Similarity.COSINE)
        {
            double cos = CosineSimilarity(inputVector, vector.Value);
            vectorMeasures.TryAdd(vector.Key, cos);
        }
        else
        {
            //Default similarity
            double distance = EuclideanDistanceSimilarity(inputVector,
                                                          vector.Value);
            vectorMeasures.TryAdd(vector.Key, distance);
        }
    });

    //Sort vector measures and take the nearest K vectors to the input vector.
    var nearestVectors = vectorMeasures.OrderBy(item => item.Value).Take(K);

    //Group by label and take the majority vote among the nearest vecotors
    var candidates = from nn in nearestVectors
                     group nn by labelVectors[nn.Key] into g
                     select new { Label = g.Key, Count = g.Count() };

    //Sort Candidates. The maximum vote comes first
    candidates = candidates.OrderByDescending(item => item.Count);
    string prediction = candidates.First().Label;

    return prediction;
}
```

Listing 4. Partial implementation of prediction model and KNN algorithm.

Running the prototype app with the prepared 12,000 traffic instances shows that the Euclidean distance similarity outperformed the Cosine similarity measurement in the miniVSM implementation. The detection accuracy was 82.63% using Cosine similarity whereas using Euclidean distance similarity yielded 99.42% detection accuracy. As discussed earlier, the purpose of the prototype is to provide insight into the new framework before actual development. The result of this prototype indicates the feasibility of the dataset training and algorithms design. However, the result also shows that the default Lucene similarity algorithm, Cosine similarity, is less effective than the Euclidean distance similarity.

3.5 Deployment and Continuous Training

While using Euclidean distance similarity measurement and KNN algorithm in the proposed custom search engine, the prediction confidence can be low. When this occurs, the ML binary classifier model is used to determine the status of the input vector; based on the result, the custom search engine is retrained. As discussed earlier, both the search engine and the ML model use the same dataset for the initial training. A larger amount of training data improves accuracy but consumes large storage and computational time, making it inefficient for real-time detection (Modi et al., 2013). This is true for both the ML engine and the search engine. With the proposed technique, the search engine has an advantage over the ML engine as it can update itself while it is running without destroying the model.

Another advantage of using such a search engine technique is that the indexed data can be stored and searched in distributed environments using parallel multisearchers, which also means the model can be consumed as a service by several client hosts. The detection framework can also be deployed on a single server.

3.6 Summary

In this chapter, the design of the new framework and its components has been described. Depending on the deployment option, the new framework can serve to analyze traffic coming to the local area network or to a single host. The framework can work with live traffic or previously captured traffic data. As shown in Figure 7, the tapped network traffic goes to the feature processor to produce a feature set and pass it to the threat detection engine. The threat detection engine uses its two models (search engine and machine learning) to determine the status of the traffic. The search engine model learns the incidents when necessary. If a threat is detected, the detection engine notifies the alert service, then the alert service dispatches the alert message to connected client apps. Similarity measurement is the core algorithm of the search engine model. Development tools and frameworks, such as Visual Studio, .NET Core, Lucene, ML.NET, and datasets selections, have also been discussed. The prototype project shows that Euclidean distance similarity is better than Cosine similarity, the default similarity in the Lucene search engine library.

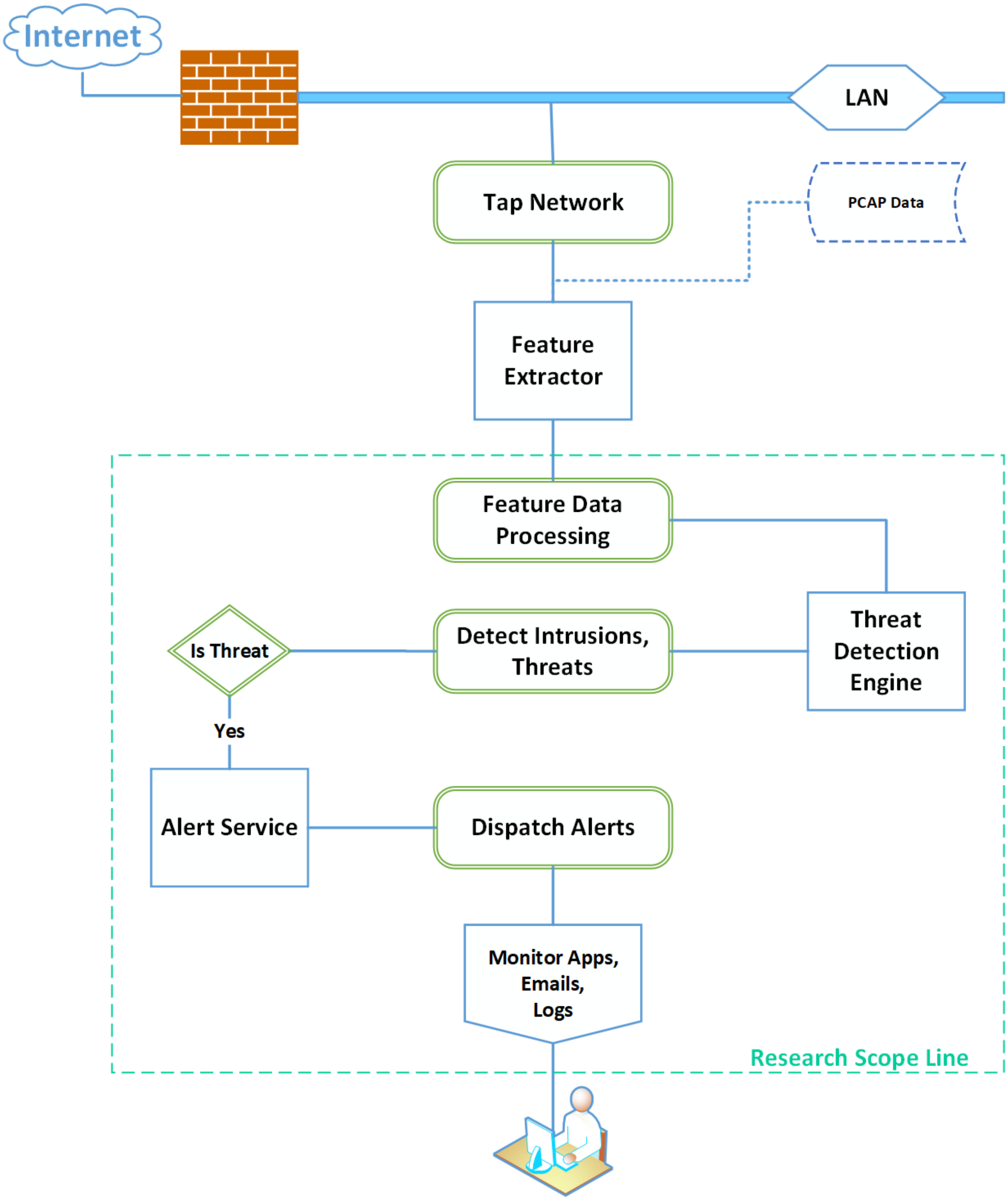


Figure 7. Process flow design.

This chapter incorporated research methodology as well as the new framework design from a software engineering perspective. Several architectural diagrams, algorithm designs, and use cases were discussed in this chapter. A prototype project analysis was included in this chapter to make design decisions on the core algorithms.

CHAPTER 4

SYSTEM DEVELOPMENT

This chapter discusses the technical details and implementations of techniques and algorithms used to develop the proposed framework as designed in Chapter 3. This chapter involves mathematical analysis and the development of algorithms and components. The proposed framework is divided into several components. Each component is developed in such a way that it can be separately tested and, therefore, is less dependent on other components. The major components of the framework include the data processor, custom search engine, ML based Binary Classifier Engine, decision engine, and alert service. The component development process involves subcomponents, implementations of algorithms, and procedures.

4.1 Data Processor

The data processing task involves converting raw PCAP files into features data, and partitioning, sampling, and balancing datasets. Figure 8. Data processor process flow diagramFigure 8 shows the process flow of the data processor component. The raw data contains two kinds of classes: malicious traffic data and benign traffic data. Feature datasets go through the data balancing process when the classes are not proportional; data balancing reduces the issue of class imbalance. Both undersampling and oversampling the training data impacts the accuracy of the detection. Undersampling can be mitigated by intelligently generating synthetic data for minority classes, which is also referred to as oversampling. Different resampling techniques have different characteristics (Chio & Freeman, 2018), and therefore, the best strategy is determined by running different experiments. To start the training process the initial dataset is assumed to be adequate, then based on the result, different sampling techniques such as semisupervised learning may be used. In semisupervised learning, the initial training model can be used to generate an additional training dataset from unlabeled data. Predictions with the highest confidence are considered correctly labeled and added to the training data.

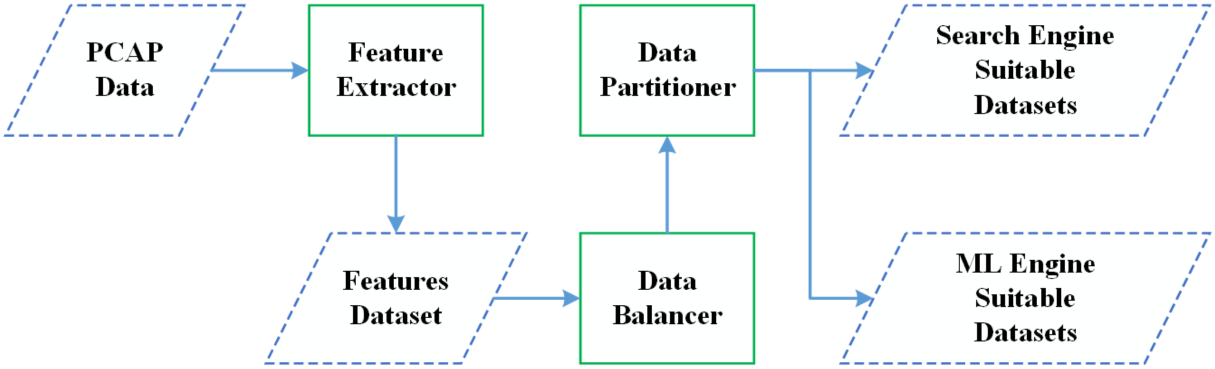


Figure 8. Data processor process flow diagram.

4.1.1 Feature Engineering

Feature extraction is the crucial step in anomaly detection. Features depend on the intrusion detection domain; host intrusion detection and network intrusion detection systems require different feature datasets. This research focuses on building network intrusion detection; therefore, features are extracted from the network traffic data. The CICFlowMeter tool is used to extract features from traffic data. According to the documentation (Canadian Institute for Cybersecurity, 2019), the CICFlowMeter uses WinPcap, the standard Windows packet capture library, to capture network packets. In addition to the packet header and footer, CICFlowMeter examines signals and statistics about the traffic. Version 4.0 of CICFlowMeter can generate 84 different features including duration, number of packets, number of bytes, length of packets, and so forth in both forward and reverse directions. CICFlowMeter is developed in Java and can run on Windows and Linux environments. Although CICFlowMeter can generate features from live traffic, extracting features from live traffic is beyond the scope of this research. Therefore, precollected PCAP files are used to generate features datasets for training in this project.

The feature extractor method uses the input directory that contains PCAP files to process the extracted features and the output directory to save them. Extracted features are saved in Comma Separated Value (CSV) format. Since CICFlowMeter is developed in Java, there is no direct Java code execution from C#. Therefore, the feature extractor method uses Windows system process components. The method starts a command prompt process by passing a batch file that accepts the directory values as parameters. The batch file then starts the CICFlowMeter executables. The output of the CICFlowMeter executables are redirected to the host process console screen as shown in Listing 5.

```
public static void ExtractFeatures()
{
    ProcessStartInfo processStartInfo = new ProcessStartInfo
    {
        FileName =  $"{AppConfigSettings.CicFlowMeterPath}cfm.bat",
        Arguments = $"\"{AppConfigSettings.InputPcapFileDirectory}\" " +
                    $"\"{AppConfigSettings.FeaturesDataDirectory}\"",
        WorkingDirectory = AppConfigSettings.FeaturesDataDirectory,
        Verb = "runas",
        UseShellExecute = false,
        RedirectStandardOutput = true
    };

    Process cicFlowMeterProcess = new Process
    {
        StartInfo = processStartInfo
    };

    cicFlowMeterProcess.OutputDataReceived +=
           new DataReceivedEventHandler((sender, e) =>
            {
                if (!string.IsNullOrEmpty(e.Data))
                {
                    Console.WriteLine(e.Data);
                }
            });

    cicFlowMeterProcess.Start();
    cicFlowMeterProcess.BeginOutputReadLine();
    cicFlowMeterProcess.WaitForExit();
    cicFlowMeterProcess.Close();
}
```

Listing 5. Feature Extractor method implementation.

4.1.2 Data Balancer

Implementation of the data balancer depends on the accuracy of the detection result and initial dataset size. Data balancing is not important when adequate and proportional training data are available. Therefore, this component is implemented during the evaluation phase whenever necessary.

4.1.3 Data Partitioning

The data partitioning process involves data preparation, randomization, training-testing data selection, normalization, and data formatting.

Data Preparation: The feature data preparation task reads all the extracted features, cleans invalid records, removes duplicates, and generates a list of instances with labels and their descriptions. Label descriptions are used by the custom search engine during indexing. The raw feature data reading is implemented as shown in Listing 6. The data splitter method is then used on the prepared feature data to generate engine-specific training data.

```
public static void ReadRawFeatureData(string fileDirectory = "",
                                      int maxGroupSize = 10000000)
{
   char[] separators = {',' };
    int errorCounter = 0;
    string directory = AppConfigSettings.RawDataDirectory;
    if (!string.IsNullOrWhiteSpace(fileDirectory))
        directory = fileDirectory;
    var files = Directory.GetFiles(directory, "*.csv");

    List<string> sampleLines = new List<string>();
    foreach (string file in files)
    {

        string fileName = Path.GetFileNameWithoutExtension(file);
        sampleLines.Add(fileName);
        Console.WriteLine($"Processing: {fileName}");
        var dataLines = File.ReadLines(file)
                            .Where(line => !(line.Contains("Infinity")
                                          || line.Contains("NaN"))).Distinct();
        sampleLines.AddRange(dataLines.Take(5));

        Console.WriteLine($"Total distinct records = {dataLines.Count()}");

        dataLines = dataLines.Skip(1);

        Console.WriteLine("Data selection");
        var attackDataLines = dataLines.Where(l => !l.ToLower()
                              .Contains("benign")).Take(maxGroupSize).ToList()
        int attackInstanceCount = attackDataLines.Count;
        Console.WriteLine($"{attackInstanceCount} - " +
               $"attack instances found. Selecting random benign instances...");
        int benignInstanceCount = (int)(attackInstanceCount * 1.5);
        var benignDataLines = dataLines.Where(l => l.ToLower().Contains("benign"))
                                       .OrderBy(r => Guid.NewGuid())
                                       .Take(benignInstanceCount).ToList();

        Console.WriteLine("Merging instances...");
```

```
var entireDataLines = attackDataLines.Union(benignDataLines);
Console.WriteLine($"{entireDataLines.Count()} - " +
                  $"total instances found.");


int dataCounter = 1;
ConcurrentDictionary<int, string> instances =
                                 new ConcurrentDictionary<int, string>();


Parallel.ForEach(entireDataLines, line =>
{
    var featureItems = line.Split(separators);
    if (!featureItems.Any(f => string.IsNullOrWhiteSpace(f)))
    {
        var featuresList = featureItems.Take(featureItems.Count() - 1);
        var label = featureItems.Last().Trim().ToLower();
        label = Regex.Replace(label, @"[^a-z]|\s+", "-")
                     .Replace("---", "-").Replace("--", "-");
        var normalizedLabel = label == "benign" ?
                              benignLabelValue : attackLabelValue;

        int featureLength = featuresList.Count();
        string featureLine = string.Empty;
        try
        {
            for (int i = 0; i < featureLength; i++)
            {
                if (!excludedColumnIndexes.Contains(i))
                {
                    double val = Convert.ToSingle(featuresList
                                                  .ElementAt(i));
                    string fVal = val.ToString();
                    featureLine += fVal + ",";
                }
            }

            featureLine = featureLine.Trim(' ', ',');
            int totalFeatureSize = featureLine.Split(separators).Length;

            if (!string.IsNullOrWhiteSpace(featureLine))
            {
                instances.TryAdd(dataCounter,
                    (featureLine + "|" + normalizedLabel + "," + label)
                    .Trim(' ', ',')); //features|labels
            }
            dataCounter++;
            if (dataCounter % 10000 == 0)
                Console.WriteLine($"{dataCounter}");
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
            Console.WriteLine(line);
            errorCounter++;
```

```
                }
            }
        }
        );
        Console.WriteLine($"Saving {fileName}.txt ...");
        File.WriteAllLines(directory + fileName + "-labeled.txt",
                           instances.Select(l => l.Value));
    }
}
```

Listing 6. Implementation of the data preparation method.

Data Splitting: In machine learning the training data and the testing data should be distinct. Usually thirty percent of the entire dataset goes to testing, and the remaining seventy percent is used for training. The test data fraction can be adjustable depending on the training data size. As shown in the implementation code in Listing 7, first, the prepared dataset is split into two groups: benign instances and attack instances. Then the test dataset is extracted from each group based on the test-fraction value. This step generates four groups of datasets: benign and attack instances for training and testing. Data selection is through randomization. In the last step, the training and testing datasets of each of the benign and attack datasets are merged and shuffled. This step generates two datasets, training and testing, and each group contains both attack and benign instances.

```
private static void PrepareDateset(int instanceCount)
{
    IEnumerable<string> entireDataLines = ReadFeatureData();

    //Split datasets into attack and benign instances
    var benignInstances = entireDataLines.Where(item =>
                              Regex.IsMatch(item, @"\|" + benignLabelValue));
    var attackInstances = entireDataLines.Where(item =>
                              Regex.IsMatch(item, @"\|" + attackLabelValue));

    //Make sure there is no intersection between the datasets
    int intersection = benignInstances.Intersect(attackInstances).Count();
    Debug.Assert(intersection == 0, "Invalid benign-attack instances");

    int totalBenignInstances = benignInstances.Count();
    int totalAttackInstances = attackInstances.Count();



    //Determine the train-test benign data proportion
    int testBenignInstancesCount = (int)(totalBenignInstances * TEST_FRACTION);
    int trainingBenignInstancesCount =
        totalBenignInstances - testBenignInstancesCount;
    //Determine the train-test attack data proportion
    int testAttackInstancesCount = (int)(totalAttackInstances * TEST_FRACTION);
    int trainingAttackInstancesCount =
        totalAttackInstances - testAttackInstancesCount;

    //Generate random data
    IEnumerable<string> testAttackInstances = attackInstances.OrderBy(r =>
                         Guid.NewGuid()).Take(testAttackInstancesCount).ToList();
    IEnumerable<string> trainingAttackInstances = attackInstances.Except(
                                                        testAttackInstances);

    //Make sure there is no intersection between the datasets
    intersection = testAttackInstances.Intersect(trainingAttackInstances).Count()
    Debug.Assert(intersection == 0, "Invalid training-test attack instances");

    IEnumerable<string> testBenignInstances = benignInstances.OrderBy(r =>
                          Guid.NewGuid()).Take(testBenignInstancesCount).ToList()
    IEnumerable<string> trainingBenignInstances = benignInstances.Except(
                                                        testBenignInstances);

    //Make sure there is no intersection between the datasets
    intersection = testBenignInstances.Intersect(trainingBenignInstances).Count()
    Debug.Assert(intersection == 0, "Invalid training-test benign instances");

    //Merge and shuffle data
    var trainingInstances = trainingAttackInstances
               .Union(trainingBenignInstances).OrderBy(r => Guid.NewGuid());
    var testingInstances = testAttackInstances.Union(testBenignInstances)
                                              .OrderBy(r => Guid.NewGuid());

    //Make sure there is no intersection between the datasets
    intersection = trainingInstances.Intersect(testingInstances).Count();
    Debug.Assert(intersection == 0, "Invalid training-test instances");

    Console.WriteLine($"            Benign\t\tAttack");
    Console.WriteLine($"Training\t{trainingBenignInstancesCount}\t" +
                      $"\t{trainingAttackInstancesCount}");
    Console.WriteLine($"Testing\t{testBenignInstancesCount}\t" +
                      $"\t{testAttackInstancesCount}");

    //Geenrate and save ML compatible datasets
    File.WriteAllLines(AppConfigSettings.MlTrainingDataPath,
                       GenerateMLCompatibleData(trainingInstances));
    File.WriteAllLines(AppConfigSettings.MlTestingDataPath,
                       GenerateMLCompatibleData(testingInstances));

    //Generate and save SE compatible datasets
    File.WriteAllLines(AppConfigSettings.SeTrainingDataPath,
                       GenerateSECompatibleData(trainingInstances));
    File.WriteAllLines(AppConfigSettings.SeTestingDataPath,
                       GenerateSECompatibleData(testingInstances));
}
```

Listing 7. Implementation of the data split method.

There are two detection engines in the proposed framework: the ML engine and the custom search engine. Both engines use different data formats. The ML engine can process CSV files and text files without further customization. But the search engine requires a further data preparation step, discussed in detail next.

4.2 Custom Search Engine Data Preparation

The primary use of a search engine is data mining, so it requires each data instance in the form of text, which is called a document in search engine terminology. During indexing, the indexer generates a multidimensional term-frequency vector for each document. The term frequency is calculated from the document. For instance, the text “the red fox and the red cat are smart” is stored as a term-frequency vector within the search engine VSM, as shown in Table 5. Articles and other stop words are not analyzed by default.

Table 5. Term Frequency Table of a Sample Text

Term	red	fox	cat	smart
Frequency	2	1	1	1

One of the main artifacts in this research is the customization of a search engine library to make it function as a threat detection engine. The customized search engine is expected to store vectors generated from traffic data features. For instance, the customized search engine should store the traffic data that contains only port number 80 and payload size 4000 bytes in its internal vector space model, as shown in Table 6.

Table 6. Feature Value Table of Sample Traffic Data Features

Feature	port	payload_size
Value	80	4000

The default Lucene search engine indexer expects the terms “port” and “payload_size” to be repeated 80 and 4,000 times, respectively, in the text to generate values, as shown in Table 6. Generating the expected vector space model by repeating terms in the documents is not an efficient approach, particularly for traffic data with several features. Therefore, the default indexer needs to be customized to generate the required vector from well-formatted text data like “port|80 payload_size|4000”, where the numbers indicate the frequency/value of the term/feature. Customization of the Lucene indexer will be discussed in detail later in the next section. Hence, the training data for the search engine should be prepared as a list of well-formatted text for efficient indexing. Listing 8 shows how the training and testing datasets are reprocessed for the custom search engine indexer. Feature values are rounded to integers because the search engine processes these values as frequency. Each feature is boosted by one to ensure that every feature is included in the search during the detection phase. However, this might change during code optimization and will be discussed later in Chapter 5. Case study.

