Doctoral Dissertation

Massively Parallel Empirical Dynamic Modeling for Network Traffic Analysis

Wassapon Watanakeesuntorn

Program of Information Science and Engineering
Graduate School of Science and Technology
Nara Institute of Science and Technology

Supervisor: Professor Hajimu Iida
Software Design and Analysis Laboratory (Division of Information Science)

Submitted on September 16, 2022

A Doctoral Dissertation
submitted to Graduate School of Science and Technology,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Wassapon Watanakeesuntorn

Massively Parallel Empirical Dynamic Modeling for Network Traffic Analysis[*]

Wassapon Watanakeesuntorn

Abstract

A nonlinear dynamical system is a dynamical system in which the next state of the system is described as a nonlinear function of time and current state. Modeling a nonlinear dynamical system is one of the challenging topics. Empirical Dynamic Modeling (EDM) is a data-driven framework for modeling nonlinear dynamical systems. EDM has traditionally been used to model nonlinear dynamical systems in biology, neurology, oceanography, and other fields. However, EDM is rarely applied in computer science. In computer science, time series prediction is a popular topic that many researches propose multiple machine learning techniques to tackle it. This presents an opportunity to introduce EDM into computer science.

In this dissertation, I aim to utilize EDM to predict incoming traffic in a network and classify anomalous traffic in real-time. To achieve this goal, I tackle the following three challenges: (1) accelerating EDM computation to enable real-time analysis of network traffic, (2) capturing network traffic in an SDN in real-time with low-overhead, and (3) applying EDM for network traffic prediction. First, a new EDM library, named *mpEDM*, is implemented that supports large-scale analysis on HPC systems. mpEDM uses an improved EDM algorithm and it is fully optimized to accelerate the computation. mpEDM is up to 1,530× faster compared to an existing EDM implementation. Second, a transparent and low-overhead monitoring system for SDN-based networks, named *Opimon*, is developed. Opimon monitors a network by interposing a proxy between the

[*]Doctoral Dissertation, Graduate School of Science and Technology, Nara Institute of Science and Technology, September 16, 2022.

controller and switches and intercepting every control message exchanged between them. This design allows Opimon to be compatible with any SDN switch or controller. The overhead imposed to the network latency by Opimon is less than 0.5 μs at maximum. Finally, I leverage EDM to predict the network traffic in SDN environments. EDM is used to predict the incoming network traffic and detect DDoS attacks. I compared the prediction and classification results with traditional machine learning algorithms, which are auto regression (AR) and Long short-term memory (LSTM). The results show that EDM is capable of predicting the time series faster than LSTM and the classification result using EDM prediction provides higher classification accuracy than the AR-based model.

Keywords:

Empirical Dynamic Modeling (EDM), High Performance Computing (HPC), Software-Defined Networking (SDN), Machine Learning, Time Series Analysis

Contents

List of Figures

List of Tables

1. Introduction

A *dynamical system* is a system where its state evolves over time based on a fixed set of rules. A *nonlinear dynamical system* is a dynamical system in which the next state of the system is described as a nonlinear function of time and current state [1]. Nonlinear dynamical systems commonly appear in a wide variety of problems in science and engineering [2, 3]. Since even a simple nonlinear dynamical system can exhibit extremely complex and seemingly unpredictable (*i.e.,* chaotic) behaviors, finding out the underlying rules of a nonlinear dynamical system and predicting its future state is challenging.

Empirical Dynamic Modeling (EDM) is a data-driven framework for modeling nonlinear dynamical systems. EDM is non-parametric, meaning that it does not assume any prior knowledge of the underlying rules of a dynamic system to be modeled. Instead, it reconstructs the trajectory of a system in the state space and uses the reconstructed trajectory to model the system. EDM is capable of predicting the future state of a system, determining the complexity of a system, and finding causal relationships between variables in a system. EDM has traditionally been used to model nonlinear dynamical systems in biology, neurology, oceanography, and other fields [4–7]. However, EDM has rarely been applied in computer science [8]. In computer science, time series prediction is a popular topic that many researches propose multiple machine learning techniques to tackle it [9]. This presents an opportunity to introduce EDM into computer science field.

