Machine Learning for Detecting Ransomware Attacks Using BGP Routing Records

by

Ana Laura Gonzalez Rios

B.Sc., Monterrey Institute of Technology and Higher Education (ITESM), 2013

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Applied Science

in the
School of Engineering Science
Faculty of Applied Sciences

Declaration of Committee

Name:	**Ana Laura Gonzalez Rios**
Degree:	**Master of Applied Science**
Thesis title:	**Machine Learning for Detecting Ransomware Attacks Using BGP Routing Records**
Committee:	**Chair:** Zhenman Fang Assistant Professor, Engineering Science
	Ljiljana Trajković Supervisor Professor, Engineering Science
	Ivan Bajić Committee Member Professor, Engineering Science
	Uwe Glässer Examiner Professor, Computing Science

Abstract

Analyzing and detecting Border Gateway Protocol (BGP) anomalies caused by evolving ransomware cyber attacks are topics of great interest in cyber security. Various anomaly detection approaches such as time series and historical-based analysis, statistical validation, reachability checks, and supervised machine learning have been applied to collected datasets. Supervised and semi-supervised machine learning techniques rely on label and unlabeled data that contain regular and anomalous events. They are publicly available from BGP collection sites such as Réseaux IP Européens (RIPE) and Route Views.

Keywords: Communication networks; BGP; intrusion detection; network anomalies; ransomware; machine learning

Dedication

To my parents and my sister, who have been the best adventure partners in this journey called life; to my grandmother Tere, who has been an example of perseverance and dedication; and to my husband Miguel and our unborn child, who are my main motivation and strength.

Acknowledgements

I would like to express my gratitude to all the individuals who offered assistance during the preparation of this thesis.

I give my sincere thanks and appreciation to my advisor Prof. Ljiljana Trajković for giving me the opportunity to join her research team and work in the field of communication networks and machine learning. She dedicated her time to guide my research and encouraged me to explore ideas and research directions that will significantly influence my career.

I would also like to thank the Chair Prof. Zhenman Fang, Committee Member Prof. Ivan V. Bajić, and Internal Examiner Prof. Uwe Glässer for reviewing this thesis and providing constructive feedback.

I am very thankful to my colleagues in the Communication Networks Laboratory. I would like to particularly thank my colleague Dr. Zhida Li for his support and guidance during my research work as well as his dedication in preparing all figures used to illustrate architectures of the various machine learning algorithms included in this thesis. I would also like to thank Hardeep Kaur Takhar and my former colleague Kamila Bekshentayeva for their valuable comments and suggestions.

I would like to thank my family for their support and care throughout my graduate studies. Last but not least, I would also like to extend my gratitude to my former managers Justin Foster, Nathan Aubry, and Benoit Essiambre as well as my current coworkers Vivek Muchalla and Cornel Sarament for inspiring and giving me the opportunity to develop valuable technical skills that have greatly influenced my contributions in this thesis.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ACK	Acknowledge
AES	Advanced Encryption Standard
AFRINIC	African Network Information Center
AI	Artificial Intelligence
APC	Asynchronous Procedure Call
APNIC	Asia-Pacific Network Information Centre
ARIN	American Registry for Internet Numbers
ASCII	American Standard Code for Information Interchange
ASes	Autonomous Systems
ASLR	Address Space Layout Randomization
ASN	AS Number
AU-ROC	Area Under Receiver Operating Characteristics Curve
BGP	Border Gateway Protocol
Bi-RNNs	Bidirectional Recurrent Neural Networks
BLS	Broad Learning System
CatBoost	Categorical Boosting
CBC	Cipher Block Chaining
CEBLS	Cascades of Enhancement Nodes BLS
CFBLS	Cascades of Mapped Features BLS
CFEBLS	Cascades of Mapped Features and Enhancement Nodes BLS
CICIDS	Canadian Institute for Cybersecurity Intrusion Detection System
CIDR	Classless Inter-Domain Routing
DCNs	Data Center Networks
DDoS	Distributed Denial of Service
DLL	Dynamic Link Library

DoS	Denial of Service
EBGP	External BGP
EFB	Exclusive Feature Bundling
EGP	Exterior Gateway Protocol
Extra-Trees	Extremely Randomized Trees
FC	Fully-Connected
FN	False Negative
FP	False Positive
FSM	Finite State Machine
GBDT	Gradient Boosting Decision Trees
GBMs	Gradient Boosting Machines
GOSS	Gradient-Based One-Side Sampling
GRU	Gated Recurrent Unit
IANA	Internet Assigned Numbers Authority
IBGP	Internal BGP
IDSs	Intrusion Detection Systems
iForest	Isolation Forest
IGP	Interior Gateway Protocol
IIS	Internet Information Service
IP	Internet Protocol
IT	Information Technology
iTrees	Isolation Trees
LACNIC	Latin America and Caribbean Network Information Centre
LightGBM	Light Gradient Boosting Machine
LSTM	Long Short-Term Memory
ML	Machine Learning
MRT	Multi-threaded Routing Toolkit
NDAE	Stacked Non-symmetric Deep Auto-encoder
NIDSs	Network Intrusion Detection Systems
NLRI	Network Layer Reachability Information

OT	Operational Techonology
RaaS	Ransomware-as-a-Service
RBF-BLS	Radial Basis Function Network BLS
RIB	Routing Information Bases
RIPE	Réseaux IP Européens
RIPE NCC	Réseaux IP Européens Network Coordination Center
RIRs	Regional Internet Registries
RIS	Routing Information Service
RNNs	Recurrent Neural Networks
rrcs	Remote Route Collectors
SMB	Server Message Block
SQL	Structured Query Language
SVM	Support Vector Machine
SYN	Synchronize
TCP	Transport Control Protocol
TN	True Negative
TP	True Positive
VCFBLS	Variable Features with Cascades BLS
VFBLS	Variable Features BLS
VNE	Virtual Network Embedding
XGBoost	eXtreme Gradient Boosting
YAML	Yet Another Markup Language

Chapter 1

Introduction

In this Chapter, the motivation of the thesis research topic is given followed by an overview of the Border Gateway Protocol (BGP), machine learning approaches, and the topic of anomaly detection. It also includes a summary of the research contributions, a list and description of the publications emanating from this work, and the roadmap of the thesis.

1.1 Motivation

The Internet has been highly susceptible to failures and attacks that greatly affect its performance. Over the past years, frequent cases of complex and challenging threats have been encountered. BGP is an incremental path vector Internet routing protocol that manages network reachability information and optimally routes data between Autonomous Systems (ASes). While BGP is a simple and flexible routing protocol, implementing routing policies is a complex and error-prone task [7, 8]. BGP also lacks security mechanisms to verify legitimate route updates. Therefore, it is prone to anomalies that impede successful delivery of reachability messages and may generate a large volume of *update* messages. Several modifications have been proposed to improve BGP security [9, 10]. BGP anomalies include worms (Slammer [11, 12], Nimda [13, 14], and Code Red I [15]), ransomware attacks (WannaCrypt [16], WestRock [17, 18]), routing misconfigurations [19], Internet Protocol (IP) prefix hijacks [20], and link failures [21] (Moscow blackout [22, 23], Pakistani power outage [24]).

Ransomware attacks, first introduced in 1996 [25], have rapidly evolved and become more popular among attackers due to:

- availability of digital currency and powerful cryptography (asymmetric encryption) methods,
- difficulty to detect new ransomware variants, and
- attractiveness of the business model that offers sophisticated tools and high revenues for affiliates to get the maximum benefit.

Between 2014 and 2015, ransomware victims reported more than USD $18 million in losses [26]. During 2020, after the beginning of the COVID-19 pandemic, more than 2,400 cases of ransomware attacks were identified with losses of more that USD $21.9 million [27]. In 2021, rectifying the impact of recent ransomware attacks in an organization had an average cost of USD $1.4 millions [28]. Typical steps followed during a ransomware attack include distribution of the of the attack (phishing emails, compromised websites), infection of the victim's host, communication to an encryption-key server, search for important files to encrypt in the victim's host, encryption of data (moving and renaming targeted files before and after successful encryption), and ransom demand (locking the host's screen and demanding payment). Recommended protection strategies consist of [26, 29]:

- raising awareness among users,
- implementing regular backup in dedicated devices with no access to the Internet,
- performing system updates regularly,
- disabling threatening or weak services/processes,
- limiting administrative permissions,
- establishing spam and web filtering rules to prevent ransomware from reaching the network and hosts, and
- restricting the number of available files shared in a network.

Most detection techniques scan hosts to identify suspicious processes or monitor for compromising websites. However, a more global detection technique may rely on detecting ransomware attacks while they are being spread through the Internet and, hence, prevent various organizations from being affected. Therefore, studying BGP anomalies cause during periods of ransomware attacks that occur worldwide may enable the development of effective detection techniques at a global level.

Machine learning algorithms [30, 31, 32, 33] have been used to address a variety of engineering and scientific problems. They may be used to classify data using a feature matrix where its rows and columns correspond to data points and feature values, respectively. By extracting relevant features, machine learning approaches help build generalized classification models and improve their performance. Machine learning algorithms are classified as supervised, unsupervised, and semi-supervised. In supervised algorithms, labeled data are employed during training. Unsupervised algorithms are used to cluster (group) unlabeled data into various categories based on similarity of features. Semi-supervised [34] algorithms are trained using a combination of labeled and unlabeled data to improve data classification and clustering. BGP anomalies [35, 6] have been classified using various supervised machine learning algorithms such as support vector machine [36], Recurrent Neural

Networks (RNNs) [31], Bidirectional Recurrent Neural Networks (Bi-RNNs) [31], Broad Learning System (BLS) [4, 37], and Gradient Boosting Decision Trees (GBDT) [38, 39, 40]. Implementing semi-supervised machine learning algorithms may improve the classification and detection of BGP anomalies when using routing records available from various sites such as Réseaux IP Européens (RIPE) [41] and Route Views [42].

1.2 Border Gateway Protocol (BGP)

The Internet consists of numerous ASes that encompass routers (peers) within a single technical administration implementing a unified routing policy. ASes use BGP to exchange network reachability information. This path-vector routing protocol was introduced in 1989 [43, 44] and was first deployed in 1994. The current version of BGP (BGP-4) has been in use since 2006 [1]. BGP-4 enables Classless Inter-Domain Routing (CIDR) by offering mechanisms such as support for advertising a set of destinations as an IP prefix, routes aggregation, incremental updates, and changes to local routing policies without the need to reset BGP connections.

BGP routers are classified as internal and external peers [1]. Internal peers are part of the same AS and use Interior Gateway Protocol (IGP) to exchange routing information. External peers reside in distinct ASes and use Exterior Gateway Protocol (EGP) for exchanging routing information. Internal BGP (IBGP) and External BGP (EBGP) protocols define connections between internal and external peers, respectively. Reliable router-to-router BGP communication is established using Transport Control Protocol (TCP) sessions (port 179). Routing information advertised by a local BGP peer is stored in Routing Information Bases (RIB). They consist of three sections: Adj-RIB-In containing unprocessed routing information, Loc-RIB having selected routing information based on applied local policies, and Adj-RIB-Out storing routes for advertisement to specific peers using *update* messages. BGP routers exchange four types of messages: *open*, *keepalive*, *update*, and *notification.* An *open* message sent after establishing a TCP connection and upon confirmation of its receipt, is followed by a *keepalive* message. Routing advertisements and withdrawals are exchanged between BGP peers using *update* messages. When an error condition is detected, a *notification* message is sent and the BGP connection is closed.

1.3 Machine Learning Approaches

The interdisciplinary field of Artificial Intelligence (AI) emerged in 1950s aiming to explore automating intellectual tasks performed by humans [45]. During its early stage, AI was conceived as a comprehensive set of rules that allowed the knowledge manipulation. This concept of AI was embodied in the dominant paradigm of symbolic AI between 1950s and 1980s. A drawback of this paradigm was its intractability when solving fuzzy problems that required complex sets of rules for data processing using classical programming. Machine

Learning (ML) arose as a new paradigm to symbolic AI, where data and expected outputs were used to train a system in order to generate a set of rules for tasks automation. ML has become a subfield of AI starting in late 1990s. It has gained popularity due to faster hardware and availability of larger datasets [46] (big data).

In machine learning, statistics is applied to large datasets by using computers to estimate functions that lead to descriptive and predictive models [47, 31]. ML algorithms rely on experience E with respect to a task T and performance measure P [48]. Based on the process employed in a machine learning system to transform an example or collection of features, a tasks T is defined as [31]:

Classification: A category k is specified for a given input $\boldsymbol{x}$ by using a function $f : \mathbb{R}^n \to \{1, \ldots, k\}$. A numeric value of f is assigned to the output y.

Classification with missing inputs: Not all elements of input $\boldsymbol{x}$ are provided and a set of functions accounts for various subsets of the input with missing values.

Regression: Its goal is to predict a numerical value given an input.

Transcription: An unstructured representation of a data type are transcribed into discrete textual form.

Machine translation: A sequence of symbols is converted to another language.

Structured prediction: Predicted are relationships between elements within a data structure.

Anomaly detection: Unusual or atypical events are identified or flagged.

Synthesis and sampling: Generated are new examples similar to the input.

Imputation of missing values: Predicted are missing values of an example $\boldsymbol{x} \in \mathbb{R}^n$.

Denoising: Predicted is a clean value $\boldsymbol{x} \in \mathbb{R}^n$ given its corrupted form $\tilde{\boldsymbol{x}} \in \mathbb{R}^n$.

Density estimation or probability mass function estimation: A probability density function or probability mass function is inferred for a continuous or discrete input, respectively. The structure of the probability distribution is implicitly captured by the machine learning algorithm.

Depending on the type of task T to be solved, employed performance measures P include accuracy, precision, sensitivity, F-Score, mean absolute error, and mean squared error. The experience E of a machine learning algorithm is related to the input data used during the training process. Machine learning algorithms are broadly categorized as supervised and unsupervised based on experience E. The process to generate machine learning models typically consist of training, validation, and testing phases while datasets are partitioned to create subsets used in each phase [49]. During the training phase, the training dataset is employed to fit a selected machine learning algorithm and its hyperparameters. The validation phase is used to evaluate various machine learning models in order to select the set of hyperparameters leading to the best performance. After fine-tuning hyperparameters, the generalization of the best generated machine learning model is evaluated using a testing dataset that contains unobserved samples.

1.3.1 Supervised Machine Learning

Supervised machine learning algorithms are trained using datasets that contain labels (targets) associated with each data point. The task of developed supervised machine learning models is to predict the correct label of a data point during the validation and test processes. A predicted label y is provided by a resident expert in order to instruct the algorithm [31].

Classification, structured prediction, and regression tasks are traditionally solved using supervised machine learning. Performance of classification and structured prediction tasks is based on their ability to correctly predict classes or values. Performance metrics include accuracy, confusion matrix, precision, sensitivity (recall), F-Score, and Area Under Receiver Operating Characteristics Curve (AU-ROC). Regression tasks have continuous outputs and, hence, require performance evaluation based on the difference between predicted values and ground truth. These performance metrics include mean absolute error, mean squared error, root mean squared error, and R^2 coefficient.

1.3.2 Unsupervised Machine Learning

Unsupervised machine learning algorithms only contain features that are used to learn properties of the data correlations without providing a target y. The task of unsupervised machine learning models is to find similarities, patterns, and probability distributions by inferring an underlying structure from the input [31, 34]. These algorithms are typically used for dimensionality reduction and clustering (division of data in clusters of similar examples).

Denoising, synthesis, denisity estimation, and anomaly detection tasks are usually solved employing unsupervised machine learning. When employing clustering algorithms to solve these tasks, performance is commonly evaluated based on the silhouette, completeness, homogeneity, adjusted mutual information, adjusted Rand score, V-measure, and contingency matrix [47]. Performance of dimensionality reduction algorithms is evaluated based on reconstruction errors while performance of algorithms used for anomaly detection are evaluated using mass-volume [50] and excess-mass [51] curves.

1.3.3 Semi-Supervised Machine Learning

Semi-supervised machine learning algorithms may be employed where labeled data are scarce due to expensive or difficult labeling. Data clustering and classification tasks are combined to extract relevant information and predict labels. Most semi-supervised learning algorithms rely on explicitly or implicitly satisfying one or more assumptions [34]:

> *Smoothness assumption*: Labels y and y' should be identical for two samples x and x' that are close in the input space.
>
> *Low-density assumption*: The decision boundary should not pass through high-density areas in the input space.
>
> *Manifold assumption*: Data points on the same low-dimensional manifold should have identical labels.

Semi-supervised approaches may be classified as inductive (boosting, cluster-then-label, manifolds algorithms) and transductive (graph-based algorithms). In inductive approaches, a classifier is constructed to generate predictions by using the unlabelled data during the training process. Transductive approaches rely on using connections between data points and, thus, result in graph-based approaches that consist of three steps: graph construction, graph weighting, and inference (assigning labels to the unlabelled data points).

1.4 Anomaly Detection

The anomaly detection task identifies data patterns that do not conform with a well-defined expected behavior [52]. Depending on the application, these patterns are referred to as anomalies, outliers, discordant observations, aberrations, surprises, peculiarities, or contaminants. Anomalies and outliers are the most commonly and interchangeably used terms. Anomaly detection was first studied by the statistics community in the 19^{th} century. It is of interest to a wide range of application domains such as cybersecurity intrusion detection, fraud detection, industrial fault or damage detection, medical and public health diagnosis, text data, and event detection in sensor networks. Analysis of detected anomalies may help improve applications and the decision making process in the onset of critical incidents. Some of the most common challenges in anomaly detection include defining the region of expected behavior, adaptation of ever-evolving anomalous activity, maintain an updated notion of expected behavior in a particular domain, applicability of developed techniques across domains, lack of labeled data for training or validating models, and presence of noise. Anomaly detection techniques may be supervised, unsupervised, or semi-supervised depending on the availability of labeled data. They may produce labels or scores (ranked list) during the testing process. Unlike labeled-based anomaly detection techniques, score-based techniques allow defining thresholds to select anomalous data points of interest. In the score-based techniques, data points are assigned a score (ranked) based on their likelihood of being anomalous. Type and number of attributes, relationship between data points, and nature or type of anomalies are relevant aspects to consider when developing anomaly detection techniques. They are explained in the following paragraphs.

Development of anomaly detection techniques rely on the type and number of attributes of input data points and their relationships. Types of attributes are binary, categorical, or continuous. Binary attributes are defined as: 0 (attribute is absent) or 1 (attribute is present). Categorical attributes (qualitative) consist of symbols (ordinal) or names (nominal) that determine the category, code, or state of the data point [53]. Continuous attributes (quantitative) are numerical values that are typically discretized and categorized into a finite number of intervals [54]. A data point may be univariate or multivariate consisting of a single or multiple attributes, respectively. Multivariate data points may contain the same or a mix of attribute types.

Based on the relationship between data points, input data may be sequential, spatial, or graph data. Sequential data consist of linearly ordered data points such as time-series data. Spatial data contain information about the location of an instance with respect to its neighbors (vehicular traffic, geolocation). Graph data represent data points as vertices and their connections as edges [52].

The nature or type of anomalies is another important aspect of developing anomaly detection techniques. Anomalies may be categorized as point, contextual, or collective. Point anomalies consider a single data point as anomalous with respect to the other data points. Due to its simplicity, anomaly detection is mainly focused on point anomalies. Contextual (conditional) anomalies define a data point as anomalous with respect to its relationships with other data points (specific conditions) [55, 56, 57]. Detection of these anomalies is based on contextual (environmental or extrinsic) and behavioral (indicator or intrinsic) attributes. Detection of contextual anomalies is commonly employed for time-series data. Collective anomalies refer to a collection of data points that are anomalous with respect to other data points when they occur together. Anomalies in sequential, spatial, and graph data have been considered as collective. Collective anomalies only occur in datasets containing related data points while point anomalies may occur in any dataset [52]. In this study, we analyze collective anomalies that occur during the periods of ransomware attacks.

1.4.1 Anomaly Detection in Communication Networks

Anomaly detection in communication networks is referred to as intrusion detection. Its most frequent challenges are computation efficiency to handle large input data, online analysis for data streaming, and a high false alarm rate. Intrusion Detection Systems (IDSs) [7, 52, 58, 59] are used to identify and classify malicious activities and may be host-based or network-based. Host-based systems protect the host (endpoint) by monitoring operating system files and processes. Network-based systems monitor incoming network traffic (IP addresses, service ports, protocols) by analyzing flows of packets or by inspecting packet headers. Their role is to enhance security by identifying suspicious events in the observed network traffic. Detecting malicious network intrusions may be signature-based or anomaly-based. In cybersecurity, signatures are used to define patterns of malicious attacks on a host or network based on the bytes sequence of a file in the network traffic, unauthorized activities in a host or network (software execution, network or directory access), or unexpected use of network privileges [60]. Anomalies in network traffic are sudden deviations from an expected (regular) behavior that may be caused by network intrusions or by changes in the network elements (misconfigurations, infrastructure failures) [61]. Signature-based intrusion techniques [62] rely on known events that follow certain rules and patterns while anomaly-based intrusion detection techniques [63, 64] rely on detecting deviations from an expected behavior. Anomaly-based intrusion detection may use classification, clustering, statistical, or information theoretic techniques [52].

1.4.1.1 Classification

Classification-based anomaly detection techniques [52] consist of training phase and testing phases. Depending on the labels of the training data points, classification-based techniques may be one-way or multi-way. In one-way classification, a boundary is defined to encompass the regular data points while data points outside this boundary are classified as anomalous. Multi-way classification may be employed when data points are labeled based on multiple regular or anomalous classes.

The computational complexity of classification-based techniques heavily depends on the classification algorithms. The training phase requires longer time than the testing phase because a classifier needs to be first generated while the testing phase employs the derived classifier. Even though data classification using these techniques have a short testing time, their performance relies on the accuracy of available labels and do not have the capability to generate scores.

1.4.1.2 Clustering

Clustering-based anomaly detection techniques may be developed using unsupervised or semi-supervised machine learning [52]. Depending on the data input, anomalies may be detected based on: their location with respect to a cluster, their proximity to the nearest cluster centroid, or the size and sparsity of the cluster. When using their location, data points that do not belong to a cluster are labeled as anomalous. In the case of proximity, data points are first clustered and the anomaly score is then generated based on the distance to their nearest cluster centroid. The size and sparsity of a cluster are used to label data points as anomalous based on a defined threshold.

Computational complexity of clustering-based techniques is quadratic if the pairwise distances are calculated for all data points while heuristic algorithms offer linear complexity. Advantages of these techniques include adaptation to various data types and short testing times because the number of clusters is relatively small. Disadvantages of these techniques include: ineffectiveness of employed clustering algorithms, lack of optimization, forming small clusters with anomalous data points, and high computational complexity.

1.4.1.3 Statistics

Statistical anomaly detection techniques consider anomalies as observations that are not generated by the used statistical model and, hence, are partially or fully irrelevant [52, 33]. Therefore, these techniques assume that regular and anomalous data points are located in the high and low probability regions, respectively. Statistical models may be developed by using parametric or non-parametric techniques [65]. In parametric techniques, anomalies may be detected based on the score of a given data point $\boldsymbol{x}$ by calculating the inverse of a probability density function $f(\boldsymbol{x}, \Theta)$, where Θ is estimated from the data. Other parametric techniques may rely on the rejection of the null hypothesis H_0 in order to declare a data point $\boldsymbol{x}$ as anomalous. Gaussian and regression models as well as mixture of parametric distributions are examples of these techniques. In contrast, non-parametric techniques determine a statistical model based on the given data instead of assuming knowledge of the underlying distribution. Typically used non-parametric techniques are histograms and kernel function.

Computational complexity of statistical anomaly detection techniques depends on the statistical models. When employing exponential parametric distributions (Gaussian, Poisson, multimodal), computational complexity is usually linear with respect to the data size and number of attributes. If using distributions based on iterative estimation techniques (mixture models, hidden Markov models) linear computational complexity in each iteration may also be exhibited albeit with slow convergence [52]. In the case of kernel-based techniques, computational complexity may be quadratic with respect to the data size. Statistical techniques may provide feasible solutions as long as the assumed underlying data distributions hold true. However, assumed statistical distributions may not hold true and constructing hypothesis tests for some distributions may be nontrivial, especially for high-dimensional datasets. These techniques may also be employed for unsupervised anomaly detection if the estimation of the distribution proves robust. Generated scores may be used during the evaluation of a given data point because they are associated with confidence intervals.

1.4.1.4 Information Theory

Information theoretic anomaly detection techniques employ various measures such as Kolmogorov complexity, entropy, and relative entropy in order to analyze the information content of a dataset [52, 33]. They assume that irregularities in the information content are introduced by anomalies in the data. Basic information theoretic techniques have exponential time complexity while some proposed approximate techniques have linear time complexity. These techniques may be used for unsupervised anomaly detection and without assuming the underlying statistical distribution of the data. However, anomaly scores may be difficult to associate with a data point during testing. Performance of information theoretic techniques highly depends on the selected measure and the size of sequential and spatial datasets.

1.5 Related Work

Network Intrusion Detection Systems (NIDSs) rely on misuse, anomaly, or hybrid detection techniques [66, 62]. Misuse approaches compare network traffic to a set of rules or patterns while anomaly detection relies on deviations of traffic from regular behavior. Hybrid detection techniques combine the two to improve detection of known attacks while decreasing the high false positive rates for unknown deviations from a regular behavior. Machine learning algorithms are used for misuse detection by capturing properties of known attacks or to detect anomalies that exploit new vulnerabilities (also known as a zero-day attacks).

Various NIDSs [67, 68] have been proposed to address a dynamically changing landscape of cyber threats. They employ diverse deep learning algorithms [63, 66, 69, 70, 71] such as convolutional neural networks, RNNs [72], deep belief networks, multilayer perceptrons [73], autoencoders [74], and Stacked Non-symmetric Deep Auto-encoder (NDAE) [75] that demonstrated promising results for anomaly detection. An example is a neural network model with four hidden layers yielding high accuracy [76]. Conventional machine learning algorithms (J48, naïve Bayes, naïve Bayes Tree, Random Forests, Random Tree, Multi-Layer Perception, and SVM) have also been evaluated [77, 66, 78, 79]. Their performance when classifying anomalies and intrusions has been evaluated [80, 59] using datasets such as BGP [41, 81], NSL-KDD [82], and Canadian Institute for Cybersecurity Intrusion Detection System (CICIDS) datasets [83].

Algorithms that have a short training time are important for network intrusion detection systems in order to adequately prevent the onset of malicious attacks. This enables system administrators to effectively and timely remove affected network elements. Training time for deep neural networks may be reduced by selecting appropriate features and parameters while maintaining high accuracy [84]. Combination of supervised learning and feature selection algorithms is employed to develop novel intrusion detection solutions that reduce the high false alarm rate by classifying previously unobserved network traffic patterns [85]. Reported results demonstrate that the proposed anomaly-based IDS employing a neural network with a wrapper feature selection outperforms other models.

1.6 Research Contributions

The main contributions of this thesis are:

- **Development of a database engine to store BGP routing records that enables analyzing routing information:**

Our past analysis of BGP datasets [86] considers only changes in BGP *update* messages. However, publicly available BGP routing records include BGP *open*, *update*, *keepalive*, and *notification* messages. They also include the TCP connection state of a peer based on the BGP finite state machine (*Idle*, *Connect*, *Active*, *OpenSent*, *OpenConfirm*, *Established*). Including the remaining BGP messages and the TCP connection state of peers may improve anomaly detection using machine learning algorithms. Hence, we have developed a *BGP Relational Database (BGP-RDB)* to store BGP *open*, *update*, *notification*, and *keepalive* messages as well as changes of the TCP connection state captured from RIPE and Route Views data collection sites. Available data in Multi-threaded Routing Toolkit (MRT) format are first parsed to American Standard Code for Information Interchange (ASCII) and stored into Yet Another Markup Language (YAML) files using the *mrtparse* [87] Python module. An ingestion engine has also been developed based on the *sqlite3* [88] Python module to insert data into the database tables. The database has been developed with the goal to be implemented in real-time anomaly detection systems. The research contribution of developing a database engine to store BGP routing records consists of:

- Analyzing the protocol to derive a relational database model
- Defining relationships between tables based on primary and foreign keys
- Specifying the primary keys of lookup and detail tables using unique integer value based on specifications of the BGP messages
- Configuring primary keys of core and list tables to be unique integer values auto-generated when inserting new data entries
- Downloading BGP routing records from RIPE and Route Views collection sites using HTTP commands available in the *Requests* Python module
- Transforming BGP routing records in MRT format to ASCII and storing them in YAML files using the publicly available *mrtparser* tool
- Developing an ingestion engine to initialize the *BGP-RDB* database tables based on the *sqlite3* Python module
- Processing data contained in the generated YAML files and adding new data entries in the *BGP-RDB* database

• **Implementation and comparison of supervised and semi-supervised machine learning algorithms for detecting ransomware attacks using BGP routing records**:

Identifying BGP anomalies caused by ransomware attacks is of great interest in cybersecurity. We classify BGP anomalies caused by WannaCrypt (May 12, 2017) and WestRock (January 23, 2021) ransomware attacks. The WannaCrypt ransomware attack has been one of the most critical attacks because it targets hosts with the legacy operating system Microsoft Windows 7. In 2017, this ransomware attack impacted users from various industries worldwide, being the healthcare and manufacturing industries the most affected. The WestRock ransomware attack recently impacted a specific manufacturing company with several locations worldwide. In early 2021, this ransomware attack affected not only the targeted systems, but also impacted the company's supply chain and production levels. We generate datasets using publicly available BGP routing records during the periods of WannaCrypt and WestRock ransomware attacks from the RIPE [41] and Route Views [42] data collection sites. Data classification is performed using supervised and semi-supervised machine learning. Employed are deep learning and fast machine learning supervised algorithms: RNNs, Bi-RNNs, BLS and its extensions, eXtreme Gradient Boosting (XGBoost), Light Gradient Boosting Machine (LightGBM), and Categorical Boosting (CatBoost). In the case of semi-supervised machine learning, Isolation Forest (iForest) unsupervised machine learning algorithm is first employed during the labeling process to identify regular data within the anomalous periods (label refinement) while classification is performed using supervised algorithms: RNNs, BLS and its extensions, XGBoost, LightGBM, and CATBoost. We compare classification performance based on training time, accuracy, F-Score, precision, sensitivity, and confusion matrix. The research contribution of implementing and comparing supervised and semi-supervised machine learning algorithms for detecting ransomware attacks using BGP routing records consists of:

- Identifying the start and end time of the WannaCrypt and WestRock ransomware attacks
- Downloading BGP routing records from RIPE and Route Views collection sites that capture two days prior and two days after the attacks as well as the periods of the attacks
- Transforming the raw BGP data from MRT format to ASCII using the *zebra-dump-parser* tool
- Extracting 37 features using the *C#* tool to generate datasets based on transformed data
- Performing label refinement by implementing the iForest unsupervised machine learning algorithm to detect regular data points with the anomalous periods
- Implementing the deep learning and fast machine learning supervised algorithms
- Partitioning datasets into training and testing subsets
- Fine tuning parameters to generate machine learning models
- Evaluating classification performance of supervised and semi-supervised machine learning techniques based on training time, accuracy, F-Score, precision, sensitivity (recall), and confusion matrix

1.7 Research Publications

I have co-authored 1 journal and 7 conference publications. An additional paper is under review by the *IEEE Communications Magazine*. Listed are the publications and their abstracts. (Note: Conference publications [3] and [5] emanated from a separate research project.)

1.7.1 Journal Publications

[1] Z. Li, **A. L. Gonzalez Rios**, and Lj. Trajković, "Machine learning for detecting the WestRock ransomware attack using BGP routing records," *IEEE Commun. Mag.*, submitted for publication.

Border Gateway Protocol (BGP) enables the Internet data routing. BGP anomalies may affect the Internet connectivity and cause routing disconnections, route flaps, and oscillations. Hence, detection of anomalous BGP routing dynamics is a topic of great interest in cybersecurity. Various anomaly and intrusion detection approaches based on machine learning have been employed to analyze BGP *update* messages collected from Réseaux IP Eropeéns and Route Views. In this paper, we survey machine learning algorithms for detecting BGP anomalies and intrusions. Gradient boosting decision trees and deep learning algorithms are evaluated by creating models using collected datasets that contain ransomware events. Furthermore, a BGP anomaly detection tool *BGPGuard* has been developed to integrate various stages of the anomaly detection process.

[2] Z. Li, **A. L. Gonzalez Rios**, and Lj. Trajković, "Machine learning for detecting anomalies and intrusions in communication networks," *IEEE J. Sel. Areas Commun.*, vol. 39, no. 7, pp. 2254–2264, July 2021.

Cyber attacks are becoming more sophisticated and, hence, more difficult to detect. Using efficient and effective machine learning techniques to detect network anomalies and intrusions is an important aspect of cybersecurity. A variety of machine learning models have been employed to help detect malicious intentions of network users. In this paper, we evaluate performance of recurrent neural networks (Long Short-Term Memory and Gated Recurrent Unit) and Broad Learning System with its extensions to classify known network intrusions. We propose two BLS-based algorithms with and without incremental learning. The algorithms may be used to develop generalized models by using various subsets of input data and expanding the network structure. The models are trained and tested using Border Gateway Protocol routing records as well as network connection records from the NSL-KDD and Canadian Institute of Cybersecurity datasets. Performance of the models is evaluated based on selected features, accuracy, F-Score, and training time.

1.7.2 Conference Publications

[3] H. K. Takhar, **A. L. Gonzalez Rios**, and Lj. Trajković, “Comparison of virtual network embedding algorithms for data center networks,” in *Proc. IEEE Int. Symp. Circuit Syst.*, Austin, Texas, USA, May 2022, pp. 1660–1664 (virtual).

Software defined networks are a new Internet architecture paradigm that allows coexistence of heterogeneous network architectures. They optimize network management (maintenance, operability, effective content delivery) by provisioning a centralized network intelligence. Virtual Network Embedding (VNE) algorithms improve scalability and utilization of physical resources in Data Center Networks (DCNs). In this paper, we implement various DCN topologies and evaluate performance of VNE algorithms using the VNE-Sim simulator. We compare performance by implementing both server-centric and switch-centric DCN topologies.

[4] Z. Li, **A. L. Gonzalez Rios**, and Lj. Trajković, “Classifying denial of service attacks using fast machine learning algorithms,” in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Melbourne, Australia, Oct. 2021, pp. 1221–1226 (virtual).

Denial of service attacks are harmful cyberattacks that diminish Internet resources and services. Hence, detecting these cyberattacks is a topic of great interest in cybersecurity. Using traditional machine learning approaches in intrusion detection systems requires long training time and has high computational complexity. Thus, we evaluate performance of fast machine learning algorithms for training and generating models to detect denial of service attacks in communication networks. We use synthetically generated datasets that captured Transmission Control Protocol and User Datagram Protocol network flows in a controlled testbed laboratory environment. Evaluated algorithms include broad learning system and its extensions as well as XGBoost, LightGBM, and CatBoost gradient boosting decision tree algorithms. Experiments indicate that boosting algorithms often require shorter training time and have better performance.

[5] **A. L. Gonzalez Rios**, K. Bekshentayeva, Maheeppartap Singh, Soroush Haeri, and Lj. Trajković, “Virtual network embedding for switch-centric data center networks,” in *Proc. IEEE Int. Symp. Circuit Syst.*, Daegu, Korea, May 2021, pp 1–5 (virtual).

Advances in software defined and data center networks have enabled network virtualization. Virtual network embedding increases resources utilization and reduces cost of network deployment. Its performance depends on embedding algorithms and data center network topologies. In this paper, we evaluate performance of virtual network embedding algorithms based on acceptance ratio, revenue to cost ratio, and node and link utilizations by simulating virtual network embeddings on Spine-Leaf, Three-Tier, and Collapsed Core data center network topologies.