```
private static IEnumerable<string> GenerateSECompatibleData(
                                         IEnumerable<string> instanceLines)
{
    List<string> indexableLines = new List<string>();

    foreach(string line in instanceLines)
    {
        string[] featureLabel = line.Split('|');
        string[] features = featureLabel[0].Split(',');

        string labels = featureLabel[1];
        string indexableLine = string.Empty;

        int featuresLength = features.Length;
        int featureCounter = 1;

        for(int i =0; i<featuresLength; i++)
        {
            if(!EngineConstants.EXCLUDED_FEATURE_INDEXES.Contains(i))
            {
                //Convert string to float and round to the ceiling integer
                int fVal = (int)Math.Ceiling(Convert.ToSingle(features[i]));
                fVal = Math.Abs(fVal);

                //fVal += 1; //Boost all by 1; this sets the minimum frequency to
                //generate n-dimensional vector which results in low performance.
                //Please read section 5.2.4 Variable Vector Dimension for details.
                if (fVal > 0)
                {
                                  // ሀ|v1 ሁ|v2 ሂ|v3 …
               indexableLine += $"{ETHIOPIC_ALPHABETS[featureCounter++]}|{fVal} ";
                }
            }
        }
        indexableLine = indexableLine.Trim();
        indexableLine += "," + labels.Replace(",", "|");
        indexableLines.Add(indexableLine.Trim().Replace("  ", " "));
    }

    return indexableLines;
}
```

Listing 8. Custom search engine-compatible data preparation method.

4.2.1 Feature Name Representation

Feature names can serve as axis names in the feature-value VSM. Feature names can be the actual feature names of the traffic such as “port”, “payload_size”, and so on. The feature value tokenizer, which will be discussed later, iterates through each character of the feature name during tokenization and while splitting the feature name from the feature value. If the traffic has several features, longer names consume memory and disk space, and the tokenizer could be slow as it iterates through every character. Therefore, using shorter names makes the VSM more efficient. One approach could be using a single character to represent a feature name like (a|v1 b|v2 c|v3 . . .). It only takes one byte to store an English alphabet character in memory; however, English alphabet characters are limited in number, and the approach fails to support models with several features. To support large models with several features, a script that contains a large number of alphabet characters and consumes less space in memory is needed. According to Unicode Standard 12.1, there are a total of 495 assigned Ethiopic characters including supplemental and extended versions. Excluding the tonal marks, numerals, and punctuation characters, there are 473 different pronounceable Ethiopic characters (Unicode, 2019). Ethiopic characters are used by Ethiopian languages such as Amharic, Geez, Tigrinya, and others. Each Ethiopic character takes two bytes in memory. There are other language scripts such as Chinese, Japanese, Korean (CJK Unified Ideographs Extension B) that contain several thousands of assigned characters but require more than two bytes to represent each character in memory. Using Ethiopic characters for feature names allows each feature name in a network traffic data that contains up to 473 features to be processed with a single character name in the search engine model. This means the model can grow up to a 473-dimensional vector space. Ethiopic characters are enough to represent the features in the dataset and provide enough space for future expansion. When using Ethiopic characters for feature names, the prepared dataset looks like the following: (ሀ|v1 ሁ|v2 ሂ|v3

…). Note that the same data generator method shown in Listing 8 is used to generate the custom search engine compatible data during the training and the detection phases.

4.3 Custom Search Engine

A search engine is an information retrieval system that can retrieve data efficiently. In general, a search engine involves two steps: indexing and searching. Indexing is a process of collecting metadata about the raw data and storing it in a systematic way for efficient searching. As discussed in Chapter 2, Apache Lucene is a popular opensource search engine library. Lucene is implemented in different programming languages such as Java, C#/.NET, and Python. In this section we focus on the customization of the C# version of Lucene, also known as Lucene.NET. At the time of this writing, the latest .NET version of Lucene is 4.8.0, which is behind the current Java version, which is 8.1.1. To make the Lucene library function as a threat detection engine, the core components such as the indexer, analyzers, searcher, and similarity should be customized. Customization depends on the Lucene platform and its version. For instance, the Java Lucene supports indexing custom term frequencies since version 7.0,[1] but this capability does not exist in the Lucene.NET. Lucene is an extendable search engine library. Figure 9 shows the basic customization architecture of Lucene to make it function as a threat detection engine. The customized engine is intended to store the feature data in the form of feature-value vectors within the Lucene Index Store and apply the custom searcher that uses a special scoring algorithm to find the best matching vectors to the input traffic feature data. Each component customization will be discussed in detail in the next sections.

[1] https://issues.apache.org/jira/browse/LUCENE-7854

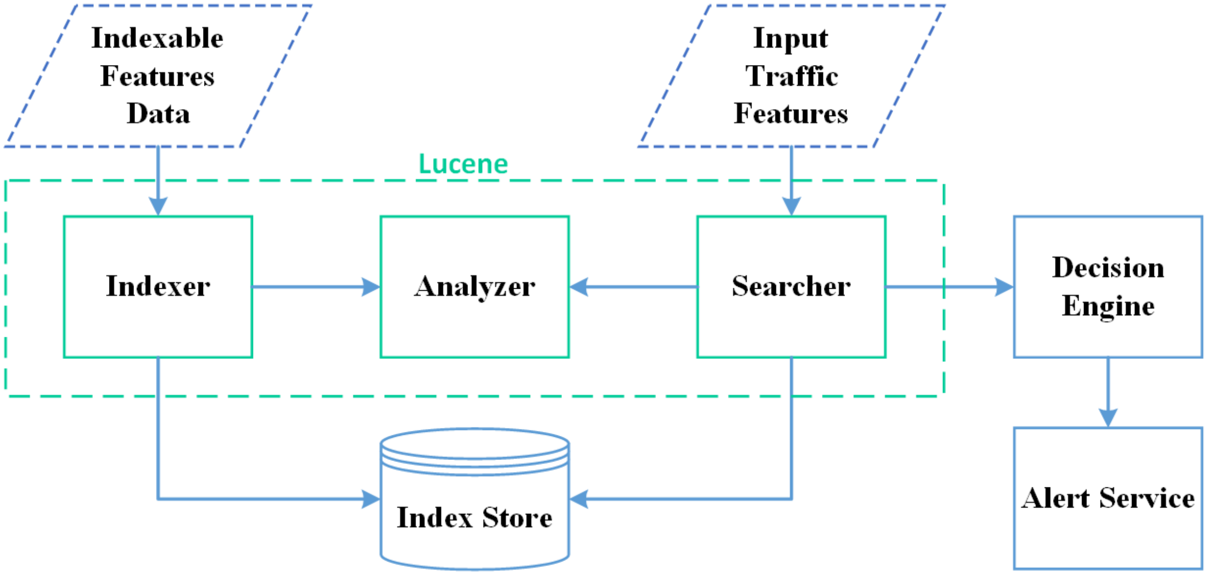


Figure 9. Lucene customization basic architecture.

4.3.1 Feature Analyzer

In Lucene library, the purpose of the Analyzer is to build tokens to extract index terms from the input text. The process of analysis converts the input text into indexable/searchable tokens. Analyzer uses Tokenizer and Token Filter components. The purpose of customizing the Analyzer is to convert the input traffic feature data into indexable feature tokens with the feature value as an attribute. Then the value attribute is used by the indexer to build a feature-value vector. The custom Analyzer, hereafter called Feature Analyzer, extends the default Lucene Analyzer, and uses a custom token filter called Feature Value Token Filter as shown in Listing 9. As discussed in the data processing section, the search engine indexable data are provided in the form of a space-separated set of feature names along with its value concatenated by pipeline character (f1|v1 f2|v2, f3|v3, …). The Feature Analyzer first splits each feature and value combination by using the Lucene built in WhitespaceTokenizer.

```
public class FeatureAnalyzer : Analyzer
{
    protected override TokenStreamComponents CreateComponents(string fieldName,
                                                              TextReader reader)
    {
        Tokenizer source = new WhitespaceTokenizer(LuceneVersion.LUCENE_48,
                                                   reader);
        FeatureValueTokenFilter filter = new FeatureValueTokenFilter(source);
        return new TokenStreamComponents(source, filter);
    }
}
```

Listing 9. Partial implementation of Feature Analyzer.

The Feature Value Token Filter splits the feature and its value, and stores them as term attributes and frequency attributes, respectively. Note that the term-frequency and feature-value terminologies are used interchangeably in this context. Lucene treats *feature* as a *term*, and *value* as a *frequency* in its internal structure. The Feature Value Token Filter extends the default Lucene TokenFilter class and overrides the IncrementToken method, which splits the feature and its values for each token as shown in Listing 10. This implementation was adapted from the Java implementation of DelimitedTermFrequencyTokenFilter.[2]

```
public sealed class FeatureValueTokenFilter : TokenFilter
{
    public static char DEFAULT_DELIMITER = '|';
    private char delimiter;
    private ICharTermAttribute featureAtt;
    private IFeatureValueAttribute featureValueAtt;

    public FeatureValueTokenFilter(TokenStream input) :
                         this(input, DEFAULT_DELIMITER)
    {
        featureAtt = m_input.AddAttribute<ICharTermAttribute>();
        featureValueAtt = m_input.AddAttribute<IFeatureValueAttribute>();
    }
    public FeatureValueTokenFilter(TokenStream input, char delimiter) :
                                   base(input)
    {
        this.delimiter = delimiter;
    }

    public override bool IncrementToken()
    {
        if (m_input.IncrementToken())
        {
            char[] buffer = featureAtt.Buffer;
            int length = featureAtt.Length;
            for (int i = 0; i < length; i++)
            {
                if (buffer[i] == delimiter)
                {
                    //Sets the feature to be the value before the delimiter
                    featureAtt.Length = i;
                    i++;
```

[2] https://lucene.apache.org/core/7_0_0/analyzers-common/org/apache/lucene/analysis/miscellaneous/DelimitedTermFrequencyTokenFilter.html

```
                    //Sets the feature value to be the value after the delimiter
                    featureValueAtt.FeatureValue = ArrayUtil.ParseInt32(buffer,
                                                                      i,
                                                                      length - i);

                    return true;
                }
            }
            return true;
        }
        return false;
    }
}
```

Listing 10. Implementation of the Feature Value Token Filter.

4.3.2 Custom Indexer

Lucene has an Index Writer class that is used to create and maintain the index. The IndexWriter class is initialized by setting the index directory that is used to create or append indexes; the configuration setting is shown in Listing 11. The custom analyzer can be assigned to the Index Writer in the configuration.

```
using (FSDirectory directory = FSDirectory.Open(indexDirectory))
{
    var freatureAnalyzer = new FeatureAnalyzer();
    var indexConfig = new IndexWriterConfig(LuceneVersion.LUCENE_48,
                                            freatureAnalyzer);

    //…
}
```

Listing 11. Assigning Feature Analyzer to the Index Writer.

The Index Writer configured with Feature Analyzer requires the custom indexing process chain to read the frequency/value attribute added to each feature/term token. In addition to the feature-value data, metadata of the traffic such as attack type can be stored in the index during the training phase, which is used to determine the details of the input traffic later during the detection phase. Customizing the Lucene.NET 4.8 indexing chain is a very convoluted process and involves customization of several components such as TermFrequencyAttribute, FieldInvertState, DocInverterPerField FreqProxTermsWriterPerField, TermVectorsConsumerPerField, and TermsHashPerField.

The customized indexing chain generates a feature-value vector for each piece of traffic data and stores it in the index directory. As shown in the customization architecture in

Figure 9, the searcher component uses the analyzer to search the stored index. In Lucene, searching depends on the similarity algorithms. There are several built-in similarity algorithms in Lucene such as TFIDFSimilarity, BM25Similarity, IBSimilarity, LMSimilarity, etc. TFIDFSimilarity is the default similarity in Lucene. The purpose of the similarity algorithm in the searcher is to score the relevancy of the stored data to the search query.

Lucene.NET has a component called Lucene.NET.Classification, which is used to classify text documents into groups. This component uses different algorithms such as KNN and Naïve Bayes. Although this component uses KNN or Naïve Bayes algorithms, the internal classification technique is based on "More-Like-This" algorithm. In Lucene search, More-Like-This algorithm is used to generate similar queries. A typical example of a More-Like-This algorithm is providing similar search suggestions, and "do you mean . . ." suggestions in search engines such as Google. Similar to the implementation in Lucene.NET, More-Like-This algorithm internally uses TFIDFSimilarity for document matching. The built-in classification implementation in the Lucene library does not function as a binary classifier for threat detection because the scoring algorithm is TFIDFSimilarity, which is based on Cosine similarity, term frequency, and inverse document frequency. TIFIDFSimilarity does not consider all the document terms to match with the query; this results in a very low detection accuracy if used as threat detection because it does not consider all the traffic data features in the search. Therefore, a custom similarity algorithm is required.

4.3.3 Custom Similarity

TFIDFSimilarity is based on Cosine similarity algorithm with VSM. In VSM, documents are represented as weighted vectors in a multidimensional space. In Chapter 3, we saw that Euclidean distance similarity outperformed the Cosine similarity using the KNN algorithm in a VSM-based intrusion detection prototype project. Euclidean similarity is not implemented in Lucene. Therefore, we need to implement Euclidean similarity by extending the base similarity component of Lucene.NET.

The base similarity class of Lucene, named SimilarityBase, provides a simple Application Programming Interface (API) for its derivative classes. Subclasses that extend SimilarityBase must apply their scoring formula by overriding the Score method. The Score method provides basic statistics of the query term, the matching term stored frequency, and the document length, as shown in Listing 12. Basic properties of the term called BasicStats contain the field name of the document containing the matching term, the matching term frequency, total number of occurrences of the term across all documents, document frequency, query boost value, etc.

```
public sealed class EuclideanDistanceSimilarity : SimilarityBase
{
    public override float Score(BasicStats stats, float freq, float docLen)
    {
        throw new NotImplementedException();
    }

    public override string ToString()
    {
        return "EuclideanDistanceSimilarity";
    }
}
```

Listing 12. Extending Lucene Similarity Base.

The Euclidean distance formula is the shortest distance between two points in a vector space. In terms of Lucene scoring, the Euclidean distance can be expressed as:

$$d = \sqrt{\sum_{i=1}^{n}(stats.TotalBoost - freq)^2}$$

Where *n* is the number of total terms/features.

The Score method exposes only the stored term frequency (freq), and the query term frequency that can be retrieved from stats.TotalBoost. The total terms and the entire vector context are not available to apply the summation and square root operations in this method. The total score of the document is calculated in the document scorers. Therefore, the Euclidean distance formula as it is cannot be implemented in the Score method. It is possible to compute the Euclidean distance formula in the document scorer components of Lucene such as Collectors. However, customizing the internals of document scorers is more involved. Therefore, we need to customize the Euclidean distance formula to fit into the Score method without losing the context. Let us start the customization by squaring both sides of the Euclidean distance equation, which gives us:

$$d^2 = \sum_{i=1}^{n} (stats.TotalBoost - freq)^2$$

Since the summation can be computed by the document score collector components, what we need to implement in the Score method is the score of the individual term score. If we compute a term score *s* as:

$$s = (stats.TotalBoost - freq)^2$$

We obtain the total document score of d^2, which is the square of the Euclidean distance. The purpose of measuring the Euclidean distance *d* of the given query vector is to find the nearest *K* vectors in the vector space to apply the KNN or other suitable algorithms for classification. What we achieved so far is the square of the Euclidean distance. However, we do not know if the square of Euclidean distance d^2 results in the same search result as Euclidean distance *d*. Therefore, we must prove that measuring the squared Euclidean distance d^2 produces the same nearest *K* vectors in the same order achieved by measuring Euclidean distance *d*. Finding the nearest vectors is attained by applying an inequality operation (less than or greater than) on the values. That means, for stored vectors *v1* and *v2*, and query vector *vq*, if *d1* is the distance between *vq* and *v1*, and *d2* is the distance between *vq* and *v2*, and if *d1* is less than *d2*, then we say *v1* is nearer to *vq*. What if $d1^2$ is less than $d2^2$? Is *v1* still be nearer to *vq*? This can be formulated as a theorem, and proving this theorem verifies the correctness of the Score calculation.

Theorem: For two Euclidean distances *d1* and *d2*, if $d1 \leq d2$, then $d1^2 \leq d2^2$**.**

Proof:

1: $d1, d2 > 0$ ……… $both\ d1\ and\ d2\ are\ measure\ of\ distances$

2: $d1 \leq d2$ ……… $given$

3: $d1 * d1 \leq d2 * d1$ ……… $multiply\ both\ sides\ by\ d1$

4: $d2 * d1 \leq d2 * d2$ … $(1\ \&\ 2)\ \&\ multiplication\ property\ of\ inequality$

5: $d1 * d1 \leq d2 * d1 \leq d2 * d2$ ……… $combining\ (3)\ and\ (4)$

6: $d1 * d1 \leq d2 * d2$ ……… $transitivity$

7: $d1^2 \leq d2^2$ ……… ■

Furthermore, since $\forall\, d1, d2 > 0 :\ d1 \leq\ d2 \Rightarrow d1^2 \leq d2^2$, the function $f(d) = d^2$ is a monotonically increasing function for $d > 0$. This implies that both the Euclidean distance and the square of Euclidean distance formulas produce the same set of K nearest vectors in VSM because the comparison yields the same result. Therefore, the Score formula of the Euclidean distance similarity can be computed as:

$$s\ =\ (stats.TotalBoost - freq)^2$$

Where s is score,

stats.TotalBoost is the term frequency in the query, and

freq is the term frequency in the matching document.

The Score method of the Euclidean distance similarity can be implemented as shown in Listing 13. Notice that the Score method implementation returns a negative value. This is intentional because in the nearest vector search, the minimum distance has higher relevancy. By default, Lucene searching sorts search results by descending score value, which means the farthest vector appears first. So, switching the sign reverses the search relevancy score, and the nearest vector appears first in the search result.

```
public sealed class EuclideanDistanceSimilarity : SimilarityBase
{
    public override float Score(BasicStats stats, float freq, float docLen)
    {
        return - (stats.TotalBoost - freq) * (stats.TotalBoost - freq);
    }

    public override string ToString()
    {
        return "EuclideanDistanceSimilarity";
    }
}
```

Listing 13. Implementation of Score method in Euclidean distance similarity.

Since the square root function, $f(x) = \sqrt{x}$ is also a monotonically increasing function for all $x > 0$, if we apply the square root on the score equation, it can be reduced to the arithmetic difference between the term frequencies as:

$$s\ = stats.TotalBoost - freq$$

However, this may result in negative values when the stored frequency is greater than the query term frequency. Therefore, the score needs to be the absolute value of the difference, and the modified score formula can be computed as:

$$s \ = |stats.TotalBoost - freq|$$

This formula is equal to the Manhattan distance formula. In the Manhattan distance formula, the distance between two points is the absolute difference of their cartesian coordinates and can be expressed as:

$$Manhattan\ distance\ (p, q) \ = \sum_{i=1}^{n} |\ p_i - q_i\ |$$

Where *p*, and *q* are points in n-dimensional space.

This shows that the Euclidean distance formula is logically reduced to the Manhattan distance formula in finding the *K* nearest vectors in VSM. Similarly, the score method of the Manhattan distance similarity in Lucene.NET can be implemented, as shown in Listing 14.

```
public sealed class ManhattanDistanceSimilarity : SimilarityBase
{
    public override float Score(BasicStats stats, float freq, float docLen)
    {
       return - Math.Abs(stats.TotalBoost - freq);
    }

    public override string ToString()
    {
        return "ManhattanDistanceSimilarity";
    }
}
```

Listing 14. Implementation of the Score method in Manhattan distance similarity.

Although the Euclidean and Manhattan distance formulas are logically related in finding the nearest *K* vectors in VSM, they may produce different set of *K* nearest vectors. The best similarity algorithm is chosen by running and analyzing different experiments, which will be discussed later in the evaluation and optimization section.

4.3.4 Custom Searcher

In Lucene, searching involves query parsing, term boosting, scoring, and sorting. The custom searcher component uses a specific searching technique. In the customized searching chain, the input query is provided as a single line string that contains a list of feature-value tokens. Then each feature-value item is added to a Boolean query as a term query. Every feature of the query should be considered in the document matching. This can be enforced by setting the search occur property to Occur.MUST if necessary. The value of each feature is added as a term boost in the query and is accessed in the similarity Score method. The searcher uses the custom similarity class specifically implemented for the nearest vector search. Every vector in the VSM is computed against the query vector, and the nearest *K* vectors are returned as a search result. Partial implementation of the customized searching chain is shown in Listing 15. Search performance optimization will be discussed later in the next chapter.