1.1 Motivation and Goal

Predicting computer network traffic is one of the challenging tasks. In particular, the recently emerged Software-Defined Networking (SDN), which provides flexibility to the network control, makes this problem even more challenging. This is because the flexible network control and management enabled by SDN allow the network traffic to change rapidly. This behavior of SDN traffic could be viewed as a nonlinear dynamical system, which is suitable for analyzing with EDM. Therefore, in this dissertation, I aim to utilize EDM to predict network traffic in SDN. In particular, I use EDM to predict incoming traffic in a network and classify anomalous traffic in real-time. To achieve this goal, I tackle the following three challenges:

1. **Accelerating EDM computation to enable real-time analysis of network traffic**: A high-performance EDM library is needed to predict large-scale time series. The current *de facto* standard EDM implementation, cppEDM, is designed for commodity PCs and does not support modern High-Performance Computing (HPC) hardware such as GPUs that enable large-scale analysis. cppEDM also suffers from performance issues due to redundant computation. As a result, processing long time series takes hours.

2. **Capturing network traffic in an SDN in real-time with low-overhead**: Time series of the network is used as an input for network traffic prediction of EDM. The network information is required to be collected from actual network traffic before converting into a time series dataset. To collect the highest-quality network traffic dataset, a low-overhead SDN monitoring system is needed to precisely collect network traffic information in real-time and feed it as an input to EDM. The example of network information is network throughput, traffic volume, and packet headers. However, a low-overhead and framework-independent SDN monitoring system does not yet exist. Existing monitoring systems either rely on a specific controller framework or require a modification to the controller. This makes it difficult to monitor production network without causing interference, which makes it difficult to monitor the network and convert the network traffic data into the time series.

3. **Applying EDM for network traffic prediction**: EDM is introduced to replace machine learning for computer network prediction. Computer network prediction with EDM is executed with the combine of the achievements from mpEDM and Opimon. The monitored network traffic is processed to time series by Opimon and fed as an input of the EDM library to predict the incoming network traffic. The results of the prediction are compared with existing machine learning techniques to evaluate the capability of EDM in network traffic prediction.

To tackle the first challenge, I implement a new EDM library, named *mpEDM*, that supports a large-scale analysis on HPC systems. mpEDM uses an improved