[6] Z. Li, **A. L. Gonzalez Rios**, and Lj. Trajković, "Detecting Internet worms, ransomware, and blackouts using recurrent neural networks," in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Toronto, Canada, Oct. 2020, pp. 2165–2172.

Analyzing and detecting Border Gateway Protocol (BGP) anomalies are topics of great interest in cybersecurity. Various anomaly detection approaches such as time series and historical-based analysis, statistical validation, reachability checks, and machine learning have been applied to BGP datasets. In this paper, we use BGP update messages collected from Réseaux IP Européens and Route Views to detect BGP anomalies caused by Slammer worm, WannaCrypt ransomware, and Moscow blackout by employing recurrent neural network machine learning algorithms.

[7] **A. L. Gonzalez Rios**, Z. Li, K. Bekshentayeva, and Lj. Trajković, "Detection of denial of service attacks in communication networks," in *Proc. IEEE Int. Symp. Circuits Syst.*, Seville, Spain, Oct. 2020, pp. 1–5 (virtual).

Detection of evolving cyber attacks is a challenging task for conventional network intrusion detection techniques. Various supervised machine learning algorithms have been implemented in network intrusion detection systems. However, traditional algorithms require long training time and have high computational complexity. Therefore, we propose detection of denial of service cyber attacks in communication networks by employing the broad learning system (BLS) that requires shorter training time while achieving comparable performance. Because designing effective detection systems relies on training and test datasets that contain anomalous network traffic data, in this paper we evaluate the performance of various BLS models by using recently generated network intrusion datasets. The best accuracy and F-Score were often achieved using BLS with cascades while BLS with incremental learning usually required shorter training time.

[8] Z. Li, **A. L. Gonzalez Rios**, G. Xu, and Lj. Trajković, "Machine learning techniques for classifying network anomalies and intrusions," in *Proc. IEEE Int. Symp. Circuits Syst.*, Sapporo, Japan, May 2019, pp. 1–5.

Using machine learning techniques to detect network intrusions is an important topic in cybersecurity. A variety of machine learning models have been designed to help detect malicious intentions of network users. We employ two deep learning recurrent neural networks with a variable number of hidden layers: Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU). We also evaluate the recently proposed Broad Learning System (BLS) and its extensions. The models are trained and tested using Border Gateway Protocol (BGP) datasets that contain routing records collected from Réseaux IP Européens (RIPE) and BCNET as well as the NLS-KDD dataset containing network connection records. The algorithms are compared based on accuracy and F-Score.

[9] **A. L. Gonzalez Rios**, Z. Li, G. Xu, A. Diaz Alonso, and Lj. Trajković, "Detecting network anomalies and intrusions in communication networks," in *Proc. 23rd IEEE Int. Conf. Intell. Eng. Syst. 2019*, Gödöllő, Hungary, April 2019, pp. 29–34.

Detecting anomalies and intrusions in communication networks is of great interest in cybersecurity. In this paper, we use Support Vector Machine (SVM) and Broad Learning System (BLS) supervised machine learning approaches to detect anomalies and intrusions in datasets collected from packet data networks. The developed models are trained and tested using data from the Internet routing tables, a simulated air force base network, and an experimental testbed. These datasets contain records of both anomalous and regular traffic data. We compare the two machine learning algorithms based on accuracy, F-Score, and training time.

1.8 Thesis Outline

In this thesis, the motivation and the importance of using supervised and semi-supervised machine learning algorithms to enhance detection of BGP anomalies were presented in Chapter 1. Machine learning approaches, principles of anomaly detection and techniques employed in communication networks applications, and survey of relevant related work were described. A summary of the research contributions and the list of publications are also provided.

Various BGP anomalies that affect the protocol performance are first introduced in Chapter 2. We also describe the RIPE and Route Views collection sites as well as their collection mechanisms. Included are the current data processing and a feature extraction method employed to generate BGP datasets.

The BGP messages and relevant information are described in Chapter 3. The *mrtparse* tool and generation of YAML files using raw data are introduced. The BGP database schema as well as the elements of each table, tables initialization, and data ingestion mechanism are also presented.

Ransomware attacks are defined in Chapter 4. We describe existing types of ransomware attacks, the mechanisms used for their execution, and some of the vulnerabilities they exploit. In particular, we provide details of WannaCrypt and WestRock, the two recent ransomware attacks that impacted users and systems located worldwide.

BLS and its extensions with and without incremental learning (RBF-BLS, CFBLS, CEBLS, CFEBLS), variable features BLS (VFBLS, VCFBLS), recurrent neural networks (LSTM, GRU), extremely randomized trees, gradient boosting decision trees (XGBoost, LightGBM, CatBoost), and isolation forest algorithms are introduced in Chapter 5.

Experimental procedure, performance metrics, experimental results, and performance comparison are presented in Chapter 6.

We conclude and describe the future work in Chapter 7.

Chapter 2

Border Gateway Protocol: Anomalies and Datasets

BGP routing data are used to analyze the Internet topology and hierarchy, infer AS relationships [89], and evaluate various intrusion and anomaly detection mechanisms [7, 90]. Data are collected using BGP trace collectors (RIPE [41] and Route Views [42]), route servers, looking glasses, and the Internet routing registries. Various BGP data collections may be combined to provide a more complete Internet topology [91]. The allocation and registration of unique numbers for ASes are managed by the Regional Internet Registries (RIRs): African Network Information Center (AFRINIC), American Registry for Internet Numbers (ARIN), Asia-Pacific Network Information Centre (APNIC), Latin America and Caribbean Network Information Centre (LACNIC), Réseaux IP Européens Network Coordination Center (RIPE NCC).

In this Chapter, BGP anomalies are first introduced followed by a description of RIPE and Route Views BGP data collection sites. Also presented are processing of BGP *update* messages and extraction of features employed to generate BGP datasets used for supervised anomaly detection.

2.1 BGP Anomalies

Network traffic anomalies are deviations from expected behavior [66, 7]. They affect BGP *update* messages and, thus, result in harmful changes in the protocol's dynamics that degrade the Internet performance and reliability. BGP anomalies may be caused by infrastructure failures, router misconfigurations, or network intrusions such as worms and ransomware attacks. Infrastructure (link) failures are caused by power outages or physical damage to network elements such as cables and routers. They cause reachability or connectivity loss between dedicated (private) connections or service provider (public) BGP peers. The Moscow power system blackout (2005) [22, 23], Mediterranean cable break (2008) [92], and Pakistani power outage (2021) [93, 24] resulted in BGP link failures that affected cities in over 20 countries.

BGP anomalies may consist of single updates (invalid AS numbers, invalid or reserved IP prefixes, prefix announced by an illegitimate AS, AS-PATH without a physical equivalent) or a set of updates (longest and shortest paths, behavioral changes in BGP traffic over time). Anomalies resulting from routers misconfigurations and hijacking attacks directly modify BGP routing configuration. Consequences of such anomalies include packet loss, unintended paths between routers, and forwarding loops. BGP routing origin and export missconfigurations [7] may cause announcements of used (hijacking) or unused (leaked routers) prefixes. Origin misconfigurations occur when non-owned prefixes are accidentally announced or private ASes are not filtered. Export misconfigurations appear when BGP policies are accidentally configured. In hijacking attacks, the attacker redirects routes from a valid AS by claiming the ownership of a prefix or sub-prefix. BGP hijacking attacks employ Denial of Service (DoS), Distributed Denial of Service (DDoS), man-in-the-middle, and phishing.

Network intrusions such as worms and ransomware attacks target Internet components and do not directly modify BGP routing configuration. They result in an increased number of prefix announcements, prefix withdrawals, implicit withdrawals, and changes in AS-Path length and packet size. Worms are self-replicating codes that exploit systems vulnerabilities and propagate through a network [94, 95]. They employ email applications or scan engines to spread to various hosts and may carry other malware as their payload. While antivirus systems may require several hours to identify worms, an Intrusion Detection System (IDS) is capable of detecting worms faster because they use large network bandwidths. Slammer [11], Nimda [13], and Code Red [15] are well-known worms that exploited vulnerabilities of Structured Query Language (SQL) and Internet Information Service (IIS). Ransomware attacks rely on advanced cryptography to lock the victim's data until a ransom is paid. They may be classified as: cryptoworm, Ransomware-as-a-Service (RaaS), and automated active adversary [16].

2.2 BGP Collection Sites: RIPE and Route Views

BGP routing information is collected from peers located in various geographical locations. BGP trace collectors receive BGP messages from their peers and periodically store the routing updates and tables into publicly available archives. Routing tables contain numerous entries from each peering AS indicating the preferred paths to destination prefixes at a given time. Routing messages indicate alternative paths and backup links. BGP *update* messages are available from global BGP monitoring systems such as RIPE [41] and Route Views [42]. They may be collected using Quagga [96], a suite derived from the Zebra [97] multi-server routing software. BGP messages are stored in MRT format by the BGP trace collectors [98].

2.2.1 RIPE

The Routing Information Service (RIS) [99] is a RIPE NCC project established in 2001 to collect and store routing data from ASes worldwide. The main collection site is located at NCC and consists of a route collector, database, and user interface. Remote Route Collectors (rrcs) installed at major topologically interesting Internet points use the Quagga routing software to collect BGP data. Routes are collected directly from the AS border routers at the rrc or from nearby routers via multi-hop BGP peering.

The raw data are collected using state dumps while batches of updates are periodically made available for each rrc. The Zebra [97] tool is used to collect BGP *update* messages every 15 minutes before July 23, 2003 and every 5 minutes after July 23, 2003. The BGP routing tables are stored every 8 hours. RIS currently consists of 25 rrcs: Europe (16), North America (4), Asia (2), South America (2), and Africa (1).

2.2.2 Route Views

Route Views [42] is the University of Oregon project to collect real-time BGP routing data from various backbone routers and locations worldwide. The publicly available data have been used for routing analysis, AS path visualization, analysis of IPv4 address space utilization, topological studies, and generation of geographic host locations. Backbone routers (Cisco, Juniper) are configured as IPv4 or IPv6 Route-Views-like route servers and connect as peers via multi-hop BGP sessions.

The Route Views project employs FRRouting, Quagga, and Cisco collectors. FRRouting and Quagga collectors are based on Zebra [97]. BGP messages and routing tables are stored in MRT format and are collected every 15 minutes and 2 hours, respectively. Data from Cisco collectors are generated every 2 hours starting at 00:00. Routes and their attributes are extracted using the Cisco command line interface. 31 Route Views collectors (16 FRRouting, 14 Quagga, and 1 Cisco) are distributed across 5 RIRs: ARIN (14), LACNIC (6), APNIC (5), AFRINIC (3), and RIPE NCC (3).

2.3 BGP Datasets: Data Processing and Feature Extraction

BGP datasets are extracted from BGP *update* messages downloaded from RIPE [41] and Route Views [42] collection sites. The data collected during periods of Internet anomalies include the days of the attack (anomalous data) as well as two days prior and two days after the attack (regular data). Employed collection sites are located near a considered anomalous event. Each day contains data extracted from the BGP *update* messages [35]. These messages are first transformed from MRT to ASCII format by using the *zebra-dump-parser* [100] tool written in Perl. GMT time is used for all BGP *update* messages in order to synchronize RIPE and Route Views collection times. A tool written in C# [101] is then used to generate datasets [102, 103] by extracting 37 continuous, categorical, and binary features classified as *AS-Path* and *volume* features, as listed in Table 2.1. Granularity of generated datasets is based on 1-minute intervals of routing records. Definitions of each feature are given in Table 2.2. The created datasets consist of collected data points and extracted features.

Table 2.1: List of features extracted from BGP *update* messages. The features have value types binary, categorical, or continuous and are categorized as *AS-Path* and *volume* features.

Feature	**Name**	**Type**	**Category**
1	Number of announcements	continuous	*volume*
2	Number of withdrawals	continuous	*volume*
3	Number of announced NLRI prefixes	continuous	*volume*
4	Number of withdrawn NLRI prefixes	continuous	*volume*
5	Average *AS-Path* length	categorical	*AS-Path*
6	Maximum *AS-Path* length	categorical	*AS-path*
7	Average unique *AS-Path* length	categorical	*AS-Path*
8	Number of duplicate announcements	continuous	*volume*
9	Number of implicit withdrawals	continuous	*volume*
10	Number of duplicate withdrawals	continuous	*volume*
11	Maximum edit distance	categorical	*AS-Path*
12	Arrival rate	continuous	*volume*
13	Average edit distance	categorical	*AS-Path*
14-23	Maximum *AS-Path* $= n$, $n = (11, ..., 20)$	binary	*AS-Path*
24-33	Maximum edit distance $= n$, $n = (7, ..., 16)$	binary	*AS-Path*
34	Number of Interior Gateway Protocol (IGP) packets	continuous	*volume*
35	Number of Exterior Gateway Protocol (EGP) packets	continuous	*volume*
36	Number of incomplete packets	continuous	*volume*
37	Packet size (B)	continuous	*volume*

Table 2.2: Definition of *AS-Path* and *volume* features extracted from BGP *update* messages.

Feature	**Name**	**Definition**
1	Number of announcements	Routes available for delivery of data
2	Number of withdrawals	Routes no longer reachable
3/4	Number of announced/withdrawn NLRI prefixes	BGP *update* messages that include announcement/withdrawal of routes
5/6/7	Average/maximum/average unique *AS-Path* length	Various *AS-Path* lengths
8/10	Number of duplicate include announcements/withdrawals	Duplicate BGP *update* messages that announcement/withdrawal of routes
9	Number of implicit withdrawals	BGP *update* messages that include announcement of routes with different *AS-Path* attribute for already announced NLRI prefixes
11/13	Maximum/average edit distance	Maximum/average of edit distances of messages per one-minute time interval
12	Arrival rate	Average number of BGP *update* messages arrived during one-minute time interval
14–23/24–33	Maximum *AS-Path* length/edit distance	Histograms with the most frequent values of maximum *AS-Path* length/edit distance
34/35/36	Number of IGP, EGP or, incomplete packets	BGP *update* messages generated by IGP, EGP, or unknown sources
37	Packet size	Average size of received BGP *update* messages in bytes

Chapter 3

Relational Database for Border Gateway Protocol: BGP-RDB

Routing information collected from BGP peers contains *open*, *update*, *notification*, and *keepalive* messages. Understanding information included in the fields of BGP message types is important not only to analyze changes in the protocol, but also to identify anomalies and intrusions. In this Chapter, an overview of these elements is given followed by a description of the developed *BGP Relational Database (BGP-RDB)*.

3.1 BGP Messages

The length of a BGP message varies between 19 and 4,096 octets, where the smallest length corresponds to a message containing only the BGP header [1]. Layout and fields lenght (bites) of the BGP header message are shown in Fig. 3.1.

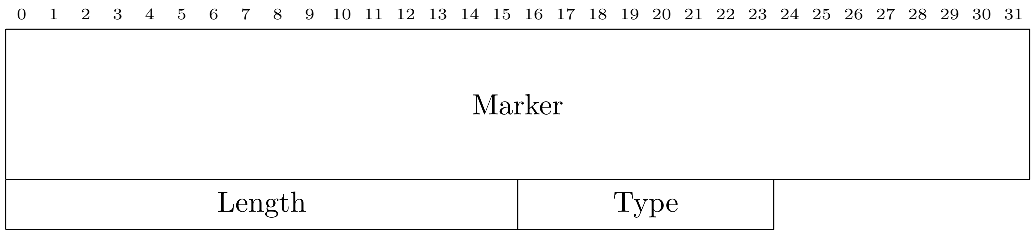


Figure 3.1: Fields in a BGP header: layout and lengths (bits) [1].

3.1.1 Open Messages

A BGP peer sends an *open* message that contains the BGP version, peer's AS number, hold time, BGP identifier, and optional parameters [1]. Its minimum length including the header is 29 octets. The current BGP version is 4 (BGP-4). An example of a BGP *open* message is illustrated in Fig. 3.2.

The AS Number (ASN) of a BGP peer is a unique 16-bit or 32-bit identifier allocated by a corresponding RIR: AFRINIC, ARIN, APNIC, LACNIC, RIPE NCC. The Internet

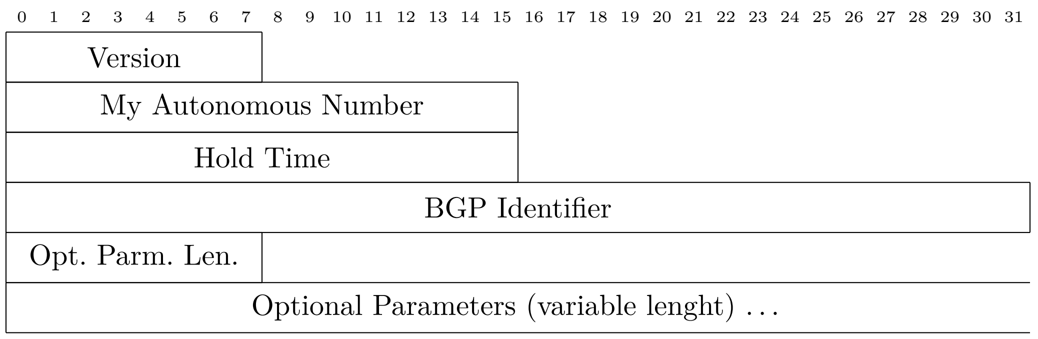


Figure 3.2: Layout of the *open* message: fields and lengths (bits) [1].

Assigned Numbers Authority (IANA) has reserved ASNs ranging between 64512 and 65534 (2-byte) or between 4200000000 and 4294967294 (4-byte) for privet use [104].

The hold time, proposed by the sender, indicates the value of the hold timer (seconds) used to confirm that BGP peers are functional and reachable. When the hold timer reaches zero after receiving successive BGP *keepalive* or *update* messages, the BGP connection is closed, routes from the corresponding BGP peer are removed, and a BGP *update* message indicating the withdrawn routes is sent.

The BGP identifier is a unique 32-bit value that corresponds to the IP address assigned to a BPG peer. This value may be manually defined or assigned based on the highest IP address available on a loopback or physical interface. A loopback interface is a logical (software) interface configured internally in a router to emulate a physical interface [105]. It is not assigned to a physical port and, hence, is always available on a functional router. Conversely, a physical interface corresponds to the port of a physical network connector such as Ethernet or serial. If a physical port is unavailable or fails, the physical interface becomes unresponsive. Therefore, loopback interface addresses are more stable and preferred over physical interface addresses [106].

Optional parameters are used to specify capabilities [107] supported by a BGP peer. Each parameter consists of a triplet indicating the capability type, length, and value as shown in Fig. 3.3. A BGP *notification* message with the error subcode "Unsupported Optional Parameter" is sent by a BGP peer if a capability is not supported.

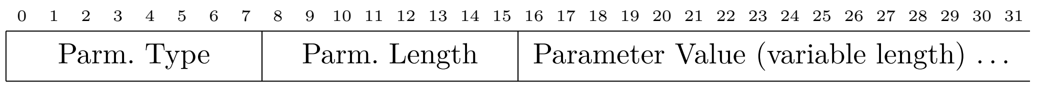


Figure 3.3: Layout of the optional parameters triplet: elements and their lengths (bits) [1].

3.1.2 Update Messages

Routing information such as advertisements or withdrawals is exchanged between BGP peers using BGP *update* messages. Relationships among ASes may be described by constructing a graph based on BGP *update* messages. These messages shall include: length of withdrawn routes, withdrawn routes, length of path attributes, path attributes, and Network Layer Reachability Information (NLRI) [1]. Fields are illustrated in Fig. 3.4. Advertisements of feasible routes and withdrawals of infeasible routes may be included within a single BGP *update* message. The minimum length of BGP *update* messages is 23 octets.

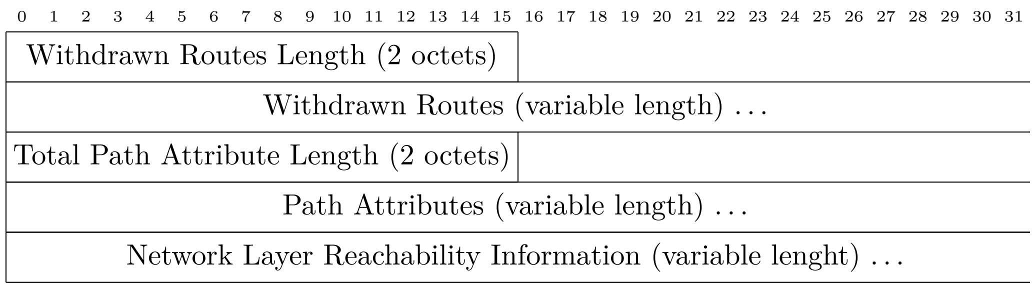

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Withdrawn Routes Length (2 octets)
Withdrawn Routes (variable length) ...
Total Path Attribute Length (2 octets)
Path Attributes (variable length) ...
Network Layer Reachability Information (variable lenght) ...

Figure 3.4: Layout of the *update* message: fields and their lengths [1].

The withdrawn routes length (a 2-octet unsigned integer) indicates the number of octets in the withdrawn routes field. The withdrawn routes field is a list of prefixes (IP addresses) to be withdrawn. Each withdrawn route is a 2-tuple containing the prefix length (bits) and address that includes trailing bits, as shown in Fig. 3.5. If no routes are withdrawn, the withdrawn routes length is 0 and no withdrawn routes are included in the message.

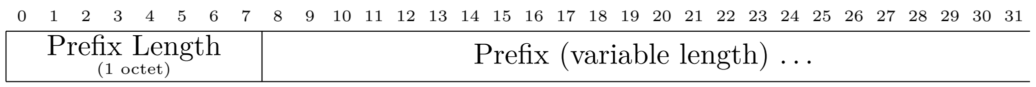


Figure 3.5: Withdrawn routes field: elements of the 2-tuple and their lengths (bits) [1].

The total path attributes length field (a 2-octet unsigned integer) contains the length of the path attributes field. If the length is 0, the BGP *update* message does not contain the NLRI and path attribute fields. The path attributes field (a 3-tuples of variable lengths) determines the type, length, and value of each attribute. The attribute type (a 2-octet field) consists of one attribute flag and one attribute type code octet, as shown in Fig 3.6. The high-order bit defines if the attribute is well-known (1) or optional (0). The second-high order bit determines if an optional attribute is non-transitive (0) or transitive (1). Well-known attributes shall have the transitive bit set to 1. The third high-order (*partial*) bit indicates if the optional transitive attribute is complete (0) or partial (1). This bit is set to 0 for both well-known and optional non-transitive attributes. The fourth high-order (*extended length*) bit is one octet (0) or two octets (1) in length . The lower-order four bits of the attribute flags octet are unused and have value 0.

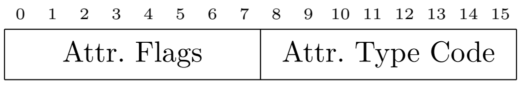


Figure 3.6: Attribute type: elements in the 2-octet and their lengths (bits) [1].

The NLRI field consists of 2-tuples indicating the prefix length (bits) and address, as indicated in Fig. 3.7. Its length is:

$$\mathit{Update\ message\ Length} - 23 - \mathit{Total\ Path\ Attributes\ Length} - \mathit{Withdrawn\ Routes\ Length}$$

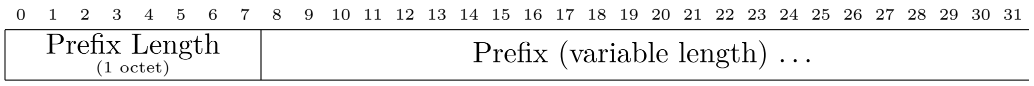


Figure 3.7: NLRI field: elements in each 2-tuple and their lengths (bits) [1].

3.1.2.1 Path Attributes: Flags and Codes

BGP path attributes are classified into four categories: well-known mandatory, well-known discretionary, optional transitive, and optional non-transitive [1]. Every BGP implementation recognizes all well-known attributes. All well-known mandatory attributes shall be included in every BGP *update* message that contains NLRI while well-known discretionary attributes may only be sent in a particular BGP *update* message. Unlike mandatory attributes, optional attributes are not supported by all BGP implementations. When a BGP *update* message contains unrecognized optional transitive attributes, BGP peers shall accept the path and pass it to other peers with the *partial* bit always set to 1. However, unrecognized optional non-transitive attributes shall be ignored and not passed to other BGP peers.

The supported attribute type codes are [1]: ORIGIN (type code 1), AS_PATH (type code 2), NEXT_HOP (type code 3), MULTI_EXIT_DISC (type code 4), LOCAL_PREF (type code 5), ATOMIC_AGGREGATE (type code 6), AGGREGATOR (type code 7). The origin of the path is defined by the well-known mandatory attribute ORIGIN and it indicates if NLRI is interior to the origin AS (value 0), learned via the EGP protocol (value 1), or from other means (value 2).

A sequence or a set of AS path segments creates the well-known mandatory attribute AS_PATH. Each segment is a triplet that contains the path segment type, path segment length, and path segment value. The segment types are the AS_SET with an unordered set of ASes (value 1) and the AS_SEQUENCE with an ordered set of ASes (value 2). The path segment length indicates the number of ASes in the path segment while the path segment value contains the AS numbers.

A BGP peer determines the outbound interface and immediate next-hop address based on information provided in the well-known mandatory attribute NEXT_HOP that defines the next IP address for forwarding packets in transit to their destination. Usually, the next hop is defined so that packets traverse the shortest path. A recursive route lookup is used to determine the immediate next-hop based on routing tables.

The optional non-transitive attribute MULTI_EXIT_DISC is used when there are multiple exit or entry points to a particular neighbouring AS. It consists of a 4-octet number (metric). A lower metric is the preferred exit point. BGP peers have mechanisms for removal of this attribute and may be configured to alter its value received over EBPG.

A BGP peer sends to internal peers the degree of preference for advertised routes using the well-known mandatory attribute LOCAL_PREF. This attribute shall not be included in BGP *update* messages sent to external peers unless defined by BGP confederations. If received in a BGP *update* message from an external BGP peer, this attribute shall be ignored except if defined by BPG confederations. A BGP confederation is a solution employed to overcome the scaling problem caused by the BGP fully mesh requirement [108]. This solution consists in dividing a large AS into sub-autonomous systems (sub-ASes) that are assigned a sub-AS number from the private AS numbers in the range 64,512 to 65,535 [109].

The well-known discretionary attribute ATOMIC_AGGREGATE should be included in a BGP *update* message if an AS_SET is no longer valid and some AS numbers in the AS_PATH are excluded from the aggregated route.

Updates formed by aggregation may include the optional transitive attribute AGGREGATOR. It contains the last AS number and IP address of the BPG peer that formed the aggregate route. The specified IP address should be the same as used for the BPG Identifier of the BPG peer.

3.1.3 Keepalive Messages

Reachability between peers is determined by periodically sending BGP *keepalive* messages so that the hold timer does not expire [1]. These messages are usually sent within one third of the hold time interval without exceeding one message per second. Repeated BGP *keepalive* messages shall not be sent when the hold time interval is 0. Only the header is included in these messages and, hence, their length is 19 octets.

3.1.4 Notification Messages

BGP *notification* messages, sent when an error is detected, are followed by the BGP connection termination [1]. These messages contain the error code, error subcode, and data (the reason for the notification), as shown in Fig. 3.8. Their minimum length is 21 octets including the message header.

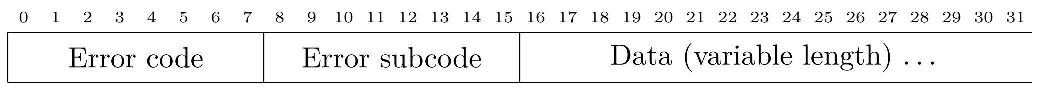


Figure 3.8: Layout of the *notification* message: fields and their lengths (bits) [1].

The six error codes indicate the type of notification: message header error, *open* message error, *update* message error, hold timer expired error, finite state machine error, and cease notification [1]. Message header errors may be caused by an unsynchronized connection, invalid message length, or invalid message type. *Open* message errors result from unsupported BGP version, invalid peer AS, invalid BGP identifier, unsupported optional parameter, and unacceptable hold time. Sources of *update* message errors include malformed attribute list, unrecognized well-known attribute, missing well-known attribute, attribute flags error, attribute length error, invalid origin attribute, invalid NEXT_HOP attribute, optional attribute error, invalid network field, and malformed AS_PATH.

3.1.5 BGP Finite State Machine

A BGP Finite State Machine (FSM) is used to describe the process of establishing a TCP connection between peers [110, 2, 1]. A BGP peer initializes a TCP connection unless configured to remain in the idle state or remain passive. In a BGP connection, the connecting (active) side is called outgoing while the listening (passive) side is referred to as incoming. The BGP connection happens when the outgoing side sends the first TCP Synchronize (SYN) packet and the incoming side sends the first SYN/Acknowledge (ACK). The six states of the BGP FSM and their transitions (events) are illustrated in Fig. 3.9.

In the initial state of the FSM (*Idle*), no BGP resources are allocated to a peer and all connections are refused. Upon a *ManualStart* or *AutomaticStart* event, BGP resources for peer connection and TCP connection with other peers are initiated. The *Connect* state occurs while waiting for the TCP connection to be completed. If the connection is successful, state transitions to *OpenSent* while an unsuccessful connection results in state transitioning to *Active*. In the *Active* state, a peer is awaiting and accepting TCP connections and transitions to *OpenSent* state upon a successful connection. After receiving a *ManualStop*, *AutomaticStop*, or *HoldTimer_Expires* event while in the *OpenSent* state, the FSM transitions back to the *Idle* state. In the case of a *TcpConnectionFails* event, the state transitions from *OpenSent* to *Active*. However, if a BGP *open* message is received, the state transitions to *OpenConfirm* and peer awaits for a BGP *keepalive* or *notification* message. After any of these messages is received, the state is updated to *Established*. While in the *Established* state, *update*, *notification*, and *keepalive* messages are exchanged until transition to the *Idle* state occurs due to an error or a *ManualStop*, *AutomaticStop*, or *HoldTimer_Expires* event.

3.2 BGP Relational Database: Data Processing and Model

Databases are files used for sorting data and are designed to rapidly and continuously insert and access large amounts of data [111]. Their primary data structures are tables, rows, and columns. Relational databases are employed to create links between multiple tables. Tables, rows, and columns are usually referred to as relation, tuple, and attribute, respectively. Links between tables are defined using logical, primary, and foreign keys:

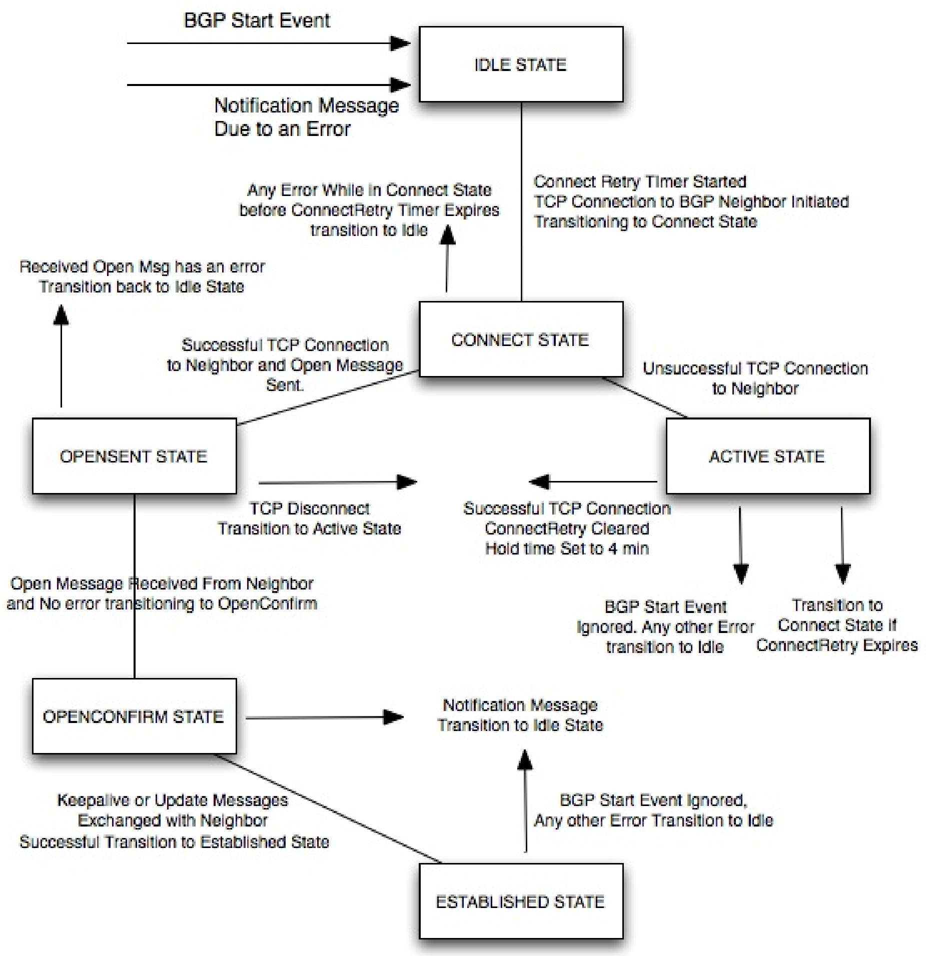

BGP Start Event
IDLE STATE
Notification Message Due to an Error
Any Error While in Connect State before ConnectRetry Timer Expires transition to Idle
Connect Retry Timer Started TCP Connection to BGP Neighbor Initiated Transitioning to Connect State
Received Open Msg has an error Transition back to Idle State
Successful TCP Connection to Neighbor and Open Message Sent.
CONNECT STATE
Unsuccessful TCP Connection to Neighbor
OPENSENT STATE
ACTIVE STATE
TCP Disconnect Transition to Active State
Successful TCP Connection ConnectRetry Cleared Hold time Set to 4 min
Open Message Received From Neighbor and No error transitioning to OpenConfirm
BGP Start Event Ignored. Any other Error transition to Idle
Transition to Connect State if ConnectRetry Expires
OPENCONFIRM STATE
Notification Message Transition to Idle State
Keepalive or Update Messages Exchanged with Neighbor Successful Transition to Established State
BGP Start Event Ignored, Any other Error Transition to Idle
ESTABLISHED STATE

Figure 3.9: BGP finite state machine: states and transition conditions [2].

- *Logical keys* are used to search for a specific row and are typically defined using unique strings.
- *Primary keys* consist of unique integer values that may be defined when creating a table or automatically assigned by the database when adding a new row.
- *Foreign keys* have number values that refer to a primary key of a row in a different table.

The primary and foreign keys are employed to generate queries that require joining multiple (two or more) tables. Data modeling is the process of defining how data are stored in tables and their relationships. The document containing such information is know as data model. Common database systems include Oracle [112], MySQL [113], Microsoft SQL Server [114], PostgreSQL [115], and SQLite [88]. In particular, SQLite system consists of a C-language library that does not require a dedicated server and may be embedded in applications to provide database support. The *sqlite3* module [116] is a Python database application program interface to integrate SQLite.

3.2.1 Data Processing

Publicly available BGP data from the RIPE and Route Views collection sites are downloaded using the *dataDownload.py* module listed in Appendix A. Raw BGP data in MRT format are converted to ASCII and stored in YAML files by using the *mrtparser* [87] Python module. The module includes an object class *Reader* used to first decode BGP headers in MRT format and then to parse the BGP message. Included is also the sample script *mrt2yaml.py* that has been modified to export converted MRT data to a YAML file, as listed in Appendix B. If extracting multiple BGP dumps to the same YAML file, the string *'- - -'* is inserted to indicate the start of a new set of BGP messages.

3.2.2 Data Model

The developed *BGP-RDB* consists of core (7), lookup (7), list (7), and detail (8) tables. An ingestion engine has been developed to include tables initialization and entries insertion based on data in the generated YAML files. Functions used by the ingestion engine and the implementation to store incoming data are listed in Appendix C and Appendix D, respectively. The database schema with connections between tables is shown in Fig. 3.10.