```
public List<NeighborVector> Search(string featureData)
{
    List<NeighborVector> nearestVectors = new List<NeighborVector>();

    string[] components = featureData.Split(',');
    string[] features = components[0].Split(' ');

    var query = new BooleanQuery();

    foreach (string featureValue in features)
    {
        string[] featureValueComponents = featureValue.Split('|');
        string feature = featureValueComponents[0];
        int value = Convert.ToInt32(featureValueComponents[1]);
        if (value > 0)
        {
            var featureQuery = new TermQuery(
                                   new Term(EngineConstants.FEATURES_NAME, feature));
            featureQuery.Boost = value;
            query.Add(featureQuery, Occur.SHOULD);
        }
    }

    TopDocs topDocs = searcher.Search(query, K);

    for (int i = 0; i < K; i++)
    {
        Document doc = searcher.Doc(topDocs.ScoreDocs[i].Doc);
        string label = doc.Get(EngineConstants.LABEL_NAME).ToString();
        string description = doc.Get(EngineConstants.DESCRIPTION_NAME).ToString();
        nearestVectors.Add(new NeighborVector
        {
            Label = label,
            Distance = Math.Abs(topDocs.ScoreDocs[i].Score),
            Description = description
        });
    }

    return nearestVectors;
}
```

Listing 15. Partial implementation of Searcher returning K nearest vectors.

4.4 Binary Classifier Engine

The proposed framework incorporates a machine learning-based binary classifier engine primarily used to train the search engine-based threat detection engine. The binary classifier engine is developed using the Microsoft ML.NET, as discussed in Chapter 3. The binary classifier engine also serves as reinforcement for the search engine-based classifier when its detection confidence is low. It has two major components: the model trainer and the predictor.

4.4.1 Model Trainer

The model trainer uses preprocessed data using the ML.NET data processor component discussed in Section 4.1. This section explains implementation of the model trainer component. The model trainer goes through three steps: first, the trainer creates a data reader and loads the input data into a DataView object. The feature values are processed as vectors, and the label is processed separately, as shown in Listing 1. After the data is loaded, in the second step, the model trainer goes through different algorithms such as normalization and classification algorithms. MinMaxNormalization algorithm is used to scale down the feature values based on observed minimum and maximum values of the data. FastTree classification algorithm is selected to train the model because FastTree is a decision tree-based classification algorithm. The literature review in Chapter 2 shows that decision tree-based algorithms outperform other algorithms in binary classification for intrusion detection. Parameter selection in the FastTree algorithm is a heuristic process. The initial values, such as the number of decision trees, leaves, and count per leaf, are set based on the prototype project discussed in Chapter 3. Further parametrization and optimization will be discussed in Chapter 5. Finally, the trained model is saved as an archive file. As of this writing, FastTree trainer algorithm is not in the list of retrainable algorithms in ML.NET (Microsoft, 2019g). Therefore, a new model needs to be trained with the updated dataset when retraining the ML-based binary classification engine is necessary.

```
public static void Train()
{
    var mlContext = new MLContext();

    // Load the data
    var reader = mlContext.Data.CreateTextLoader(
        columns: new TextLoader.Column[]
        {
            new TextLoader.Column(EngineConstants.FEATURES_NAME,
                                  DataKind.Single, 0,
                                  EngineConstants.FEATURES_LAST_INDEX),
            new TextLoader.Column(EngineConstants.LABEL_NAME,
                                  DataKind.Boolean,
                                  EngineConstants.LABEL_INDEX)
        },
        separatorChar: EngineConstants.FEATURE_DATA_SEPARATOR,
        hasHeader: false
    );
    var trainingDataset = reader.Load(AppConfigSettings.MlTrainingDataPath);

    //Train the Model
    var pipeline = mlContext.Transforms
                            .NormalizeMinMax(EngineConstants.FEATURES_NAME)
                            .AppendCacheCheckpoint(mlContext)
                            .Append(mlContext.BinaryClassification.Trainers
                                     .FastTree(numberOfLeaves: 70,
                                               numberOfTrees: 70,
                                               minimumExampleCountPerLeaf: 20));
    var model = pipeline.Fit(trainingDataset);

    //Save the model
    using (var fileStream = new FileStream(AppConfigSettings.MLModelPath,
                                           FileMode.Create,
                                           FileAccess.Write, FileShare.Write))
    {
        mlContext.Model.Save(model, trainingDataset.Schema, fileStream);
    }
}
```

Listing 16. ML Model Trainer implementation.

The accuracy and performance of the trained model is measured by running the model evaluator method against the test data, as implemented in Listing 17. The model evaluator loads the trained model that was created by the model trainer. Then for each input data the model trainer compares the prediction result with the original label value and populates the binary classification metrics object. The details of the evaluation metrics will be discussed later in Chapter 5.

```
public static EvaluationMetrics Evaluate()
{
    var mlContext = new MLContext();

    // Create data reader.
    var reader = mlContext.Data.CreateTextLoader(
        columns: new TextLoader.Column[]
        {
            new TextLoader.Column(EngineConstants.FEATURES_NAME,
                                  DataKind.Single, 0,
                                  EngineConstants.FEATURES_LAST_INDEX),
            new TextLoader.Column(EngineConstants.LABEL_NAME,
                                  DataKind.Boolean,
                                  EngineConstants.LABEL_INDEX)
        },
        separatorChar: EngineConstants.FEATURE_DATA_SEPARATOR,
        hasHeader: false
    );

    DataViewSchema modelSchema;
    var trainedModel = mlContext.Model.Load(AppConfigSettings.MLModelPath,
                                            out modelSchema);

    //Evaluate the Model
    var testingDataset = reader.Load(AppConfigSettings.MlTestingDataPath);

    IDataView predictions = trainedModel.Transform(testingDataset);

    CalibratedBinaryClassificationMetrics metrics = mlContext
                                         .BinaryClassification
                                         .Evaluate(predictions,
                                                   EngineConstants.LABEL_NAME);


    //Test speed of the prediction engine
    List<(string ActualValue, string PredictedValue)> evaluationResult =
                              new List<(string, string)>();
    IEnumerable<string> testDataList = File.ReadAllLines(
                                       AppConfigSettings.MlTestingDataPath);
    int testDataCount = testDataList.Count();

    var watch = Stopwatch.StartNew();

    var predictor = ModelPredictor.Instance;

    foreach (string dataLine in testDataList)
    {
        string[] components = dataLine.Split(',');
        string features = string.Join(",", components.Take(components.Count()-1));
        string label = components.Last();

        FeatureData featureData = new FeatureData
        {
            MLFeaturesString = features,
            Features = features.Split(EngineConstants.FEATURE_DATA_SEPARATOR)
                               .Select(val => Convert.ToSingle(val)).ToArray(),
            Label = label
        };
        MLPrediction predictionResult = predictor.Predict(featureData);

        string prediction = predictionResult.PredictedLabel? "1" : "0";
        evaluationResult.Add((label, prediction));
    }

    watch.Stop();
    double elapsedMs = watch.Elapsed.TotalMilliseconds / testDataCount;

    EvaluationMetrics evaluationMetrics = MetricsCalcualtor
                                          .Compute(evaluationResult);

    evaluationMetrics.EngineName = "ML.NET";
    evaluationMetrics.AUC = metrics.AreaUnderRocCurve;
    evaluationMetrics.AUPRC = metrics.AreaUnderPrecisionRecallCurve;
    evaluationMetrics.DetectionSpeedPerSample = elapsedMs;

    return evaluationMetrics;
}
```

Listing 17. Implementation of model evaluation.

4.4.2 Predictor

The predictor component of the binary classification engine is used to predict the status of the input traffic data. The predictor uses the trained model. The input traffic features data extracted from the FeatureExtractor are in a text format. The input data is converted to an input object called FeatureData. Instead of creating one property for each feature, all features are combined as a single float vector, as shown in Listing 18. This reduces the complexity of data conversion and makes the predictor model less dependent on the input data structure.

```
public class FeatureData
{
    public string RawFeatureString { get; set; }
    public string SEFeaturesString { get; set; }
    public string MLFeaturesString { get; set; }
    public string Label { get; set; }
    public string Description { get; set; }

    [LoadColumn(0, EngineConstants.FEATURES_LAST_INDEX)]
    [VectorType(EngineConstants.FEATURES_LAST_INDEX + 1)]
    public float[] Features { get; set; }
}
```

Listing 18. Feature Data class.

The Predict method in the Predictor class accepts input traffic feature data as a string value. The feature data then converts to a FeatureData object and is passed to the prediction engine. Finally, the prediction engine returns the prediction result. As shown in Listing 19, the ModelPredictor class uses a thread-safe singleton[3] pattern, which means no instance of this class is created in the application domain during prediction; only one instance of this class stays in the memory and performs prediction. The reason for this is to improve performance because reconstructing the prediction engine at every prediction slows down the detection process.

```
public sealed class ModelPredictor : CoreMLEngine
{
    private static PredictionEngine<FeatureData, MLPrediction> predictionEngine
    private static volatile ModelPredictor instance;
    private static readonly object syncLock = new object();
    private static readonly object threadLock = new object();
    private static MLContext mlContext;
    private ModelPredictor()
    {
         mlContext = new MLContext();
        var trainedModel = mlContext.Model.
            Load(AppConfigSettings.MLModelPath, out DataViewSchema modelSchema);
        predictionEngine = mlContext.Model
            .CreatePredictionEngine<FeatureData, MLPrediction>(trainedModel);
    }
    public static ModelPredictor Instance
    {
        get
        {
            if (instance == null)
            {
                lock (syncLock)
                {
                    if (instance == null)
                        instance = new ModelPredictor();
                }
            }
            return instance;
        }
    }

    public MLPrediction Predict(FeatureData featureData)
    {
        lock (threadLock)
        {
            return predictionEngine.Predict(featureData);
        }
    }
}
```

Listing 19. Implementation of ML Model Predictor.

[3] https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff650316(v=pandp.10)

4.5 Decision Engine

The decision engine serves as a dispatcher in the detection process. The decision engine first receives the input traffic data and sends it to the search engine searcher and collects the nearest matches. The decision engine applies the KNN algorithm on the returned search results to determine the status of the input traffic data. The decision engine also computes the confidence level of the prediction. When the confidence level is high, the prediction result becomes final. However, when the confidence level is low, the decision engine sends the input traffic data to the binary classifier engine, the binary classifier engine returns the final prediction, and the input traffic data is sent to the search engine for training. Therefore, the decision engine is responsible for continuously retraining the search engine, as discussed in Chapter 3 and shown in Figure 5's activity diagram.

4.5.1 Detection Engine

The decision engine has a detection engine that runs several subprocesses such as detecting threats, sending alerts, and triggering retraining. Performance is critical for the detection engine, especially when analyzing live traffic. Therefore, the detection engine runs in a multithreaded context where operations run in parallel. In multithreaded contexts, multiple tasks can run asynchronously without blocking each other. The detection engine creates a detection process task for every input traffic. This task is separate from the main thread. This makes the detection engine concurrently run multiple detection process tasks. Each detection process task creates another threat detection task that is responsible for executing detection algorithms and returning prediction results. After the threat detector task completes, the returned prediction result passes to other child tasks such as alert sender task and search engine retrainer task; the child tasks run in parallel. The predictor and the searcher engines are developed to be thread-safe using a singleton design pattern to handle multiple threads at the same time.

The .NET Framework supports multithreading and asynchronous programming and has several kinds of implementations. In .NET Framework there are three concepts of parallel programming: Thread, ThreadPool, and Task. Thread represents the low-level Operating System (OS) thread that allows programs the highest degree of control. ThreadPool is a wrapper of a pool of Threads maintained by the Common Language Runtime (CLR), which can be used to execute Tasks, asynchronous activities, and process timers. Task provides the benefits of both Thread and ThreadPool. Task provides efficient and scalable use of system resources, provides better control, and has a rich set of APIs. In the .NET Framework, Task Parallel Library (TPL), which is based on the concept of Tasks and asynchronous operation, is the preferred API for multithreaded, asynchronous, and parallel programming (Microsoft, 2017). Therefore, the decision engine is developed based on task-based asynchronous programming. The partial implementation of the detection engine component (the core of the decision engine) is shown in Listing 20.

```
public static EngineName ACTIVE_DETECTION_ENGINE = EngineName.SEML;
private static ConcurrentBag<Task> concurrentTasks = new ConcurrentBag<Task>();
public static ConcurrencyMethod CONCURRENCY_METHOD = ConcurrencyMethod.Ct;
private static readonly object trainerThreadLock = new object();

public static async Task RunDetectionEngineWithCt()
{
    while (!STOP_ENGINE)
    {
        if (concurrentTasks.Count > 0)
        {
            RUNNING = true;
            var firstFinishedTask = await Task.WhenAny(concurrentTasks);
            concurrentTasks.TryTake(out firstFinishedTask);
            await firstFinishedTask;
        }
    }
}
public static void DetectWithCt(FeatureData featureData)
{
    if(!STOP_ENGINE)
       concurrentTasks.Add(DetectTask(featureData));
}
private static void Detect(FeatureData featureData)
{
    DetectTask(featureData);
}
private static Task DetectTask(FeatureData featureData)
{
    var threatDetectorTask = Task.Run(() =>
    {
        //… Please see the implementation in Listing 21.
    });

    return threatDetectorTask;
}
```

Listing 20. Partial implementation of the detection engine.

4.5.2 Decision Algorithms

The decision engine uses two prediction models: the search engine and the ML engine. The search engine uses KNN as a primary classifier algorithm, and the ML binary classifier uses Fast Tree-based binary classifier algorithm. The prediction confidence of the search engine is computed as a percentage value of the number of the nearest vector labels that match the predicted label. When the confidence level is lower than a preset value, the feature data goes through the ML binary classifier. If the binary classifier returns different prediction results, the search engine is retrained. The value of the confidence level threshold determined by running different experiments will be discussed later in Chapter 5. Listing 21 shows the implementation of the detection method.

```
private static Task DetectTask(FeatureData featureData)
{
    var threatDetectorTask = Task.Run(() =>
    {
        PredictionResult predictionResult = null;
        var predictedLabel = string.Empty;
        if (ACTIVE_DETECTION_ENGINE == EngineName.SE)
           {
               predictionResult = searcher.Predict(featureData.SEFeaturesString);
               predictedLabel = predictionResult.Label;
               predictionResult.EngineName = "SE";
           }
        else if (ACTIVE_DETECTION_ENGINE == EngineName.SEML)
        {
            predictionResult = searcher.Predict(featureData.SEFeaturesString);
            predictionResult.EngineName = "SE";
            predictedLabel = predictionResult.Label;
            if (predictionResult.Confidence < GOOD_CONFIDENCE_PERCENTAGE)
            {
                var mlPrediction = mlPredictor.Predict(featureData);
                predictedLabel = mlPrediction.PredictedLabel ? "1" : "0";
                if (predictionResult.Label != predictedLabel)
                {
                    predictionResult = mlPrediction;
                    predictionResult.Confidence = mlPrediction.Probability * 100;
                    predictionResult.EngineName = "ML";
                    // … Call Retrainer.
                    // Please see the implementation in Listing
29.
                }
            }
        }
        else
        {
            var mlPrediction = mlPredictor.Predict(featureData);
            predictionResult = mlPrediction;
            predictionResult.Confidence = mlPrediction.Probability * 100;
            predictionResult.EngineName = "ML";
            predictedLabel = mlPrediction.PredictedLabel ? "1" : "0";
        }

        string[] components = featureData.RawFeatureString.Split(',');
        predictionResult.SourceIp = components[0];
        predictionResult.SourcePort = components[1];
        predictionResult.DestinationIp = components[2];
        predictionResult.DestinationPort = components[3];
        predictionResult.Protocol = components[4];
        predictionResult.TimeStamp = DateTime.Now.ToString();
        predictionResult.FlowDuration = components[5];
        predictionResult.TotalForwardPackets = components[6];
        predictionResult.TotalBackwardPackets = components[7];
        predictionResult.TotalForwardPacketsLength = components[8];
        predictionResult.TotalBackwardPacketsLength = components[9];
        predictionResult.Label = predictedLabel;

        SendAlert(predictionResult);
    });

    return threatDetectorTask;
}
```

Listing 21. Implementation of threat detection method.

When the search engine retrainer method is called, the input feature data along with the prediction label is collected to retrain the search engine. The search engine retraining is processed by updating the search engine index. Unlike the ML binary classification model, the search engine can be trained without recreating the index. This is one benefit of the search engine model. However, the updated index is not available as soon as the index is updated because the searcher is implemented using the singleton pattern, which does not release previous indexes stored in the memory for performance reasons. This process will be discussed in detail later in Chapter 5 section 5.4.1 Continuous Training.

4.6 Alert Service

Although intrusion detection systems monitor network traffic for suspicious activities, they may not be capable of stopping the activity from further propagation. Usually intrusion detection systems issue alerts when potentially malicious activities or anomalies are detected based on their configurations. Alerts could be in the form of emails, texts, log files, dashboards, etc.

The alert service in the proposed framework is implemented by extending the log4net library. Log4net is a high-performance logging library developed based on the Apache log4j logging library, which has been in development since 1996. Log4net supports multiple logging targets such as the Windows console screen, trace pages, log files, Windows Event Log, Windows Messenger service, syslog service (Linux), email address (SMTP services), memory buffer, etc. (Apache Software Foundation, 2017). Log4net uses a configuration file to set up logging targets, formatting, and other settings. Listing 22 shows the initial setup of log4net, which reads settings from the log4net config file. The config file contains different logging targets and settings, which means the custom configured logger can send email and text messages, as well as trigger alerts to connected dashboards.

```
public class AlertLogManager
{
    private static readonly log4net.ILog log = log4net.LogManager
                                              .GetLogger(typeof(AlertLogManager));
    private static bool isLoggerConfigured;
    public static log4net.ILog Log
    {
        get
        {
            if (!isLoggerConfigured)
            {
                XmlDocument log4netConfig = new XmlDocument();
                log4netConfig.Load(File.OpenRead("log4net.config"));
                var repository = log4net.LogManager
                               .CreateRepository(Assembly.GetEntryAssembly(),
                                typeof(log4net.Repository.Hierarchy.Hierarchy));
                log4net.Config.XmlConfigurator.Configure(repository,
                                              log4netConfig["log4net"]);
                isLoggerConfigured = true;
            }
            return log;
        }
    }
}
```

Listing 22. Log4net logger instance configuration.

In the customized logger, when a threat is detected, the alert service writes the alert message in a log file and in a memory buffer through the log4net MemoryAppender component. The log4net MemoryAppender class has been extended to handle a custom event when a new log message is added to the log, as shown in Listing 23.

```
public class AlertLogMemoryAppender : MemoryAppender
{
    public event EventHandler LogUpdated;

    protected override void Append(LoggingEvent loggingEvent)
    {
        base.Append(loggingEvent);
        LogUpdated?.Invoke(this, new EventArgs());
    }
}
```

Listing 23. Custom MemoryAppender implementation.

There is a separate class called AlertLogWatcher that monitors the activity of the custom MemoryAppender class, called AlertLogMemoryAppender. This class intercepts the newly appended event by handling the custom LogUpdated event of the MemoryAppender class, as shown in Listing 24.

```
public class AlertLogWatcher
{
    private AlertLogMemoryAppender memoryAppender;
    public event EventHandler LogUpdated;
    public string LogContent { get; private set; }

    public AlertLogWatcher()
    {
        memoryAppender = (AlertLogMemoryAppender)Array.Find(
               AlertLogManager.Log.Logger.Repository.GetAppenders(),
              (appender) => appender.Name.Equals("AlertLogMemoryAppender"));

        LogContent = GetEvents(memoryAppender);
        memoryAppender.LogUpdated += HandleLogUpdate;
    }

    public void HandleLogUpdate(object sender, EventArgs e)
    {
        LogContent = GetEvents(memoryAppender);
        LogUpdated?.Invoke(this, new EventArgs());
    }

    public string GetEvents(AlertLogMemoryAppender memoryAppender)
    {
        StringBuilder output = new StringBuilder();
        LoggingEvent[] logEvents = memoryAppender.GetEvents();
        if (logEvents != null && logEvents.Length > 0)
        {
            memoryAppender.Clear();
            foreach (LoggingEvent ev in logEvents)
            {
                output.Append(ev.RenderedMessage);
            }
        }
        return output.ToString();
    }
}
```

Listing 24. Implementation of AlertLogWatcher class.

The main purpose of the AlertLogWatcher class is to expose the newly added event to external services, such as the monitoring application and dashboards. By using the AlertLogWatcher class, external services are notified in real time when a new log event is triggered. This will be discussed in the next section.

4.7 Monitor Application

To block detected malicious activities, automated systems such as Intrusion Prevention Systems (IPS) or manual activities may be required. Discussing IPS is beyond the scope of this research; however, in order to demonstrate the validity of the proposed framework, a simple monitor application that shows the detection activities and provides control services has been developed. The monitor application is developed using Web technologies such as JavaScript, HTML, and CSS so that it can be remotely accessed through HTTP on different platforms such as Desktops, Tablets, and Mobile devices.

The monitor application uses WebSocket technology for real-time communication with the detection engine. According to the Internet Engineering Task Force (IETF, 2011), WebSocket is a protocol that enables two-way communication channels over TCP connections that do not rely on opening multiple HTTP connections. WebSocket technology is used in real-time communication apps such as chat, dashboard, and stock ticker apps. In .NET Framework, WebSocket can be implemented using different libraries such as SignalR. The ASP.NET Core SignalR is an opensource library that supports WebSocket as a real-time web functionality to applications. SignalR applies “server push” functionality using Remote Procedure Calls (RPC) rather than the request-response model. SignalIR uses a high-level pipeline called a hub, which allows method calls between the client and server communication. WebSocket is fully supported in Microsoft Internet Explorer, Google Chrome, and Mozilla Firefox and partially supported in Opera and Safari. When the client browser does not support WebSocket, SignalR falls back to other transports such as Server-Sent Events, ForeverFrame, and Ajax Long Polling (Microsoft, 2014, 2018).

The monitor application is an ASP.NET Core application. The application serves as a dashboard and a control board to the threat detection system. The application can send commands such as to restart and stop services and enables the system administrators to control the entire detection system using browsers. A high-level process flow diagram of the monitor application is shown in Figure 10.

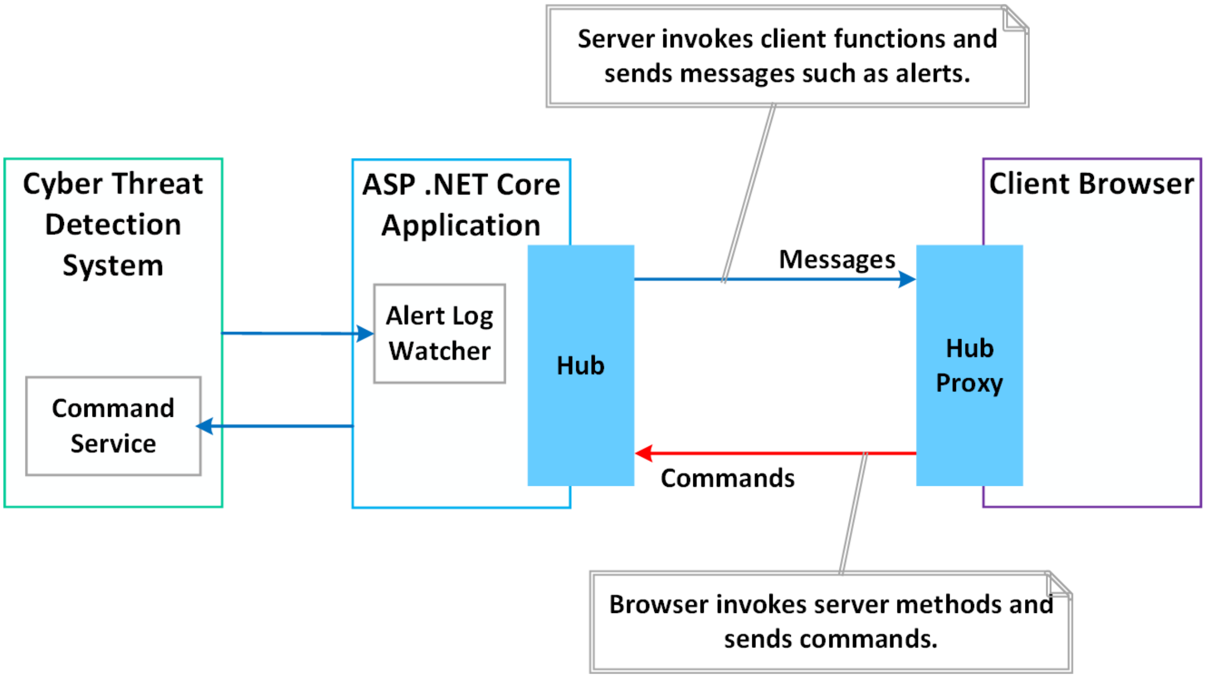

Server invokes client functions and sends messages such as alerts.
Cyber Threat Detection System
Command Service
ASP .NET Core Application
Alert Log Watcher
Hub
Messages
Commands
Client Browser
Hub Proxy
Browser invokes server methods and sends commands.

Figure 10. Process flow diagram of the monitor application.

The service hub class in the ASP.NET Core application has an instance of the AlertLogWatcher class, as shown in Listing 25. When an alert is triggered, the AlertLogWatcher_Updated method sends the alert message to all connected clients of the monitor application by invoking their ReceiveAlert function.