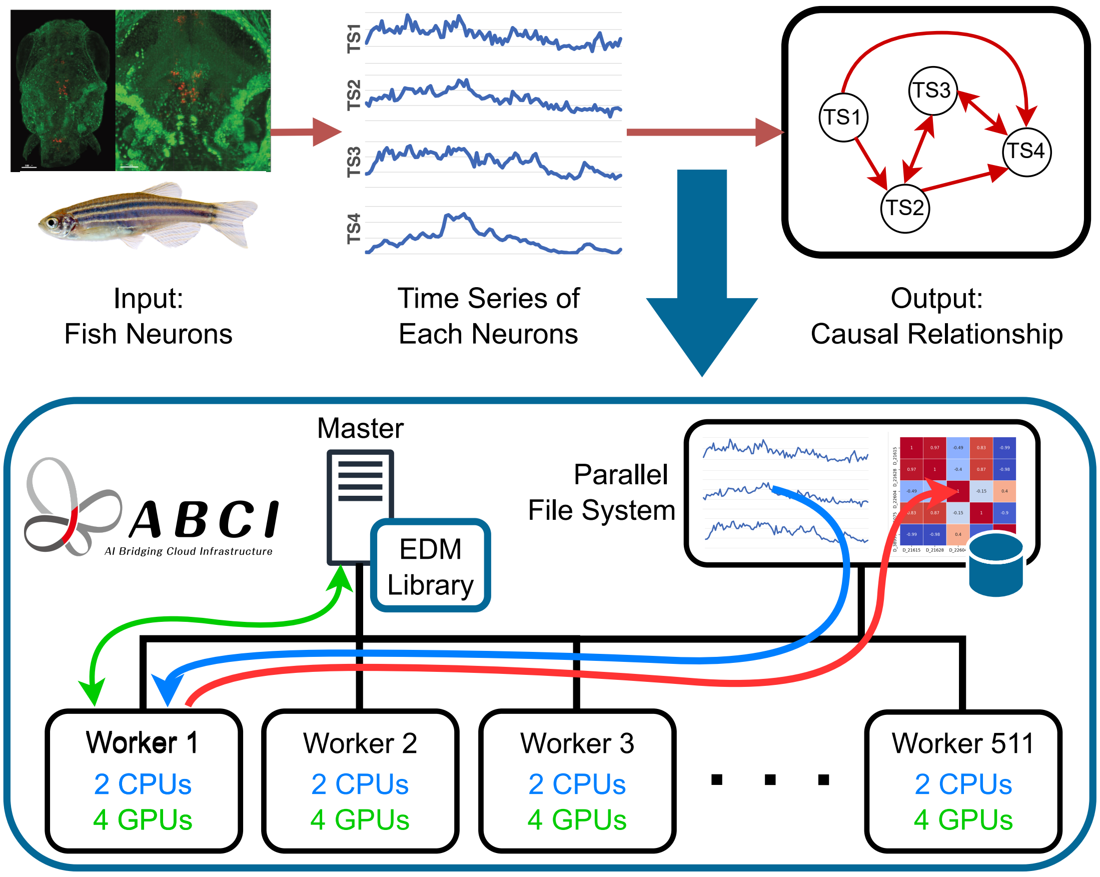


Figure 1: Concept of a high-performance EDM with mpEDM

EDM algorithm from cppEDM and it is fully optimized for modern GPU-centric supercomputers to accelerate the computation. It fully utilizes hardware resources such as GPUs and SIMD units. It also supports intra-node and inter-node parallelism, which is not supported in cppEDM. Figure 1 shows the concept of mpEDM along with its first application, building a causal map of neurons in a Zebrafish brain. The performance of mpEDM was evaluated on the AI Bridging Cloud Infrastructure (ABCI), one of the most powerful supercomputers in Japan. I compared the performance of mpEDM with cppEDM using the same dataset and hardware resources. The results show that mpEDM reduces the runtime from 8.5 hours of cppEDM down to 20 seconds, which is 1,530× faster. Additionally, mpEDM is capable of processing a 13× larger dataset under 200 seconds.