3.2.2.1 Core Tables

The core tables of the developed *BGP-RDB* capture information in BGP headers, messages, and path attributes. They consists of a field indicating the primary key of each inserted header, message, and path attribute; their timestamps; and their fields containing corresponding information. The primary key field is an integer datatype and its value is a unique auto-generated number. Timestamps are stored in both Unix (number) and date-time (yyyy-mm-dd hh:mm:ss) formats and, hence, data types used for these fields are integer and date-time, respectively. Fields containing header or message details may have text or integer data types. Fields with integer datatypes may contain foreign keys used to join lookup, list, or detail tables. Core tables are described in Appendix E.

3.2.2.2 Lookup Tables

References to types of collected MRT files, table dumps, BGP messages, and BGP categories are stored in lookup tables. They consist of a primary key (unique integer value) and related unique text value (type, subtype, category). These tables have one-to-many relationships with the core tables *bgp_headers*, *bgp_openmessates*, *bgp_updatemessages*, *bgp_notifiactionmessages*, and *bgp_keepalivemessages*. Fields included in lookup tables are given in Appendix F.

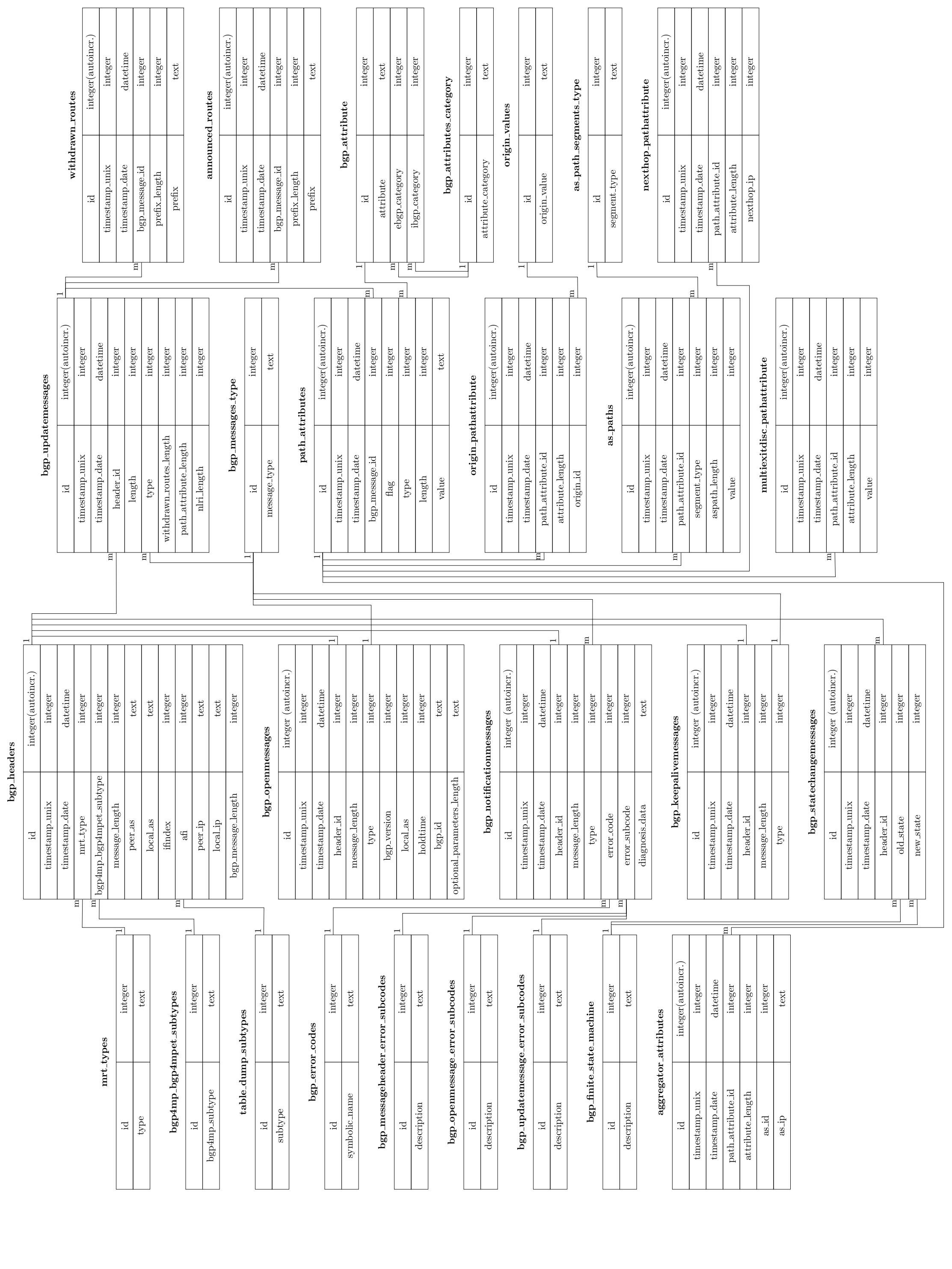


Figure 3.10: *BGP-RDA* Schema: Shown are tables, table fields and table connections indicating one-to-many (1–m) relationships.

3.2.2.3 List Tables

Information contained in BGP *update* messages are stored in list tables. Common elements included are the primary key (auto-generated unique integer value), the timestamp of related BGP *update* message in Unix and date-time formats, and specific fields in the received BGP *update* message (prefixes of withdrawn/announced routes, path attributes). Relationships between list and core tables are one-to-many. The list tables are described in Appendix G.

3.2.2.4 Detail Tables

The detail tables are used to store different types of BGP attributes, AS_Path segments, origin of an AS_Path, and *notification* messages error codes/subcodes. These tables consist of primary key (unique integer value) used to create one-to-many relationships with core tables and a logic key indicating the related attribute type, AS_Path origin, and message error code/subcode. Furthermore, table *bgp_attribute* includes two foreign keys to create one-to-many relationships with the lookup table *bgp_attributes_category*. Description of fields in the detail tables are listed in Appendix H.

Chapter 4

Ransomware Attacks

During ransomware, attackers employ advanced cryptography algorithms to lock victims' data until a ransom is paid. Types of ransomware attacks include Cryptoworm, Ransomware-as-a-Service (RaaS), and automated active adversary [16]. Cryptoworms replicate themselves to targeted hosts for maximum reach and impact. RaaS attacks, sold on the dark web as distribution kits, are typically deployed via malicious spam e-mails or exploit kits. In case of automated active adversary ransomware, the attackers scan the Internet for systems with weak protection, enter the system, and plan the attack for the maximum damage.

Ransomware attacks rely on tools and processes such as runtime packers and exploits. Runtime packers are compressed executable-files designed to avoid detection of attacks until they have completed their core task. Exploits (EternalBlue, Windows Event Viewer process, CVE-2018-8453) are tools used to ensure that the attacks gain administrative privileges by taking advantage of the vulnerabilities in an operating system. A ransomware stores the encrypted data on the used (overwrite) or available (copy) disk sectors. During the encryption, data are partially or fully renamed. Well-known ransomware attacks include WannaCrypt [16], Petya [117], and Locky [118]. In this study, we collect and analyze BGP *update* messages during the WannaCrypt and WestRock [17, 18] ransomware attacks from RIPE [41] and Route Views [42] collection sites.

4.1 WannaCrypt Ransomware Attack

WannaCrypt (WannaCry) is a cryptoworm ransomware that works by gaining administrative privileges and employs the EternalBlue exploit and DoublePulsar backdoor in systems running Microsoft Windows 7 [119, 16]. It lasted from May 12, 2017 to May 15, 2017 and infected over 230,000 computers in 150 countries. Cisco first observed requests for one of WannaCrypt killswitch domains on May 12, 2017 at 07:24 UTC [120]. A victim's data files are encrypted using 128-bit Advanced Encryption Standard (AES) in Cipher Block Chaining (CBC) mode. After the encryption is completed, data files are renamed by adding the extension ".wncry" while the string "WannaCrypt!" is added to the combination of encrypted AES key and data. Wannacrypt may copy or overwrite the data after encryption. It uses the Volume Shadow Service (vssadmin.exe) Windows utility to delete previous versions of the locked data. By manipulating the Windows Boot Configuration Data (bcdedi.exe), the attack:

- prevents Windows diagnostics-and-repair feature to automatically run after a third unsuccessful boot or
- attempts a normal boot even in case of a failed boot, shutdown, or checkpoint.

WannaCrypt flushes buffers to ensure that all encrypted data are only located in the storage drive. It replaces the Windows desktop wallpaper with a message to inform the victim that data have been locked and to demand a ransom. After the ransom is paid, the risk remains that decryption of data fails.

WannaCrypt spreads throughout a network by attempting to connect to TCP port 445. After the connection is established, the ransomware scans for the Windows Server Message Block (SMB) EternalBlue vulnerability and checks if it is already infected with the DoublePulsar backdoor. The DoublePulsar backdoor implant tool contains a code injection technique that employs an Asynchronous Procedure Call (APC) to execute code within a regular trusted process [121]. EternalBlue exploits the wrong casting, wrong parsing, and non-paged pool allocation defects of the SMB protocol as well as an address space Address Space Layout Randomization (ASLR) bypass. Exploiting the wrong casting and parsing defects causes buffer overflow and overwrite while the non-paged pool allocation and ASLR allow placing the machine code (shellcode) at a predefined executable address [122]. EternalBlue then implants the DoublePulsar backdoor in the victim's host to send the cryptoworm payload using Dynamic Link Library (DLL) injection [119, 122]. WannaCrypt replicates by querying for the non-existing domains:

- www[.]iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea[.]com
- www[.]ifferfsodp9ifjaposdfjhgosurijfaewrwergwea[.]com.

Its replication may be prevented if the victims receive a response indicating that these domains are registered [123].

4.2 WestRock Ransomware Attack

The WestRock Company [124] experienced a ransomware attack in late January 2021 [17, 18]. This is the second largest packaging company in USA that owns over 320 manufacturing facilities worldwide. The ransomware attack was detected on January 23, 2021. It impacted the company's Information Technology (IT) and Operational Techonology (OT) systems for over six days. While IT systems store, process, maintain, and operate data, OT systems monitor and control industrial processes, events, and devices. Hence, the attack caused delays in shipments of goods and production levels. The company implemented a controlled remediation plan that was executed in phases including systems shutdown and enhancement of existing security measures. In this study, we assume that the ransomware attack lasted between 1:12 UTC on January 23, 2021 and 23:59 UTC on January 29, 2021.

4.3 BGP Datasets: Data Collection and Visualization

We analyze BGP *update* messages captured from RIPE [41] and RouteViews [42] collection sites. Datasets are generated based on 1-minute intervals and 37 extracted features. Anomalous behavior is often indicated by large variations of BGP *volume* features such as number of BGP announcements, number of announced NLRI prefixes, and number of interior gateway protocol packets. We assume that regular data points (class 0) correspond to two days before and after the WannaCrypt and WestRock ransomware attacks. Data within the window of anomalous events are initially labeled as anomalies (class 1). However, the window with anomalous data points may also contain regular events. Hence, we use label refinement based on the iForest unsupervised learning algorithm to identify and label regular data points within the anomalous periods. It may help better identify anomalous data points and, thus, improve model performance. Majority of data points within each window retain the same original label (anomalies) after the label refinement using iForest algorithms. These regular data points are easier to identify using iForest because they are detected as outliers.

We collect eight days of BGP *update* messages during the WannaCrypt ransmoware attack: the days of the anomalous event as well as two days prior and two days after the attack. We select the RIPE collector rrc04 (located in Geneva, Switzerland) that contains 20 peer ASes [125] and the Route Views collector route-views2 (located in Oregon, USA) that has 77 peer ASes [126]. Generated datasets consist of 11,520 data points with the ransomware attack lasting 5,760 minutes. Examples of features that exhibit visible differences in patterns during regular and anomalous events for the WannaCrypt BGP datasets are shown in Fig. 4.1. Effect of several features extracted from RIPE and Route Views datasets is visualized in scatter plots shown in Fig. 4.2. The graphs indicate spatial separation for regular and anomalous classes. Separation into two distinct classes is more visible for WannaCrypt in the Route Views dataset. Better separation of spatial patterns usually leads to higher classification accuracy.

We collect 11 days of BGP *update* messages during the WestRock ransomware attack: the days of the attacks as well as two days prior and two days after the attacks. BGP *update* messages are downloaded from RIPE collector rrc14 (located in Palo Alto, CA, USA) consisting of 28 peer ASes [127] and Route Views collector telxatl (located in Atlanta, GA, USA) having 36 peer ASes [126]. Data collection sites are located near the consid-

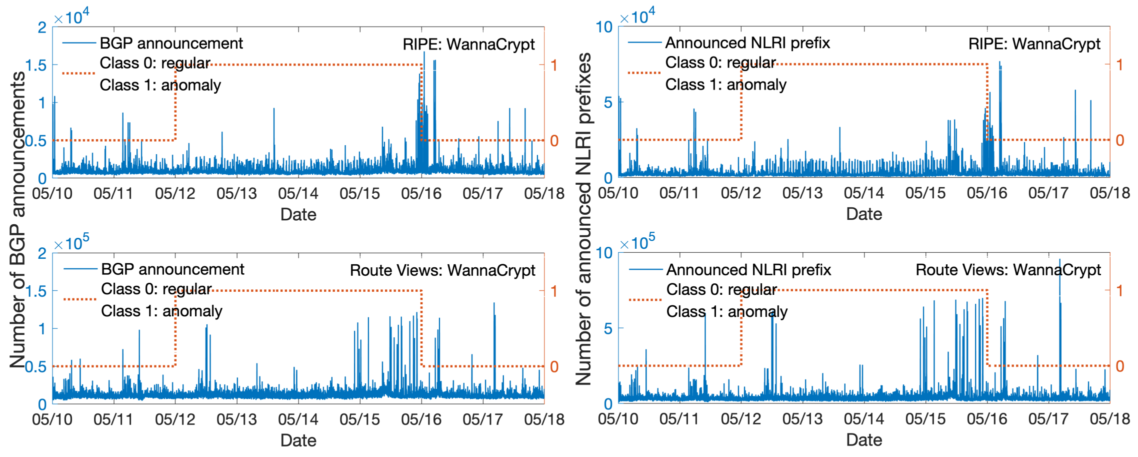


Figure 4.1: WannaCrypt ransomware attack RIPE (top) and Route Views (bottom): Number of BGP announcements (left) and announced NLRI prefixes (right).

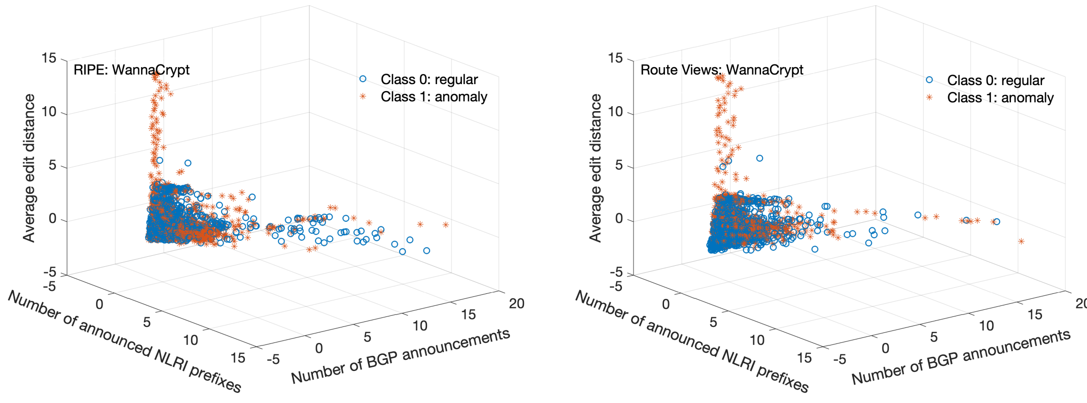


Figure 4.2: WannaCrypt ransomware attack RIPE (left) and Route Views (right): Number of announced NLRI prefixes vs. number of BGP announcements vs. average edit distance.

ered anomalous event. Generated datasets consist of 15,840 data points with ransomware attack lasting 10,008 minutes. Examples of features extracted from regular and anomalous events collected during the WestRock ransomware attack are shown in Fig. 4.3. The iForest algorithm is used for label refinement of regular data points within the windows of anomalous events. The patterns exhibit visible differences between regular and anomalous events for the WestRock ransomware dataset. Effect of several features extracted from RIPE and Route Views datasets is visualized in scatter plots shown in Fig. 4.4.

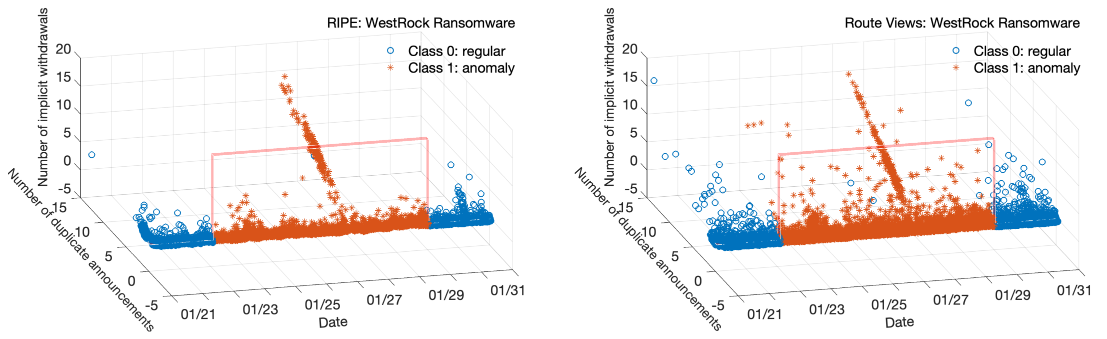


Figure 4.3: WestRock ransomware attack RIPE (left) and Route Views (right) data collection sites: Number of duplicate announcements vs. number of implicit withdrawals vs. date. Regular data points (class 0) are represented with circles while the anomalous data points (class 1) are represented with stars. Labels of data points within each anomaly window are refined using the iForest algorithm.

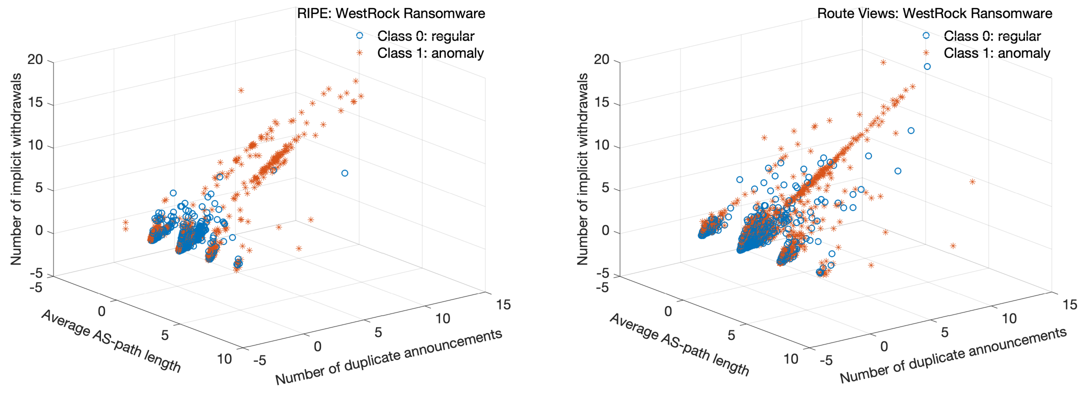


Figure 4.4: WestRock ransomware attack RIPE (left) and Route Views (right) data collection sites: Average unique AS-path vs. number of duplicate announcements vs. number of implicit withdrawals.

Chapter 5

Supervised and Semi-Supervised Machine Learning Algorithms

Supervised and semi-supervised machine learning techniques have been used to detect network anomalies. Most approaches rely on robust supervised learning algorithms, which train models using the given labels. Training and test of various supervised and semi-supervised machine learning algorithms is based on first generating a matrix $\boldsymbol{X}$ that contains N input data points (rows) and F features (columns). In the training phase, output matrix $\boldsymbol{Y}$ contains assigned labels: 0 (regular) and 1 (anomalous) data points.

5.1 Recurrent Neural Networks

RNNs are deep learning networks used to process sequential data $x^{(t)} = x(1), ..., x(\tau)$ having τ elements of fixed or variable lengths where t represents the index in the sequence [31]. These recurrent networks share parameters during the learning process: an output is a function of the previous time step (computation) and the same update rule applies to each element of the sequence. Recurrent connections may be present between hidden units or between the output and hidden units at consecutive time steps. RNNs are trained using mini-batches of the input data (sequence) where elements of the mini-batch may be of different lengths. They may be designed to produce an output at each time step or after processing an entire sequence. Bidirectional RNNs (Bi-RNNs) may be employed when the prediction depends on interpretation of past and future information. They consist of forward and backward layers that process data starting at the beginning and at the end of the sequence, respectively.

While traditional RNNs deal with sequential data, they do not effectively address long data sequences. Applying the same computation at every time step of such sequences leads to vanishing or exploding gradients thus making RNNs unable to learn long-term dependencies. The vanishing gradient problem occurs when the magnitude of eigenvalues of the weight matrix is smaller than 1 while the exploding gradient problem is caused by eigenvalues having magnitudes larger than 1. Approaches used to overcome the long-term dependency challenge in RNNs include: echo state networks, skipping connections through time, removing connections, and leaky and gated units. Long-term dependencies in echo state networks are addressed by learning only output weights while skipping connections through time introduces recurrent connections with time delays longer that one. RNN connections may be removed by replacing unit-length connections with longer connections. Leaky units are hidden RNN units that consist of linear self-connections with weight close to 1. Hence, learned past information is easily remembered over longer time steps. In contrast, gated units update connection weights at each time step thus allowing RNNs to discard old information when no longer significant. Long Short-Term Memory (LSTM) [128, 129] and Gated Recurrent Unit (GRU) [130] are gated RNNs that preserve memory by updating weights based on the previous and current time steps.

5.1.1 Long-Short Term Memory

LSTM networks are RNNs that are capable of learning long-term dependencies by connecting time intervals to form a continuous memory [128, 129]. Traditional RNN networks perform poorly when they need to bridge segments of information with long time gaps. Hence, LSTM networks were introduced to overcome long-term dependency and vanishing gradient problems. The LSTM cell shown in Fig. 5.1 is composed of: (a) *forget gate* f_t, (b) *input gate* i_t, and (c) *output gate* o_t.

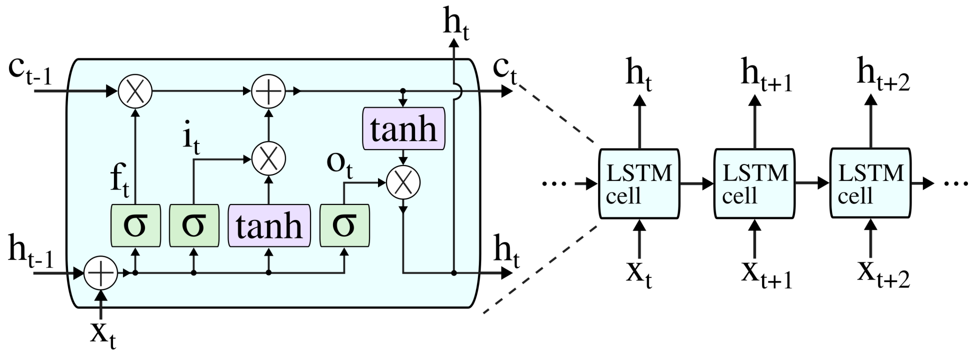


Figure 5.1: Repeating module for the LSTM neural network. States $c_t - 1$ and c_t are the previous and current cell states, respectively [3].

The *forget gate* discards irrelevant memories based on the cell state, the *input gate* controls the information that will be updated in the LSTM cell, and the *output gate* functions as a filter that controls the output. The logistic sigmoid σ and *tanh* are used as cell functions. The output of the LSTM cell is connected to the output layer and the next cell. The outputs of the *forget gate* f_t, *input gate* i_t, and *output gate* o_t at time t are [131]:

$$\begin{aligned} f_t &= \sigma(W_{if}x_t + b_{if} + U_{hf}h_{t-1} + b_{hf}) \\ i_t &= \sigma(W_{ii}x_t + b_{ii} + U_{hi}h_{t-1} + b_{hi}) \\ o_t &= \sigma(W_{io}x_t + b_{io} + U_{ho}h_{t-1} + b_{ho}), \end{aligned} \tag{5.1}$$

where $\sigma(\cdot)$ is the logistic sigmoid function, x_t is the current input, h_{t-1} is the previous output, W_{if}, U_{hf}, W_{ii}, U_{hi}, W_{io}, and U_{ho} are weight matrices, and b_{if}, b_{hf}, b_{ii}, b_{hi}, b_{io}, and b_{ho} are bias vectors. The information is stored in the cell state depending on the output i_t of the *input gate*. The sigmoid function is used to update the cell state c_t calculated as:

$$c_t = f_t * c_{t-1} + i_t * tanh(W_{ic}x_t + b_{ic} + U_{hc}h_{t-1} + b_{hc}), \tag{5.2}$$

where $*$ denotes element-wise multiplications and the *tanh* function is used to calculate the input to the next cell state. The output of the LSTM cell is:

$$h_t = o_t * tanh(c_t). \tag{5.3}$$

5.1.2 Gated Recurrent Unit

The GRU cell is derived from LSTM and has a simpler structure, as illustrated in Fig. 5.2. To make predictions, it employs gated mechanisms to control input and memory at the current timestep. While an LSTM cell consists of three gates, a GRU cell contains only *reset* r_t and *update* z_t gates [130]. The *reset gate* determines the combination of new input information and previous memory content while the *update gate* defines the content stored at the current timestep.

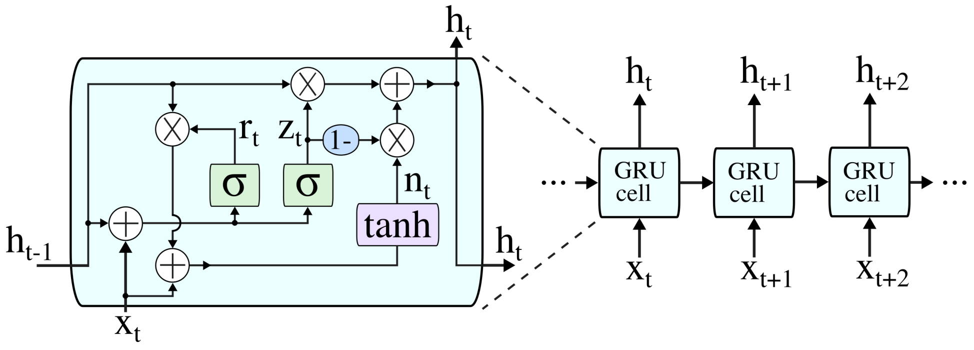


Figure 5.2: Repeating module for the GRU neural network [3].

The outputs of the *reset gate* r_t and the *update gate* z_t at time t are [131]:

$$\begin{aligned} r_t &= \sigma(W_{ir}x_t + b_{ir} + U_{hr}h_{t-1} + b_{hr}) \\ z_t &= \sigma(W_{iz}x_t + b_{iz} + U_{hz}h_{t-1} + b_{hz}), \end{aligned} \tag{5.4}$$

where $\sigma(\cdot)$ is the logistic sigmoid function, x_t is the input, h_{t-1} is the previous cell output, W_{ir}, U_{hr}, W_{iz}, and U_{hz} are the weight matrices, and b_{ir}, b_{hr}, b_{iz}, and b_{hz} are the bias vectors. The output of the GRU cell is:

$$h_t = (1 - z_t) * n_t + z_t * h_{t-1}, \tag{5.5}$$

where n_t is:

$$n_t = tanh(W_{in}x_t + b_{in} + r_t * (U_{hn}h_{t-1} + b_{hn})), \tag{5.6}$$

W_{in} and U_{hn} are the weight matrices, and b_{in} and b_{hn} are the bias vectors.

5.2 Broad Learning System

Broad learning system [4, 5] is based on the random vector functional-link neural network. It consists of a set of n mapped features ($\boldsymbol{Z}^n$) and m enhancement nodes ($\boldsymbol{H}^m$) that are concatenated in a single layer feed-forward neural network to form a broad instead of a deep network. Each group of mapped features $\boldsymbol{Z}_i$ consists of n_1 nodes generated by first multiplying matrix $\boldsymbol{X}$ by randomly generated weights $\boldsymbol{W}_{e_i}$ and including randomly generated bias $\boldsymbol{\beta}_{e_i}$. A mapping ϕ is then used to generate mapped features:

$$\boldsymbol{Z}_i = \phi(\boldsymbol{X}\boldsymbol{W}_{e_i} + \boldsymbol{\beta}_{e_i}), i = 1, 2, ..., n. \tag{5.7}$$

Matrix $\boldsymbol{Z}_i$ of dimension $N \times n_1$ corresponds to a single group of mapped features. The dimension of matrix $\boldsymbol{W}_{e_i}$ is $F \times n_1$. Matrix $\boldsymbol{Z}^n$ of dimension $N \times (n_1 \times n)$ is the concatenation of generated mapped features $\boldsymbol{Z}_i$. Each enhancement node $\boldsymbol{H}_j$ is created by first multiplying the concatenated groups of mapped features with random weights $\boldsymbol{W}_{h_j}$ and by adding random bias $\boldsymbol{\beta}_{h_j}$. The dimension of $\boldsymbol{W}_{h_j}$ is $(n_1 \times n) \times m$. A mapping ξ is then applied to generate enhancement nodes:

$$\boldsymbol{H}_j = \xi(\boldsymbol{Z}^n\boldsymbol{W}_{h_j} + \boldsymbol{\beta}_{h_j}), j = 1, 2, ..., m. \tag{5.8}$$

Mapped features and enhancement nodes are then concatenated to form the state matrix $\boldsymbol{A}_n^m$.

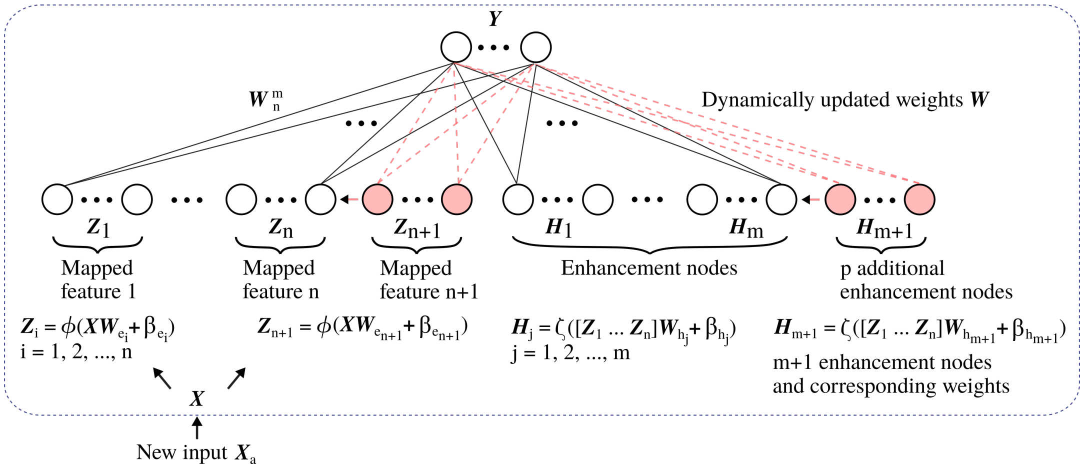
In the training phase, the Moore-Penrose pseudo-inverse or ridge regression is used to invert the state matrix and calculate the output weights $\boldsymbol{W}_n^m$ for the given labels $\boldsymbol{Y}$:

$$\boldsymbol{W}_n^m = (\lambda\boldsymbol{I} + (\boldsymbol{A}_n^m)^T\boldsymbol{A}_n^m)^{-1}(\boldsymbol{A}_n^m)^T\boldsymbol{Y}, \tag{5.9}$$

where λ is the regularization coefficient and $\boldsymbol{I}$ is the identity matrix whose dimension is $(n_1 \times n) + m$. During testing, the predicted labels are calculated using the output weights.

BLS extensions leading to improved performance include incremental learning [4], Radial Basis Function Network BLS (RBF-BLS) [132], Cascades of Mapped Features BLS (CF-BLS), Cascades of Enhancement Nodes BLS (CEBLS), and Cascades of Mapped Features and Enhancement Nodes BLS (CFEBLS) [5].

Incremental BLS algorithms, illustrated in Fig. 5.3, allow increments of input data, mapped features, and/or enhancement nodes and, thus, enable dynamical updates of BLS models. In case of incremental input data, additional data points are used to recalculate mapped features and enhancement nodes. The calculation is done only for the additional data points and there is no need to include all previously considered data points. One may also create additional mapped features by increasing their number and recalculating/updating the weights of enhancement nodes. A similar approach is applied when creating new enhancement nodes based on the existing mapped features.



$\boldsymbol{Y}$
$\boldsymbol{W}_n^m$
Dynamically updated weights $\boldsymbol{W}$
Z_1
Z_n
Z_{n+1}
H_1
H_m
H_{m+1}
Mapped feature 1
Mapped feature n
Mapped feature n+1
Enhancement nodes
p additional enhancement nodes
$\boldsymbol{Z}_i = \phi(\boldsymbol{X}\boldsymbol{W}_{e_i} + \beta_{e_i})$
i = 1, 2, ..., n
$\boldsymbol{Z}_{n+1} = \phi(\boldsymbol{X}\boldsymbol{W}_{e_{n+1}} + \beta_{e_{n+1}})$
$\boldsymbol{H}_j = \zeta([\boldsymbol{Z}_1 \dots \boldsymbol{Z}_n]\boldsymbol{W}_{h_j} + \beta_{h_j})$
j = 1, 2, ..., m
$\boldsymbol{H}_{m+1} = \zeta([\boldsymbol{Z}_1 \dots \boldsymbol{Z}_n]\boldsymbol{W}_{h_{m+1}} + \beta_{h_{m+1}})$
m+1 enhancement nodes and corresponding weights
$\boldsymbol{X}$
New input $\boldsymbol{X}_a$

Figure 5.3: Module of the BLS algorithm with increments of mapped features, enhancement nodes, and new input data [4].

The RBF-BLS extension employs the Gaussian rather than *tanh* function as the enhancement mapping ξ:

$$\xi(x) = \exp\left(-\frac{\| x - c \|^2}{\gamma^2}\right), \tag{5.10}$$

where c is the central point and γ determines the width of the Gaussian distribution. If the input data are located in a narrow region around the central point c, the RBF generates significant non-zero responses. An RBF network with k hidden nodes relies on calculating pseudoinverse to perform rapid training. The weight vectors of the output $\boldsymbol{HW}$ are deduced from:

$$\begin{aligned} \boldsymbol{W} &= (\boldsymbol{H}^T\boldsymbol{H})^{-1}\boldsymbol{H}^T\boldsymbol{Y} \\ &= \boldsymbol{H}^+\boldsymbol{Y}, \end{aligned} \tag{5.11}$$

where $\boldsymbol{W} = [w_1, w_2, ..., w_k]$ and $\boldsymbol{H} = [\xi_1, \xi_2, ..., \xi_k]$ are matrices of output weights and hidden nodes, respectively while $\boldsymbol{H}^+$ is the pseudoinverse of $\boldsymbol{H}$.