```
public class ServiceHub : Hub
{
    private AlertLogWatcher alertLogWatcher;
    protected IHubContext<ServiceHub> _context;
    public ServiceHub(IHubContext<ServiceHub> context)
    {
        if(alertLogWatcher == null)
        {
            alertLogWatcher = new AlertLogWatcher();
            alertLogWatcher.LogUpdated += AlertLogWatcher_Updated;
            EngineService.StartDetectionEngineLiveTraffic(
                        activeDetectionEngine: EngineName.SEML);
            //EngineService.StartDetectionEngineSimulation();
        }
        _context = context;

    }

    private async void AlertLogWatcher_Updated(object sender, EventArgs e)
    {
        var connectedClients = Startup.hubContext.Clients;
        if (connectedClients != null)
        {
            string msg = alertLogWatcher.LogContent;
            await connectedClients.All.SendAsync("ReceiveAlert", msg);
        }
    }
}
```

Listing 25. Partial implementation of SignalR Service Hub.

In this context, clients are user browsers. The monitor application can be hosted in a web server, so that clients can access it through a URL. On the first page load event, the connected client makes a call to the service hub through the hub URL. Once the connection is established, the client can send messages to, and receive messages from, the server in real time through the JavaScript SignalR component, as shown in Listing 26.

```
$(document).ready(function () {
    adjustResultLayout();

    var connection = new signalR.HubConnectionBuilder()
        .withUrl("/service-hub").build();

    connection.on("ReceiveAlert", function (alert) {
        displayAlert(alert);
    });

    connection.start().then(function () {
    }).catch(function (err) {
        return console.error(err.toString());
    });

});
```

Listing 26. Partial implementation of client-side service.

4.8 Summary

This chapter discussed implementation of algorithms and components for the proposed cyber threat detection framework. The main components of the framework are the data processor, the two core detection engines—the search engine and ML engine—the decision engine, alert service, and monitor application. Each component is designed and developed to be loosely coupled to one another. The search engine customization involved the introduction of new similarity algorithms and the customization of default components of Lucene such as Analyzers, Tokenizers, and the Indexing process chain. Implementation of the ML.NET-based binary classifier engine is easier than the search engine customization. The decision engine is multithreaded to speed up the detection process. When an attack is detected, the alert service records the event in a multitargeted logger service. The detection system has a monitor application subscribed to the logger service to watch alerts. If alerts are pushed, the monitor application notifies connected clients in real time. The real-time communication of the client and server is implemented using WebSocket and SignalR library. In the next chapter we will discuss the evaluation, optimization approaches, and validation of the developed framework.

This concludes Chapter 4. System Development. This chapter discussed implementation of the new framework and its components as designed in Chapter 3. This chapter involved mathematical analysis and the development of algorithms and components.

CHAPTER 5

CASE STUDY

Chapter 4 provided implementation of the core framework and its components. Each component was tested for its basic functionality during development. This chapter discusses several additional test cases such as performance, detection accuracy, scalability, load testing, and overall system validation. Setting up evaluation metrics, optimizing techniques, empirical analysis and performance results of the framework are also discussed in this chapter.

5.1. Initial Evaluation

According to Microsoft documentation (Microsoft, 2019e), metrics for the binary classification model include accuracy, precision, recall, Area Under the Curve (AUC), Area Under the Curve of a Precision Recall Curve (AUCPR), and F1-Score. In general, the performance of an IDS is evaluated in terms of detection accuracy and rate. Detection accuracy can be further analyzed by positive prediction and negative prediction rates based on the context of the problem.

5.1.1. Evaluation Metrics

Definition: In the context of this model evaluation, an attack instance is treated as a positive class, and a benign instance is treated as a negative class.

True Positive (TP): A prediction outcome where the model correctly predicts the positive class (attack class). It is a measure of positive (attack) instances detected accurately.

False Positive (FP): A prediction outcome where the model incorrectly predicts the positive class. It is a measure of negative (benign) instances detected as positive (attack).

True Negative (TN): A prediction outcome where the model correctly predicts the negative class (benign class). It is the measure of negative (benign) instances detected accurately.

False Negative (FN): A prediction outcome where the model incorrectly predicts the negative class. It is a measure of positive (attack) instances detected as negative (benign).

Accuracy (ACC): The percentage of total number of instances correctly classified. It is the proportion of the predictions of the model got right. The closer the accuracy is to 100% the better. A 100% accuracy, however, could indicate issues such as label leakage, model overfitting, or testing with the training data. An unbalanced or very small amount of test data could make the accuracy approach the extremes of 0 or 100% (Microsoft, 2019e).

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

Precision (Pr): The proportion of correctly predicted positive (attack) classes to the total number of instances predicted as positive (attack).

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

Recall (Rc): The proportion of actual positives (attacks) predicted correctly to the total number of correctly predicted positives or incorrectly predicted negatives. Recall is also called the True Positive Rate (TPR).

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

False Positive Rate (FPR): The measure of the proportion of negative (benign) instances that are correctly predicted.

$$FPR = \frac{FP}{FP + TN}$$

F1-Score (F1): The harmonic mean of the Precision and Recall. It indicates the balance between the Precision and Recall.

$$F1 = \frac{2 * TPR * Pr}{Pr + TPR}$$

Receiver Operating Characteristic (ROC) Curve: A graph that shows the performance of a binary classification model at all classification thresholds. Figure 11 shows the ROC curve. It plots the TPR or Recall versus FPR. As the ROC indicates, lowering the classification threshold makes the model classify more positives.

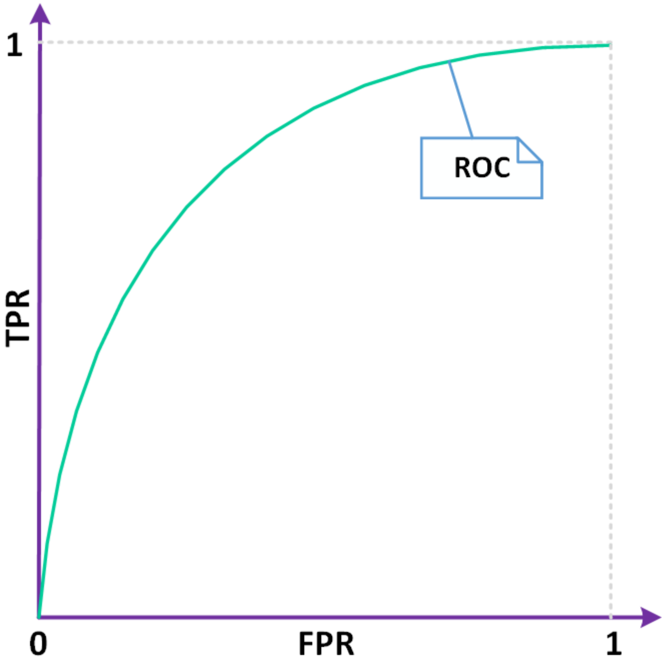


Figure 11. ROC curve.

Area Under the ROC Curve (AUC): The measure of the area under the curve created by sweeping the True Positive Rate or Recall and the False Positive Rate. For acceptable models, AUC should be greater than 0.50. The closer the AUC is to 1.0, the better.

If we let *x* be FPR and TPR be *f(x),* the area of the region under the curve can be calculated as the integral of *f(x)* by *dx*.

$$Area = \int_0^1 f(x)dx$$

$$\Rightarrow AUC \ = \int_0^1 TPR(FPR)d(TPR)$$

The ML.NET documentation defines AUC as the probability that the binary classifier ranks a randomly chosen positive instance higher than a randomly chosen negative instance, and that AUC has been implemented accordingly in the ML.NET source code.[4]

Area Under the Curve of Precision Recall Curve (AUCPR): The measure of the success of prediction when the classes are very imbalanced. An AUCPR value closer to 1.00

[4] https://github.com/dotnet/machinelearning/blob/610ffcb67083c2e5e6e1a14884ba24b1da0384c7/src/Microsoft.ML.Data/Evaluators/BinaryClassifierEvaluator.cs

shows that the binary classifier is returning high-precision accurate results, as well as returning the majority of all positive results (high recall).

Average Detection Speed Per Sample: The average time the prediction engine takes to predict the class of a single instance. Speed depends on several factors such as the testing computer performance and its configurations.

$$Detection\ Speed = \frac{Total\ detection\ time}{Number\ of\ instances}$$

Implementation of algebraic metrics for both the search engine and binary classifier engine are shown in Listing 27.

```
public class MetricsCalcualtor
{
    public static EvaluationMetrics Compute(IEnumerable<(string ActualValue,
                                    string PredictedValue)> evaluationResult)
    {
        double tp = evaluationResult.Where(item => (item.PredictedValue == "1")
                                           && item.ActualValue == "1").Count();
        double fp = evaluationResult.Where(item => (item.PredictedValue == "1")
                                           && item.ActualValue == "0").Count();
        double tn = evaluationResult.Where(item => (item.PredictedValue == "0")
                                           && item.ActualValue == "0").Count();
        double fn = evaluationResult.Where(item => (item.PredictedValue == "0")
                                           && item.ActualValue == "1").Count();

        double acc = (tp + tn) / (tp + tn + fp + fn);
        double pr = tp / (tp + fp);
        double tpr = tp / (tp + fn);
        double fpr = fp / (fp + tn);
        double f1Score = (2 * tpr * pr) / (pr + tpr);

        EvaluationMetrics metrics = new EvaluationMetrics
        {
            Accuracy = acc,
            Precision = pr,
            TPR = tpr,
            FPR = fpr,
            F1Score = f1Score
        };

        return metrics;
    }
}
```

Listing 27. Implementation of evaluation metrics.

5.1.2. Model Evaluation

For the initial evaluation, both the search engine and the ML binary classification models are trained with the same dataset. A total of 129,924 training instances and 55,253 testing instances, which is one third of the training instances, are used for building and evaluating the models. The datasets contain different kinds of network attacks such as DoS/DDoS, brute force Cross Site Scripting (XSS), SQL injection, port scan, infiltration, and FTP/SSH patator.[5] The class balance for both the training and testing datasets is one to one, which means the number of attack and benign instances are equal in each dataset. Initial parameters of the selected algorithms to train the initial models are shown in Table 7. These parameters are set based on the prototype project and theoretical assumptions.

Table 7. Initial Model Parameters

Model	**Algorithms and Parameters**	
Search Engine	**Classification Algorithm**	KNN
	Similarity Algorithm	Squared Euclidean Distance
	Number of Neighbors (K)	69
	Vector Dimension	Fixed
ML.NET	**Classification Algorithm**	FastTree (Decision Tree)
	Number of Trees	50
	Number of Leaves	50
	Sample Count Per Leaf	20

Table 8 shows the initial model evaluation result before applying any optimization. A Windows 10 developer workstation with 16GB RAM and 3.1GHz processor speed was used in this evaluation. The detection time per instance is the amount of time spent in milliseconds to process a single traffic instance in a synchronous operation. Unlike the other evaluation metrics, the detection time changes at every test run. The listed detection speed is the average value.

[5] Patator is a Python script used to make multiple brute force attacks. https://en.kali.tools/?p=147

Table 8. Initial Evaluation of Classification Models

Metrics	**Detection Models**	
	Search Engine	**ML.NET**
Accuracy	0.86	0.89
Precision	0.91	0.89
True Positive Rate (Recall)	0.81	0.87
False Positive Rate (FPR)	0.09	0.10
F1 Score	0.86	0.88
Detection Time Per Instance (ms)	531.52	145.68

The initial evaluation result shows that the search engine is slower than the ML.NET model. As the search engine is using VSM, this is expected because each vector in the VSM needs to be visited to compute distances from the input vector. Detection accuracy and speed can be improved by applying different optimization techniques, which will be discussed in the next section.

5.2. Optimization

The search engine uses the VSM model and KNN algorithm for binary classification. The time complexity (big O notation) *O* of the model to detect a single instance of traffic can be expressed as:

$$Complexity = O(n * f) \ \Rightarrow O(n)$$

Where *n* is the number of training instances (number of vectors in the VSM), and *f* is the number of features of each instance.

The time complexity of the VSM model can be optimized by either reducing the number of visited vectors *n* or reducing the number of less important features *f*.

5.2.1 Feature Reduction

Feature reduction is a process of selecting the most important features by excluding the less important features. Feature reduction reduces noise and training time, increasing the performance of the model. In machine learning, the internals of feature processing are less clear, and the models are often considered black boxes. The most important features can be selected by randomly shuffling data and calculating the delta on the performance metrics; the larger the delta is, the more the important the feature (Breiman, 2001). ML.NET classification and regression models provide feature details such as feature weight and change in metrics using Permutation Feature Importance (PFI) technique (Microsoft, 2019f). For the binary classification model, PFI computes all possible classification evaluation metrics for each feature by running a given number of iterations called the permutation count.

The PFI runs after training the first model. To obtain the importance of each feature, thirty permutations have been performed and each evaluation metric was collected as shown in the implementation Listing 28. Then the result was ordered by change in performance.

```
public static List<string> ComputePermutationFeatureImportance()
{
    var mlContext = new MLContext();

    // Load the data
    var reader = mlContext.Data.CreateTextLoader(
        columns: new TextLoader.Column[]
        {
            new TextLoader.Column(EngineConstants.FEATURES_NAME,
                                  DataKind.Single, 0,
                                  EngineConstants.FEATURES_LAST_INDEX),
            new TextLoader.Column(EngineConstants.LABEL_NAME,
                                  DataKind.Boolean,
                                  EngineConstants.LABEL_INDEX)
        },
        separatorChar: EngineConstants.FEATURE_DATA_SEPARATOR,
        hasHeader: false
    );
    var trainingDataset = reader.Load(AppConfigSettings.MlTrainingDataPath);

    //Train the Model
    var pipeline = mlContext.Transforms.NormalizeMinMax(
                                        EngineConstants.FEATURES_NAME)
                            .AppendCacheCheckpoint(mlContext)
                            .Append(mlContext.BinaryClassification.Trainers
                                    .FastTree(numberOfLeaves: 50,
                                          numberOfTrees: 50,
                                          minimumExampleCountPerLeaf: 20));
    var model = pipeline.Fit(trainingDataset);

    var transformedData = model.Transform(trainingDataset);
    var linearPredictor = model.LastTransformer;

    var permutationMetrics = mlContext.BinaryClassification
```

```
        .PermutationFeatureImportance(linearPredictor, transformedData,
        permutationCount: 30);

    var featureDetails = permutationMetrics.Select((metrics, index) => new
        {
            Index = index,
            AUC = metrics.AreaUnderRocCurve,
            metrics.Accuracy,
            metrics.F1Score,
            Precision = metrics.PositivePrecision,
            TPR = metrics.PositiveRecall,
            FPR = metrics.NegativeRecall
        }).OrderByDescending(feature => Math.Abs(feature.AUC.Mean))
          .ThenByDescending(feature => Math.Abs(feature.Accuracy.Mean))
          .ThenByDescending(feature => Math.Abs(feature.F1Score.Mean))
          .ThenByDescending(feature => Math.Abs(feature.Precision.Mean))
          .ThenByDescending(feature => Math.Abs(feature.TPR.Mean))
          .ThenByDescending(feature => Math.Abs(feature.FPR.Mean));

    string featureDetailsString = "Feature\tWeight\tAUC\tAccuracy\tF1Score" +
                                  "\tPrecison\tTPR\tFPR\tAUC-Confidence";
    List<string> featuresDetailList = new List<string>();
    featuresDetailList.Add(featureDetailsString);
    Console.WriteLine(featureDetailsString);