To tackle the second challenge, I develop Opimon, a transparent and low-

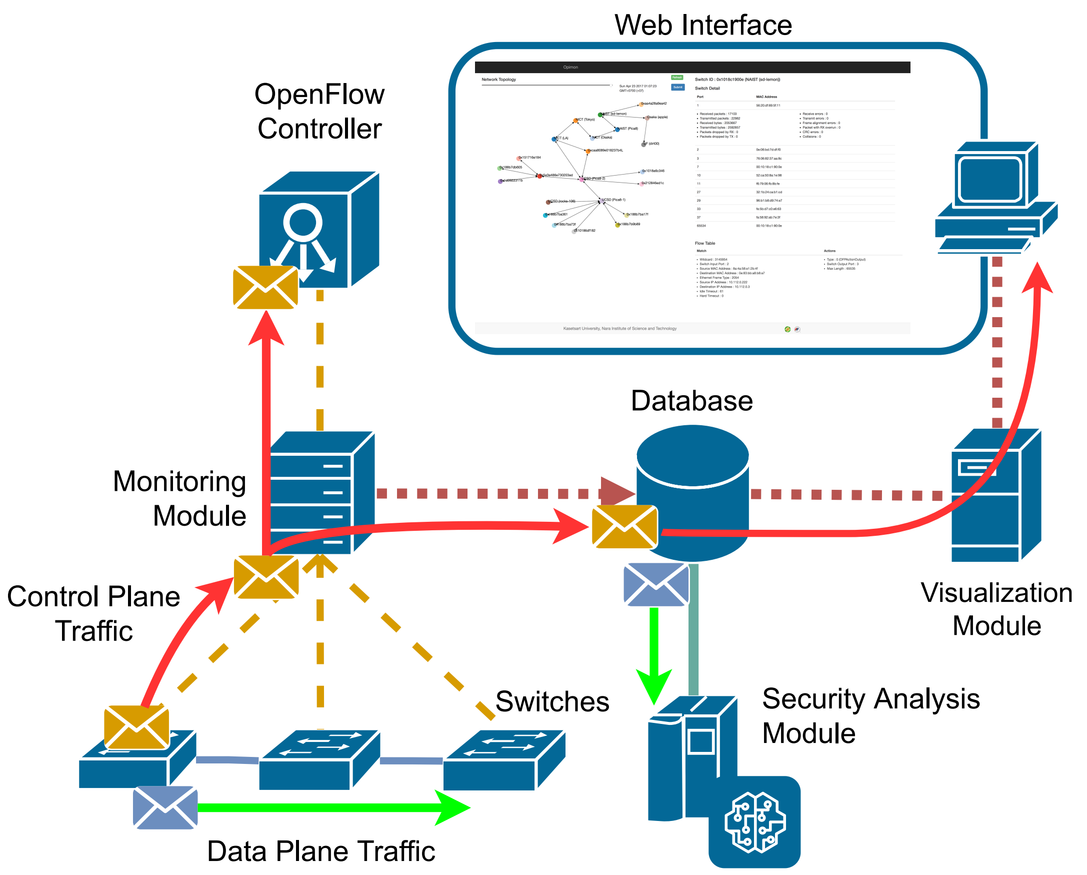


Figure 2: Concept of a transparent SDN monitoring system with Opimon

overhead monitoring system for SDN-based networks. The concept of Opimon is shown in Fig. 2. Opimon monitors a network by interposing a proxy between the controller and switches and intercepting every control message exchanged between them. This design allows Opimon to be compatible with any SDN switch or controller. Opimon monitors the network topology, switch statistics, and flow tables in a SDN network and visualizes the result through a web interface in real-time. The information from Opimon is used for visualization on web interface and analysis of the network traffic in the SDN network to detect issues on the network, such as Distributed Denial of Service (DDoS) attacks. Additionally, the monitored network traffic is converted the network packets into time series for feeding to EDM. I tested the functionalities of Opimon on a virtual network and a large-scale international SDN testbed. I also measured the monitoring overhead

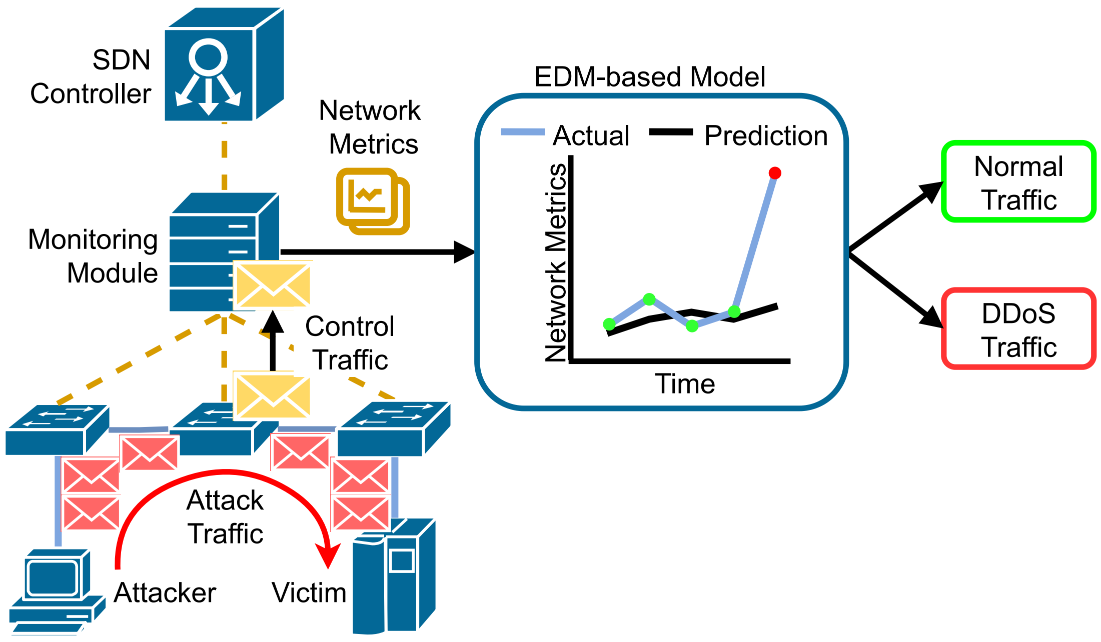


Figure 3: Concept of network traffic predictions with EDM

of Opimon. The results indicated that the overhead in terms of latency is less than 0.5 μs at maximum, which is just 3% compared to the total latency.