The cascades of mapped features and enhancement nodes add depth to the original architecture of BLS and may improve its performance. The connections within and between the mapped features and enhancement nodes define the structure of BLS. In the case of CFBLS, the first group of mapped features is based on input data and weights (5.7) while subsequent groups (k) of mapped features are created by using the previous group ($k-1$). The groups of mapped features are formulated as:

$$\begin{aligned} \boldsymbol{Z}_k &= \phi(\boldsymbol{Z}_{k-1}\boldsymbol{W}_{e_k} + \boldsymbol{\beta}_{e_k}) \\ &\triangleq \phi^k(\boldsymbol{X}; \{\boldsymbol{W}_{e_i}, \boldsymbol{\beta}_{e_i}\}_{i=1}^{k}), \text{for } k = 1, ..., n. \end{aligned} \tag{5.12}$$

The cascades of these groups $\boldsymbol{Z}^n \triangleq [\boldsymbol{Z}_1, ..., \boldsymbol{Z}_n]$ are used to generate the enhancement nodes $\{\boldsymbol{H}_j\}_{j=1}^{m}$. The first CEBLS enhancement node is generated from mapped features while subsequent nodes are generated from previous nodes creating a cascade:

$$\begin{aligned} \boldsymbol{H}_u &= \xi(\boldsymbol{H}_{u-1}\boldsymbol{W}_{e_u} + \boldsymbol{\beta}_{e_u}) \\ &\triangleq \xi^u(\boldsymbol{Z}^n; \{\boldsymbol{W}_{h_i}, \boldsymbol{\beta}_{h_i}\}_{i=1}^{u}), \text{for } u = 1, ..., m, \end{aligned} \tag{5.13}$$

where $\boldsymbol{W}_{h_i}$ and $\boldsymbol{\beta}_{h_i}$ are randomly generated. The CFBLS and CEBLS architectures are shown Fig. 5.4. The CFEBLS architecture is a combination of the two cascading approaches. The structure of incremental CFEBLS is shown in Fig. 5.5.

Based on its broad hidden layer and the use of pseudo-inverse or ridge regression, BLS offers shorter training time with comparable performance to deep learning networks such as multilayer perceptron, deep belief networks, deep Boltzmann machines, and convolutional neural networks [4]. Studies also reported comparable performance when applying BLS to data collected from communication networks [133, 134, 135, 136, 6].

5.3 Variable Features Broad Learning System

Mapping the input data to sets of mapped features is an essential step of BLS algorithms. We recently proposed Variable Features BLS (VFBLS) and Variable Features with Cascades BLS (VCFBLS) algorithms [6]. The architecture of VFBLS and VCFBLS consists of a variable number of mapped features and groups of mapped features as shown in Fig. 5.6. The algorithms employ feature selection that enable models to be trained based on a variable number of features extracted from the input data. The two algorithms also offer variants with incremental learning.

The VFBLS and VCFBLS algorithms expand the BLS network by using both original input data and subsets of input data as well as sets of groups of mapped features. They enable developing more generalized models and, hence, prevent over-fitting and enhance

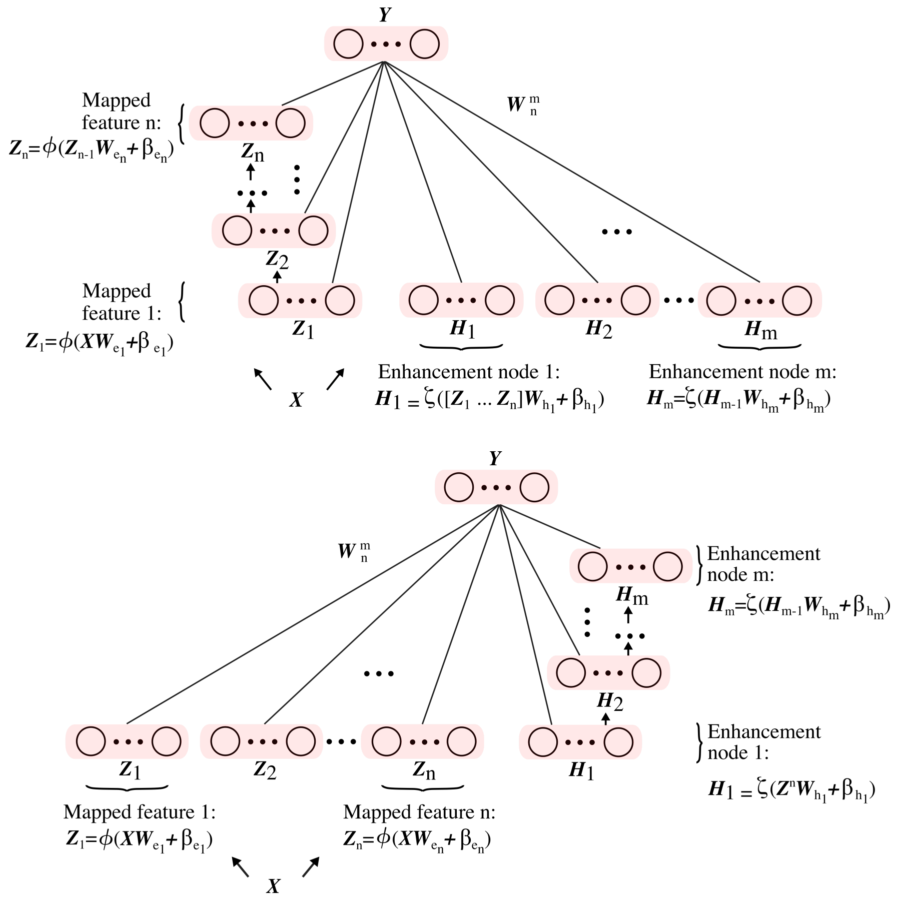

Y
Mapped feature n:
$Z_n=\phi(Z_{n-1}W_{e_n}+\beta_{e_n})$
W_n^m
Z_n
Z_2
Mapped feature 1:
$Z_1=\phi(XW_{e_1}+\beta_{e_1})$
Z_1
H_1
H_2
H_m
Enhancement node 1:
$H_1=\zeta([Z_1 \ldots Z_n]W_{h_1}+\beta_{h_1})$
Enhancement node m:
$H_m=\zeta(H_{m-1}W_{h_m}+\beta_{h_m})$
X
Y
W_n^m
Enhancement node m:
H_m
$H_m=\zeta(H_{m-1}W_{h_m}+\beta_{h_m})$
H_2
Enhancement node 1:
$H_1=\zeta(Z^nW_{h_1}+\beta_{h_1})$
Z_1
Z_2
Z_n
H_1
Mapped feature 1:
$Z_1=\phi(XW_{e_1}+\beta_{e_1})$
Mapped feature n:
$Z_n=\phi(XW_{e_n}+\beta_{e_n})$
X

Figure 5.4: Modules of the CFBLS (top) and CEBLS (bottom) algorithms. Shown are cascades of mapped features (top) and enhancement nodes (bottom) without incremental learning [5].

their performance. Generating the best BLS and incremental BLS models is rather time-consuming because they rely on multiple two-stage experiments: selecting features and generating models. In contrast, VFBLS and VCFBLS models are developed using a single experiment with integrated stages. A variable number of mapped features is used to reduce training time because the proposed algorithms introduce additional complexity by using the entire input dataset and by incorporating a feature selection algorithm and additional sets of mapped features. In case of incremental learning, features are selected in each step and ranked based on their importance. After the last step, all selected features are multiplied with weights proportional to the size of the dataset used in each step. They are then ranked and used for testing.

Data $\boldsymbol{X}$ and features that are selected based on a feature selection algorithm are used as input:

$$\boldsymbol{X}_v = \mathcal{F}(\boldsymbol{X}), v = 1, 2, ..., f, \tag{5.14}$$

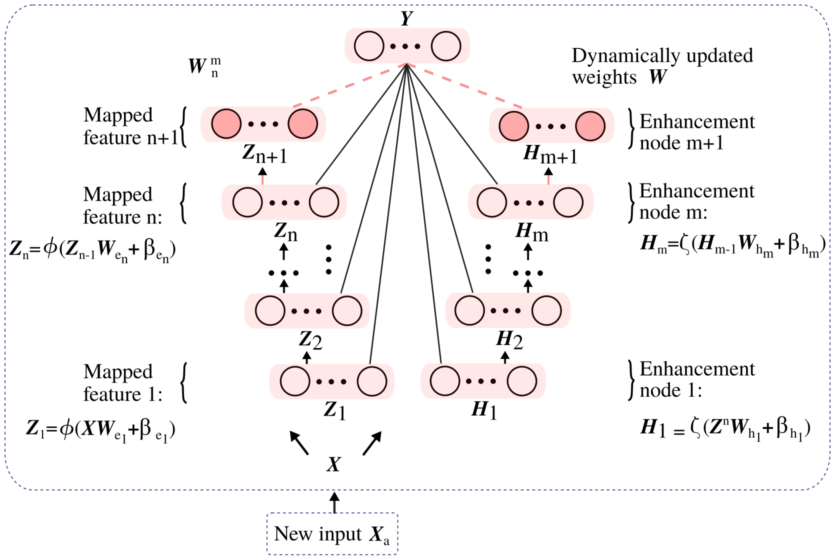


Figure 5.5: Module of the incremental CFEBLS algorithm with input data $\boldsymbol{X}$, cascades of mapped features, and cascades of enhancement nodes as well as increments of mapped features, enhancement nodes, and new input data [5].

where $\boldsymbol{X}_v$ is a subset of $\boldsymbol{X}$ with a selected set of features. Note that selecting all features in the input data to generate mapped features is a special case where $\boldsymbol{X}_v = \boldsymbol{X}$. Sets of groups of mapped features are generated as:

$$\boldsymbol{Z}^{n_v} \triangleq [\boldsymbol{Z}^{n_1}, ..., \boldsymbol{Z}^{n_f}], \tag{5.15}$$

where $\boldsymbol{Z}^{n_v}$ contains n_v mapped features. In the case of VFBLS groups of mapped features $\boldsymbol{Z}^{n_v}$, $\boldsymbol{X}_v$ and n_v correspond to $\boldsymbol{X}$ and n, respectively (5.7) while VCFBLS groups of mapped features $\boldsymbol{Z}^{n_v}$ are defined based on the previous group (5.12). Note that the first mapped feature $\boldsymbol{Z}_1$ in each set is created from $\boldsymbol{X}_1$, ..., $\boldsymbol{X}_f$. The number of mapped features and groups of mapped features may vary in sets $\boldsymbol{Z}^{n_1}$, ..., $\boldsymbol{Z}^{n_f}$. The number of mapped features in each group of a set is constant. We improve performance by including properties of the original data and concatenating input data $\boldsymbol{X}$ and sets of mapped features to create $\boldsymbol{Z}^t$ similar to the case of random vector functional-link network [137]:

$$\boldsymbol{Z}^t = [\boldsymbol{X}|\boldsymbol{Z}^{n_v}]. \tag{5.16}$$

The enhancement nodes are:

$$\boldsymbol{H}_j = \xi(\boldsymbol{Z}^t\boldsymbol{W}_{h_j} + \boldsymbol{\beta}_{h_j}), j = 1, 2, ..., m. \tag{5.17}$$

The state matrix $\boldsymbol{A}_t^m$ is the concatenation of $\boldsymbol{Z}^t$ and $\boldsymbol{H}^m$. The ridge regression algorithm is then employed to compute the weights $\boldsymbol{W}_t^m$ based on $\boldsymbol{A}_t^m$ and given labels $\boldsymbol{Y}$. The error

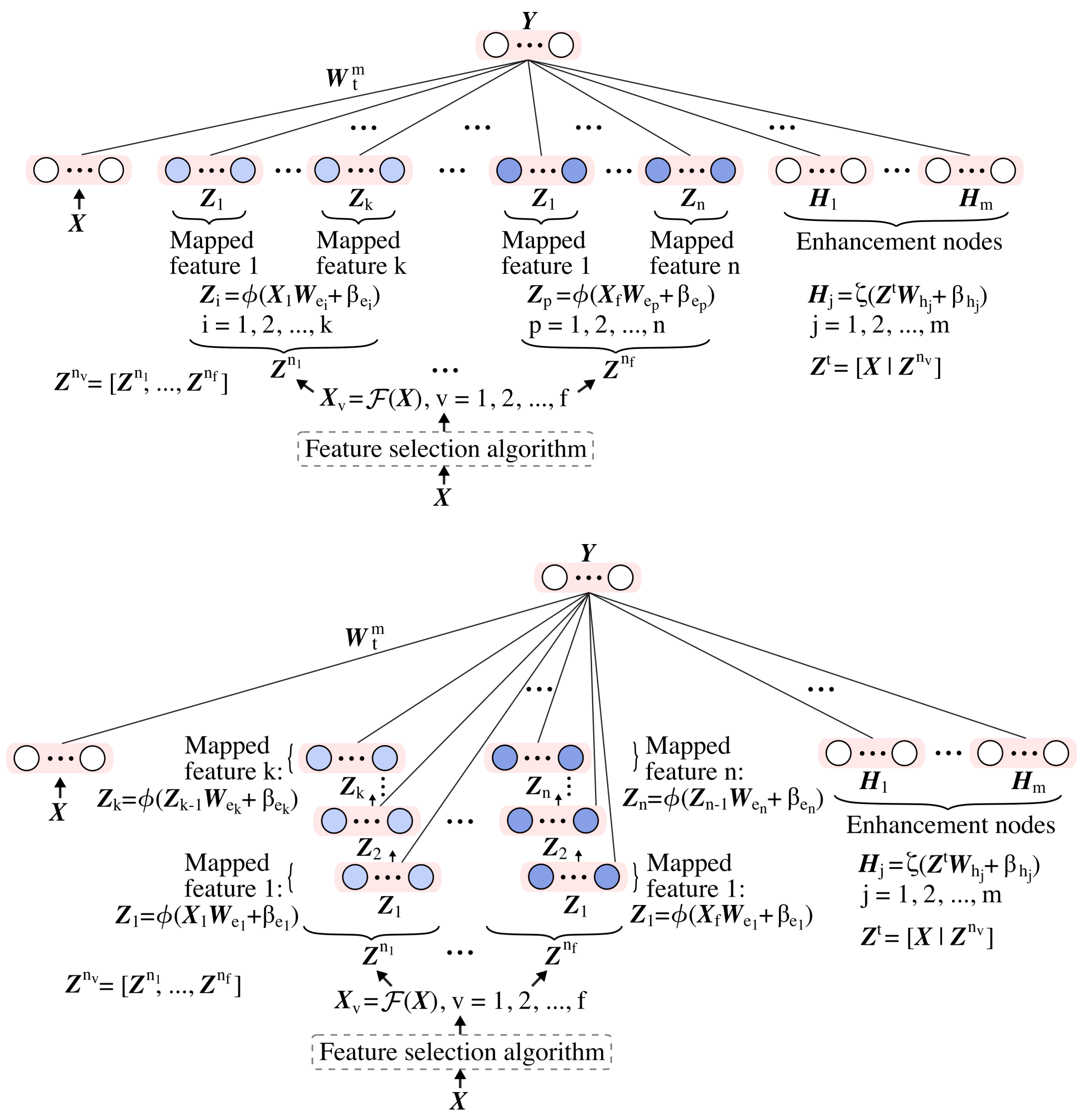


Figure 5.6: Modules of the VFBLS and VCFBLS algorithms with input data $\boldsymbol{X}$, sets of groups of mapped features with and without cascades ($\boldsymbol{Z}^{n_1}$, ..., $\boldsymbol{Z}^{n_f}$), and enhancement nodes ($\boldsymbol{H}_1$, ..., $\boldsymbol{H}_m$) [6].

function, minimized during the training process, is defined as [4]:

$$\boldsymbol{E}(\boldsymbol{W}_t^m) = ||\boldsymbol{A}_t^m \boldsymbol{W}_t^m - \boldsymbol{Y}||_2^2 + \lambda ||\boldsymbol{W}_t^m||_2^2, \tag{5.18}$$

where λ is the sparse regularization coefficient. The term $\lambda||\boldsymbol{W}_t^m||_2^2$ is used to control overfitting. The minimum of (5.18) can be found in closed-form by setting its derivative with respect to $\boldsymbol{W}_t^m$ to 0. The output weights are defined as [30]:

$$\boldsymbol{W}_t^m = (\lambda \boldsymbol{I} + (\boldsymbol{A}_t^m)^T \boldsymbol{A}_t^m)^{-1} (\boldsymbol{A}_t^m)^T \boldsymbol{Y}. \tag{5.19}$$

The incremental versions of VFBLS and VCFBLS are illustrated in Fig. 5.7 and Fig. 5.8. Pseudocodes for the VFBLS and VCFBLS algorithms and their incremental versions are listed in Appendix I.

While a variety of feature selection algorithms may be employed, in our experiments we use Extremely Randomized Trees (Extra-Trees) [138] to rank features based on importance.

5.4 Extremely Randomized Trees

Extremely randomized trees (Extra-Trees) algorithm is an improved version of the decision tree and random forests used to select relevant features for generating subsets of the input data. The extra-trees algorithm is faster than random forests because it randomly set the threshold for splitting nodes. Furthermore, it introduces additional randomness by randomly splitting nodes in order to avoid over-fitting. Features with higher importance are more relevant for a given dataset and better capture its properties. They may have better spatial separation and, thus, enhance the model's performance. The Gini importance is used to compute feature scores in a given dataset [139]:

$$Importance(\boldsymbol{X}_c) = \frac{1}{N_T} \sum_{T} \sum_{t \in T : v(s_t) = \boldsymbol{X}_c} p(t) \Delta i(s_t, t), \tag{5.20}$$

where $\boldsymbol{X}_c$ is the subset of $\boldsymbol{X}$ corresponding to one feature, N_T is the number of trees, t is the index of a node in a tree, s_t is the direction of the split, $v(s_t)$ is a randomly generated threshold, $p(t)$ is the weight, and $\Delta i(s_t, t)$ is the decrease of the node impurity equivalent to its importance.

5.5 Gradient Boosting Decision Trees

Boosting algorithms, a class of ensemble learning, are greedy algorithms that sequentially include estimators (base learners) to enhance the model performance [32]. Their goal is to minimize the loss function by including estimators that are trained based on residuals. Residuals (the difference between the target and predicted values) are calculated in each iteration and are used as the target values in the next iteration. The forward stage-wise additive modeling is used to generate boosting models. The number of training iterations is equivalent to the number of estimators because a new estimator is added to the boosting model in each iteration. Boosting models employ loss functions such as squared error, absolute error, exponential loss, or log-loss.

The Gradient Boosting Machines (GBMs) [140] are boosting algorithms that employ functional gradient descent to minimize the loss function. The Gradient Boosting Decision Trees (GBDT) algorithm is a GBM variant that employs decision trees as estimators. Optimized GBDT algorithms include XGBoost [38], LightGBM [39], and CatBoost [40].

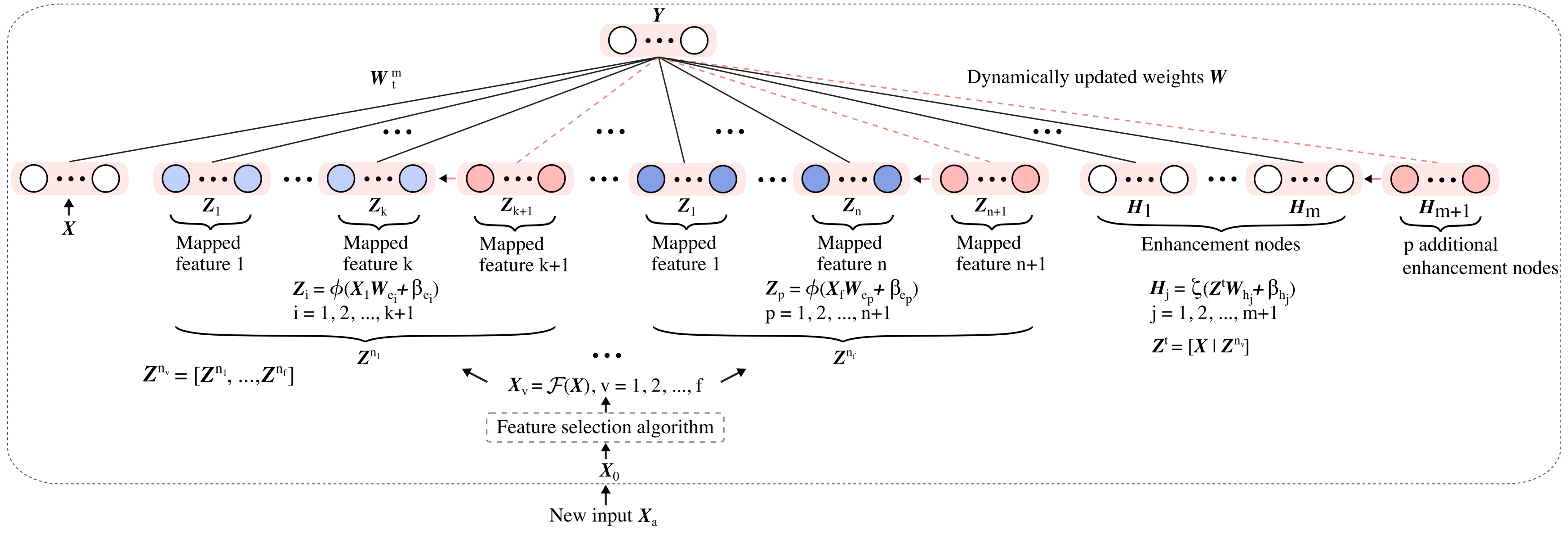
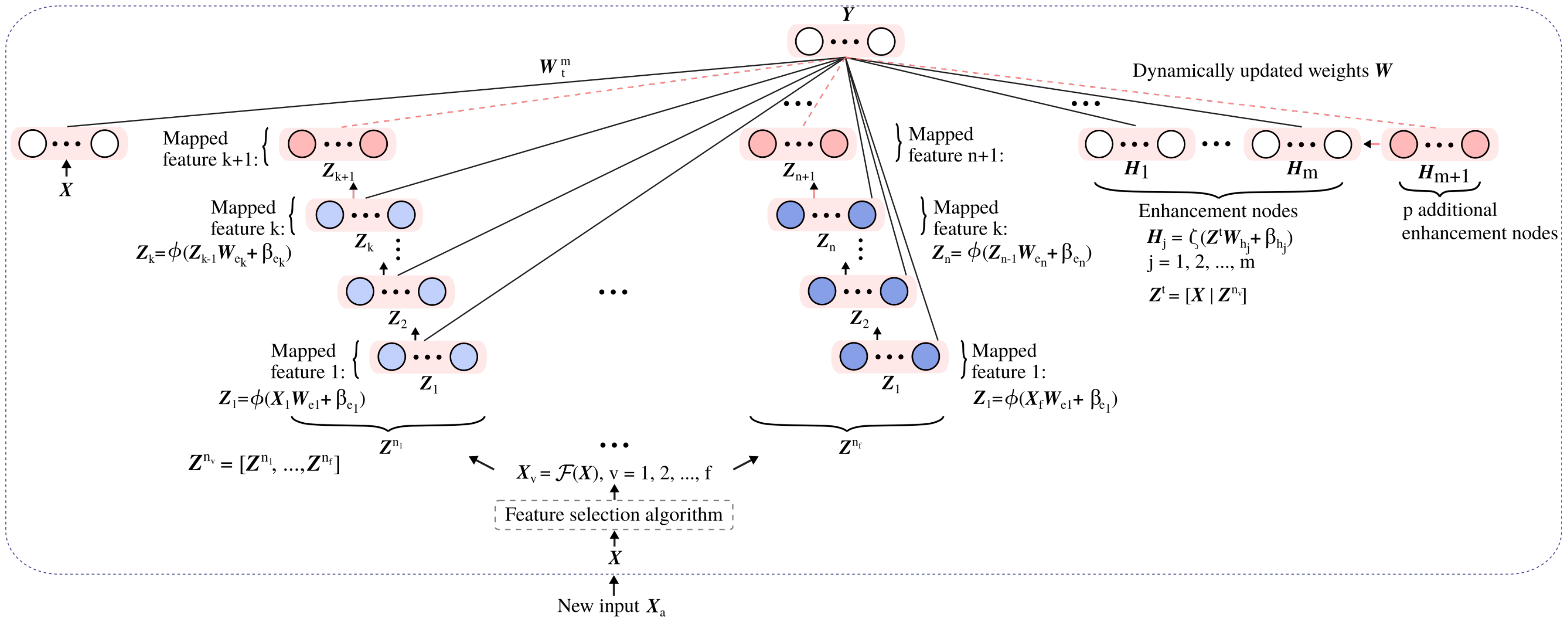


Figure 5.7: Module of the incremental VFBLS algorithms with increments of new input data $\boldsymbol{X}_a$, mapped features $\boldsymbol{Z}_{n+1}$, and enhancement nodes $\boldsymbol{H}_{m+1}$ [6].



Y
W_t^m
Dynamically updated weights W
X
Mapped feature k+1:
Z_{k+1}
Mapped feature k:
Z_k
$Z_k=\phi(Z_{k-1}W_{e_k}+\beta_{e_k})$
Z_2
Mapped feature 1:
Z_1
$Z_1=\phi(X_1W_{e1}+\beta_{e_1})$
Z^{n_1}
$Z^{n_v}=[Z^{n_1}, ..., Z^{n_f}]$
$X_v=\mathcal{F}(X)$, v = 1, 2, ..., f
Feature selection algorithm
X
New input X_a
Mapped feature n+1:
Z_{n+1}
Mapped feature k:
Z_n
$Z_n=\phi(Z_{n-1}W_{e_n}+\beta_{e_n})$
Z_2
Mapped feature 1:
Z_1
$Z_1=\phi(X_fW_{e1}+\beta_{e_1})$
Z^{n_f}
H_1
H_m
H_{m+1}
Enhancement nodes
$H_j=\zeta(Z^tW_{h_j}+\beta_{h_j})$
j = 1, 2, ..., m
$Z^t=[X \mid Z^{n_v}]$
p additional enhancement nodes

Figure 5.8: Modules of the incremental VCFBLS algorithms with increments of new input data $\boldsymbol{X}_a$, mapped features $\boldsymbol{Z}_{n+1}$, and enhancement nodes $\boldsymbol{H}_{m+1}$ [6].

When training a GBDT model [38, 32] with K estimators using N data points, the predicted output is:

$$\hat{y}_i = \sum_{k=1}^{K} f_k(\boldsymbol{x}_i), \tag{5.21}$$

where f_k is the k^{th} decision tree and $\boldsymbol{x}_i$ is the i^{th} data point. One collection sample is represented as a row vector $\boldsymbol{x}_i$ of matrix $\boldsymbol{X}$ containing input data. In the k^{th} iteration, predicted output is evaluated using the k^{th} decision tree (estimator):

$$\hat{y}_i^{(k)} = \hat{y}_i^{(k-1)} + f_k(\boldsymbol{x}_i), \tag{5.22}$$

where $\hat{y}_i^{(k)}$ is the predicted output of the i^{th} data point and $\hat{y}_i^{(k-1)}$ is the previously predicted output. The goal is to minimize the objective function:

$$\mathcal{L}^{(k)} = \sum_{i=1}^{N} l(y_i, \hat{y}_i^{(k)}) + \Omega(f_k), \tag{5.23}$$

where $l(\cdot)$ is the loss function, y_i is the label of the i^{th} input data point, and $\Omega(f_k)$ (optional) is the regularization term.

5.5.1 XGBoost Algorithm

GBDT algorithms may be improved by adding an L^2 norm regularization term to avoid overfitting. For example, XGBoost [38] employs the second-order Taylor series to approximate its objective function and a sparsity-aware algorithm to deal with the sparse data. A cache-aware block structure is employed to generate the XGBoost model when using parallel and distributed computing and to increase training speed. The regularization function is:

$$\Omega(f_k) = \gamma T + \frac{1}{2}\lambda ||\omega||^2, \tag{5.24}$$

where γ and λ are the regularization coefficients, T is the number of leaves in the tree, and ω are the leaf weights. The second-order Taylor series is used to approximate (5.23):

$$\mathcal{L}^{(k)} \simeq \sum_{i=1}^{N} \Big[l(y_i, \hat{y}_i^{(k-1)}) + g_i f_k(\boldsymbol{x}_i) + \frac{1}{2} h_i f_k^2(\boldsymbol{x}_i) \Big] + \Omega(f_k), \tag{5.25}$$

where $g_i = \frac{\partial l(y_i, \hat{y}_i^{(k-1)})}{\partial \hat{y}_i^{(k-1)}}$ and $h_i = \frac{\partial^2 l(y_i, \hat{y}_i^{(k-1)})}{\partial (\hat{y}_i^{(k-1)})^2}$ are known and $l(y_i, \hat{y}_i^{(k-1)})$ is a constant.

For a known tree structure $q(\boldsymbol{X})$, I_t is a set containing the indices of data points in leaf t. Setting the derivative of (5.25) to zero gives the optimal weight ω_t^* for leaf t:

$$\omega_t^* = -\frac{\sum_{i \in I_t} g_i}{\sum_{i \in I_t} h_i + \lambda}. \tag{5.26}$$

The optimal solution of the objective function is:

$$\mathcal{L}^{*(k)} = -\frac{1}{2}\sum_{t=1}^{T}\frac{(\sum_{i\in I_t} g_i)^2}{\sum_{i\in I_t} h_i + \lambda} + \gamma T. \tag{5.27}$$

This optimal value is used to evaluate the quality of a tree structure $q(\boldsymbol{X})$. The tree structure with the lowest optimal value is selected for each iteration.

5.5.2 LightGBM Algorithm

LightGBM [39] algorithm employs Gradient-Based One-Side Sampling (GOSS) and Exclusive Feature Bundling (EFB) techniques to significantly accelerate the training speed. It achieves performance comparable to XGBoost albeit with lower memory usage.

LightGBM relies on the histogram-based algorithm to accelerate locating the best splitting point for each feature. Using the training data points, mutually exclusive features are bundled to create feature histograms. GOSS involves sorting the training data points in a descending order based on the absolute value of their gradients. Top N_t data points (subset $\boldsymbol{A}$) with the largest gradients are selected and random sampling of the remaining input data points is performed to create a subset $\boldsymbol{B}$. The dimensions of $\boldsymbol{A}$ and $\boldsymbol{B}$ depend on predefined sampling ratios a and b, respectively. When training a LightGBM GBDT model with a given dataset, gradients are calculated in each iteration.

In a decision tree, nodes are split based on features with the largest information gain that depends on the variance gain $\tilde{V}_j(d)$ for feature j computed after splitting as [39]:

$$\begin{aligned}\tilde{V}_j(d) = &\frac{1}{N \times N_l^j(d)}(\sum_{\boldsymbol{x}_i\in\boldsymbol{A}_l} g_i + \frac{1-a}{b}\sum_{\boldsymbol{x}_i\in\boldsymbol{B}_l} g_i)^2 \\ &+\frac{1}{N \times N_r^j(d)}(\sum_{\boldsymbol{x}_i\in\boldsymbol{A}_r} g_i + \frac{1-a}{b}\sum_{\boldsymbol{x}_i\in\boldsymbol{B}_r} g_i)^2,\end{aligned} \tag{5.28}$$

where d is the splitting point, N is the number of data points, N_l^j and N_r^j are number of input data points related to left and right child nodes, respectively, and g_i is the gradient for input data point $\boldsymbol{x}_i$. The sampling ratios a and b are used to calculate the normalization coefficient $(1-a)/b$. $\boldsymbol{A}_l$ ($\boldsymbol{B}_l$) and $\boldsymbol{A}_r$ ($\boldsymbol{B}_r$) are the subsets of $\boldsymbol{A}$ ($\boldsymbol{B}$) for the left and right child nodes, respectively.

LightGBM, based on the GOSS technique, utilizes leaf-wise growth approach to grow the decision trees instead of level-wise growth used in XGBoost. Level-wise growth splits the leaves of the same layer, which enables easy control of the model complexity. However, it introduces unnecessary overhead because the leaves with low variance gains are also split. Compared to level-wise tree growth, leaf-wise is a more efficient approach because it splits the leaf that has the maximum variance gain thus reducing additional loss after a number of splits. Furthermore, it may lead to deeper decision trees resulting in over-fitting. Hence, the hyper-parameter “max depth” is introduced to limit their depth.

The EFB technique combines dataset features in order to reduce the dimension of the input data and the complexity of building the histogram from $\mathcal{O}(n_d n_f)$ to $\mathcal{O}(n_d n_b)$, where n_d, n_f, and n_b are the number of data points, features, and bundles, respectively.

5.5.3 CatBoost Algorithm

The XGBoost algorithm only accepts numerical values and employs one-hot encoding to convert categorical features to numerical values while LightGBM converts these features to gradient statistics. Hence, CatBoost [40] is introduced to deal with categorical features. It employs the ordered boosting algorithm and offers an effective approach (ordered target statistic) when compared to XGBoost and LightGBM. Target statistic was used to convert categorical to numerical features by using the values that estimate the expected labels based on the categories while keeping the dimension of the dataset unchanged.

In the GBDT models, residuals are calculated in each iteration and are used as the target values in the next training iteration. This leads to bias increase and prediction shift in subsequent iterations and, thus, model over-fitting. Hence, ordered boosting was proposed to address the prediction shift when building the decision trees during the training process. It performs permutation and trains multiple decision trees in each iteration. Each residual is calculated based on the target and predicted values generated by the previous decision tree. Symmetric (oblivious) decision trees are used to avoid over-fitting and reduce the time required to grow the tree. CatBoost offers plain and ordered boosting modes with target statistics and ordered boosting, respectively. In each iteration, the two boosting modes have the same asymptotic complexity for calculating gradients $\mathcal{O}(sN)$, updating decision trees $\mathcal{O}(sN)$, and computing ordered target statistic $\mathcal{O}(N_{TS}N)$, where s, N, and N_{TS} are the number of permutations, data points, and features using target statistics, respectively.

5.6 Isolation Forest

Isolation forests (iForests) are unsupervised algorithms used for anomaly detection [141] that offer short execution time due to their linear complexity. They exploit that the outliers (anomalous data points) have fewer instances and different attribute-values than inliers (regular data points) and, hence, are easier to isolate.

iForests are ensembles of binary decision trees called Isolation Trees (iTrees). Split values when building an iTree are randomly selected for each feature according to the range of their values. Data points are then iteratively routed through the iTrees based on the defined split values until a node has only one instance or all node data have the same values. Outliers are detected based on the average path length from root to leaf. Hence, a data point is more likely to be an outlier when the average path length is shorter while average path lengths for inliers are longer.

A path lenght $h(x)$ of a given data pint x is based on the number of edges traversed in an iTree from the root to external node. The structure of iTrees is similar to binary search trees and, hence, estimation of the average $h(x)$ of a dataset with N data points is calculated as [141]:

$$c(N) = 2H(N-1) - (2(N-1/N), \tag{5.29}$$

where $c(N)$, used to normalize $h(x)$, is the average path length of $h(x)$ given N and $H(i)$ is the harmonic number estimated as $ln(i) + 0.5772156649$ (Euler's constant). The score s of a data point is:

$$s(x, N) = 2^{-\frac{E(h(x))}{c(N)}}, \tag{5.30}$$

where $E(h(x))$ is the average of $h(x)$ for a collection of iTrees. Based on obtained scores s, a data point is considered as:

$$\text{outlier: } E(h(x)) \to 0, s \to 1, \tag{5.31}$$

$$\text{inliers: } E(h(x)) \to N, s \to 0, \tag{5.32}$$

$$\text{indistinct: } E(h(x)) \to c(n), s \to 0.5. \tag{5.33}$$

Chapter 6

Performance Evaluation and Experimental Results

BGP raw data captured from RIPE and Route Views data collection sites are processed to generate datasets. The *zebra-dump-parser* [100] tool, one of the offered tools advertised by RIPE, is first used to transform data from MRT to ASCII format. Features are then extracted using our developed C# [101] tool based on format of transformed data. Generated datasets are employed to perform two-way supervised or semi-supervised classification techniques to identify regular (0) and anomalous (1) data points using the WannaCrypt and WestRock datasets. Eight datasets are labeled with and without employing iForest to refine labels within the anomalous windows of the WannaCrypt and WestRock ransomware attacks. Generated are models based on RNNs (LSTM, GRU), Bi-RNNs (Bi-LSTM, Bi-GRU), BLS and its extensions, VFBLS, VCFBLS, and gradient boosting decision trees (GBDT) (XGBoost, LightGBM, CatBoost). Models performance is compared based on training time, accuracy, F-Score, precision, sensitivity (recall), and confusion matrix.