    VBuffer<float> featureWeights = new VBuffer<float>();
    linearPredictor.Model.SubModel.GetFeatureWeights(ref featureWeights);
    foreach (var fd in featureDetails)
    {
        string detailsString = string.Format("{0}\t{1:0.00}\t{2:G4}\t{3:G4}" +
            "\t{4:G4}\t{5:G4}\t{6:G4}\t{7:G4}\t{8:G4}",
             fd.Index,
             featureWeights.GetValues()[fd.Index],
             Math.Abs(fd.AUC.Mean),
             Math.Abs(fd.Accuracy.Mean),
             Math.Abs(fd.F1Score.Mean),
             Math.Abs(fd.Precision.Mean),
             Math.Abs(fd.TPR.Mean),
             Math.Abs(fd.FPR.Mean),
             1.96 * fd.AUC.StandardError);
        featuresDetailList.Add(detailsString);
        Console.WriteLine(detailsString);
    }
    File.WriteAllLines(AppConfigSettings.FeaturesImportancePath,
                        featuresDetailList);
    return featuresDetailList;
}
```

Listing 28. Partial implementation of Permutation Feature Importance (PFI).

There is a total of 78 features used in each instance of the initial training dataset. These features were generated from PCAP files using CICFlowMeter. Although CICFlowMeter generates over 80 features, the PFI result shows that eighteen features have no impact on any of the evaluation metrics. The complete result of features and importance metrics is listed in Appendix B: List of Features. The eighteen least important features have been removed from the dataset, and the models are retrained with the remaining 60 features. The evaluation result in Table 9 clearly shows that accuracy was not reduced because the removed features are not important. As a result of feature reduction, the search engine detection time was improved by 186 milliseconds on average per detection, and the overall performance of ML.NET model was also improved.

Table 9. Model Evaluation Metrics Using Important Features

Metrics	**Detection Models**	
	Search Engine	**ML.NET**
Accuracy	0.86	0.92
Precision	0.91	0.96
True Positive Rate (Recall)	0.81	0.87
False Positive Rate (FPR)	0.09	0.04
F1 Score	0.86	0.91
Detection Time Per Instance (ms)	345.08	117.70

5.2.2 Euclidean Distance versus Manhattan Distance

In Chapter 4, we discussed similarity algorithms, so we have seen that the square of Euclidean distance can be logically reduced to Manhattan distance in searching for nearest neighbors. This means that both the squared Euclidean distance and the Manhattan distance formulas yield the same set of nearest vectors in Lucene VSM because the square and the square root functions are monotonically increasing functions on real numbers greater or equal to one. In this section we will verify the logic by comparing the performance difference between the two similarity algorithms as shown in Table 10.

Table 10. Euclidean Distance versus Manhattan Distance

Metrics	**Search Engine Model Algorithms**	
	Euclidean Distance	**Manhattan Distance**
Accuracy	0.86	0.86
Precision	0.91	0.91
True Positive Rate (Recall)	0.81	0.81
False Positive Rate (FPR)	0.09	0.09
F1 Score	0.86	0.86
Detection Time Per Instance (ms)	345.08	336.19

Both the Euclidean distance and the Manhattan distance similarities yielded the same set of nearest vectors and detection performance except the Manhattan distance is little faster than the Euclidean distance. This result verifies that using the square of Euclidean distance and Manhattan distance in Lucene VSM yields the same accuracy in the KNN algorithm.

5.2.3 Changing the Number of Nearest Vectors in KNN Algorithm

As discussed in Chapters 2 and 4, the KNN algorithm depends on the number of nearest neighbors *K*. Varying the value of *K* impacts the performance, and finding the optimal value of *K* is a heuristic process. Usually the simple approach to choose the value of *K* for binary classification is to take an odd integer closer to the square root of the total training dataset (Chio & Freeman, 2018). Table 11 shows the accuracy variation of ten randomly chosen values of $K \in [3, \sqrt{n}]$; where $\sqrt{n} \cong 359$. The initial value of *K* was 69; changing *K* to 29 slightly improves the performance because it reduces the number of nearest vectors, and there is no performance difference when switching between these values.

Table 11. Number of Nearest Vectors versus Accuracy

K	3	9	19	29	43	69	89	119	229	359
Accuracy (%)	85	83	84	86	86	86	85	82	82	83

5.2.4 Variable Vector Dimension

In the initial training, all features with zero values were boosted to 1 to keep all the stored vectors at the same dimension. Documents in the search engine can have variable term lengths, so the corresponding term-frequency vectors can have variable dimensions. The training data has 60 features, but all these features may not have non-zero values all the time. In this process the search engine model was retrained by excluding the zero values, which creates vectors with variable dimensions in Lucene VSM. This improved both the detection accuracy and the speed, as shown in Table 12. The training used Manhattan distance similarity and 29 nearest vectors (K=29).

Table 12. Performance of Variable versus Fixed Dimension Vectors

Metrics	**VSM Vector Dimension**	
	Variable Dimension	**Fixed Dimension**
Accuracy	0.89	0.86
Precision	0.93	0.91
True Positive Rate (Recall)	0.84	0.81
False Positive Rate (FPR)	0.06	0.09
F1 Score	0.88	0.86
Detection Time Per Instance (ms)	195.83	336.19

5.2.5 ML.NET Binary Classifier Optimization

The ML.NET binary classifier engine uses FastTree algorithm, which is a decision tree-based algorithm. This algorithm takes three parameters: the number of trees, leaves, and sample counts per leaf. Changing these parameters yields different metrics. The data normalization algorithm also impacts the accuracy of the model. After running several combinations of the parameter values and normalization algorithms, using the min-max normalization algorithm and setting the number of decision trees to 70 with 70 leaves per tree and 20 minimum sample count per leaf yielded the optimal result, as shown in Table 13.

Table 13. Optimized ML.NET Binary Classifier Model

Metrics	Value
Accuracy	0.92
Precision	0.95
True Positive Rate (Recall)	0.87
False Positive Rate (FPR)	0.04
F1 Score	0.91
AUC	0.98
AUCPR	0.97
Detection Time Per Instance (ms)	117.70

5.2.6 Class Balancing

The model training so far used equally partitioned numbers of attack and benign instances. In real cases, most traffic instances in a network are not attacks. With this logic in mind, we increased the number of benign instances by one third in the training dataset without changing the number of attack instances. This made the attack-benign ratio 2:3. The new model result shows improvement on the detection accuracy, precision, and FPR. However, both models were slightly reduced in F1Score, TPR, and detection speed (for the search engine), as shown in Table 14.

Table 14. Detection Performance with 2:3 Attack-Benign Ratio Training Dataset

Metrics	Detection Models	
	Search Engine	**ML.NET**
Accuracy	0.92	0.94
Precision	0.96	0.97
True Positive Rate (Recall)	0.80	0.84
False Positive Rate (FPR)	0.02	0.02
F1 Score	0.87	0.90
Detection Time Per Instance (ms)	275.95	117.70

5.3 Final Evaluation

Algorithm selection and parameter optimization depend on the type of the dataset used for training and testing. Table 15 shows the optimized parameters and algorithms for each detection model. Table 16 also shows the final evaluation results of the models.

Table 15. Optimized Model Parameters

Model	**Algorithms and Parameters**	
Search Engine	**Classification Algorithm**	KNN
	Similarity Algorithm	Manhattan Distance
	Number of Neighbors (K)	29
ML.NET	**Classification Algorithm**	FastTree (Decision Tree)
	Number of Trees	70
	Number of Leaves	70
	Sample Count Per Leaf	30

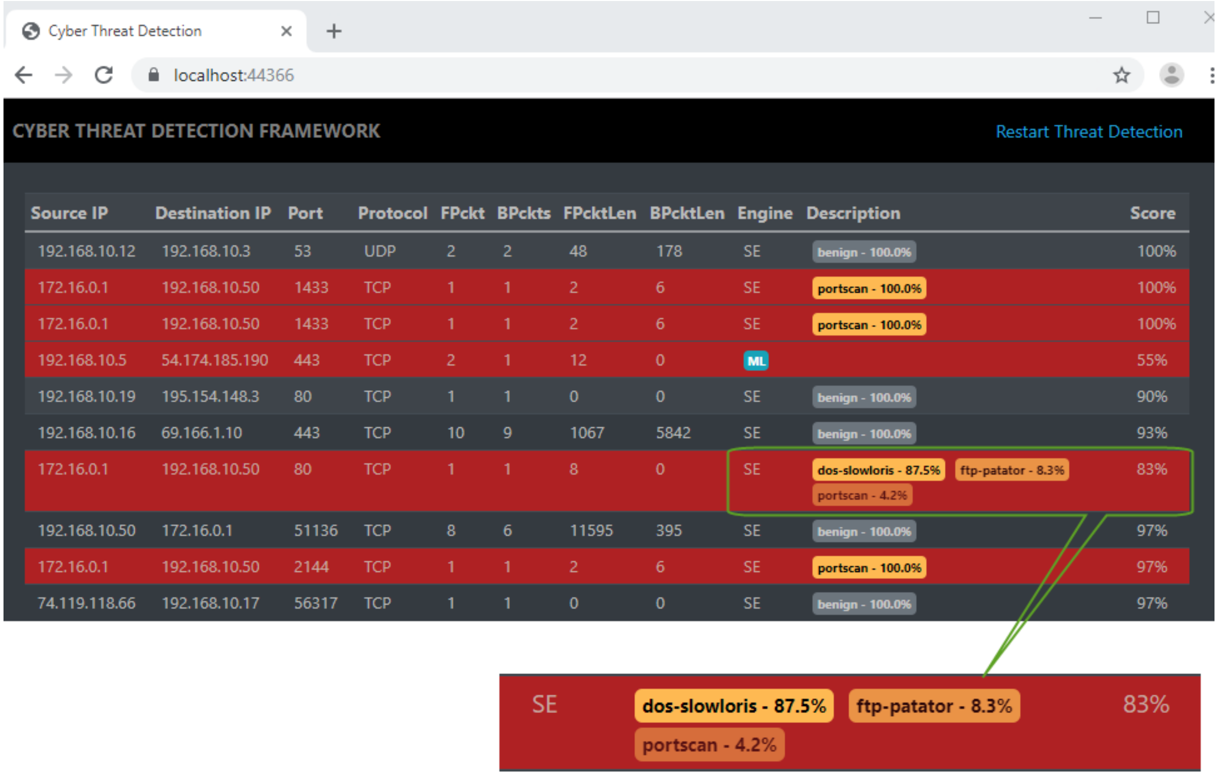
Table 16. Final Model Evaluation Result

Metrics	**Detection Models**	
	Search Engine	**ML.NET**
Accuracy	0.92	0.94
Precision	0.96	0.97
True Positive Rate (Recall)	0.80	0.84
False Positive Rate (FPR)	0.02	0.02
F1 Score	0.87	0.90
Detection Time Per Instance (ms)	275.95	117.70

The evaluation and optimization techniques used so far show that the models can be further tuned up by changing techniques and parameter values based on business requirements. One technique or parameter may not improve all the metrics in all instances. For example, class unbalancing improves accuracy and precision but reduces the TPR and F1 Score. Although the overall detection accuracy of the search engine and the ML.NET are closer to each other, ML.NET is faster than the search engine when processing a single traffic instance in a synchronous operation. However, the detection engine runs asynchronously, which means that several detection operations can run in parallel without blocking each other. A new traffic instance runs into the detection engine as soon as it arrives, without waiting for the previous instance to complete. When running the engine with asynchronous parallel tasks, the search engine takes an average of 87 milliseconds to process a single traffic instance, whereas ML.NET takes 0.03 milliseconds.

5.4 System Testing

The threat detection framework incorporates components such as a data processor, detection engine, alert service, and dashboard application. In this section, system testing addresses the integration of these components in a complete system environment in different scenarios. Since feature extraction is beyond the scope of this research, a prelabeled flood of traffic instances was used to simulate high volume network traffic. About 10,000 traffic instances continuously passed to a running detection engine using a loop code. The detection engine uses the search engine as the primary detection, and the ML engine backs it up when the search engine detection confidence is low. Figure 12 shows the dashboard application with the detection results. Unlike the ML engine, the search engine can predict the details of the attack such as the type of the attack. The description column shows the probability of the attacks by type. For instance, as shown in Figure 12, the expanded description cell shows the probability of possible attack types, which means out of selected *K* nearest vectors, 17% of the vectors are labeled benign, and the remaining 83% are labeled as attacks. Out of 83% of attack labeled vectors, 87.5% are labeled as DOS Slowloris, 8.3% are labeled as FTP Patator, and 4.2% are labeled as port scan.



Source IP	Destination IP	Port	Protocol	FPckt	BPckts	FPcktLen	BPcktLen	Engine	Description	Score
192.168.10.12	192.168.10.3	53	UDP	2	2	48	178	SE	benign - 100.0%	100%
172.16.0.1	192.168.10.50	1433	TCP	1	1	2	6	SE	portscan - 100.0%	100%
172.16.0.1	192.168.10.50	1433	TCP	1	1	2	6	SE	portscan - 100.0%	100%
192.168.10.5	54.174.185.190	443	TCP	2	1	12	0	ML		55%
192.168.10.19	195.154.148.3	80	TCP	1	1	0	0	SE	benign - 100.0%	90%
192.168.10.16	69.166.1.10	443	TCP	10	9	1067	5842	SE	benign - 100.0%	93%
172.16.0.1	192.168.10.50	80	TCP	1	1	8	0	SE	dos-slowloris - 87.5% ftp-patator - 8.3% portscan - 4.2%	83%
192.168.10.50	172.16.0.1	51136	TCP	8	6	11595	395	SE	benign - 100.0%	97%
172.16.0.1	192.168.10.50	2144	TCP	1	1	2	6	SE	portscan - 100.0%	97%
74.119.118.66	192.168.10.17	56317	TCP	1	1	0	0	SE	benign - 100.0%	97%

SE dos-slowloris - 87.5% ftp-patator - 8.3% 83%
portscan - 4.2%

Figure 12. Detection framework dashboard.

5.4.1 Continuous Training

As previously discussed in Chapters 3 and 4, when search engine detection confidence is below some configurable value, the ML engine takes the prediction task and the search engine learns the incident. In this research, the default confidence value is set to be 95, which means a prediction score of less than 95% is considered low. When the detection confidence of the search engine is low, the decision engine uses the ML model to process the traffic feature. If the ML model predicts a different result than the search engine does, the decision engine takes the prediction result of the ML model, and the search engine learns the incident in a separate task, as shown in Listing 29.

During the retraining process, the input traffic data has the result predicted from the ML model as a label. The search engine model gets updated with the new training data. This changes the future prediction for similar traffic data because the nearest vectors set may include the newly added vector. Therefore, continuous training makes the search engine grow smarter as it increases the search engine prediction confidence.

```
private static Task DetectTask(FeatureData featureData)
{
    var threatDetectorTask = Task.Run(() =>
    {
        PredictionResult predictionResult = null;
        var predictedLabel = string.Empty;
        if (ACTIVE_DETECTION_ENGINE == EngineName.SE)
            {
               //…
            }
     else if (ACTIVE_DETECTION_ENGINE == EngineName.SEML)
        {
            predictionResult = searcher.Predict(featureData.SEFeaturesString);
            predictionResult.EngineName = "SE";
            predictedLabel = predictionResult.Label;
            if (predictionResult.Confidence < GOOD_CONFIDENCE_PERCENTAGE)
            {
                var mlPrediction = mlPredictor.Predict(featureData);
                predictedLabel = mlPrediction.PredictedLabel ? "1" : "0";
                if (predictionResult.Label != predictedLabel)
                {
                    predictionResult = mlPrediction;
                    predictionResult.Confidence = mlPrediction.Probability * 100;
                    predictionResult.EngineName = "ML";

                    var searchEngineRetrainerTask = Task.Factory.StartNew(() =>
                           {
                               ReTrainSearchEngine(featureData.SEFeaturesString,
                                   predictedLabel);
                           });
                }
            }
        }
        else
        {
            var mlPrediction = mlPredictor.Predict(featureData);
            predictionResult = mlPrediction;
            predictionResult.Confidence = mlPrediction.Probability * 100;
            predictionResult.EngineName = "ML";
            predictedLabel = mlPrediction.PredictedLabel ? "1" : "0";
        }

        //…

        SendAlert(predictionResult);
    });

    return threatDetectorTask;
}
```

Listing 29. Partial implementation of threat detection task.

The searcher component of the search engine is managed in a singleton instance. After the retraining task is completed, the singleton instance should be updated to include the newly trained model. Starting the searcher component takes a few seconds on average. To mitigate this issue, when the retraining task starts, the decision engine uses the ML engine until the training is complete and the model is refreshed, as shown in Listing 30.

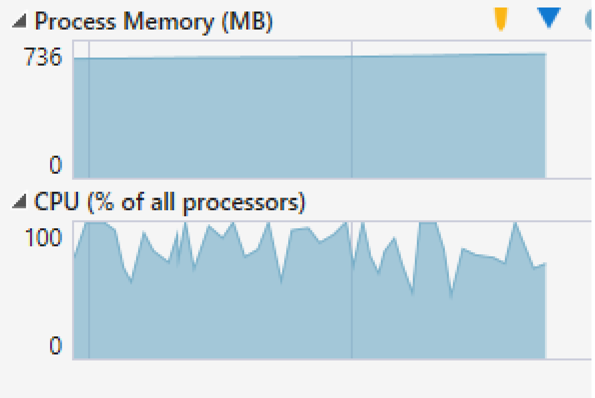
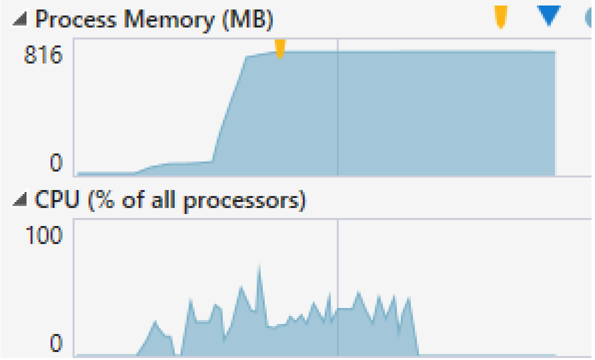
```
private static void ReTrainSearchEngine(string seFeatureString,
                                        string predictedLabel)
{
    lock (trainerThreadLock)
    {
        ACTIVE_DETECTION_ENGINE = EngineName.ML;

        string labelDescription = "benign";
        if (predictedLabel == "1")
            labelDescription = "attack"; //ML BC cannot predict the attack type
        string featureData = seFeatureString + "," + predictedLabel + "|" +
                             labelDescription;
        var indexer = new Indexer();
        indexer.UpdateIndex(featureData);

        Searcher.ResetInstance();
        searcher = Searcher.Instance; //restart searcher

        ACTIVE_DETECTION_ENGINE = EngineName.SEML;
    }
}
```

Listing 30. Implementation of search engine retraining.

5.4.2 Enterprise Scale Testing

In this section we will discuss the performance of the detection engine with respect to enterprise testing metrics. Table 17 shows the Visual Studio diagnostic tools results, such as CPU, memory usage, and system stability, while the detection engine is processing approximately 10,000 simulated flood of traffic instances being passed to the detection engine at a time.

Table 17. Load Testing Performance Comparison Table

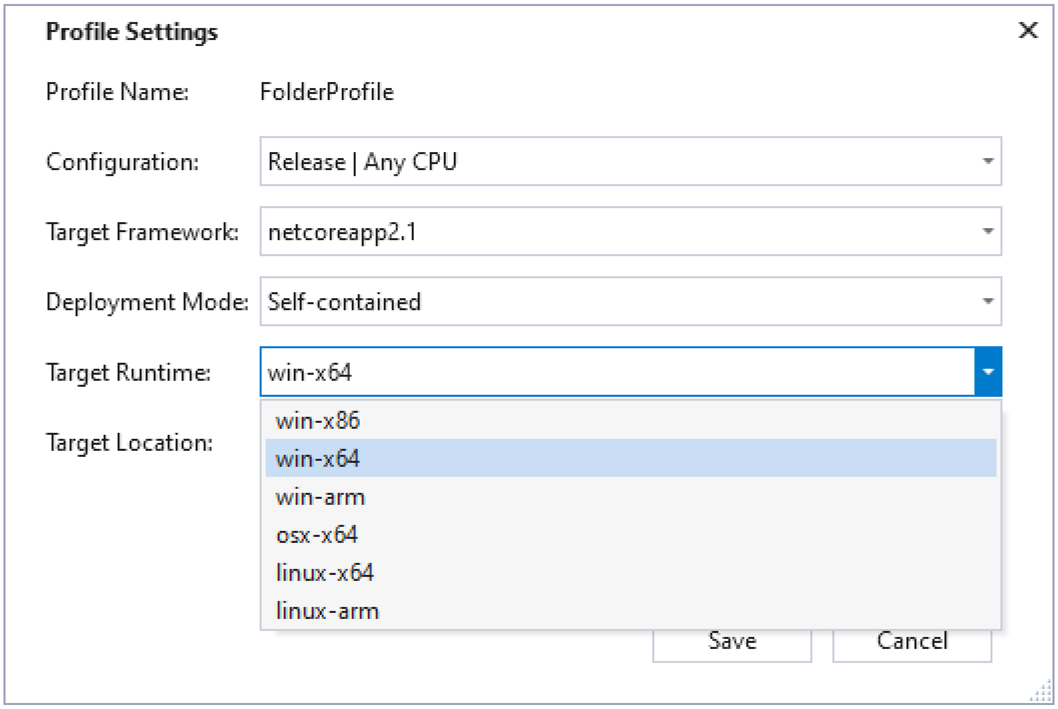
Search Engine	**ML.NET**
 Process Memory (MB) 736 0 CPU (% of all processors) 100 0	 Process Memory (MB) 816 0 CPU (% of all processors) 100 0
Detection Time Per Instance: 87ms	**Detection Time Per Instance**: 0.03ms
Exceptions: 0	**Exceptions:** 0
Crashes: 0	**Crashes**: 0

Load Testing: As the performance comparison table shows, the search engine consumes more resources than the ML engine. The search engine consumes up to all the available CPU power while processing the flood of traffic instances. This implies that the search engine-based detection is CPU intensive and requires more CPU resources for high volume traffic detection. Testing the search engine detection on a different machine (Intel Xeon CPU E5-2430 v2 2.5Ghz) reduced the average single instance detection time from 87ms to 70ms. This indicates that increasing the CPU power increases the performance of the search engine.

Stability: The detection framework uses the popular logger log4net for logging and alerting services. Any exception during the detection can be logged. The logger can also be further configured for notification. During the testing session, both engines completed without a failure or exception. The detection engine has a restart option in case of crash recovery.

Extensibility: As the detection engine is based on a search engine library, it can be trained to function as a text classifier such as a spam detector. By extending the data processor, indexer, and searcher components of the engine based on the input data, a new detection model can be generated without further customizing the internals of the search engine.

Portability: The whole framework and its components are developed using Microsoft .NET Core framework. The .NET Core framework is a multiplatform framework that supports Windows, Linux, and macOS operating systems. For instance, Figure 13 shows the publish settings of the framework project to target multiple platforms. The detection framework has been designed and developed to run on these platforms but tested only on Windows.



Profile Settings

Profile Name: FolderProfile

Configuration: Release | Any CPU

Target Framework: netcoreapp2.1

Deployment Mode: Self-contained

Target Runtime: win-x64

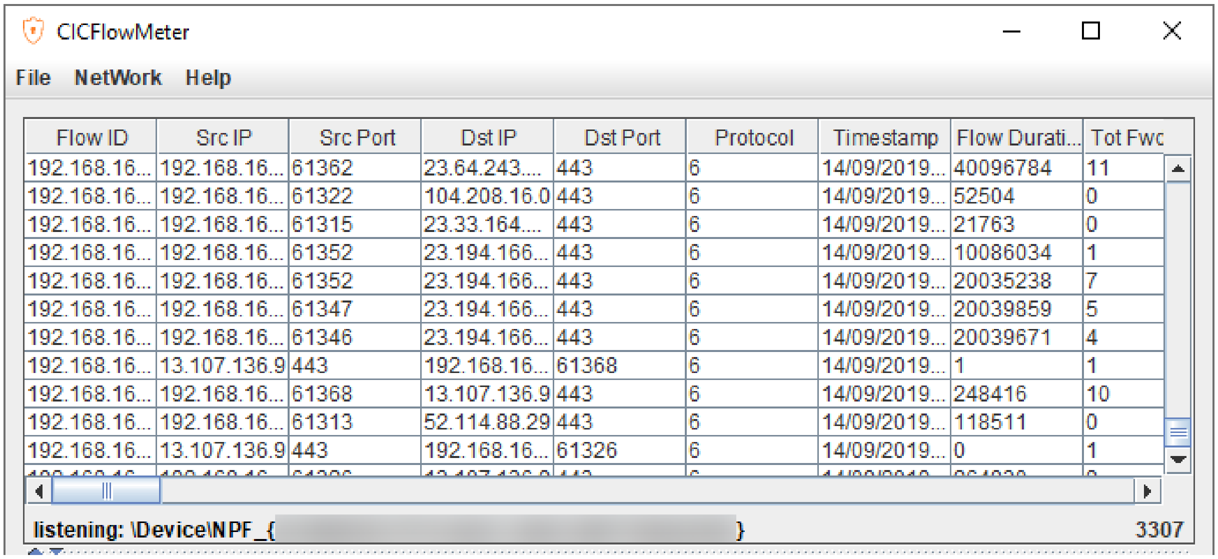
Target Location:

Figure 13. Project publish settings.

Deployment: The new cyber threat detection framework was deployed as a host-based threat detection system in a real environment for validation purposes. Although feature extraction from live traffic is beyond the scope of this research, CICFlowMeter was used to extract features from live traffic. CICFlowMeter saves the extracted feature to a csv file during live traffic feature extraction. To connect the CICFlowMeter process to the detection engine, the detection engine uses a file watcher component, which triggers an event when the CICFlowMeter appends a new feature data to the csv file. When the new feature data arrives, it is passed to the running detection engine, as shown in the code of Listing 31. Then the detection engine determines the status of the traffic data and passes the result to the dashboard through the alert service. Figure 14 and Figure 15 show both CICFlowMeter and the detection engine running at the same time.