To solve the third challenge, I leverage EDM to predict the network traffic in SDN environments. The concept of this work is shown in Fig. 3. In this research, EDM is used to predict the future values of various network metrics (*e.g.,* throughput of the network traffic) from recent observations. Simplex projection function, one of the fundamental EDM algorithms, is used to predict and evaluate the prediction. In the preliminary experiment, the data plane dataset only is used to evaluate the predictions. Opimon is used to collect, pre-process, and extract various network metrics from the network traffic. kEDM, an improved version of mpEDM, is used to carry out EDM computations. The prediction result is compared with the prediction results from auto regression (AR) and long short-term memory (LSTM). A classification model is trained to distinguish between normal traffic and anomalous traffic, which is assumed to be a DDoS attack traffic. The results show that EDM is capable of predicting the time series faster than LSTM and the classification result using EDM prediction provides higher classification accuracy than the AR-based model.

1.2 Organization of the Dissertation

The rest of this dissertation is structured as follows. Chapter 2 describes the design and implementation of a massive parallel implementation of Empirical Dynamic Modeling (EDM) library, mpEDM. It also describes the first application to evaluate the performance of mpEDM, which is a causal inference of a whole Zebrafish brain at single neuron resolution. Chapter 3 describes the design and implementation of an OpenFlow network monitoring system, Opimon. This chapter describes the architecture of Opimon to monitor the network, visualize the information of the network on web interface, and detect an DDoS attack in the network with machine learning. I also describe the experimental results of Opimon on a simulated network and a real OpenFlow network testbed. In this chapter, I benchmark Opimon for evaluating the practicality of the tool under normal and heavily-loaded traffic conditions. Additionally, I describe the preliminary result of the performance comparison of several machine learning techniques in terms of detection accuracy and runtime. Chapter 4 describes traffic prediction and classification with EDM. This chapter describes a feasibility study of applying EDM to network traffic predictions. I also compared the predicted performance of EDM, auto regression (AR), and long short-term memory (LSTM) to predict the upcoming traffic loads. I also used the predicted traffic results to classify between normal traffic and DDoS attack traffic. Finally, Chapter 5 concludes this dissertation and discusses future work.

2. A Massively Parallel Implementation of Empirical Dynamic Modeling (EDM)

2.1 Introduction

Reverse-engineering and building a digital reconstruction of the brain is one of the greatest scientific challenges of today. A recent study on the mouse cortex [10] showed that 97% of the possible connections between neurons exist. This result suggests that it is likely more informative to investigate the dynamic interactions between neurons rather than the static connectivity between them to fully understand the function of the brain. Based on this insight, mathematical and computational tools are being built to analyze the dynamic interactions between neurons based on Empirical Dynamic Modeling (EDM).

EDM is a nonlinear time series causal inference framework based on the generalized Takens' embedding theorem on state space reconstruction [11]. EDM is used to study and predict the behavior of nonlinear dynamical systems. Convergent Cross Mapping (CCM) is one of the EDM algorithms that allows to estimate the existence and strength of the causal strength between two time series in a dynamical system [12].

This study utilizes CCM to infer the causal relationships between every neuron in an entire brain and construct a causal map that describes the dynamic interactions among neurons. For this purpose, the neural activity (*i.e.* firing rate) of an entire larval zebrafish brain (*Danio rerio*) have been recorded at singe-neuron resolution by using light sheet fluorescence microscopy. The original implementation of EDM, cppEDM, has mostly been used for individual time series of relatively short length and and mostly small numbers of variables for its computational cost. Since a larval zebrafish brain contains approximately 10^5 neurons, a staggering number of 10^{10} cross mappings need to be performed in total. CCM of this enormous scale has never been achieved so far because of the sheer amount of computation required.

The goal of this paper is to develop a highly scalable and optimized implementation of EDM that is able to analyze the whole zebrafish brain dataset within a reasonable time. I present mpEDM[1], a parallel distributed implementation of EDM optimized for execution on modern GPU-centric supercomputers. I improve the original algorithm in cppEDM to reduce redundant computation and optimize the implementation to fully utilize hardware resources such as GPUs and SIMD units.

An evaluation on AI Bridging Cloud Infrastructure (ABCI), Japan's most high performance supercomputer as of today, demonstrated the unprecedented performance of mpEDM. mpEDM was used to analyze a dataset containing the activity of 53,053 neurons in only 20 seconds using 512 ABCI nodes. In contrast, cppEDM took 8.5 hours to analyze the same dataset using the same number of nodes [13]. Furthermore, mpEDM analyzed a larger dataset containing 101,729 neurons in 199 seconds on 512 nodes. To my knowledge, this is the largest CCM calculation achieved to date. This result shows the potential for mpEDM and ABCI to analyze even larger datasets in the future.