The experiments include cross-validation and testing. They are conducted using a supercomputer managed by Compute Canada [142]. The Cedar [143] cluster consists of 94,528 CPU cores. We used 64 GB memory and an Intel E5-2683 v4 Broadwell (2.1 GHz) processor with 8 cores. Python 3.6 and libraries [144] *NumPy* (a scientific computing library), *PyTorch* (a Python framework for deep learning), *scikit-learn* (a machine learning library), and gradient boosting frameworks (*XGBoost*, *LightGBM*, *CatBoost*), are used to create input matrices and to train and test the machine learning models.

6.1 Experimental Procedure

The experimental procedure consists of the nine steps shown in Fig. 6.1:

(1) Collecting 10 and 11 days of BGP *update* messages for WannaCrypt and WestRock ransomware attacks, respectively: the days of each attacks as well as two days before and two days after each attacks;

(2) Processing BGP raw data from RIPE and Route Views to extract 37 features;

(3) Labeling regular (0) and anomalous (1) data points based on start and end times of the WannaCrypt (00:00 on 12.05.2017 to 23:59 on 15.05.2017) and WestRock (1:12 UTC on 23.01.2021 to 23:59 UTC on 29.01.2021) ransomware attacks;

(4) Refining labels for data points within the windows of anomalous events;

(5) Normalizing training and test datasets to have mean 0 and standard deviation 1 employing the *zscore* function;

(6) Partitioning data to extract subsets for training (60 % of anomalies) and testing (40 % of anomalies);

(7) Using 10-fold cross-validation based on the time series split to train and tune parameters;

(8) Generating machine learning (ML) models; and

(9) Testing and evaluating generated ML models based on training time, accuracy, F-Score, precision, sensitivity (recall), and confusion matrix.

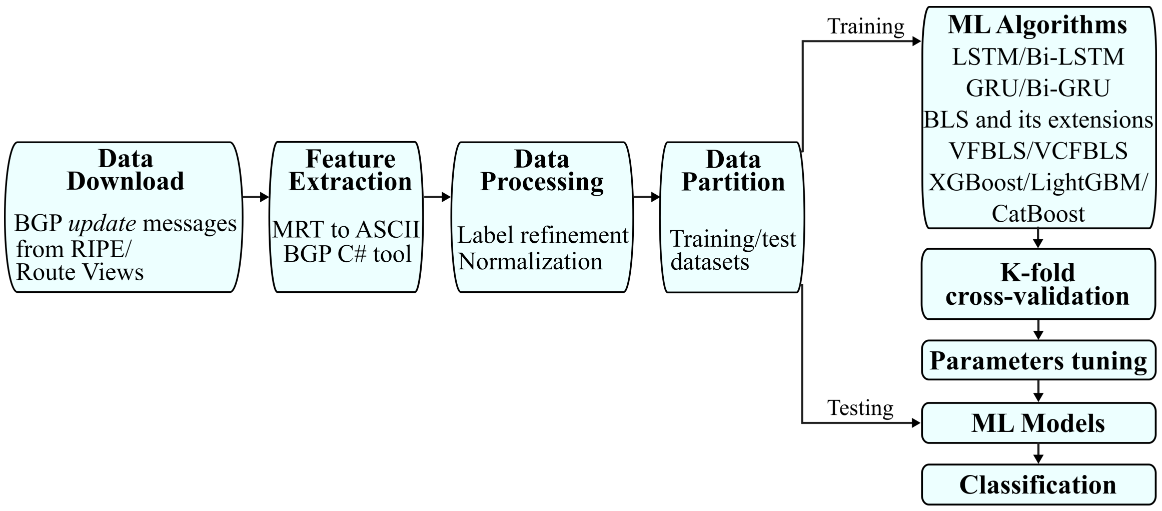


Figure 6.1: Experimental procedure for detecting BGP anomalies caused by WannaCrypt and WestRock ransomware attacks.

The partitioning process for training and validation datasets for the WannaCript and WestRock 10-fold cross-validation based on the time series split is illustrated in Fig. 6.2. In each fold, 572 (WannaCrypt) and 814 (WestRock) data points are used as the validation dataset. In the first step (Fold 1), 580 (WannaCrypt) and 820 (WestRock) data points are used for training. The training dataset in each subsequent fold is the concatenation of the previous training and validation datasets.

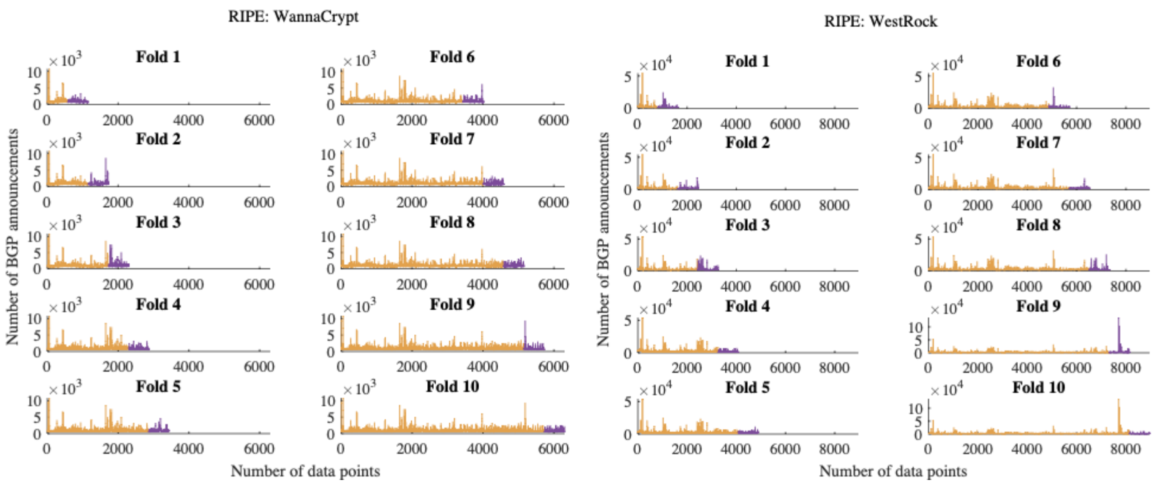


Figure 6.2: Time series split for the 10-fold cross-validation of the WannaCrypt (left) and WestRock (right) training datasets. Illustrated are the generations of training (orange) and validation (purple) datasets.

RNN and Bi-RNN models consist of one RNN and one Bi-RNN layer, respectively, and up to three Fully-Connected (FC) layers. The generated models have 2 (LSTM_2/Bi-LSTM_2, GRU_2/Bi-GRU_2), 3 (LSTM_3/Bi-LSTM_3, GRU_3/Bi-GRU_3), and 4 (Bi-LSTM_4, Bi-GRU_4) hidden layers. The first layers in the LSTM and GRU deep learning neural network models consist of 37 RNNs or Bi-RNNs. A model with 4 hidden layers is shown in Fig. 6.3. ReLU is used as the RNN activation function. Parameters leading to the best performance using WannaCrypt and WestRock datasets are listed in Table 6.1. The optimization algorithm "Adam" [145] is selected to train the RNN and Bi-RNN models. A deep learning neural network model with four hidden layers consisting of 37 RNNs, 64 FC_1, 32 FC_2, and 16 FC_3 fully connected (FC) hidden nodes is shown in Fig. 6.3.

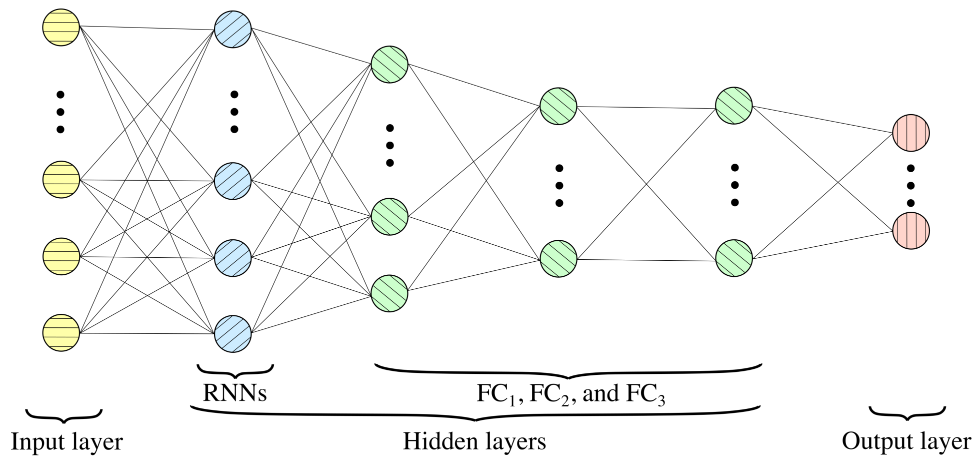


Figure 6.3: Deep learning neural network model. It consists of 37 RNNs, 64 FC_1, 32 FC_2, and 16 FC_3 fully connected (FC) hidden nodes.

Table 6.1: RNNs and Bi-RNNs Parameters Leading to the Best Performance: WannaCrypt and WestRock Datasets.

Label refinement	**Collection site**	**Dataset**	**Input sequence length**	**No. of epochs**	**No. of hidden nodes (FC_1, FC_2, FC_3)**	**Learning rate**
RNNs						
None	**RIPE**	WannaCrypt	100	30	64, 32, 16	0.01
		WestRock	160	50	80,32,16	0.001
	Route Views	WannaCrypt	100	30	64, 32, 16	0.1
		WestRock	40	50	80,32,16	0.001
iForest	**RIPE**	WannaCrypt	100	50	64, 32, 16	0.001
		WestRock	160	50	64, 32, 16	0.001
	Route Views	WannaCrypt	100	50	32, 32, 16	0.001
		WestRock	160	50	64, 32, 16	0.001
Bi-RNNs						
None	**RIPE**	WannaCrypt	100	30	64, 32, 16	0.01
		WestRock	160	50	64, 32, 16	0.001
	Route Views	WannaCrypt	100	30	64, 32, 16	0.01
		WestRock	160	50	64, 32, 16	0.001
iForest	**RIPE**	WannaCrypt	100	50	32, 32, 16	0.001
		WestRock	160	50	64, 32, 16	0.001
	Route Views	WannaCrypt	100	50	80, 32, 16	0.001
		WestRock	160	50	64, 32, 16	0.001

The CFBLS, CEBLS, and CFEBLS models were implemented by modifying the original BLS functions [136]. For the cross-validation of incremental and non-incremental BLS models, we vary the number of mapped features (50–300), groups of mapped features (5–20), and enhancement nodes (50–200). Parameters of the incremental BLS models when using datasets without label refinement are: *incremental learning steps* = 2 (WannaCrypt RIPE, WestRock RIPE), 3 (WannaCrypt Route Views, WestRock Route Views); *enhancement nodes/step* = 40 (WannaCrypt RIPE, WannaCrypt Route Views, WestRock RIPE), 20 (WestRock Route Views); and *data points/step* = 1,260 (WannaCrypt RIPE), 840 (WannaCrypt Route Views), 1,792 (WestRock RIPE), and 1,195 (WestRock Route Views). In the case of datasets with label refinement using iForest, parameters of the incremental BLS models are: *incremental learning steps* = 2 (WannaCrypt Route Views, WestRock RIPE, WestRock Route Views), 3 (WannaCrypt RIPE); *enhancement nodes/step* = 40 (WannaCrypt RIPE, WestRock Route Views), 20 (WannaCrypt Route Views, WestRock RIPE); and *data points/step* = 1,260 (WannaCrypt Route Views), 840 (WannaCrypt RIPE), and 1,792 (WestRock RIPE, WestRock Route Views). Training parameters that generate the best performance results are listed in Table 6.2.

In the cross-validation of VFBLS and VCFBLS models, we vary mapped features (20–50), groups of mapped features (10–30), and enhancement nodes (50–100). In the case of incremental VFBLS and VCFBLS models, *feature weight for initial step* = 0.9 and *enhance-*

Table 6.2: BLS and Incremental BLS Parameters Leading to the Best Performance: WannaCrypt and WestRock Datasets.

Label refinement	**Collection site**	**Dataset**	**Model**	**Mapped features features**	**Groups of mapped nodes**	**Enhancement**
BLS						
None	**RIPE**	WannaCrypt	RBF-BLS	300	5	100
		WestRock	RBF-BLS	300	5	100
	Route Views	WannaCrypt	CFBLS	50	5	200
		WestRock	RBF-BLS	100	10	200
iForest	**RIPE**	WannaCrypt	RBF-BLS	50	10	50
		WestRock	RBF-BLS	50	20	100
	Route Views	WannaCrypt	BLS	300	20	50
		WestRock	RBF-BLS	300	5	200
Incremental BLS						
None	**RIPE**	WannaCrypt	BLS	200	20	200
		WestRock	RBF-BLS	100	5	200
	Route Views	WannaCrypt	CEBLS	100	10	200
		WestRock	CEBLS	100	20	200
iForest	**RIPE**	WannaCrypt	CFEBLS	50	20	50
		WestRock	RBF-BLS	300	20	50
	Route Views	WannaCrypt	CEBLS	100	5	200
		WestRock	RBF-BLS	200	10	200

ment nodes/step = 20. When using datasets without label refinement, remaining parameters of the incremental VFBLS are: *incremental learning steps* = 2 and *data points/step* = 315 (WannaCrypt), 448 (WannaCrypt RIPE), while remaining parameters of the incremental VCFBLS models are: *incremental learning steps* = 2 (WannaCrypt RIPE, WestRock RIPE), 3 (WannaCrypt Route Views, WestRock Route Views) and *data points/step* = 315 (WannaCrypt RIPE), 210 (WannaCrypt Route Views), 448 (WannaCrypt RIPE), 299 (WestRock Route Views). In the case of datasets with label refinement, remaining parameters of the incremental VFBLS are: *incremental learning steps* = 2 (WannaCrypt RIPE, WannaCrypt Route Views, WestRock Route Views), 3 (WestRock RIPE) and *data points/step* = 315 (WannaCrypt RIPE, WannaCrypt Route Views), 448 (WestRock Route Views), 299 (WestRock RIPE), while remaining parameters of the incremental VCFBLS models are: *incremental learning steps* = 2 (WannaCrypt RIPE, WestRock RIPE, WestRock Route Views), 3 (WannaCrypt Route Views) and *data points/step* = 315 (WannaCrypt RIPE), 210 (WannaCrypt Route Views), 448 (WestRock RIPE, WestRock Route Views). Training parameters that generate the best performance results are listed in Table 6.3 and Table 6.4.

Training hyper-parameters of the GBDT models that generate the best performance results are listed in Table 6.5. Additional hyper-parameters for GBDT algorithms are: *maximum depth in a tree* = 6 (XGBoost, CatBoost), *maximum number of leaves* = 31

Table 6.3: VFBLS and VCFBLS Parameters Leading to the Best Performance: WannaCrypt and WestRock Datasets.

Label Refinement	**Collection site**	**Number of features**	**Dataset**	**Mapped features**	**Groups of mapped features**	**Enhance-ment nodes**
VFBLS						
None	**RIPE**	37, 16, 8	WannaCrypt	40, 30, 40	20, 20, 30	100
			WestRock	40, 30, 50	20, 10, 30	50
	Route Views	37, 16, 8	WannaCrypt	40, 40, 40	10, 20, 30	100
			WestRock	40, 30, 50	20, 20, 30	50
iForest	**RIPE**	37, 16, 8	WannaCrypt	40, 30, 40	20, 20, 30	100
			WestRock	40, 40, 50	10, 20, 20	50
	Route Views	37, 16, 8	WannaCrypt	20, 30, 50	10, 10, 20	10
			WestRock	30, 30, 40	10, 20, 30	50
VCFBLS						
None	**RIPE**	37, 16, 8	WannaCrypt	30, 30, 50	10, 10, 30	100
			WestRock	20, 30, 40	10, 20, 20	50
	Route Views	37, 16, 8	WannaCrypt	40, 40, 50	20, 10, 20	100
			WestRock	20, 40, 50	10, 10, 30	50
iForest	**RIPE**	37, 16, 8	WannaCrypt	30, 30, 50	10, 10, 30	100
			WestRock	40, 40, 50	20, 10, 30	50
	Route Views	37, 16, 8	WannaCrypt	40, 30, 40	10, 10, 30	50
			WestRock	20, 40, 50	10, 10, 30	50

(LightGBM, CatBoost), and *loss function* = log-loss. We implement *gbtree* (XGBoost), *gbdt* (LightGBM), and *Plain* (CatBoost) boosting modes. For cross-validation of LightGBM models, we vary the number of estimators (10–200) and learning rate (0.01–0.1).

6.2 Performance Metrics

Performance of supervised and semi-supervised classification models is evaluated based on training time, accuracy, F-Score, precision, sensitivity (recall), and confusion matrix.

6.2.1 Confusion Matrix

The number of classified data points for each class k are analyzed and compared by using a confusion matrix [146]. In the case of binary classification, only two classes are evaluated and, hence, this matrix consists of True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN) as shown in Table 6.6. TP and FN indicate the number of anomalous data points that are correctly (anomaly) and incorrectly (regular) classified, respectively. FP and TN refer to the number of regular data points that are classified as anomaly and regular, respectively. The confusion matrix is used to calculate other performance metrics including accuracy, precision, and sensitivity (recall).

Table 6.4: Incremental VFBLS and VCFBLS Parameters Leading to the Best Performance: WannaCrypt and WestRock Datasets.

Label Refinement	Collection site	Number of features	Dataset	Mapped features	Groups of mapped features	Enhance-ment nodes
Incremental VFBLS						
None	**RIPE**	37, 16, 8	WannaCrypt	40, 30, 50	10, 20, 30	50
			WestRock	40, 30, 50	10, 10, 30	50
	Route Views	37, 16, 8	WannaCrypt	30, 30, 40	10, 10, 30	100
			WestRock	30, 30, 40	20, 20, 20	100
iForest	**RIPE**	37, 16, 8	WannaCrypt	40, 30, 50	10, 20, 30	100
			WestRock	40, 40, 40	20, 20, 20	50
	Route Views	37, 16, 8	WannaCrypt	30, 30, 40	10, 10, 30	100
			WestRock	20, 40, 40	20, 20, 30	100
Incremental VCFBLS						
None	**RIPE**	37, 16, 8	WannaCrypt	30, 40, 40	10, 20, 30	50
			WestRock	40, 30, 50	20, 10, 30	100
	Route Views	37, 16, 8	WannaCrypt	40, 40, 40	20, 20, 30	100
			WestRock	20, 30, 40	20, 20, 20	50
iForest	**RIPE**	37, 16, 8	WannaCrypt	30, 40, 40	10, 10, 30	100
			WestRock	20, 40, 50	10, 20, 30	50
	Route Views	37, 16, 8	WannaCrypt	30, 30, 40	20, 10, 30	100
			WestRock	30, 30, 50	20, 10, 30	100

6.2.2 Accuracy

Accuracy is used [147] to evaluate a model's ability to correctly classify anomalous data points. It is broadly defined as:

$$\text{accuracy} = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions}. \tag{6.1}$$

If used to evaluate binary classification, then accuracy is calculated based on the number of TP and TN with respect to predicted labels:

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN}. \tag{6.2}$$

When one class has a significantly larger number of data points (imbalanced dataset) [148], accuracy does not correctly reflect a model's classification ability. Methods employed to balance [149] datasets include oversampling (duplicating data points from minority class) or undersampling (removing data points from majority class).

6.2.3 Precision

Precision is a performance metric that indicates the portion of data points correctly classified as anomalous (TP) among predicted anomalous data points ($TP\ +\ FP$) [150, 151]. This

Table 6.5: XGBoost, LightGBM, and CatBoost Hyper-Parameters Leading to the Best Performance: WannaCrypt and WestRock Datasets.

Label Refinement	**Collection Site**	**Model**	**Dataset**	**Number of estimators**	**Learning rate**
None	**RIPE**	XGBoost	WannaCrypt	300	0.10
			WestRock	100	0.01
		LightGBM	WannaCrypt	200	0.10
			WestRock	150	0.02
		CatBoost	WannaCrypt	300	0.10
			WestRock	100	0.01
	Route Views	XGBoost	WannaCrypt	300	0.10
			WestRock	50	0.01
		LightGBM	WannaCrypt	250	0.10
			WestRock	150	0.02
		CatBoost	WannaCrypt	200	0.05
			WestRock	100	0.01
iForest	**RIPE**	XGBoost	WannaCrypt	300	0.01
			WestRock	100	0.1
		LightGBM	WannaCrypt	300	0.05
			WestRock	100	0.05
		CatBoost	WannaCrypt	200	0.1
			WestRock	100	0.01
	Route Views	XGBoost	WannaCrypt	150	0.1
			WestRock	50	0.01
		LightGBM	WannaCrypt	300	0.05
			WestRock	100	0.1
		CatBoost	WannaCrypt	200	0.05
			WestRock	150	0.01

metric defines the reliability of a model when identifying true anomalies and is calculated as:

$$\text{precision} = \frac{TP}{TP + FP}. \tag{6.3}$$

6.2.4 Sensitivity (Recall)

The performance metric used to measure the portion of data points correctly classified as anomalous (TP) with respect to actual anomalous data points ($TP + FN$) is referred to as sensitivity (recall) [151, 152]. This metric determines a model's ability to predict true anomalous data points and is defined as:

$$\text{sensitivity (recall)} = \frac{TP}{TP + FN}. \tag{6.4}$$

Table 6.6: Confusion matrix.

	Predicted class	
Actual class	Anomaly (positive)	Regular (negative)
Anomaly (positive)	TP	FN
Regular (negative)	FP	TN

6.2.5 F-Score

During the evaluation performance of a machine learning model, the trade-off between precision and sensitivity should be considered given that improving one metric reduces the effectiveness of the other. A metric employed to balance this trade-off is F-Score [150]. It is defined as the harmonic mean of precision and sensitivity:

$$\text{F-Score} = 2 \times \frac{\text{precision} \times \text{sensitivity}}{\text{precision} + \text{sensitivity}}. \tag{6.5}$$

F-Score measures a model's ability to discriminate between correctly and incorrectly classified anomalous data points.

6.2.6 Training Time

We also consider the time required to train a machine learning algorithm (training time) as a performance metric. Training time depends on factors such as datasets size and computational complexity of employed algorithms. Algorithms having short training times are important for decision making at the onset of malicious attacks.

6.2.7 Experimental Results

The best performance results of RNNs, Bi-RNNs, BLS and its extensions, VFBLS, VCFBLS, and GBDT algorithms using the WannaCrypt and WestRock ransomware are listed in Table 6.7 and Table 6.8, respectively. The best classification model for WannaCrypt ransomware attack is Bi-GRU$_4$ generated based on the Route Views data with label refinement. In the case of WestRock ransomware attack, Bi-GRU$_4$ model generates the best results using RIPE data and label refinement. BLS and its extensions (incremental learning, RBF-BLS, CEBLS, CFBLS, CFEBLS), BLS with variable features (VFBLS, VCFBLS), and GBDT (XGBoost, LightGBM, CatBoost) models exhibit comparable performance with short training time in most cases. WannaCrypt and WestRock datasets from RIPE collection site often generate better accuracy and F-Score. LightGBM models offer the shortest training times albeit of lower accuracy and F-Score compared to Bi-GRU$_4$ models. RNN and Bi-RNN models require longer training times. Incremental VFBLS and VCFBLS models achieve the highest sensitivity for both datasets. Incremental BLS, CEBLS, and CFEBLS generate high sensitivity using WannaCrypt datasets. RNN and Bi-RNN models lead to the highest precision.

Experimental results indicate that RNN and Bi-RNN models using the WannaCrypt and WestRock datasets achieve more balanced precision and sensitivity that lead to higher F-Score when compared with other models. Increasing the number of hidden layers may result in higher performance albeit longer training time. Employing Bi-RNN models often improves classification performance because models are trained based on outputs of past and future time steps. The implementation of Bi-RNN models consists of one forward and one backward layer and, hence, it requires longer training time compared to RNN models.

BLS, RBF-BLS, and CFBLS models using the WannaCrypt datasets generate the best classification performance compared to other BLS extensions. These models achieve balanced precision and sensitivity albeit lower F-Score when compared to RNN and Bi-RNN models. Incremental learning models based on increments of input data and enhancement nodes when using the WannaCrypt datasets generate higher sensitivity and, hence, F-Score improves. The RBF-BLS models using the WestRock datasets achieve very high sensitivity and very low precision that lead to F-Score comparable to most RNN and Bi-RNN models. Generating incremental learning models based on increments of input data and enhancement nodes when using the WestRock datasets leads to sensitivity higher than 99.00 % while precision is comparable to precision generated using RBF-BLS models, which results in higher F-Score. When compared to RNN and Bi-RNN models, most BLS models based on the incremental learning require comparable training time despite of models only requiring dynamic update of output weights instead of retraining.

VFBLS and VCFBLS models with and without incremental learning achieve comparable classification performance with shorter training time when compared to other BLS models. Higher sensitivity and comparable precision are achieved when using VFBLS and VCFBLS models with incremental learning based on increments of input data and enhancement nodes. The increased sensitivity resulted in higher F-Score. Implementing VFBLS and VCFBLS with incremental learning results in longer training time by a factor of approximately 2. This increase may be due to feature selection required at each time step when adding new input data.

Generated GBDT models using WannaCrypt and WestRock datasets achieve comparable classification performance with shorter training time when compared to BLS models with and without incremental learning. Unlike other models, the best generated results using GBDT models are based on a smaller number of features for most cases. Implementing feature selection is an important step when generating models in order to remove redundant information that may degrade a model's performance. Another advantage of GBDT models is the reduced number of parameters and hyper-parameters that need tuning in order to improve classification performance.

In the case of semi-supervised machine learning approach, the iForest anomaly detection unsupervised algorithm is employed to identify regular data points during anomalous periods. However, most anomalous data points retain the same label after the refinement and only few data points are re-labeled as regular. This semi-supervised machine learning approach based on label refinement offers only comparable classification performance when compared to supervised machine learning.

Table 6.7: The Best Performance of RNN, Bi-RNN, BLS, VFBLS, VCFBLS, and GBDT Models Using WannaCrypt Dataset.

Model	**Refinement**	**Collection site**	**No. Ftr.**	**Training time (s)**	**Accuracy (%)**	**F-Score (%)**	**Precision (%)**	**Sensitivity (%)**	**TP**	**FP**	**TN**	**FN**
LSTM$_3$	none	RIPE	37	12.67	65.48	63.22	60.73	65.94	1,543	998	1,862	797
GRU$_3$		Route Views	37	12.01	72.63	74.14	64.50	87.18	2,040	1,123	1,737	300
GRU$_2$	iForest	RIPE	37	17.16	67.89	63.38	65.10	61.78	1,445	776	2,085	894
LSTM$_4$		Route Views	37	22.83	76.46	76.00	70.20	82.86	1,938	823	2,038	401
Bi-LSTM$_4$	none	RIPE	37	41.85	66.96	64.30	62.58	66.11	1,547	925	1,935	793
Bi-GRU$_4$		Route Views	37	27.98	79.39	79.60	71.79	89.27	2,089	821	2,039	251
Bi-LSTM$_4$	iForest	RIPE	37	29.82	64.08	65.95	57.48	77.34	1,809	1,338	1,523	530
Bi-GRU$_4$		Route Views	37	29.10	80.79	79.70	75.97	83.79	1,960	620	2,241	379
RBF-BLS	none	RIPE	37	3.67	55.73	56.68	50.48	64.62	1,512	1,483	1,397	828
CFBLS		Route Views	37	0.62	50.67	55.21	46.55	67.82	1,587	1,822	1,058	753
RBF-BLS	iForest	RIPE	37	1.02	55.61	56.46	50.37	64.22	1,502	1,480	1,401	837
BLS		Route Views	37	19.07	50.79	52.38	46.24	60.41	1,413	1,643	1,238	926
Incr. BLS	none	RIPE	37	16.44	46.97	62.10	45.69	96.92	2,268	2,696	184	72
Incr. CEBLS		Route Views	37	16.73	56.65	63.97	50.98	85.85	2,099	1,932	948	33
Incr. CFEBLS	iForest	RIPE	37	3.36	50.96	61.73	47.46	88.29	2,065	2,286	595	274
Incr. CEBLS		Route Views	37	14.81	56.82	60.98	51.24	75.29	1,761	1,676	1,205	578
VFBLS	none	RIPE	37, 16, 8	6.49	55.06	46.07	49.85	42.82	1002	1008	1872	1338
		Route Views	37, 16, 8	5.51	48.60	53.33	44.97	65.51	1,533	1,876	1,004	807
	iForest	RIPE	37, 16, 8	6.36	55.04	46.06	49.80	42.84	1,002	1,010	1,871	1,337
		Route Views	37, 16, 8	3.50	48.14	52.88	44.60	64.94	1,519	1,887	994	820
VCFBLS	none	RIPE	37, 16, 8	3.97	54.92	46.70	49.69	44.06	1,031	1,044	1,836	1,309
		Route Views	37, 16, 8	4.92	49.18	53.17	45.29	64.36	1,506	1,819	1,061	834
	iForest	RIPE	37, 16, 8	3.98	54.92	46.73	49.66	44.12	1,032	1,046	1,835	1,307
		Route Views	37, 18, 8	3.84	50.09	53.57	45.94	64.26	1,503	1,769	1,112	836
Incr. VFBLS	none	RIPE	37, 16, 8	6.66	53.22	64.36	48.87	94.23	2,205	2,307	573	135
		Route Views	37, 16, 8	4.86	56.82	64.10	51.10	85.98	2,012	1,926	954	328
	iForest	RIPE	37, 16, 8	6.88	53.30	64.13	48.89	93.16	2,179	2,278	603	160
		Route Views	37, 16, 8	4.83	57.09	64.10	51.27	85.46	1,999	1,900	981	340
Incr. VCFBLS	none	RIPE	37, 16, 8	8.64	53.20	64.32	48.86	94.10	2,202	2,305	575	138
		Route Views	37, 16, 8	10.30	55.98	64.24	50.51	88.21	2,064	2,022	858	276
	iForest	RIPE	37, 16, 8	6.84	53.01	64.40	48.75	94.83	2,218	2,332	549	121
		Route Views	37, 16, 8	8.88	55.10	64.07	49.94	89.35	2,090	2,095	786	249
XGBoost	none	RIPE	8	0.54	59.87	61.18	54.01	70.56	1,651	1,406	1,474	689
		Route Views	16	0.87	53.05	59.56	48.51	77.14	1,805	1,916	964	535
	iForest	RIPE	8	1.31	61.19	61.69	55.31	69.73	1,631	1,318	1,563	708
		Route Views	16	1.02	52.20	59.12	47.93	77.13	1,804	1,960	921	535
LightGBM	none	RIPE	8	0.09	60.25	61.48	54.35	70.77	1,656	1,391	1,489	684
		Route Views	37	0.14	52.74	59.18	48.29	76.41	1,788	1,915	965	552
	iForest	RIPE	8	0.15	66.08	61.41	54.17	70.88	1,658	1,403	1,478	681
		Route Views	37	0.23	52.38	58.95	48.02	76.31	1,785	1,932	949	554
CatBoost	none	RIPE	8	1.09	60.31	62.04	54.30	72.35	1,693	1,425	1,455	647
		Route Views	16	0.70	52.30	59.30	48.01	77.48	1,813	1,963	917	527
	iForest	RIPE	8	0.66	60.48	61.98	54.47	71.91	1,682	1,406	1,475	657
		Route Views	16	0.70	52.32	59.30	48.01	77.51	1,813	1,963	918	526

Table 6.8: The Best Performance of RNN, Bi-RNN, BLS, VFBLS, VCFBLS, and GBDT Models Using WestRock Dataset.

Model	**Refinement**	**Collection site**	**No. Ftr.**	**Training time (s)**	**Accuracy (%)**	**F-Score (%)**	**Precision (%)**	**Sensitivity (%)**	**TP**	**FP**	**TN**	**FN**
GRU_4	none	RIPE	37	13.99	75.23	80.24	74.84	86.48	3,459	1,163	1,717	541
$LSTM_4$		Route Views	37	18.95	55.42	70.72	57.20	92.60	3,704	2,771	109	296
$LSTM_2$	iForest	RIPE	37	12.63	75.36	79.73	76.41	83.35	3,333	666	1,029	1,852
$LSTM_3$		Route Views	37	13.77	60.00	69.06	62.75	76.80	3,072	1,824	1,056	928
Bi-GRU_4	none	RIPE	37	20.59	78.49	81.92	80.10	83.83	3,353	833	2,047	647
Bi-GRU_3		Route Views	37	21.89	62.50	69.70	65.73	74.18	2,967	1,547	1,333	1,033
Bi-GRU_4	iForest	RIPE	37	23.73	84.27	86.90	84.23	89.75	3,589	672	2,209	410
Bi-GRU_3		Route Views	37	20.23	64.74	72.19	66.67	78.70	3,148	1,574	1,306	852
RBF-BLS	none	RIPE	37	3.98	55.70	70.75	57.41	92.18	3,687	2,735	145	313
		Route Views	37	2.60	54.74	69.99	56.95	90.78	3,631	2,745	135	369
	iForest	RIPE	37	2.20	55.73	70.77	57.42	92.20	3,687	2,734	147	312
		Route Views	37	3.97	54.61	69.81	56.91	90.28	3,611	2,734	146	389
Incr. RBF-BLS	none	RIPE	37	1.71	58.20	73.55	58.18	99.98	3,999	2,875	5	1
Incr. CEBLS		Route Views	37	23.33	57.89	73.31	58.05	99.48	3,979	2,876	4	21
Incr. RBF-BLS	iForest	RIPE	37	33.28	58.20	73.54	58.16	99.98	3,998	2,876	5	1
		Route Views	37	7.01	58.15	73.52	58.16	99.93	3,997	2,876	4	3
VFBLS	none	RIPE	37, 16, 8	7.31	55.15	70.18	57.19	90.80	3,632	2,718	162	368
		Route Views	37, 16, 8	7.99	54.75	69.92	56.99	90.45	3,618	2,731	149	382
	iForest	RIPE	37, 16, 8	6.18	54.74	69.81	57.00	90.05	3,601	2,716	165	398
		Route Views	37, 16, 8	5.67	54.23	69.41	56.76	89.33	3,573	2,722	158	427
VCFBLS	none	RIPE	37, 16, 8	4.14	55.33	70.31	57.30	90.95	3,638	2,711	169	362
		Route Views	37, 16, 8	4.62	54.68	69.73	56.99	89.80	3,592	2,710	170	408
	iForest	RIPE	37, 16, 8	6.56	54.72	69.86	56.98	90.27	3,610	2,726	155	389
		Route Views	37, 18, 8	4.66	54.43	69.55	56.87	89.53	3,581	2,716	164	419
Incr. VFBLS	none	RIPE	37, 16, 8	6.77	58.17	73.54	58.16	100	4,000	2,878	2	0
		Route Views	37, 16, 8	6.82	58.18	73.55	58.16	100	4,000	2,877	3	0
	iForest	RIPE	37, 16, 8	11.60	58.27	73.55	58.23	99.80	3,991	2,863	18	8
		Route Views	37, 16, 8	7.62	58.20	73.55	58.18	99.98	3,999	2,875	5	1
Incr. VCFBLS	none	RIPE	37, 16, 8	12.04	58.23	73.57	58.19	99.98	3,999	2,873	7	1
		Route Views	37, 16, 8	9.08	58.30	73.57	58.25	99.85	3,994	2,863	17	6
	iForest	RIPE	37, 16, 8	11.27	58.15	73.53	58.14	99.98	3,998	2,878	3	1
		Route Views	37, 16, 8	10.40	58.20	73.56	58.17	100	4,000	2,876	4	0
XGBoost	none	RIPE	8	0.54	60.44	73.38	60.26	93.80	3,752	2,474	406	248
		Route Views	37	0.27	55.83	70.94	57.44	92.73	3,709	2,748	132	291
	iForest	RIPE	8	0.52	59.84	73.05	59.88	93.62	3,744	2,508	373	255
		Route Views	8	0.38	55.58	70.42	57.46	90.93	3,637	2,693	187	363
LightGBM	none	RIPE	16	0.05	58.37	72.20	59.01	92.98	3,719	2,583	297	281
		Route Views	8	0.06	57.50	72.16	58.27	94.73	3,789	2,713	167	211
	iForest	RIPE	37	0.10	57.66	71.42	58.77	91.02	3,640	2,554	327	359
		Route Views	16	0.05	57.72	72.81	58.14	97.38	3,895	2,804	76	105
CatBoost	none	RIPE	8	0.33	55.60	71.36	57.09	95.15	3,806	2,861	19	194
		Route Views	8	0.31	58.17	73.53	58.16	99.95	3,998	2,876	4	2
	iForest	RIPE	16	0.32	55.58	71.34	57.07	95.12	3,804	2,861	20	195
		Route Views	8	0.48	58.24	73.53	58.22	99.78	3,991	2,864	16	9

Chapter 7

Conclusion and Future Work

In this thesis, we implemented and compared supervised and semi-supervised machine learning algorithms for detecting ransomware attacks using BGP routing records. In order to enable a comprehensive analysis of routing information at the onset of Internet anomalies and intrusions, we also developed a relational database *BGP-RDB* to store BGP routing records.