```
public static void StartLiveTrafficFeatureExtraction()
{
    string featuresFileIdentifier = DateTime.Now.ToString("yyy-MM-dd");
    string featuresFileName = $"{featuresFileIdentifier}_Flow.csv";
    FileSystemWatcher watcher = new FileSystemWatcher();
    watcher.Path = AppConfigSettings.LiveTrafficFeaturePath;
    watcher.Filter = featuresFileName;
    watcher.NotifyFilter = (NotifyFilters.LastWrite);
    watcher.Changed += new FileSystemEventHandler(OnNewFeatureDataAdded);
    watcher.Created += new FileSystemEventHandler(OnNewFeatureDataAdded);

    var featuresFileStream = new FileStream(
                   AppConfigSettings.LiveTrafficFeaturePath + featuresFileName,
                   FileMode.Open, FileAccess.Read, FileShare.ReadWrite);
    featuresFileReader = new StreamReader(featuresFileStream);

    watcher.EnableRaisingEvents = true;
}

private static void OnNewFeatureDataAdded(object sender, FileSystemEventArgs e)
{
    var lines = featuresFileReader.ReadToEnd();
    IEnumerable<string> featureLines = lines.Split('\n').TakeLast(10);
    foreach (string line in featureLines)
    {
        if (!string.IsNullOrWhiteSpace(line))
        {
            if (!line.StartsWith("Flow") && !line.Contains("Infinity")
                 && !line.Contains("NaN"))
            {
                var features = line.Split(',').SkipLast(1).ToList();
                features.RemoveAt(6);
                features.RemoveAt(0);
                string featureString = string.Join(",", features);
                var featureData = DataConverter.ConvertToFeatureDataObject(
                                                               featureString);