2.2 Background

2.2.1 Causal Map of the Zebrafish Brain at Single Neuron Resolution

To understand the human brain activity dynamics with a complexity of 10^{11} neurons and 10^{15} synapses at single neuron resolution is currently a technically impossible task. Similarly a mouse brain with 7.6×10^7 neurons is not tractable because mammalian brains are opaque and it is impossible to image a complete mouse brain. With this in mind, the zebrafish embryo is an attractive model system with 120,000 neurons and transgenic technology as well as natural brain transparency. The zebrafish embryo is sufficiently complex to exhibit interesting behaviors and is technologically feasible to study to infer basic principles of systems neuroscience. Even in the case of the larval zebrafish with about 120,000 neurons we do not have the physical connectivity map, that is the connectome of the larval zebrafish, nor do we have the synaptic strengths which are pieces of information required to understand the brain starting from the physical connectivity.

Complicating this notion, recent work from the mouse brain shows that 97%

[1]`https://github.com/keichi/mpEDM`

of possible physical connections exist within the mouse cortex thus making it difficult to analyze. Given this difficulty, using an analogy of a city; to understand how a city works it will be easier to understand the city from the traffic patterns than from the street map. Thus, we wished to analyze the fish brain at single neuron resolution from a network activity dynamics perspective. Although imperfect, neural activity imaging data of an entire brain at single cell resolution in a behaving larval zebrafish (a transparent vertebrate) is used to extract all relationships in an intact vertebrate brain.

To achieve this, Whole brain neural activity patterns are recorded in multiple animals experiencing hypoxia using a Selective Plane Illumination Microscope (SPIM) [14]. The data was obtained from the entire 5-day-old larval brain (120,000 neurons) at 2 Hz in response to hypoxia for varying amounts of time typically ranging from 1,500 time steps to up to 8,000+ [15].

CCM allows the inference of causation from nonlinear time series even with substantial noise and complete absence of correlation [16, 17]. CCM and other tools from the EDM framework are used for the inference of existence, strength and sign of causal relationships within the neural activity network of the transparent larval fish brain [14]. CCM determines whether and how much causality exists between individual neurons. The adjacency in the network is determined by time delay cross mapping [17]. Predictive accuracy values give the interaction strength allow us to infer relationships within the neural network without observing the physical connectivity. A test case has collected multiple data sets of lengths around 1600 time steps at 2 Hz which contain 50,000–80,000 active neurons in most cases. This data has been analyzed and show that the generated time series are suitable for causal network inference using the EDM framework and thus demonstrated a proof of principle of computational tractability.

2.2.2 Empirical Dynamic Modeling

EDM is a mathematical framework designed for studying nonlinear dynamical systems. EDM is based upon the concept of state space reconstruction (SSR) [18]. Takens' theorem states that the attractor manifold of a multivariate dynamical system can be reconstructed from time lagged coordinates of a single time series variable [19]. Figure 4 illustrates the concept of state spaces reconstruction. In