Data modeling of the *BGP-RDB* database was based on the BGP *open*, *update*, *keepalive*, and *notification* messages as well as the state transitions of TCP connections captured from RIPE and Route Views data collection sites. Analysis and inclusion of BGP messages and operational status of peers may improve anomaly detection using machine learning algorithms. The database was developed using *sqlite3* Python module in order to enable easy integration with Python-based anomaly detection systems. Development of the *BGP-RDB* database required to first understand information contained in the fields of BGP message types as well as the type of data present in the fields (integer, datetime, text/strings). We then defined various types of tables (core, lookup, list, detail) used to store BGP routing records and their relationships (one-to-many).

Detection of WannaCrypt and WestRock ransomware attacks using BGP routing records was performed using supervised and semi-supervised machine learning techniques. BGP datasets were generated based on BGP *update* messages from the RIPE and Route Views data collection sites. In the case of supervised machine learning techniques, we evaluated performance of RNN (LSTM, GRU), Bi-RNN (Bi-LSTM, Bi-GRU), BLS and its extensions (incremental learning, RBF-BLS, CFBLS, CEBLS, CFEBLS), BLS with variable features (VFBLS, VCFBLS), and GBDT (XBoost, LightGBM, CatBoost) models. Semi-supervised machine learning was implemented by using the iForest unsupervised algorithm for label refinement of data points during the periods of anomalies while classification was based on the supervised algorithms (RNNs, Bi-RNNs, BLS, VFBLS, VCFBLS, GBDT). Performance evaluation was based on training time, accuracy, F-Score, precision, sensitivity (recall), and confusion matrix. The semi-supervised technique based on label refinement only offered comparable classification performance when compared to supervised machine learning. The best performance (accuracy, F-Score) was generated using WannaCrypt and WestRock BGP *update* messages collected from Route Views and RIPE, respectively. Bi-GRU$_4$ model generated the best classification performance for both datasets. Developing RNN and Bi-RNN models with a large number of hidden layers required longer training time. Implemented BLS models with incremental learning required only dynamic update of output weights instead of retraining the model. These models often required comparable training time when compared to RNN and Bi-RNN models. In most cases, GBDT models offered comparable classification performance with BLS. They consisted of a small number of parameters and hyper-parameters and, hence, required shorter training times.

Future work may include generating new features based on BGP *open*, *keepalive*, and *notification* messages as well as transitions in the BGP FMS based on the developed *BGP-RDB* database. The existing BGP datasets currently generated using the *zebra-dump-parser* and C# tools should be re-created by querying the *BGP-RDB* database. Finally, the *BGP-RDB* database may be integrated into real-time anomaly detection systems.

Future work for detecting BGP anomalies should explore other anomaly detection unsupervised algorithms (local outlier factor, one-class SVM) and clustering algorithms (k-means, density-based spatial clustering of applications with noise, Bayesian networks) for label refinement. Of particular interest is implementation of various feature selection algorithms (autoencoders, Fisher, decision trees) to be integrated with VFBLS and VCFBLS algorithms.

Bibliography

[1] (2022, Aug.) A Border Gateway Protocol 4 (BGP-4). [Online]. Available: https://datatracker.ietf.org/doc/html/rfc4271.

[2] (2022, Aug.) BGP routing protocol overview. [Online]. Available: http://gponsolution.com/bgp-routing-protocol-overview.html.

[3] (2022, Aug.) Understanding LSTM networks. [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs.

[4] C. L. P. Chen and Z. Liu, "Broad learning system: an effective and efficient incremental learning system without the need for deep architecture," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 1, pp. 10–24, Jan. 2018.

[5] C. L. P. Chen, Z. Liu, and S. Feng, "Universal approximation capability of broad learning system and its structural variations," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 4, pp. 1191–1204, Apr. 2019.

[6] Z. Li, A. L. Gonzalez Rios, and Lj. Trajković, "Machine learning for detecting anomalies and intrusions in communication networks," *IEEE J. Sel. Areas Commun.*, vol. 39, no. 7, pp. 2254–2264, July 2021.

[7] B. Al-Musawi, P. Branch, and G. Armitage, "BGP anomaly detection techniques: a survey," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 1, pp. 377–396, 2017.

[8] Y. Song, A. Venkataramani, and L. Gao, "Identifying and addressing reachability and policy attacks in 'secure' BGP," *IEEE/ACM Trans. Netw.*, vol. 24, no. 5, pp. 2969–2982, Oct. 2016.

[9] D. Dolev, S. Jamin, O. Mokryn, and Y. Shavitt, "Internet resiliency to attacks and failures under BGP policy routing," *Comput. Netw.*, vol. 50, no. 16, pp. 3183–3196, Nov. 2006.

[10] A. Lutu, M. Bagnulo, C. Pelsser, O. Maennel, and J. Cid-Sueiro, "The BGP visibility toolkit: detecting anomalous internet routing behavior," *IEEE/ACM Trans. Netw.*, vol. 24, no. 2, pp. 1237–1250, Apr. 2016.

[11] (2022, Aug.) MS SQL Slammer/Sapphire worm, SANS Institute GIAC Certifications. [Online]. Available: https://www.giac.org/paper/gsec/3091/ms-sql-slammer-sapphire-worm/105136.

[12] (2022, Aug.) Attack of Slammer worm - a practical case study, SANS Institute GIAC Certifications. [Online]. Available: https://www.giac.org/paper/gcih/414/attack-slammer-worm-practical-case-study/103632.

[13] (2022, Aug.) Responding to the Nimda worm: recommendations for addressing blended threats, Symantec, Cupertino, CA, USA. [Online]. Available: https://vxug.fakedoma.in/archive/Symantec/nimda-worm-recommendations-blended-threats-01-en.pdf.

[14] (2022, Aug.) A challenging response to Nimda, SANS Institute GIAC Certifications. [Online]. Available: https://www.giac.org/paper/gcih/273/challenging-response-nimda/102847.

[15] (2022, Aug.) The Code Red worm, SANS Institute Information Security Reading Room. [Online]. Available: https://sansorg.egnyte.com/dl/TJ3l4DtpBX.

[16] (2022, Aug.) How ransomware attacks, SophosLabs. [Online]. Available: https://www.sophos.com/en-us/medialibrary/pdfs/technical-papers/sophoslabs-ransomware-behavior-report.pdf.

[17] (2022, Aug.) WestRock provides update on ransomware incident. [Online]. Available: https://ir.westrock.com/press-releases/press-release-details/2021/WestRock-Provides-Update-on-Ransomware-Incident-8dfde2fca/default.aspx.

[18] (2022, Aug.) Packaging giant WestRock says ransomware attack impacted OT systems. [Online]. Available: https://www.securityweek.com/packaging-giant-westrock-says-ransomware-attack-impacted-ot-systems.

[19] R. Mahajan, D. Wetherall, and T. Anderson, “Understanding BGP misconfiguration,” in *Proc. 2002 Conf. Appl. Technologies Architectures Protocols Comput. Commun.*, Pittsburgh, Pennsylvania, USA, Aug. 2002, pp. 3–16.

[20] C. Testart, P. Richter, A. King, A. Dainotti, and D. Clark, “Profiling BGP serial hijackers: capturing persistent misbehavior in the global routing table,” in *Proc. ACM Internet Meas. Conf.*, Amsterdam, Netherlands, Oct. 2019, pp. 420–434.

[21] J. L. Sobrinho and T. Quelhas, “A theory for the connectivity discovered by routing protocols,” *IEEE/ACM Trans. Netw.*, vol. 20, no. 3, pp. 677–689, June 2012.

[22] (2022, Aug.) RAO “UES of Russia” annual report 2005. [Online]. Available: http://www.rustocks.com/put.phtml/EESR_2005_sec.pdf.

[23] (2022, Aug.) Report on the investigation of the accident in the UES of Russia on May 25, 2005. [Online]. Available: http://www.kef.ru/art_010.shtml.

[24] (2022, Aug.) Explained: What was the cause of Pakistan’s nationwide electricity outage? [Online]. Available: https://indianexpress.com/article/explained/explained-what-led-to-the-nationwide-power-outage-in-pakistan-7140969.

[25] A. Young and M. Yung, “Cryptovirology: extortion-based security threats and countermeasures,” in *Proc. 1996 IEEE Symp. Secur. Privacy*, Oakland, CA, USA, May 1996, pp. 129–140.

[26] (2022, Aug.) Understanding ransomware and strategies to defet it. [Online]. Available: https://www.mcafee.com/enterprise/en-us/assets/white-papers/wp-understanding-ransomware-strategies-defeat.pdf.

[27] (2022, Aug.) The next ransomware attack is likely to be launched using an actual employee's credentials: How to accurately underwrite cybersecurity insurance. [Online]. Available: https://www.marshall.usc.edu/sites/default/files/2022-03/Experian-Cyber-White-Paper.pdf.

[28] (2022, Aug.) The state of ransomware 2022. [Online]. Available: https://assets.sophos.com/X24WTUEQ/at/4zpw59pnkpxxnhfhgj9bxgj9/sophos-state-of-ransomware-2022-wp.pdf.

[29] A. Tandon and A. Nayyar, "A comprehensive survey on ransomware attack: a growing havoc cyberthreat," in *Proc. Int. Conf. Data Manage., Analytics, Innov.*, Pune, India, Jan. 2018, pp. 403–420.

[30] C. M. Bishop, *Pattern Recognition and Machine Learning.* Secaucus, NJ, USA: Springer-Verlag, 2006.

[31] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* Cambridge, MA, USA: The MIT Press, 2016.

[32] K. P. Murphy, *Machine Learning: A Probabilistic Perspective.* Cambridge, MA, USA: The MIT Press, 2012.

[33] ——, *Probabilistic Machine Learning: An introduction.* Cambridge, MA, USA: The MIT Press, 2022.

[34] J. E. van Engelen and H. H. Hoos, "A survey on semi-supervised learning," *Mach. Learn.*, vol. 109, no. 2, p. 373–440, Feb. 2020.

[35] Q. Ding, Z. Li, S. Haeri, and L. Trajković, "Application of machine learning techniques to detecting anomalies in communication networks," in *Cyber Threat Intelligence*, A. Dehghantanha, M. Conti, and T. Dargahi, Eds. Berlin: Springer, 2018, pp. 47–70 and pp. 71–92.

[36] C. Cortes and V. Vapnik, "Support-vector networks," *J. Mach. Learn.*, vol. 20, no. 3, pp. 273–297, Sept. 1995.

[37] (2022, Aug.) Broadlearning. [Online]. Available: http://www.broadlearning.ai.

[38] T. Chen and C. Guestrin, "XGBoost: a scalable tree boosting system," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, San Francisco, CA, USA, Aug. 2016, pp. 785–794.

[39] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "LightGBM: a highly efficient gradient boosting decision tree," in *Proc. Int. Conf. Neural Inform. Process. Syst.*, Long Beach, CA, USA, Dec. 2017, pp. 3146–3154.

[40] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin, "CatBoost: unbiased boosting with categorical features," in *Proc. Int. Conf. Neural Inform. Process. Syst.*, Montreal, Québec, Canada, Dec. 2018, pp. 6639–6649.

[41] (2022, Aug.) RIPE NCC. [Online]. Available: https://www.ripe.net/analyse.

[42] (2022, Aug.) University of Oregon Route Views projects. [Online]. Available: http://www.routeviews.org.

[43] (2022, Aug.) A Border Gateway Protocol (BGP). [Online]. Available: https://datatracker.ietf.org/doc/html/rfc1105.

[44] (2022, July) Key words for use in RFCs to indicate requirement levels. [Online]. Available: https://www.ietf.org/rfc/rfc2119.txt.

[45] A. M. Truing, "Computing machinery and intelligence," *Mind*, vol. LIX, no. 236, pp. 433–460, Oct. 1950.

[46] F. Chollet, *Deep Learning with Python.* New York, NY, USA: Manning Publications, 2017.

[47] G. Bonaccorso, *Hands-On Unsupervised Learning with Python.* New York, NY, USA: Manning Publications, 2017.

[48] T. M. Mitchell, *Machine Learning.* New York, NY, USA: McGraw-Hill, 1997.

[49] (2022, Aug.) AI data engineering lifecycle checklist: following steps for AI project sucess. [Online]. Available: https://www.cloudera.com/content/dam/www/marketing/resources/whitepapers/ai-data-lifecycle-checklist-cloudera-whitepaper.pdf?daqp=true.

[50] S. Clémençon and C. Jakubowicz, "Scoring anomalies: a M-estimation formulation," in *16th Int. Conf. Artif. Intell. Statis.*, Scottsdale, Arizona, USA, Apr. 2013, pp. 659–667.

[51] N. Goix, A. Sabourin, and S. Clémençon, "On anomaly ranking and excess-mass curves," in *18th Int. Conf. Artif. Intell. Statis.*, San Diego, CA, USA, May. 2015, pp. 287–295.

[52] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: a survey," *ACM Comput. Surv.*, vol. 41, no. 3, pp. 15:1–15:58, July 2009.

[53] A. Taha and A. S. Hadi, "Anomaly detection methods for categorical data: A review," *ACM Comput. Surv.*, vol. 52, no. 2, pp. 1–35, Mar. 2020.

[54] S. García, J. Luengo, J. A. Sáez, V. López, and F. Herrera, "A survey of discretization techniques: Taxonomy and empirical analysis in supervised learning," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 4, pp. 734–750, Apr. 2013.

[55] J.-P. Schulze, A. Mrowca, E. Ren, H.-A. Loeliger, and K. Böttinger, "Context by proxy: Identifying contextual anomalies using an output proxy," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Anchorage, AK, USA, Aug. 2019, p. 2059–2068.

[56] M. A., J. Hayes, and M. A. Capretz, "Contextual anomaly detection framework for big sensor data," *J. Big Data*, vol. 2, no. 2, pp. 1–22, Feb. 2015.

[57] X. Song, M. Wu, C. Jermaine, and S. Ranka, "Conditional anomaly detection," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 5, pp. 631–645, May 2007.

[58] M. C. Libicki, L. Ablon, and T. Webb, *The Defender's Dilemma: Charting a Course Toward Cybersecurity.* Santa Monica, CA, USA: RAND Corporation, June 2015.

[59] J. Zhang, J. Rexford, and J. Feigenbaum, "Learning-based anomaly detection in BGP updates," in *Proc. Workshop Mining Netw. Data*, Philadelphia, PA, USA, Aug. 2005, pp. 219–220.

[60] (2022, Aug.) What are signatures and how does signature-based detection work? [Online]. Available: https://home.sophos.com/en-us/security-news/2020/what-is-a-signature.

[61] T. Ahmed, B. Oreshkin, and M. Coates, "Machine learning approaches to network anomaly detection," in *Proc. USENIX Workshop Tackling Comput. Syst. Problems with Mach. Learn. Techn.*, Cambridge, MA, USA, Apr. 2007, pp. 1–6.

[62] P. Mishra, V. Varadharajan, U. Tupakula, and E. S. Pilli, "A detailed investigation and analysis of using machine learning techniques for intrusion detection," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 1, pp. 686–728, First Quarter 2019.

[63] P. García-Teodoro, J. Díaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, "Anomaly-based network intrusion detection: techniques, systems and challenges," *Computers & Security*, vol. 28, no. 1–2, pp. 18–28, Feb.–Mar. 2009.

[64] R. Samrin and D. Vasumathi, "Review on anomaly based network intrusion detection system," in *Proc. Int. Conf. Elect., Electron., Commun., Comput., Optim. Techn.*, Mysuru, India, Dec. 2017, pp. 141–147.

[65] F. A. R. G. Muruti and Z. bin Ibrahim, "A survey on anomalies detection techniques and measurement methods," in *Proc. 2018 IEEE Conf. Appl. Inf. Netw. Secur.*, Langkawi, Malaysia, Nov. 2018, pp. 81–86.

[66] A. L. Buczak and E. Guven, "A survey of data mining and machine learning methods for cyber security intrusion detection," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 2, pp. 1153–1176, 2016.

[67] W. Alhakami, A. Alharbi, R. A. S. Bourouis, and N. Bouguila, "Network anomaly intrusion detection using a nonparametric bayesian approach and feature selection," *IEEE Access*, vol. 7, pp. 52 181–52 190, Apr. 2019.

[68] M. Labonne, A. Olivereau, B. Polvé, and D. Zeghlache, "A cascade-structured meta-specialists approach for neural network-based intrusion detection," in *Proc. IEEE Annu. Consum. Commun. Netw. Conf.*, Las Vegas, NV, USA, Jan. 2019, pp. 1–6.

[69] J. Du, C. Jiang, J. Wang, Y. Ren, and M. Debbah, "Machine learning for 6G wireless networks: carrying forward enhanced bandwidth, massive access, and ultrareliable/low-latency service," *IEEE Veh. Technol. Mag.*, vol. 15, no. 4, pp. 122–134, Dec. 2020.

[70] G. Karatas, O. Demir, and O. K. Sahingoz, “Deep learning in intrusion detection systems,” in *Proc. Int. Congr. Big Data, Deep Learn. Fighting Cyber Terrorism*, Ankara, Turkey, Dec. 2018, pp. 113–116.

[71] S. Naseer, Y. Saleem, S. Khalid, M. K. Bashir, J. Han, M. M. Iqbal, and K. Han, “Enhanced network anomaly detection based on deep neural networks,” *IEEE Access*, vol. 6, pp. 48 231–48 246, Aug. 2018.

[72] C. Yin, Y. Zhu, J. Fei, and X. He, “A deep learning approach for intrusion detection using recurrent neural networks,” *IEEE Access*, vol. 5, pp. 21 954–21 961, Nov. 2017.

[73] J. P. A. Maranhão, J. P. C. L. da Costa, E. P. de Freitas, E. Javidi, and R. T. de Sousa, Jr., “Noise-robust multilayer perceptron architecture for distributed denial of service attack detection,” *IEEE Commun. Lett.*, vol. 25, no. 2, pp. 402–406, Feb. 2021.

[74] T. Kim, S. C. Suh, H. Kim, J. Kim, and J. Kim, “An encoding technique for CNN-based network anomaly detection,” in *Proc. IEEE Int. Conf. Big Data*, Seattle, WA, USA, Dec. 2018, pp. 2960–2965.

[75] N. Shone, T. N. Ngoc, V. D. Phai, and Q. Shi, “A deep learning approach to network intrusion detection,” *IEEE Trans. Emerg. Topics Comput. Intell.*, vol. 2, no. 1, pp. 41–50, Feb. 2018.

[76] Y. Jia, M. Wang, and Y. Wang, “Network intrusion detection algorithm based on deep neural network,” *IET Inf. Secur.*, vol. 13, no. 1, pp. 48–53, Jan. 2019.

[77] D. J. Weller-Fahy, B. J. Borghetti, and A. A. Sodemann, “A survey of distance and similarity measures used within network intrusion anomaly detection,” *IEEE Commun. Surveys Tuts.*, vol. 17, no. 1, pp. 70–91, 2015.

[78] T. A. Tang, L. Mhamdi, D. McLernon, S. A. R. Zaidi, and M. Ghogho, “Deep learning approach for network intrusion detection in software defined networking,” in *Proc. Wireless Netw. Mobile Commun.*, Fez, Morocco, Oct. 2016, pp. 258–263.

[79] J. Zhang and M. Zulkernine, “A hybrid network intrusion detection technique using random forests,” in *Proc. First Int. Conf. Availability, Rel. Secur.*, Vienna, Austria, Apr. 2006, pp. 262–269.

[80] Q. Ding, Z. Li, S. Haeri, and Lj. Trajković, “Application of machine learning techniques to detecting anomalies in communication networks: datasets and feature selection algorithms,” in *Cyber Threat Intelligence*, A. Dehghantanha, M. Conti, and T. Dargahi, Eds. Berlin: Springer, 2018, pp. 47–70.

[81] (2022, Aug.) BCNET. [Online]. Available: http://www.bc.net.

[82] (2022, Aug.) NSL-KDD Data Set. [Online]. Available: https://www.unb.ca/cic/datasets/nsl.html.

[83] (2022, Aug.) Canadian Institute for Cybersecurity datasets. [Online]. Available: https://www.unb.ca/cic/datasets/index.html.

[84] J. Woo, J. Song, and Y. Choi, “Performance enhancement of deep neural network using feature selection and preprocessing for intrusion detection,” in *Proc. Int. Conf. Artif. Intell. Inform. Commun.*, Okinawa, Japan, Feb. 2019, pp. 415–417.

[85] K. A. Taher, B. M. Y. Jisan, and M. M. Rahman, “Network intrusion detection using supervised machine learning technique with feature selection,” in *Proc. 2019 Int. Conf. Robot., Elect. and Signal Process. Techn.*, Dhaka, Bangladesh, Jan. 2019, pp. 643–646.

[86] (2022, Aug.) Border Gateway Protocol Routing Records from Réseaux IP Européens (RIPE) and BCNET. [Online]. Available: http://ieee-dataport.org/1977.

[87] (2022, Aug.) mrtparse. [Online]. Available: https://github.com/t2mune/mrtparse.

[88] (2022, Aug.) SQLite. [Online]. Available: https://www.sqlite.org/index.html.

[89] H. Yan, R. Oliveira, K. Burnett, D. Matthews, L. Zhang, and D. Massey, “BGPmon: a real-time, scalable, extensible monitoring system,” in *Proc. Cybersecurity Appl. Technol. Conf. Homeland Secur.*, Washington, DC, USA, Mar. 2009, pp. 212–223.

[90] K. Sriram, O. Borchert, O. Kim, P. Gleichmann, and D. Montgomery, “A comparative analysis of BGP anomaly detection and robustness algorithms,” in *Proc. Cybersecurity Appl. Technol. Conf. Homeland Secur.*, Washington, DC, USA, Mar. 2009, pp. 25–38.

[91] B. Zhang, R. Liu, D. Massey, and L. Zhang, “Collecting the Internet AS-level topology,” *ACM Computer Communication Review (CCR)*, vol. 35, no. 1, pp. 53–62, Jan. 2005.

[92] (2022, Aug.) Mediterranean fibre cable cut - a RIPE NC analysis. [Online]. Available: https://www.ripe.net/analyse/archived-projects/mediterranean-fibre-cable-cut.

[93] (2022, Aug.) Pakistan Internet connectivity collapses amid nation-scale power outage. [Online]. Available: https://netblocks.org/reports/pakistan-internet-connectivity-collapses-amid-nation-scale-power-outage-YAEvvoB3.

[94] (2022, Aug.) Malware 101 - Viruses, SANS Institute. [Online]. Available: https://sansorg.egnyte.com/dl/PNA5KifxmM.

[95] (2022, Aug.) Threat chaos: making sense of the online threat landscape, Webroot Software, Inc. [Online]. Available: https://www.webroot.com/pdf/WP_Threat_0105.pdf.

[96] (2022, Aug.) Quagga routing suite. [Online]. Available: https://www.nongnu.org/quagga/docs/quagga.html.

[97] (2022, Aug.) Zebra. [Online]. Available: http://www.zebra.org.

[98] (2022, Aug.) Multi-threaded routing toolkit (MRT) border gateway protocol (BGP) routing information export format with geo-location extensions. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6397.

[99] (2022, Aug.) RIPE NCC routing information service. [Online]. Available: https://www.ripe.net/publications/docs/ripe-200.

[100] (2022, Aug.) zebra-dump-parser. [Online]. Available: https://github.com/rfc1036/zebra-dump-parser.

[101] (2022, Aug.) BGP C sharp tool. [Online]. Available: https://github.com/communication-networks-laboratory/BGP_c_sharp_tool.

[102] N. Al-Rousan, S. Haeri, and Lj. Trajković, "Feature selection for classification of bgp anomalies using bayesian models," in *Proc. Int. Conf. Mach. Learn. Cybern.*, Xi'an, China, July 2012, pp. 140–147.

[103] N. Al-Rousan and Lj. Trajković, "Machine learning models for classification of BGP anomalies," in *Proc. IEEE Conf. High Perform. Switching Routing*, Belgrade, Serbia, June 2012, pp. 103–108.

[104] (2022, Aug.) Autonomous system (AS) numbers. [Online]. Available: https://www.iana.org/assignments/as-numbers/as-numbers.xhtml.

[105] (2022, Aug.) Cisco Networking Academy's introduction to routing concepts. [Online]. Available: https://www.ciscopress.com/articles/article.asp?p=2180208&seqNum=6.

[106] (2022, Aug.) IP routing: BGP configuration guide, Cisco IOS release 15M&T. [Online]. Available: https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/iproute_bgp/configuration/15-mt/irg-15-mt-book/irg-router-id.html.

[107] (2022, Aug.) Capabilities advertisement with BGP-4. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc3392.

[108] (2022, Aug.) Autonomous system confederations for BGP. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc5065.

[109] (2022, Aug.) BGP confederations for IBGP scaling. [Online]. Available: https://www.juniper.net/documentation/us/en/software/junos/bgp/topics/topic-map/bgp-confederations-for-scaling.html.

[110] (2022, Aug.) BGP fundamentals. [Online]. Available: https://www.ciscopress.com/articles/article.asp?p=2756480&seqNum=4.

[111] C. R. Severance, S. Blumenberg, and E. Hauser, *Python for everybody: exploring data in Python 3*. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2016.

[112] (2022, Aug.) Oracle database. [Online]. Available: https://www.oracle.com/ca-en/database.

[113] (2022, Aug.) MySQL database. [Online]. Available: https://www.mysql.com.

[114] (2022, Aug.) Microsoft sql server. [Online]. Available: https://www.microsoft.com/en-ca/sql-server.

[115] (2022, Aug.) PostgreSQL. [Online]. Available: https://www.postgresql.org.

[116] (2022, Aug.) sqlite3 — db-api 2.0 interface for sqlite databases. [Online]. Available: https://docs.python.org/3/library/sqlite3.html.

[117] (2022, Aug.) Petya ransomware goes low level. [Online]. Available: https://www.bitdefender.com/blog/labs/petya-ransomware-goes-low-level.

[118] (2022, Aug.) Stemming the exploitation of ict threats and vulnerabilities, United Nations Institute for Disarmament Research (UNIDIR). [Online]. Available: https://unidir.org/files/publications/pdfs/stemming-the-exploitation-of-ict-threats-and-vulnerabilities-en-805.pdf.

[119] (2022, Aug.) Reverse engineering of WannaCry worm and anti exploit snort rules, SANS Institute. [Online]. Available: https://sansorg.egnyte.com/dl/ebq95gaCdT.

[120] (2022, Aug.) Player 3 has entered the game: Say hello to WannaCry. [Online]. Available: https://blog.talosintelligence.com/2017/05/wannacry.html.

[121] (2022, Aug.) Exploits explained: Comprehensive exploit prevention. [Online]. Available: https://assets.sophos.com/X24WTUEQ/at/cpbtrb4w4mfxqf4bn5cx8fh/sophos-comprehensive-exploit-prevention-wp.pdf.

[122] (2022, Aug.) EternalBlue: a prominent threat actor of 2017–2018, Virus Bulletin. [Online]. Available: https://www.virusbulletin.com/uploads/pdf/magazine/2018/201806-EternalBlue.pdf.

[123] (2022, Aug.) Technical whitepaper tracking the WannaCry ransomware, Nominet. [Online]. Available: https://satisnet.co.uk/wp-content/uploads/2019/04/WannaCry-Whitepaper.pdf.

[124] (2022, Aug.) Westrock. [Online]. Available: https://ir.westrock.com.

[125] (2022, Aug.) RRC04 – CIXP, Geneva, Switzerland – Peer List. [Online]. Available: https://www.ris.ripe.net/peerlist/rrc04.shtml.

[126] (2022, Aug.) RouteViews Collector Map. [Online]. Available: http://www.routeviews.org/routeviews/index.php/map.

[127] (2022, Aug.) RRC14 – PAIX, Palo Alto, California, US – Peer List. [Online]. Available: https://www.ris.ripe.net/peerlist/rrc14.shtml.

[128] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Oct. 1997.

[129] K. Greff, R. K. Srivastava, J. Koutnik, B. R. Steunebrink, and J. Schmidhuber, "LSTM: a search space odyssey," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 28, no. 10, pp. 2222–2232, Oct. 2017.

[130] K. Cho, B. van Merriënboer, C. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translations," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, Doha, Qatar, Oct. 2014, pp. 1724–1734.

[131] (2022, Aug.) PyTorch. [Online]. Available: https://pytorch.org/docs/stable/nn.html.

[132] Z. Liu and C. L. P. Chen, "Broad learning system: structural extensions on single-layer and multi-layer neural networks," in *Proc. Int. Conf. Secur., Pattern Anal., Cybern.*, Shenzhen, China, Dec. 2017, pp. 136–141.

[133] A. L. Gonzalez Rios, G. X. Z. Li, A. D. Alonso, and Lj. Trajković, "Detecting network anomalies and intrusions in communication networks," in *Proc. 23rd IEEE Int. Conf. Intell. Eng. Syst.*, Gödöllő, Hungary, Apr. 2019, pp. 29–34.

[134] A. L. Gonzalez Rios, Z. Li, K. Bekshentayeva, and Lj. Trajković, "Detection of denial of service attacks in communication networks," in *Proc. IEEE Int. Symp. Circuits Syst.*, Seville, Spain, Oct. 2020.

[135] Z. Li, P. Batta, and Lj. Trajković, "Comparison of machine learning algorithms for detection of network intrusions," in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Miyazaki, Japan, Oct. 2018, pp. 4248–4253.

[136] Z. Li, A. L. Gonzalez Rios, G. Xu, and Lj. Trajković, "Machine learning techniques for classifying network anomalies and intrusions," in *Proc. IEEE Int. Symp. Circuits Syst.*, Sapporo, Japan, May 2019, pp. 1–5.

[137] Y.-H. Pao, G.-H. Park, and D. J. Sobajic, "Learning and generalization characteristics of the random vector functional-link net," *Neurocomputing*, vol. 6, no. 2, pp. 163–180, Apr. 1994.

[138] P. Geurts, D. Ernst, and L.Wehenkel, "Extremely randomized trees," *Mach. Learn.*, vol. 63, no. 1, pp. 3–42, Apr. 2006.

[139] G. Louppe, L. Wehenkel, A. Sutera, and P. Geurts, "Understanding variable importances in forests of randomized trees," in *Proc. Int. Conf. Neural Inform. Process. Syst.*, Lake Tahoe, NV, USA, Dec. 2013, pp. 431–439.

[140] J. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of Statistics*, vol. 29, no. 5, pp. 1189–1232, Apr. 2001.

[141] F. T. Liu, K. M. Ting, and Z. Zhou, "Isolation forest," in *IEEE Int. Conf. Data Mining*, Pisa, Italy, Dec. 2008, pp. 413–422.

[142] (2022, Aug.) Compute canada. [Online]. Available: https://ccdb.computecanada.ca/security/login.

[143] (2022, Aug.) Cedar. [Online]. Available: https://docs.computecanada.ca/wiki/Cedar.

[144] (2022, Aug.) Python package index. [Online]. Available: https://pypi.org.

[145] D. P. Kingma and J. Ba, "Adam: a method for stochastic optimization," in *Proc. 3rd Int. Conf. Learn. Representations*, San Diego, CA, USA, May 2015, pp. 1–15.

[146] (2022, Aug.) Trusted artificial intelligence: Towards certification of machine learning applications. [Online]. Available: https://arxiv.org/pdf/2103.16910.pdf.

[147] (2022, Aug.) Classification: Accuracy. [Online]. Available: https://developers.google.com/machine-learning/crash-course/classification/accuracy.

[148] (2022, Aug.) Handling imbalanced datasets in machine learning. [Online]. Available: https://towardsdatascience.com/handling-imbalanced-datasets-in-machine-learning-7a0e84220f28.

[149] (2022, Aug.) Oversampling and undersampling. [Online]. Available: https://towardsdatascience.com/oversampling-and-undersampling-5e2bbaf56dcf.

[150] (2022, Aug.) Metrics for multi-class classification: an overview. [Online]. Available: https://arxiv.org/pdf/2008.05756.pdf.

[151] (2022, Aug.) Classification: Precision and recall. [Online]. Available: https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall.

[152] (2022, Aug.) Machine learning – sensitivity vs specificity difference. [Online]. Available: https://vitalflux.com/ml-metrics-sensitivity-vs-specificity-difference.

Appendix A

BGP-RDB: Data Download Module

Listed are functions used to retrieve a single repository containing BGP routing record from RIPE and Route Views collection sites. The function *updateMessageName* generates the file name of a BGP repository based on the year (yyyy), month (mm), hour (hh), and minute (mm) strings. Its output is a 2-tuple with the strings: *updates.yyyymmdd.hhmm* and *yyyy.mm*. Function *data_downloader_single* is used to retrieve a BGP repository by specifying the file name (updates.yyyymmdd.hhmm), date of required file (yyyy.mm), site (RIPE or RouteViews), and name of the collector.

Listing A.1: BGP-RDA Module to Retrieve BGP Messages from RIPE and Route Views Collection Sites.

```python
"""
    @author Ana Laura Gonzalez Rios
    @email anag@sfu.ca
    @date June 2022
    @version: 1.0.0
    @description:
                This file contains functions to download BGP repositories
                from RIPE or Route Views collection sites.
    @copyright Copyright (c) June 2022
        All Rights Reserved

    Python code (version 3.10.2)
"""

import requests

# Name of the update_message_file generation
def updateMessageName(year, month, day, hour, minute):
    # updates.YYYYMMDD.HHMM.gz,  minute: 5 minutes interval

    data_date = "%s.%s" % (year, month)  # used for data_link in Function
    data_downloader().

    update_message_file = "updates.%s%s%s.%s%s" % (year, month, day, hour,
    minute)

    return update_message_file, data_date

# Download specific file from RIPE or RouteViews
def data_downloader_single(update_message_file, data_date, site,
    collector_ripe = 'rrc04',collector_routeviews = 'route-views2'):

    data_file = update_message_file

    if site == 'RIPE':
        data_link = 'http://data.ris.ripe.net/%s/%s/%s.gz' % (collector_ripe
    , data_date, data_file)
        print (data_link)
        r = requests.get(data_link, allow_redirects = True)
        open('%s.gz' % (data_file), 'wb').write(r.content)

    if site == 'RouteViews':
        if collector_routeviews == 'route-views2': collector_routeviews = ''
        data_link = 'http://archive.routeviews.org/%s/bgpdata/%s/UPDATES/%s.
    bz2' % (collector_routeviews,data_date,data_file)
        print(data_link)
        r = requests.get(data_link, allow_redirects = True)
        # add condition, if request is 200 then save file, otherwhise,
    return reponse code
        open('%s.bz2' % (data_file), 'wb').write(r.content)
```

Appendix B

BGP-RDB: Modified mrt2yaml.py Script

Listed is the modified script *mrt2yaml.py* used to convert MRT data to ASCII and generate YAML files to be employed as input to the ingestion engine. Parsed data are first mapped to Python ordered dictionaries and then inserted in the YAML file with indentations that indicate lists and other dictionaries contained in parsed BGP message. MRT file to be parsed and destination YAML file are required inputs to the main function *mrt2yaml*. This function inserts parsed lines from the MRT file into the YAML file. Functions *represent_ordereddict*, *dict_representer*, and *dict_constructor* are used to ensure that elements of the ordered dictionaries are correctly formatted in the YAML file.