                DetectionEngine.DetectWithCt(featureData);
            }
        }
    }
}
```

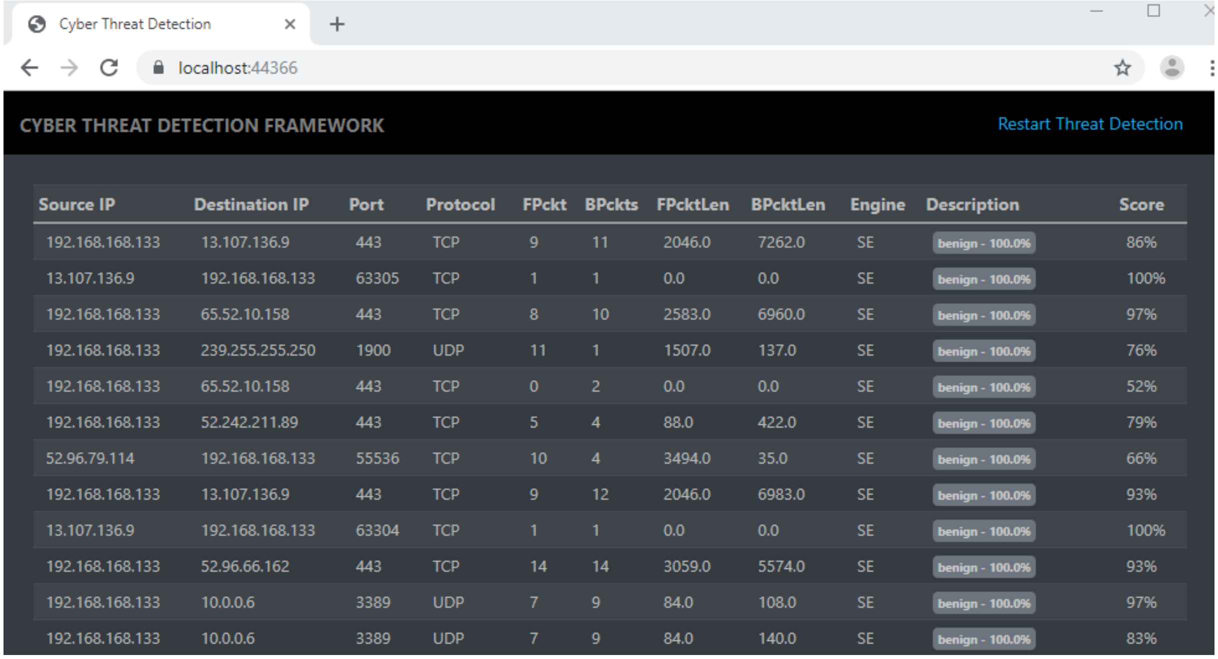
Listing 31. Live traffic feature file watcher implementation.



Flow ID	Src IP	Src Port	Dst IP	Dst Port	Protocol	Timestamp	Flow Durati...	Tot Fwc
192.168.16...	192.168.16...	61362	23.64.243....	443	6	14/09/2019...	40096784	11
192.168.16...	192.168.16...	61322	104.208.16.0	443	6	14/09/2019...	52504	0
192.168.16...	192.168.16...	61315	23.33.164....	443	6	14/09/2019...	21763	0
192.168.16...	192.168.16...	61352	23.194.166...	443	6	14/09/2019...	10086034	1
192.168.16...	192.168.16...	61352	23.194.166...	443	6	14/09/2019...	20035238	7
192.168.16...	192.168.16...	61347	23.194.166...	443	6	14/09/2019...	20039859	5
192.168.16...	192.168.16...	61346	23.194.166...	443	6	14/09/2019...	20039671	4
192.168.16...	13.107.136.9	443	192.168.16...	61368	6	14/09/2019...	1	1
192.168.16...	192.168.16...	61368	13.107.136.9	443	6	14/09/2019...	248416	10
192.168.16...	192.168.16...	61313	52.114.88.29	443	6	14/09/2019...	118511	0
192.168.16...	13.107.136.9	443	192.168.16...	61326	6	14/09/2019...	0	1

listening: \Device\NPF_{[redacted]} 3307

Figure 14. CICFlowMeter extracting features from live traffic.



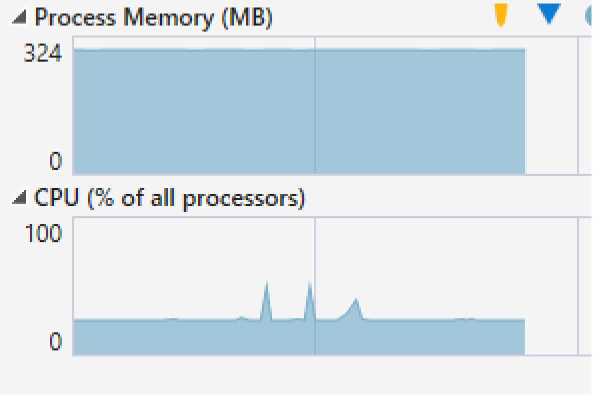
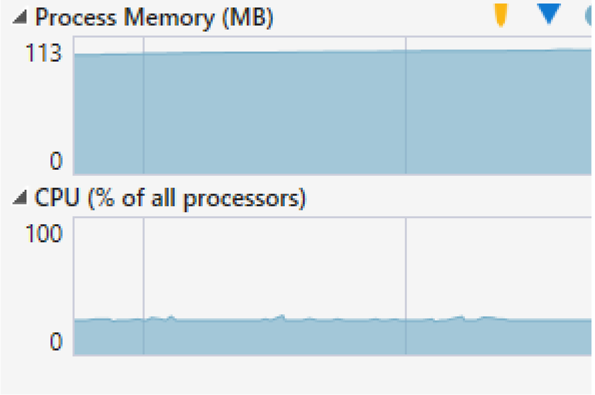
Source IP	Destination IP	Port	Protocol	FPckt	BPckts	FPcktLen	BPcktLen	Engine	Description	Score
192.168.168.133	13.107.136.9	443	TCP	9	11	2046.0	7262.0	SE	benign - 100.0%	86%
13.107.136.9	192.168.168.133	63305	TCP	1	1	0.0	0.0	SE	benign - 100.0%	100%
192.168.168.133	65.52.10.158	443	TCP	8	10	2583.0	6960.0	SE	benign - 100.0%	97%
192.168.168.133	239.255.255.250	1900	UDP	11	1	1507.0	137.0	SE	benign - 100.0%	76%
192.168.168.133	65.52.10.158	443	TCP	0	2	0.0	0.0	SE	benign - 100.0%	52%
192.168.168.133	52.242.211.89	443	TCP	5	4	88.0	422.0	SE	benign - 100.0%	79%
52.96.79.114	192.168.168.133	55536	TCP	10	4	3494.0	35.0	SE	benign - 100.0%	66%
192.168.168.133	13.107.136.9	443	TCP	9	12	2046.0	6983.0	SE	benign - 100.0%	93%
13.107.136.9	192.168.168.133	63304	TCP	1	1	0.0	0.0	SE	benign - 100.0%	100%
192.168.168.133	52.96.66.162	443	TCP	14	14	3059.0	5574.0	SE	benign - 100.0%	93%
192.168.168.133	10.0.0.6	3389	UDP	7	9	84.0	108.0	SE	benign - 100.0%	97%
192.168.168.133	10.0.0.6	3389	UDP	7	9	84.0	140.0	SE	benign - 100.0%	83%

Figure 15. Detection engine processing live traffic features.[6]

Table 18 shows the performance of the detection engine running as a host-based threat detection system using the search engine and the ML.NET engine. As the result shows, neither engine consumes much resources.

[6] The detection engine processed 60 different features for each traffic data, but the dashboard grid shows only selected features.

Table 18. Performance of the Detection Engine Running as a Host-Based Detection System

Search Engine	ML.NET
 Process Memory (MB) 324 0 CPU (% of all processors) 100 0	 Process Memory (MB) 113 0 CPU (% of all processors) 100 0

5.5 Summary

In this chapter we have discussed several evaluation, validation, and optimization techniques. Evaluation metrics include accuracy, precision, and recall rates of the detection engine. The initial evaluation result was optimized by applying different techniques such as choosing important features and optimizing scoring algorithms. The search engine-based detection engine yielded a detection accuracy closer to the ML.NET engine; however, the ML.NET engine is faster and consumes fewer CPU resources than the search engine, especially in high-volume traffic detection. Unlike the ML.NET binary model, the search engine can be continuously trained without destroying the original model. Both engines have passed through load testing and stability testing by simulating high-volume traffic. The newly developed cyber threat detection framework has the capabilities of both engines.

This concludes Chapter 5. Case Study. This chapter discussed setting up evaluation metrics, evaluation of algorithms and components, optimizing techniques, and validating and deploying the new framework. Empirical analysis and performance results of the framework were discussed in this chapter.

CHAPTER 6

CONCLUSIONS

This chapter concludes with the overall outcome of the research, a list of contributions, limitations of the research, and recommendations for future research.

This research shows the possibility of a search engine serving as a cyber threat detection framework. It also shows that the search engine-based threat detection engine can be reinforced by a machine learning-based engine. As a result, using the search- and machine learning-based engines working together improves the capability of the cyber threat detection system.

Using two different technologies, search engines and machine learning, the newly developed framework was focused on the analysis of network traffic for cyber threat detection. Several artifacts such as traffic feature processors, detection engines, an alert service, and dashboard applications have been developed. A search engine library, Lucene was customized to function as a threat detection tool. During customization, several algorithms and techniques have also been investigated. The developed framework was tested and validated. In this chapter we will explore the outcomes of the research, its contributions and limitations, and recommendations for future research.

6.1 Contributions

1. A VSM-based search engine can function as a cyber threat detection engine. The primary purpose of search engines is text mining and document ranking. However, this research showed that search engines can also function as cyber threat detection systems. The opensource search engine library Lucene was used to validate the theory that search engines could function as cyber threat detection frameworks. To achieve this, customization of Lucene components such as the analyzer, tokenizer, indexer chain, scoring algorithms, and searcher was necessary. The detection accuracy of the customized search engine was initially closer (2% accuracy difference) to that of the Microsoft machine learning framework

ML.NET used by cyber security tools such as Microsoft Defender. The search engine can improve its detection accuracy through self-learning.

2. This cyber threat detection framework uses Artificial Intelligence and classical technologies. Using mixed technologies in intrusion detection systems improves detection performance (Samrin & Vasumathi, 2017). Search engine and machine learning technologies were used in the development of the cyber threat detection framework. The developed threat detection framework incorporated two detection engines: the search engine-based and the ML.NET-based machine learning engine. The two detection engines each have their own benefits and drawbacks, but the proposed framework is able to take the best of each engine. For instance, the machine learning binary classification model has a faster detection speed and better detection accuracy than the search engine; however, it cannot predict the details of the attack. Another drawback of the machine learning model is the inability to train the model while it is running without destroying the previous model because the FastTree algorithm, which the binary classification is built on, is not a retrainable algorithm. The search engine based cyber threat detection model can predict details of the attack and update itself to improve its accuracy without destroying its model.

3. This cyber threat detection framework is self-learning. Although the search engine model has slower detection speed and slightly lower detection accuracy at the beginning, it can predict the details of the attacks and can be retrained while it is running without destroying the model in use. The developed framework can run both the search engine and the machine learning models at the same time. When the two models run together, the machine learning model serves as a continuous trainer to the search engine model—with this, the search engine learns new incidents and improves its accuracy as it continues running. Eventually, the continuous training makes the search engine grow smarter as it is trained by the machine learning engine.

4. Manhattan distance similarity has better performance than Euclidean distance and Cosine similarity in the Lucene search engine library for KNN-based classification. One of the search engine customization tasks was to implement classification and similarity algorithms. The KNN algorithm was used in the search engine model to classify nearby traffic instances. By default, Lucene search engine uses Cosine similarity. In this research, Cosine similarity has been proven to be less accurate in measuring similarity between two traffic instances represented by vectors in the search engine model. Usually Euclidean distance similarity is used with KNN algorithm for classification. In this research the square of the Euclidean distance similarity has the same classification accuracy as the Euclidean distance similarity. While the equivalency of the similarity algorithms was being proven, it was also discovered that the Manhattan distance similarity has the same classification accuracy as the Euclidean distance and the square of Euclidean distance similarities, but the Manhattan distance similarity has a slightly better performance in the Lucene search engine similarity scoring.

5. Variable-dimension VSM has achieved better accuracy than n-dimensional VSM in the Lucene-based cyber threat classification model. The search engine-based threat detection model uses VSM. The dimension of the vector space is the same as the number of features extracted from the traffic instance. The VSM model in search engines has variable dimensions because the indexed documents do not have the same number of terms/words all the time. The feature extractor generates a fixed number of features from the traffic instance. Indexing each feature set makes the VSM model have a fixed dimensional model. For example, if the feature generator generates *n* features, the respective model will be *n*-dimensional VSM. Indexing non-zero value feature sets gives the VSM a variable dimension. This research showed that variable-dimensional VSM model yielded better accuracy and performance than *n*-dimensional VSM because it runs fewer similarity score computations.

6. Every network traffic feature is not equally important for determining the status of the traffic. A feature extractor component can generate several features from the traffic instance to inspect the traffic behavior. However, not all the features are equally important for classification; some features have higher value in determining the status of the traffic. The Permutation Feature Importance (PFI) technique was used to filter out less-important features. Feature reduction reduces noise and the training time of the model.

Particularly for the search engine-based model, feature reduction improves the detection time because it reduces the number of computations in the VSM.

7. The use of different cutting-edge technologies in the cyber threat detection development leads to better results. The developed framework used the latest technologies in the artifact development. For example, ML.NET is the most recent machine learning framework used by Microsoft and was recently released as opensource. The alert service uses the popular logger component log4net used in several enterprise software (Apache Software Foundation, 2017) and in development for many years; the dashboard application engages in real-time communication using WebSocket protocol to display alerts as soon as a threat is detected. The entire framework was developed using the latest Microsoft .NET Core framework, which targets multiple platforms and operating systems. This makes the developed framework run on different operating systems such as Windows, macOS, and Linux.

6.2 Limitations and Future Research

This research was intended to develop a threat detection framework, not a complete cyber threat detection appliance. In order for the framework to grow into a complete cyber threat detection application, at least the following features need to be researched and incorporated:

1. Native feature extraction component: Feature extraction is an important step in developing threat detection systems. Because developing feature extraction was not within the scope of this research, a third-party feature extractor tool was used to extract features from live traffic to validate the framework. To advance the developed framework to a complete cyber threat detection application, a feature extractor component needs to be developed natively as part of the framework.

2. Host-based anomaly detection engine: This research focused on network traffic analysis. In addition to network-based detection, it is important to analyze the impact of the traffic on the destination host for end-to-end threat intelligence. Therefore, a host-based anomaly detection component needs to be developed and integrated as part of the framework to build an end-to-end threat intelligence system.

3. Heuristic-based analysis: The threat detection engines in the framework are based on machine learning, which sometimes produce false positives or false negatives. A separate analysis framework needs to be implemented to automatically discard false positives and false negatives by further analyzing the traffic activity, such as the reputation of the TCP connection properties, originating process behaviors, activities on the host device, etc.

4. Robust reporting service: The developed framework incorporates an alert service component that can broadcast alerts in real-time communication to the connected clients such as dashboard applications. However, the dashboard application included in the framework has a limited reporting service. Adding a robust reporting service would increase the usability and functionality of the framework.

6.3 Summary

This research has introduced a new cyber threat detection approach by combining two complementary technologies: search engine and machine learning. The research also showed that search engines can function as threat detection systems with closer accuracy as a machine learning-based threat detection framework. The main advantages of combining search engine and machine learning for cyber threat detection are the following: a) the ability of the system to self-learn from its mistakes and grow smarter as it continues running; b) the capability of the system in predicting attack details since the search engine can store and analyze metadata of the training traffic data.

Several components of the search engine such as analyzers, tokenizers, indexer, searcher, indexing, and scoring techniques have been customized to make the search engine function as a cyber threat detection engine. During the search engine customization process, this research also proved additional related findings, which are: a) Manhattan distance similarity has better performance than Euclidean distance and Cosine similarity in the Lucene search engine library for the KNN-based cyber threat classification model; b) variable-dimension VSM has achieved better accuracy than n-dimensional VSM in the Lucene-based cyber threat classification model; c) every network traffic feature is not equally important to determine the status of the traffic.

With all these features, the new cyber threat detection framework improves existing cyber threat detection approaches and contributes to the mitigation of the ongoing cyber security problems we are facing.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., … Zheng, X. (2016). TensorFlow: A system for large-scale machine learning. *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, 21.

Albin, E. (2011). *A comparative analysis of the Snort and Suricata intrusion-detection systems*.

Al-Jarrah, O., & Arafat, A. (2015). Network Intrusion Detection System Using Neural Network Classification of Attack Behavior. *Journal of Advances in Information Technology*, 1–8. https://doi.org/10.12720/jait.6.1.1-8

Ando, R. K., & Zhang, T. (2005). *A Framework for Learning Predictive Structures from Multiple Tasks and Unlabeled Data*. 37.

Apache Software Foundation. (2017). *Apache log4net Documentation*. https://logging.apache.org/log4net/release/features.html

Bello, F. L., Ravulakollu, K., & Amrita. (2015). Analysis and evaluation of hybrid intrusion detection system models. *2015 International Conference on Computers, Communications, and Systems (ICCCS)*, 93–97. https://doi.org/10.1109/CCOMS.2015.7562879

Białecki, A., Muir, R., & Ingersoll, G. (2012). Apache Lucene 4. *Proceedings of the SIGIR 2012 Workshop on Open Source Information Retrieval*, 17–23.

Breiman, L. (2001). *Random Forests*.

Bruneau, G. (2001). *The History and Evolution of Intrusion Detection*. 8.

CAIDA. (2019). *CAIDA Data—Overview of Datasets, Monitors, and Reports*. http://www.caida.org/data/overview/

Canadian Institute for Cybersecurity. (2019). *CICFlowMeter Documentation*. https://github.com/ISCX/CICFlowMeter

Chauhan, V., Jaiswal, A., & Khan, J. (2015). Web Page Ranking Using Machine Learning Approach. *2015 Fifth International Conference on Advanced Computing & Communication Technologies*, 575–580. https://doi.org/10.1109/ACCT.2015.56

Chio, C., & Freeman, D. (2018). *Machine Learning and Security*. O'Reilly.

Choudhury, S., & Bhowal, A. (2015). Comparative analysis of machine learning algorithms along with classifiers for network intrusion detection. *2015 International Conference on Smart Technologies and Management for Computing, Communication, Controls, Energy and Materials (ICSTM)*, 89–95. https://doi.org/10.1109/ICSTM.2015.7225395

Cisco. (2019). *Snort Documentation*. https://www.snort.org/documents

Creswell, J. W. (2014). *Research design: Qualitative, quantitative, and mixed method approaches* (4th ed). Sage Publications.

Day, D. J., & Burns, B. M. (2011). *A Performance Analysis of Snort and Suricata Network Intrusion Detection and Prevention Engines*. 6.

García-Teodoro, P., Díaz-Verdejo, J., Maciá-Fernández, G., & Vázquez, E. (2009). Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers & Security*, *28*(1–2), 18–28. https://doi.org/10.1016/j.cose.2008.08.003

Garg, T., & Khurana, S. S. (2014). Comparison of classification techniques for intrusion detection dataset using WEKA. *International Conference on Recent Advances and Innovations in Engineering (ICRAIE-2014)*, 1–5. https://doi.org/10.1109/ICRAIE.2014.6909184

Gómez, J., Gil, C., Padilla, N., Baños, R., & Jiménez, C. (2009). Design of a Snort-Based Hybrid Intrusion Detection System. In S. Omatu, M. P. Rocha, J. Bravo, F. Fernández, E. Corchado, A. Bustillo, & J. M. Corchado (Eds.), *Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living* (Vol. 5518, pp. 515–522). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-02481-8_75

Gracia, L. (2008). Programming with Libpcap Sniffing the Network From Our Own Application. *Hard Core IT Security Magazine*, *3*(2).

Habibi Lashkari, A., Draper Gil, G., Mamun, M. S. I., & Ghorbani, A. A. (2017). Characterization of Tor Traffic using Time based Features: *Proceedings of the 3rd International Conference on Information Systems Security and Privacy*, 253–262. https://doi.org/10.5220/0006105602530262

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009). The WEKA data mining software: An update. *ACM SIGKDD Explorations Newsletter, 11*(1), 10. https://doi.org/10.1145/1656274.1656278

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design Science in Information Systems Research. *MIS Quarterly*, *28*(1), 75–105.

Hu, Q., Asghar, M. R., & Brownlee, N. (2017). Evaluating network intrusion detection systems for high-speed networks. *2017 27th International Telecommunication Networks and Applications Conference (ITNAC),* 1–6. https://doi.org/10.1109/ATNAC.2017.8215374

Hung, C.-Y., Chen, W.-C., Lai, P.-T., Lin, C.-H., & Lee, C.-C. (2017). Comparing deep neural network and other machine learning algorithms for stroke prediction in a large-scale population-based electronic medical claims database. *2017 39th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC),* 3110–3113. https://doi.org/10.1109/EMBC.2017.8037515

IETF. (2011). *The WebSocket Protocol*. https://tools.ietf.org/html/rfc6455

Kemmerer, R. A., & Vigna, G. (2002). Intrusion detection: A brief history and overview. *Computer, 35*(4), supl27–supl30. https://doi.org/10.1109/MC.2002.1012428

Kissel, R. (2013). *Glossary of key information security terms* (NIST IR 7298r2). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.IR.7298r2

Kong, L., Huang, G., Wu, K., Tang, Q., & Ye, S. (2018). Comparison of Internet Traffic Identification on Machine Learning Methods. *2018 International Conference on Big Data and Artificial Intelligence (BDAI),* 38–41. https://doi.org/10.1109/BDAI.2018.8546682

Koomen, T., & Pol, M. (1999). *Test Process Improvement—A Practical Step-by-Step Guide to Structured Testing*. Addison Wesley.

Li, W., & Zeng, S. (2016). A Vector Space Model based spam SMS filter. *2016 11th International Conference on Computer Science & Education (ICCSE),* 553–557. https://doi.org/10.1109/ICCSE.2016.7581640

Lin, W.-H., Lin, H.-C., Wang, P., Wu, B.-H., & Tsai, J.-Y. (2018). Using convolutional neural networks to network intrusion detection for cyber threats. *2018 IEEE*

International Conference on Applied System Invention (ICASI), 1107–1110. https://doi.org/10.1109/ICASI.2018.8394474