Listing B.1: Script *mrt2yaml.py* to Export Parsed MRT Data to a YAML File.

```python
#!/usr/bin/python

import yaml
import sys
import collections
from mrtparse import *

def represent_ordereddict(dumper, instance):
    return dumper.represent_mapping('tag:yaml.org,2002:map', instance.items
    ())

yaml.add_representer(collections.OrderedDict, represent_ordereddict)

def dict_representer(dumper, data):
    return dumper.represent_dict(data.iteritems())

def dict_constructor(loader, node):
    return collections.OrderedDict(loader.construct_pairs(node))

def mrt2yaml(file_to_parse,dest_file):
    with open(dest_file,'w') as file_descriptor:
        file_descriptor.write('---' + '\n')
        print('---')
        ord_dics = []
        for entry in Reader(file_to_parse):
            #print(yaml.dump([entry.data])[:-1])
            yaml.dump([entry.data],file_descriptor)
        file_descriptor.write('...' + '\n')
```

Appendix C

BGP-RDB: Ingestion Engine Functions

Listed are functions used to initialize tables and insert data entries in core and list tables. Initialization functions for lookup and detail tables include insertion of logic keys. Function *create_database* is employed to initialize tables when creating a new BGP database. The function may be executed by invoking *ingestion.py*.

Listing C.1: *BGP-RDA* Ingestion Engine Functions: Tables Initialization and Data Entry.

```
"""
    @author Ana Laura Gonzalez Rios
    @email anag@sfu.ca
    @date June 2022
    @version: 1.0.0
    @description:
                This file contains methods to create DB tables for collected
                BGP messagesas well as to insert data in tables of an
                existing BGP database.

    @copyright Copyright (c) June 2022
        All Rights Reserved

    Python code (version 3.10.2)
    SQLite3 commands (version 3.28.0)
"""

import sqlite3
from sqlite3 import Error

# Function to create connection with specified DB (db_file)
def create_connection(db_file):
    """ create a database connection to a SQLite database """
    conn = None
    try:
        conn = sqlite3.connect(db_file)
        print("Successfully connected to Database! Database version is: ",
    sqlite3.version)
        return conn
    except Error as e:
        print(e)

# Function to create table 'mrt_types.' If table exists, it will be first
    removed.
def create_mrttype_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript('''
    DROP TABLE IF EXISTS mrt_types;
    CREATE TABLE IF NOT EXISTS mrt_types
    /* Table containing MRT Types */
    (
            id      INTEGER NOT NULL PRIMARY KEY,   -- Primary key to join
    bgp_headers table on column 'mrt_type'
            type    TEXT UNIQUE                     -- MRT Type
    );
    ''')
    types = [(12,'TABLE_DUMP'),
                (13,'TABLE_DUMP_V2'),
                (16,'BGP4MP'),
                (17,'BGP4MP_ET')]
    cur.executemany('''INSERT INTO mrt_types (id,type)
                VALUES(?,?);''',types)
    db_connection.commit()

# Function to create table 'table_dump_subtypes.' If table exists, it will
    be first removed.
def create_tabledumpsubtype_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript('''
    DROP TABLE IF EXISTS table_dump_subtypes;
    CREATE TABLE IF NOT EXISTS table_dump_subtypes
    /* Table containing Table DUMP Sutbtypes */
    (
            id          INTEGER NOT NULL PRIMARY KEY,    -- Primary key to
    join bgp_headers table on column 'afi'
            subtype     TEXT UNIQUE                      -- Table DUMP
    subtype
    );
    ''')
    types = [(1,'AFI_IPv4'),
                (2,'AFI_IPv6')]
    cur.executemany('''INSERT INTO table_dump_subtypes (id,subtype)
                VALUES(?,?);''',types)
    db_connection.commit()

# Function to create table 'bgp4mp_bgp4mpet_subtypes.' If table exists, it
    will be first removed.
# Subtyes  6 - 11 not implemented in DB
def create_bgp4mpbpg4mpetsubtype_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript('''
    DROP TABLE IF EXISTS bgp4mp_bgp4mpet_subtypes;
    CREATE TABLE IF NOT EXISTS bgp4mp_bgp4mpet_subtypes
    (
    /* Table containing BGP4MP/BGP4MP_ET Subtype */
            id                  INTEGER NOT NULL PRIMARY KEY,     -- Primary
    key to join bgp_headers table on column 'bgp4mp_bgp4mpet_subtype'
```

```
            bgp4mp_subtype    TEXT UNIQUE                       -- BGP4MP/BGP4MP_ET Subtype
    );
    ''')
    types = [(0,'BGP4MP_STATE_CHANGE'),
                (1,'BGP4MP_MESSAGE'),
                (4,'BGP4MP_MESSAGE_AS4'),
                (5,'BGP4MP_STATE_CHANGE_AS4'),
                (6,'BGP4MP_MESSAGE_LOCAL'),
                (7,'BGP4MP_MESSAGE_AS4_LOCAL'),
                (8,'BPG4MP_MESSAGE_ADDPATH'),
                (9,'BGP4MP_MESSAGE_AS4_ADDPATH'),
                (10,'BP4MP_MESSAGE_LOCAL_ADDPATH'),
                (11,'BGP4MP_MESSAGE_AS4_LOCAL_ADDPATH')]
    cur.executemany('''INSERT INTO bgp4mp_bgp4mpet_subtypes (id, bgp4mp_subtype)
                VALUES(?,?);''',types)
    db_connection.commit()

# Function to create table 'bgp_attribute.' If table exists, it will be first removed.
def create_bgpattribute_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript('''
    DROP TABLE IF EXISTS bgp_attribute;
    CREATE TABLE IF NOT EXISTS bgp_attribute
    /* Table containing BGP attributes */
    (
            id              INTEGER NOT NULL PRIMARY KEY,   -- Primary key to join table path_attributes on column 'type'
            attribute       TEXT UNIQUE,                    -- BGP attribute
            ebgp_category   INTEGER,                        -- To join table bgp_attributes_category on foreign key 'id'
            ibgp_category   INTEGER                         -- To join table bgp_attributes_category on foreign key 'id'
    );
    ''')
    # Implemented: ORIGIN, AS_PATH, NEXT_HOP, MULTI_EXIT_DISC,AGGREGATOR
    # Pending to implement attributes 6, 8 - 128. Check if examples are avalable for non-implemented attributes
    types = [(1,'ORIGIN',1,1),
                (2,'AS_PATH',1,1),
                (3,'NEXT_HOP',1,1),
                (4,'MULTI_EXIT_DISC',2,2),
                (5,'LOCAL_PREF',2,1),
                (6,'ATOMIC_AGGREGATE',2,2),
                (7,'AGGREGATOR',2,2),
                (8,'COMMUNITY',3,3),
                (9,'ORIGINATOR_ID',4,4),
                (10,'CLUSTER_LIST',4,4),
                (14,'MP_REACH_NLRI',4,4),
                (15,'MP_UNREACH_NLRI',4,4),
                (16,'EXTENDED COMMUNITIES',3,3),
                (17,'AS4_PATH',3,3),
                (18,'AS4_AGGREGATOR',3,3),
                (26,'AIGP',4,4),
                (32,'LARGE_COMMUNITY',3,3),
                (128,'ATTR_SET',3,3)]
    cur.executemany('''INSERT INTO bgp_attribute (id, attribute, ebgp_category, ibgp_category)
                VALUES(?,?,?,?);''',types)
    db_connection.commit()

# Function to create table 'bgp_attributes_category.' If table exists, it will be first removed.
def create_bgpattributescategories_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript('''
    DROP TABLE IF EXISTS bgp_attributes_category;
    CREATE TABLE IF NOT EXISTS bgp_attributes_category
    (
    /* Table containing categories of BGP attributes */
            id                      INTEGER NOT NULL PRIMARY KEY,     -- Priamry key to join bgp_attribute table on column 'ebgp_category' or 'ibgp_category'
            attribute_category      TEXT UNIQUE                       -- BGP attribute category
    );
    ''')
    types = [(1,'WELL-KNOWN MANDATORY'),
                (2,'WELL-KNOWN DISCRETIONARY'),
                (3,'OPTIONAL TRANSITIVE'),
                (4,'OPTIONAL NON-TRANSITIVE')]
    cur.executemany('''INSERT INTO bgp_attributes_category (id, attribute_category)
                VALUES(?,?);''',types)
    db_connection.commit()

# Function to create table 'bgp_messages.' If table exists, it will be first removed.
def create_bgpmessagestype_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript('''
    DROP TABLE IF EXISTS bgp_messages_type;
    CREATE TABLE IF NOT EXISTS bgp_messages_type
    (
    /* Table containing BGP message types */
            id              INTEGER NOT NULL PRIMARY KEY,   -- Primary key to join bgp_messages table on column 'type'
            message_type    TEXT UNIQUE                     -- BGP message type
    );
    ''')
    types = [(1,'OPEN'),
                (2,'UPDATE'),
                (3,'NOTIFICATION'),
                (4,'KEEPALIVE')]
    cur.executemany('''INSERT INTO bgp_messages_type (id, message_type)
                VALUES(?,?);''',types)
    db_connection.commit()

# Function to create table 'as_path_segments_type.' If table exists, it will be first removed.
def create_aspathsegmentstype_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript('''
    DROP TABLE IF EXISTS as_path_segments_type;
    CREATE TABLE IF NOT EXISTS as_path_segments_type
    (
    /* Table containing AS_PATH segment types */
            id                INTEGER NOT NULL PRIMARY KEY,    -- Primary key
    to join as_paths table on column 'segment_type'
            segment_type     TEXT UNIQUE                       -- AS_PATH
    segment type
    );
    ''')
    types = [(1,'AS_SET'),
                (2,'AS_SEQUENCE')]
    cur.executemany('''INSERT INTO as_path_segments_type (id, segment_type)
                VALUES(?,?);''',types)
    db_connection.commit()

# Function to create table 'bgp_headers.' If table exists, it will be first
    removed.
def create_bgpheader_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript('''
    DROP TABLE IF EXISTS bgp_headers;
    CREATE TABLE IF NOT EXISTS bgp_headers
    /* Table containing headers of collected BGP message */
    (
            id                            INTEGER NOT NULL PRIMARY KEY
    AUTOINCREMENT UNIQUE,  -- Primary key to join table bgp_openmessages,
    bgp_updatemessages, bgp_keepalivemessages, bgp_notificationmessages, or
    bgp_statechangemessages on column 'header_id'
            timestamp_unix                INTEGER,
                    -- Time stamp of collected BGP message in Unix format
            timestamp_date                DATETIME,
                    -- Time stamp of collected BGP message in yyyy-mm-dd hh:
    mm:ss format
            mrt_type                      INTEGER,
                    -- To join table mrt_types on column 'id'
            bgp4mp_bgp4mpet_subtype       INTEGER,
                    -- To join table bgp4mp_bgp4mpet_subtypes on column 'id'
            message_length                INTEGER,
                    -- Length of header + BGP message
            peer_as                       TEXT,
                    -- AS sending the BGP message
            local_as                      TEXT,
                    -- AS receiving the BGP message
            ifindex                       INTEGER,
                    -- Interface index
            afi                           INTEGER,
                    -- To join table table_dump_subtypes on 'id'
            peer_ip                       TEXT,
                    -- IP (prefix) of sending AS
            local_ip                      TEXT
                    -- IP (prefix) of receiving AS
    );
    ''')
    db_connection.commit()
#   bgp_message_length          INTEGER
            -- Length of BGP message

```

```

# Function to create table 'bgp_updatemessages.' If table exists, it will be first removed.
def create_bgpupdatemessages_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript('''
    DROP TABLE IF EXISTS bgp_updatemessages;
    CREATE TABLE IF NOT EXISTS bgp_updatemessages
    (
    /* Table containing collected BGP messages */
            id                          INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,  -- Primary key to join tables withdawn_routes, announced_routes, or path_attributes on bgp_message_id
            timestamp_unix              INTEGER,            -- Time stamp of collected BGP message in Unix format
            timestamp_date              DATETIME,           -- Time stamp of collected BGP message in yyyy-mm-dd hh:mm:ss format
            header_id                   INTEGER,            -- To join table bgp_headers on column 'id'
            message_length              INTEGER,            -- BGP message length
            type                        INTEGER,            -- To join table bgp_messages_type on column 'id'
            withdrawn_routes_length     INTEGER,            -- Length of withdrawn routes field
            path_attribute_length       INTEGER,            -- Length of path attributes field
            nlri_length                 INTEGER             -- Number of announced routes
    );
    ''')
    db_connection.commit()

# Function to create table 'path_attributes.' If table exists, it will be first removed.
def create_pathattributes_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript('''
    DROP TABLE IF EXISTS path_attributes;
    CREATE TABLE IF NOT EXISTS path_attributes
    (
    /* Table containing path attributes of collected BGP UPDATE messages */
            id                          INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,  -- Primary key to join tables as_paths, aggregator_attribute, origin_pathattribute, nexthop_pathattribute, or multiexitdisc_pathattribute on column 'path_attribute_id'
            timestamp_unix              INTEGER,            -- Time stamp of collected BGP message in Unix format
            timestamp_date              DATETIME,           -- Time stamp of collected BGP message in yyyy-mm-dd hh:mm:ss format
            bgp_message_id              INTEGER,            -- To join table bgp_messages on column 'id'
            flag                        INTEGER,            -- Path attribute flag
            type                        INTEGER,            -- To join table bgp_attribute on column 'id'
            length                        INTEGER
                    -- Path attribute length
    );
    ''')
    db_connection.commit()

# Function to create table 'as_paths.' If table exists, it will be first
    removed.
def create_aspaths_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript('''
    DROP TABLE IF EXISTS as_paths;
    CREATE TABLE IF NOT EXISTS as_paths
    (
    /* Table containing sequences or sets of ASes (route BGP message has
    traversed) of collected BGP UPDATE messages */
            id                            INTEGER NOT NULL PRIMARY KEY
    AUTOINCREMENT UNIQUE,  -- Primary key
            timestamp_unix                INTEGER,
                    -- Time stamp of collected BGP message in Unix format
            timestamp_date                DATETIME,
                    -- Time stamp of collected BGP message in yyyy-mm-dd hh:
    mm:ss format
            path_attribute_id             INTEGER,
                    -- To join table path_attributes on column 'id'
            segment_type                  INTEGER,
                    -- To join table as_path_segments_type on column 'id'
            aspath_length                 INTEGER,
                    -- Length of AS_PATH path attribute
            as_path                       INTEGER
                    -- Sequence or set of ASes
    );
    ''')
    db_connection.commit()

# Function to create table 'aggregator_attributes.' If table exists, it will
     be first removed.
def create_aggregatorattributes_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript('''
    DROP TABLE IF EXISTS aggregator_attributes;
    CREATE TABLE IF NOT EXISTS aggregator_attributes
    /* Table containing AGGREGATOR path attribute of collected BGP UPDATE
    messages */
    (
            id                            INTEGER NOT NULL PRIMARY KEY
    AUTOINCREMENT UNIQUE,  -- Primary key
            timestamp_unix                INTEGER,
                    -- Time stamp of collected BGP message in Unix format
            timestamp_date                DATETIME,
                    -- Time stamp of collected BGP message in yyyy-mm-dd hh:
    mm:ss format
            path_attribute_id             INTEGER,
                    -- To join table path_attributes on column 'id'
            attribute_length              INTEGER,
                    -- Length of the AGGREGATOR path attribute
            as_id                         INTEGER,
                    -- Last AS number that formed the aggregate route
            as_ip                         TEXT
                    -- IP/prefix of the last AS that formed the aggregate
    route
    );
    ''')
    db_connection.commit()

# Function to create table 'withdrawn_routes.' If table exists, it will be
    first removed.
def create_withdrawnroutes_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript('''
    DROP TABLE IF EXISTS withdrawn_routes;
    CREATE TABLE IF NOT EXISTS withdrawn_routes
    (
    /* Table containing Withdrawn routes of collected BGP UPDATE messages */
            id                            INTEGER NOT NULL PRIMARY KEY
    AUTOINCREMENT UNIQUE,  -- Primary key
            timestamp_unix                INTEGER,
                    -- Time stamp of collected BGP message in Unix format
            timestamp_date                DATETIME,
                    -- Time stamp of collected BGP message in yyyy-mm-dd hh:
    mm:ss format
            bgp_message_id                INTEGER,
                    -- To join table bgp_updatemessages on column 'id'
            prefix_length                 INTEGER,
                    -- Length of the withdrawn routes prefix field
            prefix                        TEXT
                    -- Withdrawn IPs (prefixes)
    );
    ''')
    db_connection.commit()

# Function to create table 'announced_routes.' If table exists, it will be
    first removed.
def create_announcedroutes_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript('''
    DROP TABLE IF EXISTS announced_routes;
    CREATE TABLE IF NOT EXISTS announced_routes
    (
        /* Table containing Announced routes of collected BGP UPDATE
    messages */
            id                            INTEGER NOT NULL PRIMARY KEY
    AUTOINCREMENT UNIQUE,  -- Primary key
            timestamp_unix                INTEGER,
                    -- Time stamp of collected BGP message in Unix format
            timestamp_date                DATETIME,
                    -- Time stamp of collected BGP message in yyyy-mm-dd hh:
    mm:ss format
            bgp_message_id                INTEGER,
                    -- To join table bgp_messages on column 'id'
            prefix_length                 INTEGER,
                    -- Length of the NLRI field (announced routes)
            prefix                        TEXT
                    -- Announced IPs (prefixes)
    );
    ''')
    db_connection.commit()

# Function to create table 'origin_pathattribute.' If table exists, it will
    be first removed.
def cerate_originpathattribute_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript('''
    DROP TABLE IF EXISTS origin_pathattribute;
    CREATE TABLE IF NOT EXISTS origin_pathattribute
    (
    /* Table containing ORIGIN path attribute of collected BGP UPDATE
    messages */
            id                          INTEGER NOT NULL PRIMARY KEY
    AUTOINCREMENT UNIQUE,  -- Primary key
            timestamp_unix              INTEGER,
                    -- Time stamp of collected BGP message in Unix format
            timestamp_date              DATETIME,
                    -- Time stamp of collected BGP message in yyyy-mm-dd hh:
    mm:ss format
            path_attribute_id           INTEGER,
                    -- To join table path_attributes on column 'id'
            attribute_length            INTEGER,
                    -- Length of ORIGIN path attribute
            origin_id                   INTEGER
                    -- To join table origin_values on column 'id'
    );
    ''')
    db_connection.commit()

# Function to create table 'nexthop_pathattribute.' If table exists, it will
     be first removed.
def cerate_nexthopattribute_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript('''
    DROP TABLE IF EXISTS nexthop_pathattribute;
    CREATE TABLE IF NOT EXISTS nexthop_pathattribute
    (
    /* Table containing NEXT_HOP path attribute of collected BGP UPDATE
    messages */
            id                          INTEGER NOT NULL PRIMARY KEY
    AUTOINCREMENT UNIQUE,  -- Primary key
            timestamp_unix              INTEGER,
                    -- Time stamp of collected BGP message in Unix format
            timestamp_date              DATETIME,
                    -- Time stamp of collected BGP message in yyyy-mm-dd hh:
    mm:ss format
            path_attribute_id           INTEGER,
                    -- To join table path_attributes on column 'id'
            attribute_length            INTEGER,
                    -- Length of NEXT_HOP path attribute
            nexthop_ip                  TEXT
                    -- IP (prefix) of router that should be used as next hop
    );
    ''')
    db_connection.commit()

# Function to create table 'origin_values.' If table exists, it will be
    first removed.
```

```
def cerate_originvalues_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript(’’’
    DROP TABLE IF EXISTS origin_values;
    CREATE TABLE IF NOT EXISTS origin_values
    (
    /* Table containing values of ORIGIN BGP attribute */
            id              INTEGER NOT NULL PRIMARY KEY,   -- Primary key to join origin_pathattribute table on column ’origin_id’
            origin_value    TEXT UNIQUE                     -- Origin of the path information
    );
    ’’’)
    types = [(0,’EGP’),
             (1,’IGP’),
             (2,’INCOMPLETE’)]
    cur.executemany(’’’INSERT INTO origin_values (id, origin_value)
             VALUES(?,?);’’’,types)
    db_connection.commit()

# Function to create table ’multiexitdisc_pathattribute.’ If table exists, it will be first removed.
def cerate_multiexitdiscattribute_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript(’’’
    DROP TABLE IF EXISTS multiexitdisc_pathattribute;
    CREATE TABLE IF NOT EXISTS multiexitdisc_pathattribute
    /* Table containing MULTI_EXIT_DISC path attribute of collected BGP UPDATE messages */
    (
            id                          INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,  -- Primary key
            timestamp_unix              INTEGER,
                    -- Time stamp of collected BGP message in Unix format
            timestamp_date              DATETIME,
                    -- Time stamp of collected BGP message in yyyy-mm-dd hh:mm:ss format
            path_attribute_id           INTEGER,
                    -- To join table path_attributes on column ’id’
            attribute_length            INTEGER,
                    -- Length of MULTI_EXIT_DISC path attribute
            value                       INTEGER
                    -- Preferred value to be used when discriminating among multiple entry/exit points to the same AS neighbor
    );
    ’’’)
    db_connection.commit()

# Function to create table ’bgp_keepalivemessages.’ If table exists, it will be first removed.
def create_bgpkeepalivemessages_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript(’’’
    DROP TABLE IF EXISTS bgp_keepalivemessages;
    CREATE TABLE IF NOT EXISTS bgp_keepalivemessages
    /* Table containing fields included in collected BGP OPEN messages */
    (
            id                            INTEGER NOT NULL PRIMARY KEY
    AUTOINCREMENT UNIQUE,  -- Primary key
            timestamp_unix                INTEGER,
                    -- Time stamp of collected BGP message in Unix format
            timestamp_date                DATETIME,
                    -- Time stamp of collected BGP message in yyyy-mm-dd hh:
    mm:ss format
            header_id                     INTEGER,
                    -- To join table bgp_headers on column 'id'
            message_length                INTEGER,
                    -- Length of header + BGP message
            type                          INTEGER
                    -- To join table bgp_messages_type on column 'id'
    );
    ''')
    db_connection.commit()

# Function to create table 'bgp_openmessages.' If table exists, it will be
    first removed.
def create_bgpopenmessages_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript('''
    DROP TABLE IF EXISTS bgp_openmessages;
    CREATE TABLE IF NOT EXISTS bgp_openmessages
    /* Table containing fields included in collected BGP OPEN messages */
    (
            id                            INTEGER NOT NULL PRIMARY KEY
    AUTOINCREMENT UNIQUE,  -- Primary key
            timestamp_unix                INTEGER,
                    -- Time stamp of collected BGP message in Unix format
            timestamp_date                DATETIME,
                    -- Time stamp of collected BGP message in yyyy-mm-dd hh:
    mm:ss format
            header_id                     INTEGER,
                    -- To join table bgp_headers on column 'id'
            message_length                INTEGER,
                    -- Length of BGP message
            type                          INTEGER,
                    -- To join table bgp_messages_type on column 'id'
            bgp_version                   INTEGER,
                    -- Length of MULTI_EXIT_DISC path attribute
            local_as                      INTEGER,
                    -- AS number of receiving router (peer)
            holdtime                      INTEGER,
                    -- Time between receipt of successive KEPALIVE and/or
    UPDATE messages
            bgp_id                        TEXT,
                    -- IP of receiving router (peer)
            optional_parameters_length   TEXT
                    -- Length of optional parameters included in the OPEN
    message
    );
    ''')
    db_connection.commit()

# Function to create table 'bgp_notificationmessages.' If table exists, it
    will be first removed.
def create_bgpnotificationmessages_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript('''
    DROP TABLE IF EXISTS bgp_notificationmessages;
    CREATE TABLE IF NOT EXISTS bgp_notificationmessages
    /* Table containing fields included in collected BGP NOTIFICATION messages */
    (
            id                          INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,  -- Primary key
            timestamp_unix              INTEGER,        -- Time stamp of collected BGP message in Unix format
            timestamp_date              DATETIME,       -- Time stamp of collected BGP message in yyyy-mm-dd hh:mm:ss format
            header_id                   INTEGER,        -- To join table bgp_headers on column 'id'
            message_length              INTEGER,        -- Length of BGP message
            type                        INTEGER,        -- To join table bgp_messages_type on column 'id'
            error_code                  INTEGER,        -- To join table bgp_error_codes on column 'id'
            error_subcode               INTEGER,        -- To join table bgp_error_subcodes on column 'id'
            diagnosis_data              TEXT            -- Addtional data included to diagnose errors
    );
    ''')
    db_connection.commit()

# Function to create table 'bgp_error_codes.' If table exists, it will be first removed.
def create_bgperrorcodes_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript('''
    DROP TABLE IF EXISTS bgp_error_codes;
    CREATE TABLE IF NOT EXISTS bgp_error_codes
    /* Table containing BGP error codes */
    (
            id                          INTEGER NOT NULL PRIMARY KEY,  -- Primary key, to join table bgp_notificationmessages on column 'error_code'
            symbolic_name               TEXT                           -- Description of error condition
    );
    ''')
    types = [(1,'Message Header Error'),
            (2,'OPEN Message Error'),
            (3,'UPDATE Message Error'),
            (4,'Hold Timer Expired'),
            (5,'Finite State Machine Error'),
            (6,'Cease')]
    cur.executemany('''INSERT INTO bgp_error_codes (id, symbolic_name)
            VALUES(?,?);''',types)
    db_connection.commit()

# Function to create table 'bgp_messageheader_error_subcodes.' If table exists, it will be first removed.
```

```
def create_bgpmessageheadererrorsubcodes_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript('''
    DROP TABLE IF EXISTS bgp_messageheader_error_subcodes;
    CREATE TABLE IF NOT EXISTS bgp_messageheader_error_subcodes
    /* Table containing BGP message Header error subcodes */
    (
            id                          INTEGER NOT NULL PRIMARY KEY,   --
    Primary key, to join table bgp_notificationmessages on column '
    error_subcode'
            description                 TEXT                            --
    Description of message header error subcode
    );
    ''')
    types = [(0,'Unspecific'),
                (1,'Connection Not Synchronized'),
                (2,'Bad Message Length'),
                (3,'Bad Message Type')]
    cur.executemany('''INSERT INTO bgp_messageheader_error_subcodes (id,
    description)
                VALUES(?,?);''',types)
    db_connection.commit()

# Function to create table 'bgp_openmessage_error_subcodes.' If table exists
    , it will be first removed.
def create_bgpopenmessageerrorsubcodes_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript('''
    DROP TABLE IF EXISTS bgp_openmessage_error_subcodes;
    CREATE TABLE IF NOT EXISTS bgp_openmessage_error_subcodes
    /* Table containing BGP OPEN message error subcodes */
    (
            id                          INTEGER NOT NULL PRIMARY KEY,  --
    Primary key, to join table bgp_notificationmessages on column '
    error_subcode'
            description                 TEXT                           --
    Description of open message error subcode
    );
    ''')
    types = [(0,'Unspecific'),
                (1,'Unsupported Version Number'),
                (2,'Bad Peer AS'),
                (3,'Bad BGP Identifier'),
                (4,'Unsupported Optional Parameter'),
                (5,'Authentication Failure'),
                (6,'Unacceptable Hold Time')]
    cur.executemany('''INSERT INTO bgp_openmessage_error_subcodes (id,
    description)
                VALUES(?,?);''',types)
    db_connection.commit()

# Function to create table 'bgp_updatemessage_error_subcodes.' If table
    exists, it will be first removed.
def create_bgpupdatemessageerrorsubcodes_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript('''
    DROP TABLE IF EXISTS bgp_updatemessage_error_subcodes;
    CREATE TABLE IF NOT EXISTS bgp_updatemessage_error_subcodes
    /* Table containing BGP UPDATE message error subcodes */
    (
            id                          INTEGER NOT NULL PRIMARY KEY,  --
    Primary key, to join table bgp_notificationmessages on column '
    error_subcode'
            description                 TEXT                           --
    Description of message header error subcode
    );
    ''')
    types = [(0,'Unspecific'),
                (1,'Malformed Attribute List'),
                (2,'Unrecognized Well-known Attribute'),
                (3,'Missing Well-known Attribute'),
                (4,'Attribute Flags Error'),
                (5,'Attribute Length Error'),
                (6,'Invalid ORIGIN Attribute'),
                (7,'AS Routing Loop'),
                (8,'Invalid NEXT_HOP Attribute'),
                (9,'Optional Attribute Error'),
                (10,'Invalid Network Field'),
                (11,'Malformed AS_PATH')]
    cur.executemany('''INSERT INTO bgp_updatemessage_error_subcodes (id,
    description)
                VALUES(?,?);''',types)
    db_connection.commit()

# Add table for optional parameters OPEN messages?

def create_bgpstatechangemessages_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript('''
    DROP TABLE IF EXISTS bgp_statechangemessages;
    CREATE TABLE IF NOT EXISTS bgp_statechangemessages
    /* Table containing fields old and new state in the BGP finite state
    machine */
    (
            id                          INTEGER NOT NULL PRIMARY KEY
    AUTOINCREMENT UNIQUE,  -- Primary key
            timestamp_unix              INTEGER,
                    -- Time stamp of collected BGP message in Unix format
            timestamp_date              DATETIME,
                    -- Time stamp of collected BGP message in yyyy-mm-dd hh:
    mm:ss format
            header_id                   INTEGER,
                    -- To join table bgp_headers on column 'id'
            old_state                   INTEGER,
                    -- Previous state in the BGP finite state machine
            new_state                   INTEGER
                    -- New state in the BGP finite state machine
    );
    ''')
    db_connection.commit()

def create_bgpfinitesatemachine_table(db_connection):
    cur = db_connection.cursor()
    cur.executescript('''
    DROP TABLE IF EXISTS bgp_finite_state_machine;
    CREATE TABLE IF NOT EXISTS bgp_finite_state_machine
    /* Table containing description of each sate in the BGP finite state machine */
    (
            id                          INTEGER NOT NULL PRIMARY KEY,  -- Primary key, to join table bgp_statemessages on columns 'old_state' and 'new_state'
            description                 TEXT                           -- Description of states in the BGP finite state machine
    );
    ''')
    types = [(1,'Idle'),
                (2,'Connect'),
                (3,'Active'),
                (4,'Open Sent'),
                (5,'Open Confirm'),
                (6,'Established')]
    cur.executemany('''INSERT INTO bgp_finite_state_machine (id, description)
                VALUES(?,?);''',types)
    db_connection.commit()

# Function to insert a new row in table 'bgp_headers.' Arugment 'db_connection' must be created while argument 'entries' should be a list of elements following order shown
# in command 'cur.execute(). Function returns the index of added row.
def insert_bgpheader_table(db_connection,entries):
    cur = db_connection.cursor()
    cur.execute('''INSERT OR IGNORE INTO bgp_headers (timestamp_unix,timestamp_date,mrt_type,bgp4mp_bgp4mpet_subtype,
                message_length,peer_as,local_as,ifindex,afi,peer_ip,local_ip)
                VALUES(?,?,?,?,?,?,?,?,?,?,?);''',entries)
    last_row = cur.lastrowid
    db_connection.commit()
    return last_row
#   bgp_message_length

# Function to insert a new row in table 'bgp_updatemessages.' Arugment 'db_connection' must be created while argument 'entries' should be a list of elements following order shown
# in command 'cur.execute().
def insert_bgpupdatemessages_table(db_connection,entries):
    cur = db_connection.cursor()
    cur.execute('''INSERT INTO bgp_updatemessages (timestamp_unix,timestamp_date,header_id,message_length,type,withdrawn_routes_length,
                path_attribute_length,nlri_length) VALUES(?,?,?,?,?,?,?,?);''',entries)
    db_connection.commit()

# Function to insert a new row in table 'path_attributes.' Arugment 'db_connection' must be created while argument 'entries' should be a list of elements following order shown
# in command 'cur.execute(). Function returns the index of added row.
def insert_pathattributes_table(db_connection,entries):
    cur = db_connection.cursor()
    cur.execute('''INSERT INTO path_attributes (timestamp_unix,timestamp_date,bgp_message_id,flag,type,
                length) VALUES(?,?,?,?,?,?);''',entries)
    last_row = cur.lastrowid
    db_connection.commit()
    return last_row

# Function to insert a new row in table 'as_paths.' Arugment 'db_connection' must be created while argument 'entries' should be a list of elements following order shown
# in command 'cur.execute().
def insert_aspaths_table(db_connection,entries):
    cur = db_connection.cursor()
    cur.execute('''INSERT INTO as_paths (timestamp_unix,timestamp_date,path_attribute_id,segment_type,aspath_length,
                as_path) VALUES(?,?,?,?,?,?);''',entries)
    db_connection.commit()

# Function to insert a new row in table 'aggregaror_attributes.' Arugment 'db_connection' must be created while argument 'entries' should be a list of elements following order shown
# in command 'cur.execute().
def insert_aggregatorattributes_table(db_connection,entries):
    cur = db_connection.cursor()
    cur.execute('''INSERT INTO aggregator_attributes (timestamp_unix,timestamp_date,path_attribute_id,attribute_length,as_id,as_ip)
                VALUES(?,?,?,?,?,?);''',entries)
    db_connection.commit()

# Function to insert a new row in table 'withdrawn_routes.' Arugment 'db_connection' must be created while argument 'entries' should be a list of elements following order shown
# in command 'cur.execute().
def insert_withdrawnroutes_table(db_connection,entries):
    cur = db_connection.cursor()
    cur.execute('''INSERT INTO withdrawn_routes (timestamp_unix,timestamp_date,bgp_message_id,prefix_length,prefix)
                VALUES(?,?,?,?,?);''',entries)
    db_connection.commit()

# Function to insert a new row in table 'announced_routes.' Arugment 'db_connection' must be created while argument 'entries' should be a list of elements following order shown
# in command 'cur.execute().
def insert_announcedroutes_table(db_connection,entries):
    cur = db_connection.cursor()
    cur.execute('''INSERT INTO announced_routes (timestamp_unix,timestamp_date,bgp_message_id,prefix_length,prefix)
                VALUES(?,?,?,?,?);''',entries)
    db_connection.commit()

# Function to insert a new row in table 'origin_pathattribute.' Arugment 'db_connection' must be created while argument 'entries' should be a list of elements following order shown
# in command 'cur.execute().
def insert_originpathattribute_table(db_connection,entries):
    cur = db_connection.cursor()
    cur.execute('''INSERT INTO origin_pathattribute (timestamp_unix,timestamp_date,path_attribute_id,attribute_length,origin_id)
                VALUES(?,?,?,?,?);''',entries)
    db_connection.commit()
```

```

# Function to insert a new row in table 'nexthop_pathattribute.' Arugment 'db_connection' must be created while argument 'entries' should be a list of elements following order shown
# in command 'cur.execute().
def insert_nexthopattribute_table(db_connection,entries):
    cur = db_connection.cursor()
    cur.execute('''INSERT INTO nexthop_pathattribute (timestamp_unix,timestamp_date,path_attribute_id,attribute_length,nexthop_ip)
                VALUES(?,?,?,?,?);''',entries)
    db_connection.commit()

# Function to insert a new row in table 'multiexitdisc_pathattribute.' Arugment 'db_connection' must be created while argument 'entries' should be a list of elements following order
# shown in command 'cur.execute().
def insert_multiexitdiscattribute_table(db_connection,entries):
    cur = db_connection.cursor()
    cur.execute('''INSERT INTO multiexitdisc_pathattribute (timestamp_unix,timestamp_date,path_attribute_id,attribute_length,value)
                VALUES(?,?,?,?,?);''',entries)
    db_connection.commit()

def insert_bgpkeepalive_table(db_connection,entries):
    cur = db_connection.cursor()
    cur.execute('''INSERT INTO bgp_keepalivemessages (timestamp_unix,timestamp_date,header_id,message_length,type)
                    VALUES(?,?,?,?,?);''',entries)
    db_connection.commit()

def insert_bgpopenmessages_table(db_connection,entries):
    cur = db_connection.cursor()
    cur.execute('''INSERT INTO bgp_openmessages (timestamp_unix,timestamp_date,header_id,message_length,type,bgp_version,local_as,holdtime,bgp_id,optional_parameters_length)
                    VALUES(?,?,?,?,?,?,?,?,?,?);''',entries)
    db_connection.commit()

def insert_bgpnotification_table(db_connection,entries):
    cur = db_connection.cursor()
    cur.execute('''INSERT INTO bgp_notificationmessages (timestamp_unix,timestamp_date,header_id,message_length,type,error_code,error_subcode,diagnosis_data)
                    VALUES(?,?,?,?,?,?,?,?);''',entries)
    db_connection.commit()

def insert_bgpstatechangemessages_table(db_connection,entries):
    cur = db_connection.cursor()
    cur.execute('''INSERT INTO bgp_statechangemessages (timestamp_unix,timestamp_date,header_id,old_state,new_state)
                    VALUES(?,?,?,?,?);''',entries)
    db_connection.commit()

# Pending:

# Function that returns the last row added in a table.
def last_row(db_connection):
    cur = db_connection.cursor()
    return cur.lastrowid

# Function that creates all required tables in the BGP database. Modify the
    path according to location where DB will reside.
def create_database():
    conn = create_connection(r"/Users/analauragonzalezrios/Documents/SFU/MSc
     Thesis/Experiments/bgpsqldb.db")
    create_mrttype_table(conn)
    create_tabledumpsubtype_table(conn)
    create_bgp4mpbpg4mpetsubtype_table(conn)
    create_bgpattribute_table(conn)
    create_bgpmessagestype_table(conn)
    create_aspathsegmentstype_table(conn)
    create_bgpheader_table(conn)
    create_bgpupdatemessages_table(conn)
    create_pathattributes_table(conn)
    create_aspaths_table(conn)
    create_aggregatorattributes_table(conn)
    create_bgpattributescategories_table(conn)
    create_withdrawnroutes_table(conn)
    create_announcedroutes_table(conn)
    cerate_originpathattribute_table(conn)
    cerate_nexthopattribute_table(conn)
    cerate_originvalues_table(conn)
    cerate_multiexitdiscattribute_table(conn)
    create_bgpkeepalivemessages_table(conn)
    create_bgpopenmessages_table(conn)
    create_bgpnotificationmessages_table(conn)
    create_bgperrorcodes_table(conn)
    create_bgpmessageheadererrorsubcodes_table(conn)
    create_bgpopenmessageerrorsubcodes_table(conn)
    create_bgpupdatemessageerrorsubcodes_table(conn)
    create_bgpstatechangemessages_table(conn)
    create_bgpfinitesatemachine_table(conn)


# Main function, invoques the crate_database() function.
if __name__ == '__main__':
    create_database()
```

Appendix D

BGP-RDB: Implementation of Data Download and Ingestion Engine Functions

Listed is the implementation of the ingestion engine functions used to store incoming data in corresponding tables. Required information includes year, month, day, hour, minute, data collection site, and collector name. Function *updateMessageName* generates the RIPE or Route Views repository file name while function *data_downloader_single* retrieves a BGP repository from a specified collection site and the collector. After collected MRT files are parsed, the YAML files are generated by using function *mrt2yaml*. YAML file used to insert data in the *BGP-RDB* is defined in the variable *stream*. Content of the YAML file is iteratively assessed to insert data in corresponding tables using *for loops* and *if-else statements*.

Listing D.1: *BGP-RDA* Ingestion Engine Functions: Tables Initialization and Data Entry.

```python
"""
    @author Ana Laura Gonzalez Rios
    @email anag@sfu.ca
    @date June 2022
    @version: 1.0.0
    @description:
                This file contains the implementation of functions to
                retrieve BGP repositories, parse MRT files to YAML
                fiels, and insert collected data in corresponding DB
                tables.

    @copyright Copyright (c) June 2022
        All Rights Reserved

    Python code (version 3.10.2)
    SQLite3 commands (version 3.28.0)
"""

from dataDownload import * #data_downloader_single
from mrt2yaml import mrt2yaml
import yaml
```

```
from ingestion import *

# Information used to define URL of a RIPE repository given the collector
    name, year, month, day, hour, and minute
collector_ripe = 'rrc04'
year = '2003'
month = '01'
day = '01'
hour = '00'
minute = '00'
data_file_ripe = updateMessageName(year, month, day, hour, minute)[0]
data_date_ripe = updateMessageName(year, month, day, hour, minute)[1]
site = 'RIPE'

print ('Downloading repository "%s" for month "%s"' % (data_file_ripe,
    data_date_ripe))

# Collects specified repository of BGP messages from selected RIPE collector
data_downloader_single(data_file_ripe, data_date_ripe, site, collector_ripe)

# Information used to define URL of a RIPE repository given the collector
    name, year, month, day, hour, and minute
collector_routeviews = 'route-views2'
year = '2021'
month = '01'
day = '21'
hour = '01'
minute = '30'
data_file_routeviews = updateMessageName(year, month, day, hour, minute)[0]
data_date_routeviews = updateMessageName(year, month, day, hour, minute)[1]
site = 'RouteViews'

print ('Downloading repository "%s" for month "%s"' % (data_file_routeviews,
     data_date_routeviews))

# Collects specified repository of BGP messages from selected Route Views
    collector
data_downloader_single(data_file_routeviews, data_date_routeviews, site,
    collector_routeviews)

# Collected BGP messages from RIPE are converted from MRT to YAML format
mrt2yaml(data_file_ripe + '.gz','ripe.yaml')

# Collected BGP messages from Route Views are converted from MRT to YAML
    format
mrt2yaml(data_file_routeviews + '.bz2','routeviews.yaml')

# Opens YAML file containing update messages
stream = open('routeviews.yaml')

# Loads YAML file
parsed_yaml_file = yaml.load(stream,Loader=yaml.FullLoader)
print('YAML file loaded')

# Opens connection with BGP database
conn = create_connection(r"/Users/analauragonzalezrios/Documents/SFU/MSc
    Thesis/Experiments/bgpsqldb.db")

```

```
for item in parsed_yaml_file:
    timestamp_unix = list(item['timestamp'].keys())[0]
    timestamp_date = item['timestamp'][timestamp_unix]
    mrt_type = list(item['type'].keys())[0]
    mrt_subtype = list(item['subtype'].keys())[0]
    length = item['length']
    peer_as = item['peer_as']
    local_as = item['local_as']
    ifindex = item['ifindex']
    afi = list(item['afi'].keys())[0]
    peer_ip = item['peer_ip']
    local_ip = item['local_ip']
    print(timestamp_unix,timestamp_date,mrt_type,mrt_subtype,length,peer_as,local_as,ifindex,afi,peer_ip,local_ip)

    # New entry in bgpheader_table
    bgp_header_id = insert_bgpheader_table(conn,(list(item['timestamp'].keys())[0],
                                                    item['timestamp'][list(item['timestamp'].keys())[0]],
                                                    list(item['type'].keys())[0],list(item['subtype'].keys())[0],item['length'],
                                                    item['peer_as'],item['local_as'],item['ifindex'],list(item['afi'].keys())[0],
                                                    item['peer_ip'],item['local_ip']))

    # New entry in bgp_openmessages, bgp_updatemessates and related tables, bgp_notificationmessages, and bgp_keepalivemessages:
    if (list(item['subtype'].keys())[0] == 1) or (list(item['subtype'].keys())[0] == 4):

        # New entry in bgp_openmessages table
        if list(item['bgp_message']['type'].keys())[0] == 1:
            insert_bgpopenmessages_table(conn,(list(item['timestamp'].keys())[0],
                                                item['timestamp'][list(item['timestamp'].keys())[0]],bgp_header_id,
                                                item['bgp_message']['length'],list(item['bgp_message']['type'].keys())[0],
                                                item['bgp_message']['version'],item['bgp_message']['local_as'],item['bgp_message']['holdtime'],
                                                item['bgp_message']['bgp_id'],len(item['bgp_message']['optional_parameters'])))

        # New entry in bgp_updatepmessages table
        elif list(item['bgp_message']['type'].keys())[0] == 2:
            insert_bgpupdatemessages_table(conn,(list(item['timestamp'].keys())[0],
                                                    item['timestamp'][list(item['timestamp'].keys())[0]],bgp_header_id,
                                                    item['bgp_message']['length'],list(item['bgp_message']['type'].keys())[0],
                                                    item['bgp_message']['withdrawn_routes_length'],
                                                    item['bgp_message']['path_attribute_length'],
                                                    len(item['bgp_message']['nlri'])))

            # New entries in announcedroutes_table
            for nlri in item['bgp_message']['nlri']:
                #print(nlri['prefix_length'])
                insert_announcedroutes_table(conn,(list(item['timestamp'].keys())[0],
                                                    item['timestamp'][list(item['timestamp'].keys())[0]],
                                                    bgp_header_id,nlri['prefix_length'],nlri['prefix']))

            # New entries in withdrawnroutes_table
            for withdrawal in item['bgp_message']['withdrawn_routes']:
                insert_withdrawnroutes_table(conn,(list(item['timestamp'].keys())[0],
                                                    item['timestamp'][list(item['timestamp'].keys())[0]],
                                                    bgp_header_id,withdrawal['prefix_length'], withdrawal['prefix']))

            # New entries in path attributes tables
            for path_attribute in item['bgp_message']['path_attributes']:
                path_attribute_id = insert_pathattributes_table(conn,(list(item['timestamp'].keys())[0],
                                                                        item['timestamp'][list(item['timestamp'].keys())[0]],
                                                                        bgp_header_id,path_attribute['flag'],
                                                                        list(path_attribute['type'].keys())[0],
                                                                        path_attribute['length']))

                # New entries in ORIGIN path attribute table
                if list(path_attribute['type'].keys())[0] == 1:
                    insert_originpathattribute_table(conn,(list(item['timestamp'].keys())[0],
                                                            item['timestamp'][list(item['timestamp'].keys())[0]],path_attribute_id,
                                                            path_attribute['length'],list(path_attribute['value'].keys())[0]))

                # New entries in AS_PATH attribute table
                elif list(path_attribute['type'].keys())[0] == 2:
                    insert_aspaths_table(conn,(list(item['timestamp'].keys())[0],item['timestamp'][list(item['timestamp'].keys())[0]],
                                                path_attribute_id,list(path_attribute['value'][0]['type'].keys())[0],
                                                    path_attribute['value'][0]['length'],','.join(path_attribute['value'][0]['value'])))

                # New entries in NEXT_HOP attribute table
                elif list(path_attribute['type'].keys())[0] == 3:
                    insert_nexthopattribute_table(conn,(list(item['timestamp'].keys())[0],item['timestamp'][list(item['timestamp'].keys())[0]],
                                                        path_attribute_id,path_attribute['length'],path_attribute['value']))

                # New entries in MULTI_EXIT_DISC attribute table
                elif list(path_attribute['type'].keys())[0] == 4:
                    insert_multiexitdiscattribute_table(conn,(list(item['timestamp'].keys())[0],item['timestamp'][list(item['timestamp'].keys())[0]],
                                                        path_attribute_id,path_attribute['length'],path_attribute['value']))

                # New entries in ATTOMIC AGGREGATE attribute (table not implemented)
                #elif list(path_attribute['type'].keys())[0] == 6:
                #    print('atomic aggegate:', list(path_attribute['type'].keys())[0])

                # New entries in AGGREGATOR attribute table
                elif list(path_attribute['type'].keys())[0] == 7:
                    insert_aggregatorattributes_table(conn,(list(item['timestamp'].keys())[0],item['timestamp'][list(item['timestamp'].keys())[0]],
                                                        path_attribute_id,path_attribute['length'],path_attribute['value']['as'],path_attribute['value']['id']))

        # New entry in bgp_notificationmessages table
        elif list(item['bgp_message']['type'].keys())[0] == 3:
            insert_bgpnotification_table(conn,(list(item['timestamp'].keys())[0],
                                                item['timestamp'][list(item['timestamp'].keys())[0]],bgp_header_id,
                                                item['bgp_message']['length'],list(item['bgp_message']['type'].keys())[0],
                                                list(item['bgp_message']['error_code'].keys())[0], list(item['bgp_message']['error_subcode'].keys())[0],
                                                item['bgp_message']['data'])
    )

        # New entry in bgp_keepalivemessages table
        elif list(item['bgp_message']['type'].keys())[0] == 4:
            insert_bgpkeepalive_table(conn,(list(item['timestamp'].keys())[0],
                                            item['timestamp'][list(item['timestamp'].keys())[0]],bgp_header_id,
                                            item['bgp_message']['length'],list(item['bgp_message']['type'].keys())[0]))

    # New entry in bgp_statechangemessages table
    elif (list(item['subtype'].keys())[0] == 0) or (list(item['subtype'].keys())[0] == 5):
        insert_bgpstatechangemessages_table(conn,(list(item['timestamp'].keys())[0],
                                    item['timestamp'][list(item['timestamp'].keys())[0]],bgp_header_id,
                                    list(item['old_state'].keys())[0],list(item['new_state'].keys())[0]))
```

Appendix E

BGP-RDB: Descriptions of Core Tables

Fields with header, message, or path attribute details for each core table are described in Table E.1 to Table E.7.

Table E.1: Table *bgp_headers*: Description of Fields.

Field name	**Data Type**	**Description**
id	INTEGER AUTOINCR. UNIQUE	Primary key used to join tables *bgp_openmessages*, *bgp_updatemessages*, *bgp_keepalivemessages*, *bgp_notificationmessages*, or *bgp_statechangemessages* on column 'header_id'
timestamp_unix	INTEGER	Timestamp of collected BGP message in Unix format
timestamp_date	DATETIME	Timestamp of collected BGP message in *yyyy-mm-dd hh:mm:ss* format
mrt_type	INTEGER	To join table *mrt_types* on column 'id'
bgp4mp_bgp4mpet_ subtype	INTEGER	Foreign key used to join table *bgp4mp_bgp4mpet_subtypes* on column 'id'
message_length	INTEGER	Indicates the length of header + BGP message
peer_as	TEXT	AS number of peer sending the BGP message
local_as	TEXT	AS number of peer receiving the BGP message
ifindex	INTEGER	Interface index
afi	INTEGER	To join table table_dump_subtypes on 'id'
peer_ip	TEXT	IP (prefix) of sending peer
local_ip	TEXT	IP (prefix) of receiving peer

Table E.2: Table *bgp_openmessages*: Description of Fields.

Field name	**Data Type**	**Description**
id	INTEGER AUTOINCR. UNIQUE	Primary key
timestamp_unix	INTEGER	Time stamp of collected BGP message in Unix format
timestamp_date	DATETIME	Timestamp of collected BGP message in *yyyy-mm-dd hh:mm:ss* format
header_id	INTEGER	Foreign key used to join table *bgp_headers* on column 'id'
message_length	INTEGER	Length of BGP *open* message
type	INTEGER	Foreign key used to join table *bgp_messages_type* on column 'id'
bgp_version	INTEGER	Length of MULTI_EXIT_DISC path attribute
local_as	INTEGER	AS number of receiving peer
holdtime	INTEGER	Time between receipt of successive *keepalive*/*update* messages
bgp_id	TEXT	IP of receiving peer
optional_parameters_ length	TEXT	Length of optional parameters included in the *open* message

Table E.3: Table *bgp_updatemessages*: Description of Fields.

Field name	**Data Type**	**Description**
id	INTEGER AUTOINCR. UNIQUE	Primary key used to join tables withdawn_routes, announced_routes, or path_attributes on 'bgp_message_id'
timestamp_unix	INTEGER	Time stamp of collected BGP message in Unix format
timestamp_date	DATETIME	Timestamp of collected BGP message in *yyyy-mm-dd hh:mm:ss* format
header_id	INTEGER	Foreign key used to join table *bgp_headers* on column 'id'
message_length	INTEGER	Length of BGP *update* message
type	INTEGER	Foreign key used to join table *bgp_messages_type* on column 'id'
withdrawn_routes_ length	INTEGER	Length of withdrawn routes field
path_attribute_ length	INTEGER	Length of path attributes field
nlri_length	INTEGER	Number of announced routes

Table E.4: Table *bgp_keepalivemessages*: Description of Fields.

Field name	Data Type	Description
id	INTEGER AUTOINCR. UNIQUE	Primary key
timestamp_unix	INTEGER	Time stamp of collected BGP message in Unix format
timestamp_date	DATETIME	Timestamp of collected BGP message in *yyyy-mm-dd hh:mm:ss* format
header_id	INTEGER	Foreign key used to join table *bgp_headers* on column 'id'
message_length	INTEGER	Length of BGP *update* message
type	INTEGER	Foreign key used to join table *bgp_messages_type* on column 'id'

Table E.5: Table *bgp_notificationmessages*: Description of Fields.

Field name	Data Type	Description
id	INTEGER AUTOINCR. UNIQUE	Primary key
timestamp_unix	INTEGER	Time stamp of collected BGP message in Unix format
timestamp_date	DATETIME	Timestamp of collected BGP message in *yyyy-mm-dd hh:mm:ss* format
header_id	INTEGER	Foreign key used to join table *bgp_headers* on column 'id'
message_length	INTEGER	Length of BGP *update* message
type	INTEGER	Foreign key used to join table *bgp_messages_type* on column 'id'
error_code	INTEGER	To join table *bgp_error_codes* on column 'id'
error_subcode	INTEGER	To join table *bgp_error_subcodes* on column 'id'
diagnosis_data	TEXT	Addtional data included to diagnose errors

Table E.6: Table *bgp_statechangemessages*: Description of Fields.

Field name	Data Type	Description
id	INTEGER AUTOINCR. UNIQUE	Primary key
timestamp_unix	INTEGER	Time stamp of collected BGP message in Unix format
timestamp_date	DATETIME	Timestamp of collected BGP message in *yyyy-mm-dd hh:mm:ss* format
header_id	INTEGER	Foreign key used to join table *bgp_headers* on column 'id'
old_state	INTEGER	Foreign key to join table *bgp_finite_state_manchine* that indicates the previous state in the BGP finite state machine
new_state	INTEGER	Foreign key to join table *bgp_finite_state_manchine* that indicates the new state in the BGP finite state machine

Table E.7: Table *bgp_attributes*: Description of Fields.

Field name	**Data Type**	**Description**
id	INTEGER AUTOINCR. UNIQUE	Primary key
timestamp_unix	INTEGER	Time stamp of collected BGP message in Unix format
timestamp_date	DATETIME	Timestamp of collected BGP message in *yyyy-mm-dd hh:mm:ss* format
bgp_message_id	INTEGER	Foreign key used to join table *bgp_updatemessages* on column 'id'
flag	INTEGER	Path attribute flag
type	INTEGER	Foreign key used to join table *bgp_attribute* on column 'id'
length	INTEGER	Path attribute length

Appendix F

BGP-RDB: Descriptions of Lookup Tables

Description of fields in lookup tables is given in Table F.1 to Table F.6.

Table F.1: Table *mrt_types*: Description of Fields.

Field name	**Data Type**	**Description**	**Unique Value**
id	INTEGER NOT NULL UNIQUE	Primary key used to join table *bgp_headers* on column ‘mrt_type’	
type	TEXT UNIQUE	Indicates MRT dump/message type	TABLE_DUMP TABLE_DUMP_V2 BGP4MP BGP4MP_ET

Table F.2: Table *table_dump_subtypes*: Description of Fields.

Field name	**Data Type**	**Description**	**Unique Value**
id	INTEGER NOT NULL UNIQUE	Primary key used to join table *bgp_headers* on column ‘afi’	
subtype	TEXT UNIQUE	Indicates table dump subtype	AFI_IPv4 AFI_IPv6

Table F.3: Table *table_v2_dump_subtypes*: Description of Fields.

Field name	**Data Type**	**Description**	**Unique Value**
id	INTEGER NOT NULL UNIQUE	Primary key	
subtype	TEXT UNIQUE	Indicates table dump v2 subtype	PEER_INDEX_TABLE RIB_IPV4_UNICAST RIB_IPV4_MULTICAST RIB_IPV6_UNICAST RIB_IPV6_MULTICAST RIB_GENERIC RIB_IPV4_UNICAST_ADDPATH RIB_IPV4_MULTICAST_ADDPATH RIB_IPV6_UNICAST_ADDPATH RIB_IPV6_MULTICAST_ADDPATH RIB_GENERIC_ADDPATH

Table F.4: Table *bgp4mp_bgp4mpet_subtypes*: Description of Fields.

Field name	**Data Type**	**Description**	**Unique Value**
id	INTEGER NOT NULL UNIQUE	Primary key to join table *bgp_headers* on column 'bgp4mp_ bgp4mpet_ subtype'	
bgp4mp_subtype	TEXT UNIQUE	Indicates the BGP4MP/ BGP4MP_ET subtype	BGP4MP_STATE_CHANGE BGP4MP_MESSAGE BGP4MP_MESSAGE_AS4 BGP4MP_STATE_CHANGE_AS4 BGP4MP_MESSAGE_LOCAL BGP4MP_MESSAGE_AS4_LOCAL BPG4MP_MESSAGE_ADDPATH BGP4MP_MESSAGE_AS4_ADDPATH BP4MP_MESSAGE_LOCAL_ADDPATH BGP4MP_MESSAGE_AS4_LOCAL_ ADDPATH

Table F.5: Table *bgp_messages_type*: Description of Fields.

Field name	**Data Type**	**Description**	**Unique Value**
id	INTEGER NOT NULL UNIQUE	Primary key to join table *bgp_messages* on column 'type'	
message_type	TEXT UNIQUE	Indicates the BGP message type	OPEN UPDATE NOTIFICATION KEEPALIVE

Table F.6: Table *bgp_attributes_category*: Description of Fields.

Field name	**Data Type**	**Description**	**Unique Value**
id	INTEGER NOT NULL UNIQUE	Priamry key to join tabe *bgp_attribute* on column ‘ebgp_category’ or ‘ibgp_category’	
attribute_category	TEXT UNIQUE	Indicates the BGP attribute category	WELL-KNOWN MANDATORY WELL-KNOWN DISCRETIONARY OPTIONAL TRANSITIVE OPTIONAL NON-TRANSITIVE

Appendix G

BGP-RDB: Descriptions of List Tables

Fields contained in list tables are described in Table G.1 to Table G.7.

Table G.1: Table *withdrawn_routes*: Description of Fields.

Field name	**Data Type**	**Description**
id	INTEGER AUTOINCR. UNIQUE	Primary key
timestamp_unix	INTEGER	Time stamp of collected BGP message in Unix format
timestamp_date	DATETIME	Timestamp of collected BGP message in *yyyy-mm-dd hh:mm:ss* format
bgp_message_id	INTEGER	Foreign key used to join table *bgp_updatemessages* on column 'id'
prefix_length	INTEGER	Length of the withdrawn routes prefix field
prefix	TEXT	IPs (prefixes) of withdrawn routes

Table G.2: Table *announced_routes*: Description of Fields.

Field name	**Data Type**	**Description**
id	INTEGER AUTOINCR. UNIQUE	Primary key
timestamp_unix	INTEGER	Time stamp of collected BGP message in Unix format
timestamp_date	DATETIME	Timestamp of collected BGP message in *yyyy-mm-dd hh:mm:ss* format
bgp_message_id	INTEGER	Foreign key used to join table *bgp_updatemessages* on column 'id'
prefix_length	INTEGER	Length of the NLRI field containing announced routes
prefix	TEXT	IPs (prefixes) of announced routes

Table G.3: Table *as_paths*: Description of Fields.

Field name	**Data Type**	**Description**
id	INTEGER AUTOINCR. UNIQUE	Primary key
timestamp_unix	INTEGER	Time stamp of collected BGP message in Unix format
timestamp_date	DATETIME	Timestamp of collected BGP message in *yyyy-mm-dd hh:mm:ss* format
path_attribute_id	INTEGER	Foreign key used to join table *path_attributes* on column 'id'
segment_type	INTEGER	Foreign key used to join table *as_path_segments_type* on column 'id'
aspath_length	INTEGER	Indicates the length of AS_PATH path attribute
as_path	INTEGER	Indicates the sequence or set of ASes

Table G.4: Table *aggregator_attributes*: Description of Fields.

Field name	**Data Type**	**Description**
id	INTEGER AUTOINCR. UNIQUE	Primary key
timestamp_unix	INTEGER	Time stamp of collected BGP message in Unix format
timestamp_date	DATETIME	Timestamp of collected BGP message in *yyyy-mm-dd hh:mm:ss* format
path_attribute_id	INTEGER	Foreign key used to join table *path_attributes* on column 'id'
length	INTEGER	Indicates the length of the AGGREGATOR path attribute
as_id	INTEGER	Last AS number that formed the aggregate route
as_ip	TEXT	IP/prefix of the last AS that formed the aggregate route

Table G.5: Table *origin_pathattribute*: Description of Fields.

Field name	**Data Type**	**Description**
id	INTEGER AUTOINCR. UNIQUE	Primary key
timestamp_unix	INTEGER	Time stamp of collected BGP message in Unix format
timestamp_date	DATETIME	Timestamp of collected BGP message in *yyyy-mm-dd hh:mm:ss* format
path_attribute_id	INTEGER	Foreign key used to join table *path_attributes* on column 'id'
attribute_length	INTEGER	Indicates the length of ORIGIN path attribute
origin_id	INTEGER	Foreign key used to join table *origin_values* on column 'id'

Table G.6: Table *nexthop_pathattribute*: Description of Fields.

Field name	**Data Type**	**Description**
id	INTEGER AUTOINCR. UNIQUE	Primary key
timestamp_unix	INTEGER	Time stamp of collected BGP message in Unix format
timestamp_date	DATETIME	Timestamp of collected BGP message in *yyyy-mm-dd hh:mm:ss* format
path_attribute_id	INTEGER	Foreign key used to join table *path_attributes* on column ‘id’
attribute_length	INTEGER	Indicates the length of NEXT_HOP path attribute
nexthop_ip	TEXT	IP (prefix) of router that should be used as next hop

Table G.7: Table *multiexitdisc_pathattribute*: Description of Fields.

Field name	**Data Type**	**Description**
id	INTEGER AUTOINCR. UNIQUE	Primary key
timestamp_unix	INTEGER	Time stamp of collected BGP message in Unix format
timestamp_date	DATETIME	Timestamp of collected BGP message in *yyyy-mm-dd hh:mm:ss* format
path_attribute_id	INTEGER	Foreign key used to join table *path_attributes* on column ‘id’
attribute_length	INTEGER	Indicates the length of MULTI_EXIT_DISC path attribute
value	INTEGER	Preferred value to be used when discriminating among multiple entry/exit points to the same AS neighbor

Appendix H

BGP-RDB: Descriptions of Detail Tables

Descriptions of fields for detail tables are listed in Table H.1 to Table H.8.

Table H.1: Table *bgp_attribute*: Description of Fields.

Field name	**Data Type**	**Description**	**Unique Value**
id	INTEGER NOT NULL	Primary key to join table *path_attributes* on column 'type'	
attribute	TEXT UNIQUE	Indicates the BGP attribute	ORIGIN AS_PATH NEXT_HOP MULTI_EXIT_DISC LOCAL_PREF ATOMIC_AGGREGATE AGGREGATOR COMMUNITY ORIGINATOR_ID CLUSTER_LIST MP_REACH_NLRI MP_UNREACH_NLRI EXTENDED COMMUNITIES AS4_PATH AS4_AGGREGATOR AIGP LARGE_COMMUNITY ATTR_SET
ebgp_category	INTEGER	Foreign key used to join table *bgp_attributes_category* on foreign key 'id'	
ibgp_category	INTEGER	Foreign key used to join table *bgp_attributes_category* on foreign key 'id'	

Table H.2: Table *as_path_segments_types*: Description of Fields.

Field name	Data Type	Description	Unique Value
id	INTEGER NOT NULL	Primary key to join table *as_paths* on column 'segment_type'	
segment_type	TEXT UNIQUE	Indicates the AS_PATH segment type	AS_SET AS_SEQUENCE

Table H.3: Table *origin_values*: Description of Fields.

Field name	Data Type	Description	Unique Value
id	INTEGER NOT NULL	Primary key to join table *origin_pathattribute* on column 'origin_id'	
segment_type	TEXT UNIQUE	Indicates the origin of the path	EGP IGP INCOMPLETE

Table H.4: Table *bgp_error_codes*: Description of Fields.

Field name	Data Type	Description	Unique Value
id	INTEGER NOT NULL	Primary key to join table *bgp_notificationmessages* on column 'error_code'	
symbolic_name	TEXT UNIQUE	Description of of the error code condition	Message Header Error OPEN Message Error UPDATE Message Error Hold Timer Expired Finite State Machine Error Cease

Table H.5: Table *bgp_messageheader_error_subcodes*: Description of Fields.

Field name	Data Type	Description	Unique Value
id	INTEGER NOT NULL	Primary key to join table *bgp_notificationmessages* on column 'error_subcode'	
description	TEXT UNIQUE	Description of message header error subcode	Unspecific Connection Not Synchronized Bad Message Length Bad Message Type

Table H.6: Table *bgp_openmessage_error_subcodes*: Description of Fields.

Field name	Data Type	Description	Unique Value
id	INTEGER NOT NULL	Primary key to join table *bgp_notificationmessages* on column 'error_subcode'	
description	TEXT UNIQUE	Description of open message error subcode	Unspecific Unsupported Version Number Bad BGP Identifier Unsupported Optional Parameter Authentication Failure Unacceptable Hold Time

Table H.7: Table *bgp_updatemessage_error_subcodes*: Description of Fields.

Field name	Data Type	Description	Unique Value
id	INTEGER NOT NULL	Primary key to join table *bgp_notificationmessages* on column 'error_subcode'	
description	TEXT UNIQUE	Description of update message error subcode	Unspecific Unsupported Version Number Malformed Attribute List Unrecognized Well-known Attribute Missing Well-known Attribute Attribute Flags Error Attribute Length Error Invalid ORIGIN Attribute AS Routing Loop Invalid NEXT_HOP Attribute Optional Attribute Error Invalid Network Field Malformed AS_PATH

Table H.8: Table *bgp_finite_state_machine*: Description of Fields.

Field name	Data Type	Description	Unique Value
id	INTEGER NOT NULL	Primary key to join table *bgp_statemessages* on column 'old_state' and 'new_state'	
description	TEXT UNIQUE	Description of states in the BGP finite state machine	Idle Connect Active Open Sent Open Confirm Established

Appendix I

VFBLS and VCFBLS Algorithms: Pseudocode

Algorithm 1 VFBLS and VCFBLS Algorithms: Pseudocode

```
1: procedure VFBLS AND VCFBLS(training dataset $\boldsymbol{X}$ with labels $\boldsymbol{Y}$)
2:     Initialize:
3:     Number of subsets $v$ of input data
4:     Number $f$ of features to be selected in each subset
5:     Subsets of input data $\boldsymbol{X}_v$, $v = (1, ..., f)$
6:     Sets of groups of mapped features $\boldsymbol{Z}^{n_v}$, $n_v = (n_1, ..., n_f)$
7:     Groups of mapped features in each set $\boldsymbol{Z}_i$, $i = (k, ..., p)$
8:     Number of mapped features in each group
9:     for each $f$ in $v$ do
10:        Calculate feature importance and create $\mathcal{F}(\boldsymbol{X})$ by ranking features using a feature
           selection algorithm
11:        Generate subset $\boldsymbol{X}_v = \mathcal{F}(\boldsymbol{X}), v = 1, 2, ..., f$
12:        for each set $\boldsymbol{Z}^{n_i}$ in $\boldsymbol{Z}^{n_v}$ do
13:            Initialize the set of groups of mapped features $\boldsymbol{Z}^{n_i}$
14:            for each group $\boldsymbol{Z}_i$ in set $\boldsymbol{Z}^{n_i}$ do
15:                switch Algorithm do
16:                    case VFBLS
17:                        Generate $\boldsymbol{Z}_i$ based on $\boldsymbol{X}_v$ and the number of mapped features
18:                    case VCFBLS
19:                        Generate $\boldsymbol{Z}_1$ based on $\boldsymbol{X}_v$ and the number of mapped features
20:                        Generate subsequent groups $\boldsymbol{Z}_i$ based on $\boldsymbol{Z}_{i-1}$
21:                Insert $\boldsymbol{Z}_i$ into $\boldsymbol{Z}^{n_i}$
22:            end for
23:            Insert $\boldsymbol{Z}^{n_i}$ into $\boldsymbol{Z}^{n_v}$
24:        end for
25:    end for
26:    Construct matrix $\boldsymbol{Z}^t = [\boldsymbol{X}|\boldsymbol{Z}^{n_v}]$
27:    Generate enhancement nodes $\boldsymbol{H}^m = [\boldsymbol{H}_1, ..., \boldsymbol{H}_m]$ based on $\boldsymbol{Z}^t$
28:    Concatenate $\boldsymbol{Z}^t$ and $\boldsymbol{H}^m$ to create the state matrix $\boldsymbol{A}_t^m$
29:    Compute weights $\boldsymbol{W}_t^m$ based on $\boldsymbol{A}_t^m$ and labels $\boldsymbol{Y}$ using the ridge regression algorithm
30: end procedure
```

Algorithm 2 Incremental VFBLS and VCFBLS Algorithms: Pseudocode

```
 1: procedure INCREMENTAL VFBLS AND VCFBLS(training dataset $\boldsymbol{X}$ with labels $\boldsymbol{Y}$)
 2:     Extract initial input subset $\boldsymbol{X}_0$ from dataset $\boldsymbol{X}$
 3:     Extract initial labels subset $\boldsymbol{Y}_0$ from $\boldsymbol{Y}$
 4:     Initialize:
 5:     Number of incremental learning steps: $l$
 6:     Number of data points per step: $d$
 7:     Number of enhancement nodes per step: $e$
 8:     Calculate feature weight vector $W_i = [w_0, w_1, ..., w_l]$: $w_0 = \boldsymbol{X}_0/\boldsymbol{X}$; $w_1, ..., w_l = (1 - w_0)/l$
 9:     for each step in $l$ do
10:         Generate $\boldsymbol{X}_a$ based on $\boldsymbol{X}$, $\boldsymbol{X}_0$, and $d$
11:         Generate $\boldsymbol{Y}_a$ based on $\boldsymbol{Y}$, $\boldsymbol{Y}_0$, and $d$
12:         Calculate feature importance and create $\mathcal{F}(\boldsymbol{X}_a)$ by
            ranking features using a feature selection algorithm
13:         Generate additional mapped features $\boldsymbol{Z}_{n+1}$ and additional
            enhancement nodes $\boldsymbol{H}_{m+1}$ using Algorithm 1
14:         Update $\boldsymbol{A}_t^m$
15:         Update weights $\boldsymbol{W}_t^m$
16:     end for
17:     Rank and select features to be used in testing based on:
        selected features and their importance in each step
        and the weight vector $W_i$
18: end procedure
```