Mehmood, T., & Rais, H. B. M. (2016). Machine learning algorithms in context of intrusion detection. *2016 3rd International Conference on Computer and Information Sciences (ICCOINS)*, 369–373. https://doi.org/10.1109/ICCOINS.2016.7783243

Microsoft. (2014). *Introduction to SignalR*. https://docs.microsoft.com/en-us/aspnet/signalr/overview/getting-started/introduction-to-signalr

Microsoft. (2016). *Microsoft Message Analyzer Operating Guide*. https://docs.microsoft.com/en-us/message-analyzer/microsoft-message-analyzer-operating-guide

Microsoft. (2017). *Task-based asynchronous programming*. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-based-asynchronous-programming

Microsoft. (2018). *Introduction to ASP.NET Core SignalR*. https://docs.microsoft.com/en-us/aspnet/core/signalr/introduction?view=aspnetcore-2.1

Microsoft. (2019a). *ML.NET- An open source and cross-platform machine learning framework*. https://dotnet.microsoft.com/apps/machinelearning-ai/ml-dotnet

Microsoft. (2019b). *The Microsoft Cognitive Toolkit*. https://www.microsoft.com/en-us/cognitive-toolkit/

Microsoft. (2019c). *Visual Studio*. https://visualstudio.microsoft.com/

Microsoft. (2019d). *Custom AI Models with Azure Machine Learning Studio and ML.NET*. https://devblogs.microsoft.com/premier-developer/custom-ai-models-with-azure-machine-learning-studio-and-ml-net/

Microsoft. (2019e). *Model evaluation metrics in ML.NET*. https://docs.microsoft.com/en-us/dotnet/machine-learning/resources/metrics

Microsoft. (2019f). *Explain model predictions using Permutation Feature Importance*. https://docs.microsoft.com/en-us/dotnet/machine-learning/how-to-guides/explain-machine-learning-model-permutation-feature-importance-ml-net

Microsoft. (2019g). *ML.NET - Re-train a model*. https://docs.microsoft.com/en-us/dotnet/machine-learning/how-to-guides/retrain-model-ml-net

Microsoft. (2020). *Microsoft Defender Uses ML.NET to Stop Malware*. https://dotnet.microsoft.com/apps/machinelearning-ai/ml-dotnet/customers/microsoft-defender

Mitra, B., Diaz, F., & Craswell, N. (2017). Luandri: A Clean Lua Interface to the Indri Search Engine. *ArXiv:1702.05042 [Cs]*. http://arxiv.org/abs/1702.05042

Modi, C., Patel, D., Borisaniya, B., Patel, H., Patel, A., & Rajarajan, M. (2013). A survey of intrusion detection techniques in Cloud. *Journal of Network and Computer Applications*, *36*(1), 42–57. https://doi.org/10.1016/j.jnca.2012.05.003

Mowla, N., Doh, I., & Chae, K. (2017). Evolving neural network intrusion detection system for MCPS. *2017 19th International Conference on Advanced Communication Technology (ICACT)*, 183–187. https://doi.org/10.23919/ICACT.2017.7890080

Narudin, F. A., Feizollah, A., Anuar, N. B., & Gani, A. (2016). Evaluation of machine learning classifiers for mobile malware detection. *Soft Computing*, *20*(1), 343–357. https://doi.org/10.1007/s00500-014-1511-6

Netresec. (2019). *NetworkMiner Documentation*. https://www.netresec.com/?page=NetworkMiner

Olan, M. (2003). *Unit testing: Test early, Test Often*. ResearchGate.

OpenNMS. (2019). *OpenNMS Documentation*. https://www.opennms.com/documentation/

Pathak, B., & Lal, N. (2017). Information retrieval from heterogeneous data sets using moderated IDF-cosine similarity in vector space model. *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, 3793–3799. https://doi.org/10.1109/ICECDS.2017.8390174

Pries-Heje, J., Baskerville, R., & Venable, J. R. (2008). *Strategies for Design Science Research Evaluation*. 13.

Qi, X., Wang, T., & Liu, J. (2017). Comparison of Support Vector Machine and Softmax Classifiers in Computer Vision. *2017 Second International Conference on Mechanical, Control and Computer Engineering (ICMCCE)*, 151–155. https://doi.org/10.1109/ICMCCE.2017.49

Salton, G., Wong, A., & Yang, C. S. (1975). A vector space model for automatic indexing. *Communications of the ACM*, *18*(11), 613–620. https://doi.org/10.1145/361219.361220

Samrin, R., & Vasumathi, D. (2017). Review on anomaly based network intrusion detection system. *2017 International Conference on Electrical, Electronics, Communication, Computer, and Optimization Techniques (ICEECCOT)*, 141–147. https://doi.org/10.1109/ICEECCOT.2017.8284655

Santos, P. S. M. dos, & Travassos, G. H. (2009). *Action Research Use in Software Engineering: An Initial Survey*. 414–417.

Sewak, M., Sahay, S. K., & Rathore, H. (2018). Comparison of Deep Learning and the Classical Machine Learning Algorithm for the Malware Detection. *2018 19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, 293–296. https://doi.org/10.1109/SNPD.2018.8441123

Sharafaldin, I., Habibi Lashkari, A., & Ghorbani, A. A. (2018). Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization: *Proceedings of the 4th International Conference on Information Systems Security and Privacy*, 108–116. https://doi.org/10.5220/0006639801080116

Singh, K., & Agrawal, S. (2011). Comparative analysis of five machine learning algorithms for IP traffic classification. *2011 International Conference on Emerging Trends in Networks and Computer Communications (ETNCC)*, 33–38. https://doi.org/10.1109/ETNCC.2011.5958481

Statista. (2019). *Annual Detections of New Malware Worldwide from 2015 to May 2019*. https://www.statista.com/statistics/680953/global-malware-volume/

Suricata. (2019). *Suricata Documentation*. https://suricata-ids.org/docs/

Tcpdump. (2019). *TCPDump & Libpcap Documentations*. https://www.tcpdump.org/

Telerik. (2019). *Telerik Fiddler Documentation*. https://docs.telerik.com/fiddler

The Apache Software Foundation. (2019). *The Apache Lucene*. http://lucene.apache.org/

The Zeek Project Revision. (2019). *Zeek Documentation*. https://docs.zeek.org/en/stable/intro/index.html

Thongkanchorn, K., Ngamsuriyaroj, S., & Visoottiviseth, V. (2013). Evaluation studies of three intrusion detection systems under various attacks and rule sets. *2013 IEEE International Conference of IEEE Region 10 (TENCON 2013)*, 1–4. https://doi.org/10.1109/TENCON.2013.6718975

Turney, P. D., & Pantel, P. (2010). From Frequency to Meaning: Vector Space Models of Semantics. *Journal of Artificial Intelligence Research*, *37*, 141–188. https://doi.org/10.1613/jair.2934

Turtle, H., Hegde, Y., & Rowe, S. A. (2012). Yet another comparison of Lucene and Indri performance. *Proceedings of the SIGIR 2012 Workshop on Open Source Information Retrieval*, 4.

Unicode. (2019). *Unicode 12.1 Character Code Charts*. http://www.unicode.org/versions/Unicode12.1.0/

University of Glasgow. (2019). *Terrier Documentation*. http://terrier.org/

Van Gysel, C., Kanoulas, E., & de Rijke, M. (2017). Pyndri: A Python Interface to the Indri Search Engine. *ArXiv:1701.00749 [Cs]*. http://arxiv.org/abs/1701.00749

Wieringa, R. (2014). *Design Science Methodology for Information Systems and Software Engineering*. Springer.

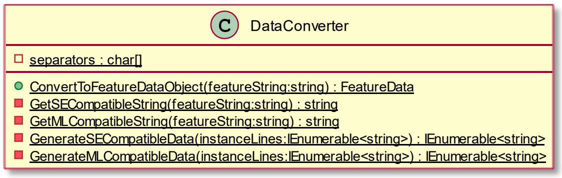
Wieringa, R., Daneva, M., & Condori-Fernandez, N. (2011). The Structure of Design Theories, and an Analysis of their Use in Software Engineering Experiments. *2011 International Symposium on Empirical Software Engineering and Measurement*, 295–304. https://doi.org/10.1109/ESEM.2011.38

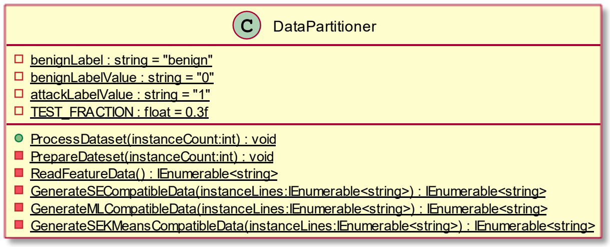
Wireshark. (2019). *Wireshark Documentations*. https://www.wireshark.org/docs/

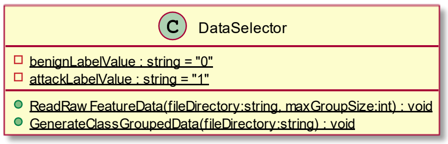
Yin, D., Nobata, C., Langlois, J.-M., Chang, Y., Hu, Y., Tang, J., Daly, T., Zhou, M., Ouyang, H., Chen, J., Kang, C., & Deng, H. (2016). Ranking Relevance in Yahoo Search. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16*, 323–332. https://doi.org/10.1145/2939672.2939677

APPENDIX A: CLASS DIAGRAMS

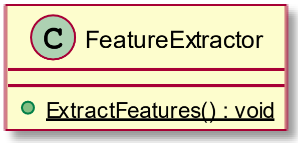
Data Processor

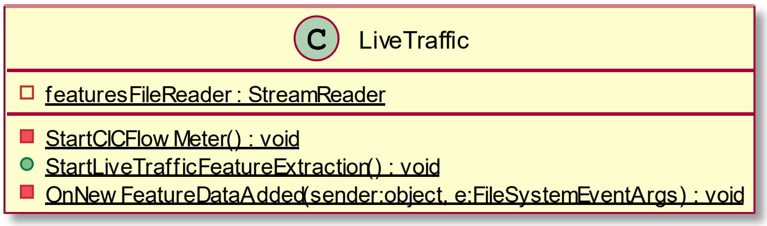

C DataConverter
□ separators : char[]
● ConvertToFeatureDataObject(featureString:string) : FeatureData
■ GetSECompatibleString(featureString:string) : string
■ GetMLCompatibleString(featureString:string) : string
■ GenerateSECompatibleData(instanceLines:IEnumerable<string>) : IEnumerable<string>
■ GenerateMLCompatibleData(instanceLines:IEnumerable<string>) : IEnumerable<string>


C DataPartitioner
□ benignLabel : string = "benign"
□ benignLabelValue : string = "0"
□ attackLabelValue : string = "1"
□ TEST_FRACTION : float = 0.3f
● ProcessDataset(instanceCount:int) : void
■ PrepareDateset(instanceCount:int) : void
■ ReadFeatureData() : IEnumerable<string>
■ GenerateSECompatibleData(instanceLines:IEnumerable<string>) : IEnumerable<string>
■ GenerateMLCompatibleData(instanceLines:IEnumerable<string>) : IEnumerable<string>
■ GenerateSEKMeansCompatibleData(instanceLines:IEnumerable<string>) : IEnumerable<string>

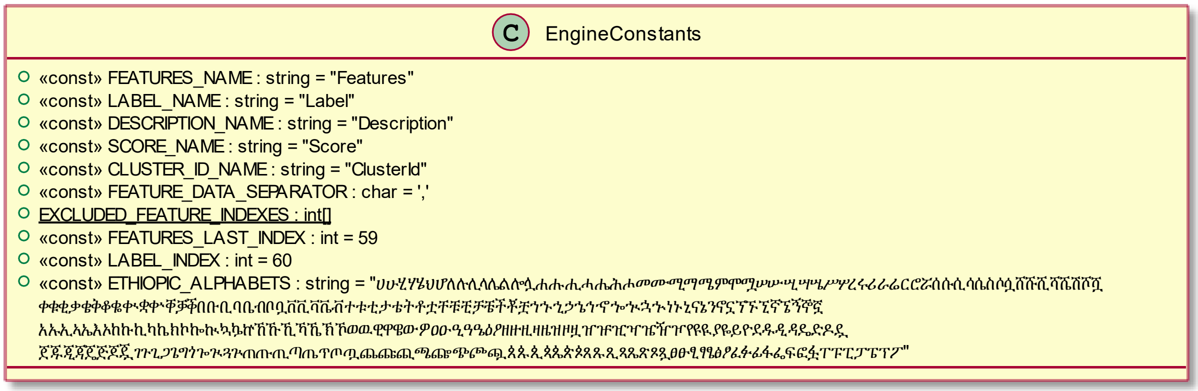

C DataSelector
□ benignLabelValue : string = "0"
□ attackLabelValue : string = "1"
● ReadRaw FeatureData(fileDirectory:string, maxGroupSize:int) : void
● GenerateClassGroupedData(fileDirectory:string) : void

Feature Generator

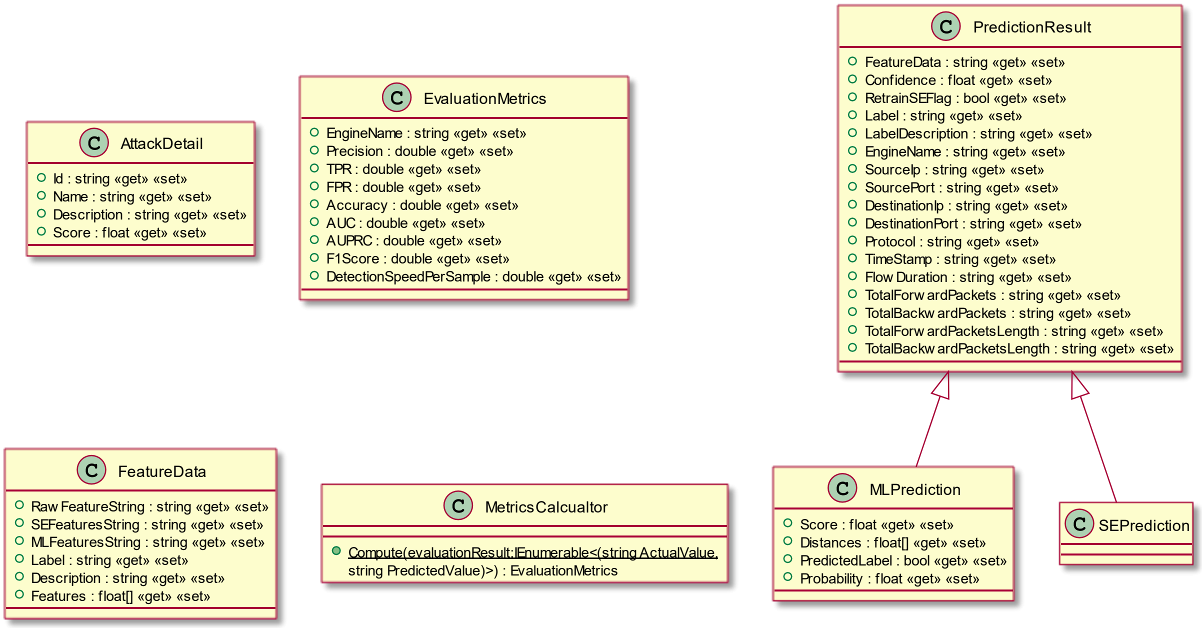



C LiveTraffic
□ featuresFileReader : StreamReader
■ StartCICFlow Meter() : void
● StartLiveTrafficFeatureExtraction() : void
■ OnNew FeatureDataAdded(sender:object, e:FileSystemEventArgs) : void

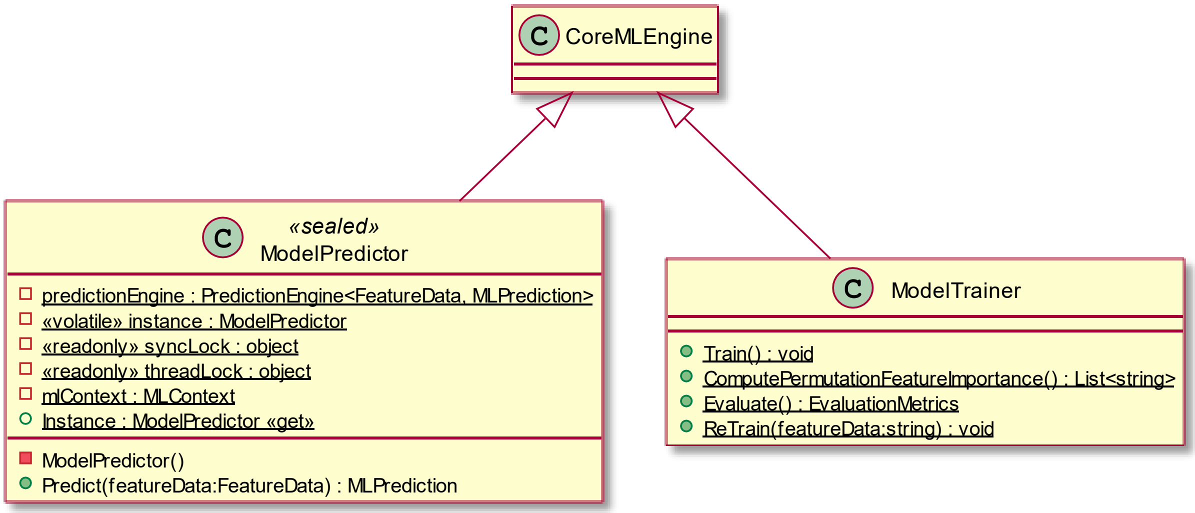
Engine Constants


C EngineConstants
○ «const» FEATURES_NAME : string = "Features"
○ «const» LABEL_NAME : string = "Label"
○ «const» DESCRIPTION_NAME : string = "Description"
○ «const» SCORE_NAME : string = "Score"
○ «const» CLUSTER_ID_NAME : string = "ClusterId"
○ «const» FEATURE_DATA_SEPARATOR : char = ','
○ EXCLUDED_FEATURE_INDEXES : int[]
○ «const» FEATURES_LAST_INDEX : int = 59
○ «const» LABEL_INDEX : int = 60
○ «const» ETHIOPIC_ALPHABETS : string = "ሀሁሂሃሄህሆለሉሊላሌልሎሏሐሑሒሓሔሕሖመሙሚማሜምሞሟሠሡሢሣሤሥሦሧረሩሪራሬርሮሯሰሱሲሳሴስሶሷሸሹሺሻሼሽሾሿ
ቀቁቂቃቄቅቆቈቊቋቌቍበቡቢባቤብቦቧቨቩቪቫቬቭቮቯተቱቲታቴትቶቷቸቹቺቻቼችቾቿኀኁኂኃኄኅኆኈኊኋኌኍነኑኒናኔንኖኗኘኙኚኛኜኝኞኟ
አኡኢኣኤእኦከኩኪካኬክኮኰኲኳኴኵኸኹኺኻኼኽኾወዉዊዋዌውዎዐዑዒዓዔዕዖዘዙዚዛዜዝዞዟዠዡዢዣዤዥዦዧየዩዪያዬይዮደዱዲዳዴድዶዷ
ጀጁጂጃጄጅጆጇገጉጊጋጌግጎጐጒጓጔጕጠጡጢጣጤጥጦጧጨጩጪጫጬጭጮጯጰጱጲጳጴጵጶጷጸጹጺጻጼጽጾጿፀፁፂፃፄፅፆፈፉፊፋፌፍፎፏፐፑፒፓፔፕፖ"

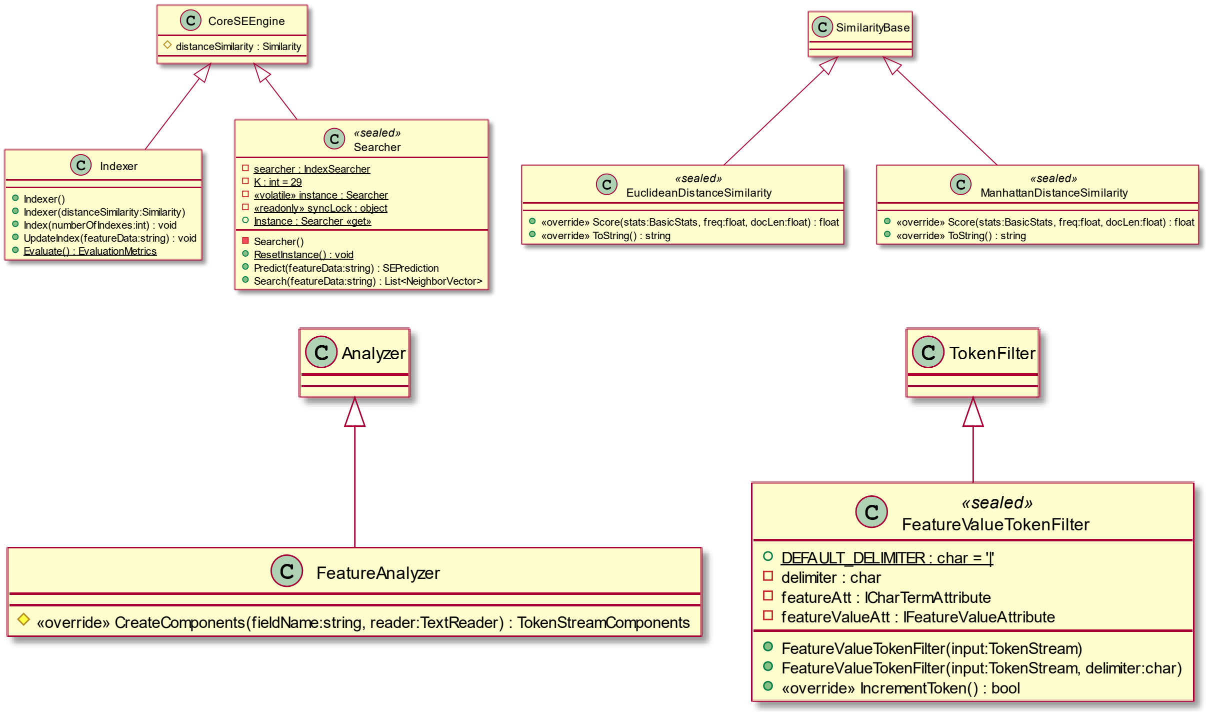
Core Engine


C AttackDetail
○ Id : string «get» «set»
○ Name : string «get» «set»
○ Description : string «get» «set»
○ Score : float «get» «set»
C EvaluationMetrics
○ EngineName : string «get» «set»
○ Precision : double «get» «set»
○ TPR : double «get» «set»
○ FPR : double «get» «set»
○ Accuracy : double «get» «set»
○ AUC : double «get» «set»
○ AUPRC : double «get» «set»
○ F1Score : double «get» «set»
○ DetectionSpeedPerSample : double «get» «set»
C PredictionResult
○ FeatureData : string «get» «set»
○ Confidence : float «get» «set»
○ RetrainSEFlag : bool «get» «set»
○ Label : string «get» «set»
○ LabelDescription : string «get» «set»
○ EngineName : string «get» «set»
○ SourceIp : string «get» «set»
○ SourcePort : string «get» «set»
○ DestinationIp : string «get» «set»
○ DestinationPort : string «get» «set»
○ Protocol : string «get» «set»
○ TimeStamp : string «get» «set»
○ FlowDuration : string «get» «set»
○ TotalForwardPackets : string «get» «set»
○ TotalBackwardPackets : string «get» «set»
○ TotalForwardPacketsLength : string «get» «set»
○ TotalBackwardPacketsLength : string «get» «set»
C FeatureData
○ RawFeatureString : string «get» «set»
○ SEFeaturesString : string «get» «set»
○ MLFeaturesString : string «get» «set»
○ Label : string «get» «set»
○ Description : string «get» «set»
○ Features : float[] «get» «set»
C MetricsCalcualtor
● Compute(evaluationResult:IEnumerable<(string ActualValue, string PredictedValue)>) : EvaluationMetrics
C MLPrediction
○ Score : float «get» «set»
○ Distances : float[] «get» «set»
○ PredictedLabel : bool «get» «set»
○ Probability : float «get» «set»
C SEPrediction

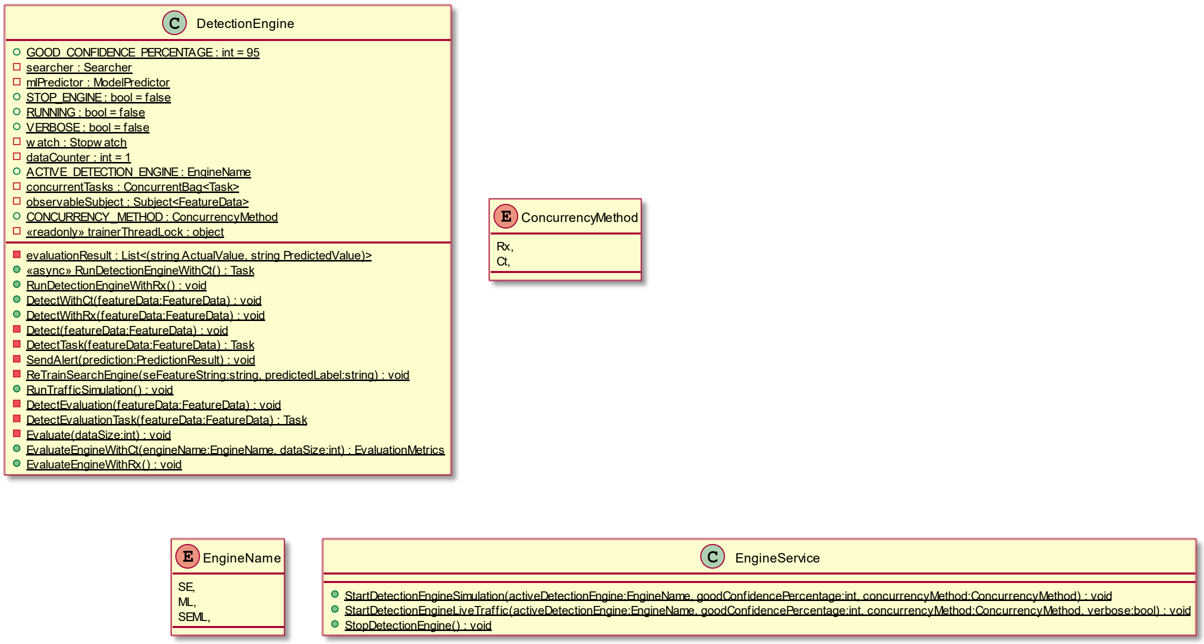
ML Engine


C CoreMLEngine
C «sealed» ModelPredictor
□ predictionEngine : PredictionEngine<FeatureData, MLPrediction>
□ «volatile» instance : ModelPredictor
□ «readonly» syncLock : object
□ «readonly» threadLock : object
□ mlContext : MLContext
○ Instance : ModelPredictor «get»
■ ModelPredictor()
● Predict(featureData:FeatureData) : MLPrediction
C ModelTrainer
● Train() : void
● ComputePermutationFeatureImportance() : List<string>
● Evaluate() : EvaluationMetrics
● ReTrain(featureData:string) : void

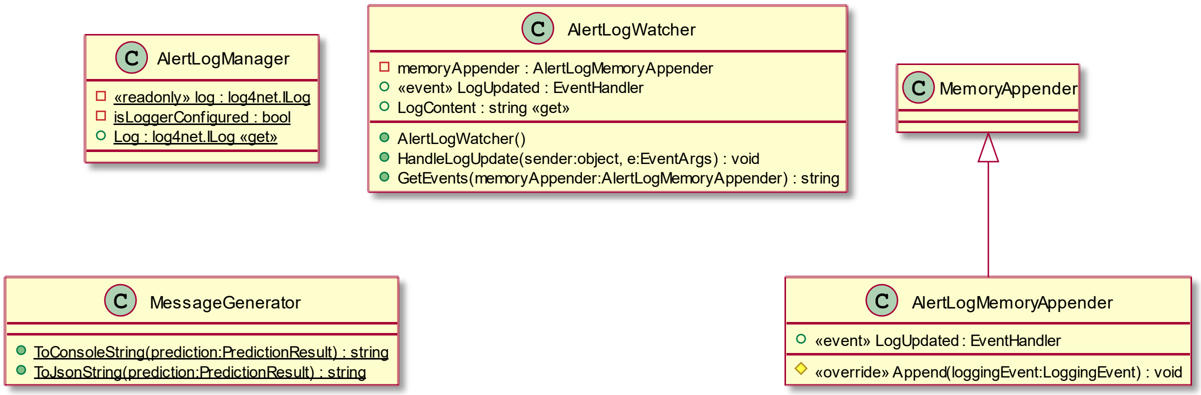
Search Engine


CoreSEEngine
distanceSimilarity : Similarity
Indexer
Indexer()
Indexer(distanceSimilarity:Similarity)
Index(numberOfIndexes:int) : void
UpdateIndex(featureData:string) : void
Evaluate() : EvaluationMetrics
«sealed»
Searcher
searcher : IndexSearcher
K : int = 29
«volatile» instance : Searcher
«readonly» syncLock : object
Instance : Searcher «get»
Searcher()
ResetInstance() : void
Predict(featureData:string) : SEPrediction
Search(featureData:string) : List<NeighborVector>
SimilarityBase
«sealed»
EuclideanDistanceSimilarity
«override» Score(stats:BasicStats, freq:float, docLen:float) : float
«override» ToString() : string
«sealed»
ManhattanDistanceSimilarity
«override» Score(stats:BasicStats, freq:float, docLen:float) : float
«override» ToString() : string
Analyzer
FeatureAnalyzer
«override» CreateComponents(fieldName:string, reader:TextReader) : TokenStreamComponents
TokenFilter
«sealed»
FeatureValueTokenFilter
DEFAULT_DELIMITER : char = '|'
delimiter : char
featureAtt : ICharTermAttribute
featureValueAtt : IFeatureValueAttribute
FeatureValueTokenFilter(input:TokenStream)
FeatureValueTokenFilter(input:TokenStream, delimiter:char)
«override» IncrementToken() : bool

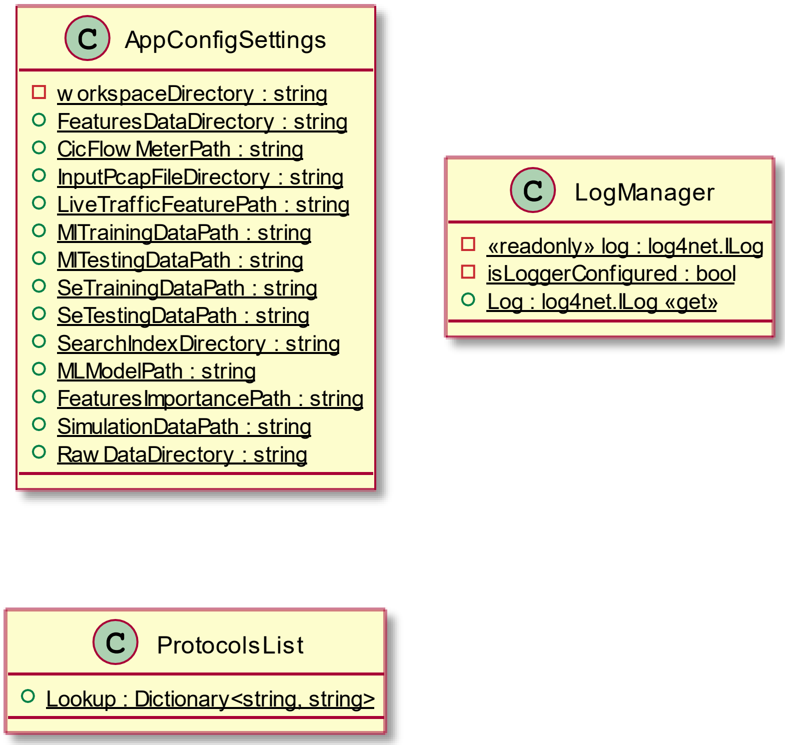
Detection Engine


DetectionEngine
GOOD_CONFIDENCE_PERCENTAGE : int = 95
searcher : Searcher
mlPredictor : ModelPredictor
STOP_ENGINE : bool = false
RUNNING : bool = false
VERBOSE : bool = false
w atch : Stopw atch
dataCounter : int = 1
ACTIVE_DETECTION_ENGINE : EngineName
concurrentTasks : ConcurrentBag<Task>
observableSubject : Subject<FeatureData>
CONCURRENCY_METHOD : ConcurrencyMethod
«readonly» trainerThreadLock : object
evaluationResult : List<(string ActualValue, string PredictedValue)>
«async» RunDetectionEngineWithCt() : Task
RunDetectionEngineWithRx() : void
DetectWithCt(featureData:FeatureData) : void
DetectWithRx(featureData:FeatureData) : void
Detect(featureData:FeatureData) : void
DetectTask(featureData:FeatureData) : Task
SendAlert(prediction:PredictionResult) : void
ReTrainSearchEngine(seFeatureString:string, predictedLabel:string) : void
RunTrafficSimulation() : void
DetectEvaluation(featureData:FeatureData) : void
DetectEvaluationTask(featureData:FeatureData) : Task
Evaluate(dataSize:int) : void
EvaluateEngineWithCt(engineName:EngineName, dataSize:int) : EvaluationMetrics
EvaluateEngineWithRx() : void
ConcurrencyMethod
Rx,
Ct,
EngineName
SE,
ML,
SEML,
EngineService
StartDetectionEngineSimulation(activeDetectionEngine:EngineName, goodConfidencePercentage:int, concurrencyMethod:ConcurrencyMethod) : void
StartDetectionEngineLiveTraffic(activeDetectionEngine:EngineName, goodConfidencePercentage:int, concurrencyMethod:ConcurrencyMethod, verbose:bool) : void
StopDetectionEngine() : void

Alert Service


C AlertLogManager
«readonly» log : log4net.ILog
isLoggerConfigured : bool
Log : log4net.ILog «get»
C AlertLogWatcher
memoryAppender : AlertLogMemoryAppender
«event» LogUpdated : EventHandler
LogContent : string «get»
AlertLogWatcher()
HandleLogUpdate(sender:object, e:EventArgs) : void
GetEvents(memoryAppender:AlertLogMemoryAppender) : string
C MemoryAppender
C MessageGenerator
ToConsoleString(prediction:PredictionResult) : string
ToJsonString(prediction:PredictionResult) : string
C AlertLogMemoryAppender
«event» LogUpdated : EventHandler
«override» Append(loggingEvent:LoggingEvent) : void

Utility


C AppConfigSettings
w orkspaceDirectory : string
FeaturesDataDirectory : string
CicFlow MeterPath : string
InputPcapFileDirectory : string
LiveTrafficFeaturePath : string
MlTrainingDataPath : string
MlTestingDataPath : string
SeTrainingDataPath : string
SeTestingDataPath : string
SearchIndexDirectory : string
MLModelPath : string
FeaturesImportancePath : string
SimulationDataPath : string
Raw DataDirectory : string
C LogManager
«readonly» log : log4net.ILog
isLoggerConfigured : bool
Log : log4net.ILog «get»
C ProtocolsList
Lookup : Dictionary<string, string>

APPENDIX B: LIST OF FEATURES

This table shows a list of features with Permutation Feature Importance (PFI) metrics in descending order of their importance.

Feature	Wght	Δ-AUC	Δ-Accuracy	Δ-F1Score	Δ-Precision	Δ-TPR	Δ-FPR
Init Win bytes forward	1	0.04999	0.07875	0.09242	0.03301	0.1343	0.02321
Init Win bytes backward	0.88	0.03566	0.06658	0.07038	0.07063	0.06984	0.06333
Destination Port	0.57	0.03256	0.0526	0.058	0.03891	0.0726	0.0326
Min seg size forward	0.39	0.01505	0.006006	0.007327	0.005831	0.0178	0.005784
RST Flag Count	0.29	0.01023	0.01293	0.01537	0.005775	0.0319	0.006041
Bwd Packet Length Std	0.34	0.007186	0.01722	0.01448	0.05425	0.0209	0.05534
Total Length of Bwd Packets	0.69	0.006158	0.01003	0.01176	0.003128	0.02354	0.003482
Fwd Header Length	0.48	0.005696	0.01152	0.01241	0.009226	0.01494	0.008107
Bwd Header Length	0.11	0.004672	0.007109	0.008432	0.003787	0.01816	0.003946
Fwd Packet Length Max	0.1	0.003927	0.005658	0.005235	0.01465	0.00265	0.01396
Fwd IAT Min	0.25	0.003677	0.007736	0.008168	0.007976	0.008288	0.007185
ECE Flag Count	0.06	0.002934	0.007737	0.009223	0.004564	0.02017	0.004695
Flow IAT Min	0.3	0.002397	0.004355	0.005122	0.002041	0.01088	0.002176
Fwd Packet Length Mean	0.09	0.002	0.00332	0.002338	0.0178	0.01074	0.01739
Bwd Packet Length Max	0.15	0.001483	0.00147	0.001601	0.0008714	0.002191	0.0007488
Bwd Packets/s	0.25	0.001301	0.002939	0.002837	0.00635	7.50E-05	0.005953
Bwd IAT Max	0.09	0.001216	0.003019	0.003676	0.003097	0.009138	0.003099
Min Packet Length	0.1	0.001211	0.002028	0.002568	0.003413	0.007402	0.003346
Total Length of Fwd Packets	0.11	0.001159	0.003792	0.004531	0.002716	0.01037	0.00278
Packet Length Mean	0.13	0.001117	0.001468	0.001614	0.0006655	0.002384	0.0005517
Average Packet Size	0.09	0.001038	0.002037	0.002595	0.003631	0.007625	0.003551
Fwd Packet Length Min	0.16	0.001001	0.00315	0.003832	0.00318	0.009483	0.003183
Flow IAT Max	0.11	0.0009405	0.002664	0.003015	0.0001611	0.005327	5.17E-07
Bwd Packet Length Mean	0.19	0.0009155	0.001395	0.00165	0.0008625	0.003692	0.0009018
Bwd IAT Min	0.08	0.0008875	0.001396	0.001487	0.001254	0.001672	0.00112

Fwd IAT Max	0.1	0.0008791	0.001574	0.001775	0.0001418	0.003103	4.50E-05
Packet Length Std	0.11	0.0008547	0.001794	0.002026	0.0001415	0.003556	3.21E-05
Fwd Packets/s	0.12	0.0008529	0.001372	0.001699	0.001848	0.00458	0.001836
Flow Bytes/s	0.19	0.0008471	0.002014	0.002456	0.002177	0.006209	0.002181
Fwd Packet Length Std	0.07	0.0008109	0.001569	0.001832	0.0006511	0.003849	0.0007105
Fwd IAT Total	0.11	0.0006778	0.001282	0.001434	0.0002697	0.00238	0.0001841
Flow Duration	0.09	0.0006725	0.0005556	0.0003552	0.003588	0.002315	0.003426
Bwd Packet Length Min	0.13	0.000654	0.001259	0.001516	0.001144	0.00368	0.001162
Flow IAT Std	0.08	0.0006259	0.000952	0.0009441	0.001766	0.0002648	0.001639
Bwd IAT Std	0.07	0.0006047	0.0005874	0.0006614	6.17E-05	0.00115	2.53E-05
Bwd IAT Mean	0.05	0.0004883	0.0002079	4.33E-05	0.003658	0.003101	0.003517
Fwd IAT Std	0.09	0.0004167	0.001047	0.0009723	0.002794	0.0005316	0.002625
Fwd PSH Flags	0.1	0.0003863	0.0004812	0.0004331	0.001472	0.0004225	0.001385
PSH Flag Count	0.06	0.0003835	2.43E-05	0.0002883	0.003432	0.003361	0.003312
Flow IAT Mean	0.06	0.0003634	0.0007583	0.0009749	0.001505	0.002994	0.001477
Idle Std	0.07	0.0003266	0.0008638	0.0008698	0.001433	0.0004039	0.001324
Max Packet Length	0.09	0.0003095	0.0004628	0.0003398	0.00242	0.001375	0.002301
Fwd IAT Mean	0.07	0.0002825	0.0006422	0.0005944	0.00175	0.0003578	0.001642
Bwd IAT Total	0.07	0.000251	0.000379	0.0004215	0.0001076	0.0006769	8.12E-05
Active Std	0.07	0.0002138	0.0004005	0.0003939	0.0007891	6.77E-05	0.0007332
Idle Mean	0.04	0.0002068	0.001414	0.001769	0.002135	0.004937	0.00211
Total Backward Packets	0.05	0.0001904	0.0003131	0.000219	0.001784	0.00107	0.001696
Flow Packets/s	0.05	0.0001876	0.0002143	0.0002295	0.0001765	0.000272	0.0001567
Total Fwd Packets	0.13	0.0001851	0.0003478	0.000413	0.0002473	0.000952	0.0002565
Act data pkt fwd	0.13	0.0001441	0.0003284	0.0003296	0.0005605	0.0001386	0.0005181
Active Min	0.08	0.0001357	0.0003397	0.0004051	0.0002624	0.0009499	0.0002704
Active Max	0.03	8.60E-05	0.0003188	0.0002923	0.0009074	0.0002141	0.0008517
Idle Max	0.06	8.21E-05	0.001972	0.002467	0.002933	0.006836	0.002892
ACK Flag Count	0.05	5.86E-05	7.65E-05	6.11E-05	0.0003381	0.0001665	0.0003196
Down/Up Ratio	0.11	3.67E-05	0.0001753	0.0002022	4.62E-05	0.0004049	5.43E-05
URG Flag Count	0.03	3.22E-05	9.18E-05	6.51E-05	0.0005132	0.000303	0.0004866
Idle Min	0.06	2.50E-05	0.0001407	0.0002137	0.000717	0.0009747	0.0006934
FIN Flag Count	0.01	1.34E-05	1.81E-06	2.45E-06	5.29E-06	8.79E-06	5.17E-06
Active Mean	0.04	3.31E-06	0.0003074	0.0004256	0.001015	0.001602	0.0009871
Fwd URG Flags	0.04	1.75E-06	2.59E-07	7.41E-07	5.92E-06	6.21E-06	5.69E-06
Protocol	0	0	0	0	0	0	0
Bwd PSH Flags	0	0	0	0	0	0	0

Bwd URG Flags	0	0	0	0	0	0	0
Packet Length Variance	0	0	0	0	0	0	0
SYN Flag Count	0	0	0	0	0	0	0
CWE Flag Count	0	0	0	0	0	0	0
Avg Fwd Segment Size	0	0	0	0	0	0	0
Avg Bwd Segment Size	0	0	0	0	0	0	0
Fwd Avg Bytes/Bulk	0	0	0	0	0	0	0
Fwd Avg Packets/Bulk	0	0	0	0	0	0	0
Fwd Avg Bulk Rate	0	0	0	0	0	0	0
Bwd Avg Bytes/Bulk	0	0	0	0	0	0	0
Bwd Avg Packets/Bulk	0	0	0	0	0	0	0
Bwd Avg Bulk Rate	0	0	0	0	0	0	0
Subflow Fwd Packets	0	0	0	0	0	0	0
Subflow Fwd Bytes	0	0	0	0	0	0	0
Subflow Bwd Packets	0	0	0	0	0	0	0
Subflow Bwd Bytes	0	0	0	0	0	0	0