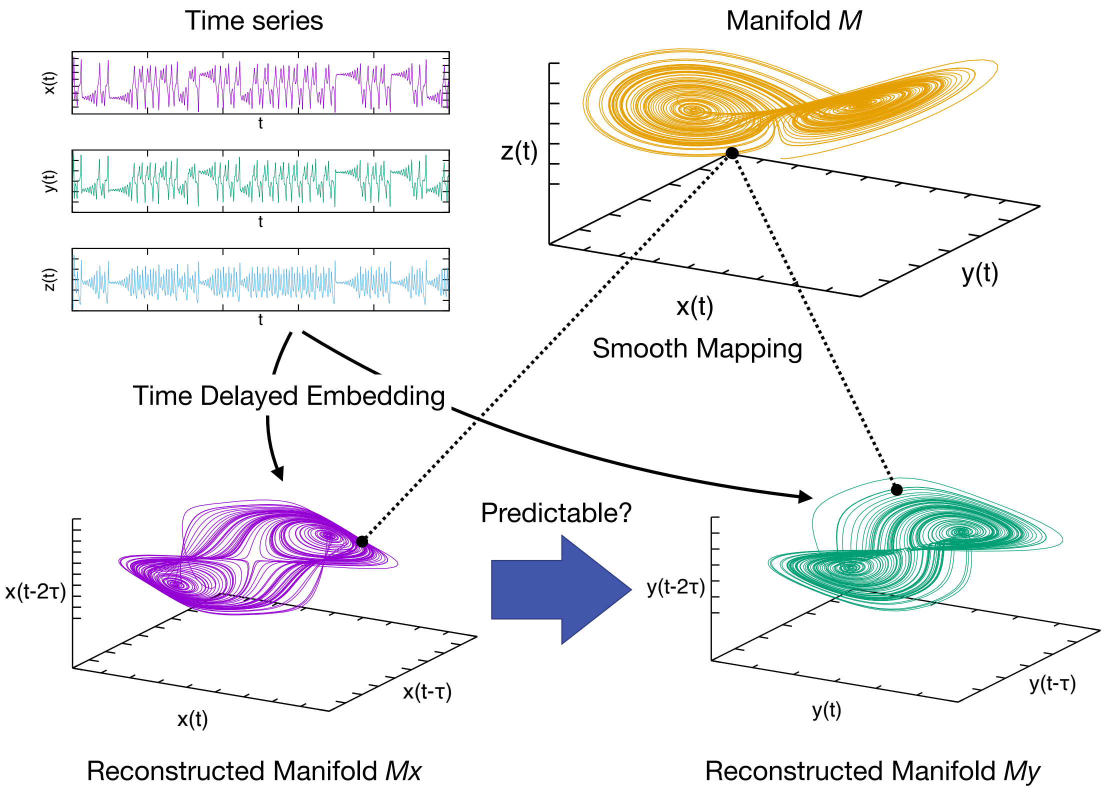


Figure 4: Basic idea behind Empirical Dynamic Modeling (EDM)

this example, three causally related time series variables $x(t)$, $y(t)$ and $z(t)$ that constitute a dynamical system form an attractor manifold M in the state space. A shadow manifold M_x can be reconstructed using the time delayed embeddings of x $(x(t), x(t-\tau), x(t-2\tau), \dots)$, where τ denotes the time lag. In the same manner, lags of y form a shadow manifold M_y. Takens' theorem states that the reconstructed manifolds M_x and M_y preserve essential mathematical properties (such as the topology) of the true manifold M. In particular, there exist *smooth* mappings between M, M_x, and M_y, suggesting that neighbors in M_x are neighbors in M_y as well.

Simplex projection is a nonlinear forecasting algorithm often used for estimating the dimensionality of a dynamical system. In simplex projection, the input time series is split into two halves: library x and target y. Both halves are embedded into E-dimensional state space by using delayed embeddings. Given a point $\boldsymbol{y}(t_p) = (y(t_p), y(t_p - 1), \dots, y(t_p - E + 1))$ in the target state space, its $E+1$ nearest neighbors (*i.e.* vertices of the simplex enclosing $\boldsymbol{y}(t_p)$) are searched from the embedded library. Suppose those neighbors are $\boldsymbol{x_1}(t_1), \boldsymbol{x_2}(t_2), \dots, \boldsymbol{x}_{E+1}(t_{E+1})$. A forecast $\boldsymbol{y}(t_p + 1)$ can be made by averaging the future of the neighbors in the library: $\boldsymbol{x_1}(t_1 + 1), \boldsymbol{x_2}(t_2 + 1), \dots, \boldsymbol{x}_{E+1}(t_{E+1} + 1)$. This prediction is performed for every point in $\boldsymbol{y}$ and the results are compared with the true y to evaluate the prediction accuracy. This entire procedure is repeated for different E values and the E that achieves the highest prediction accuracy is determined as the optimal embedding dimension of the dynamical system.

CCM determines the existence and strength of causality between two time series variables [20]. It works similar to simplex projection, but instead of predicting within a single time series, CCM predicts one time series from another. If y can be predicted from x with significant accuracy, I conclude that *y CCM causes x*.

There have been extensive studies on causal inference. Structural Causal Model (SCM) is one of the most popular causal models [21] based on statistical modeling of equilibrium systems. In contrast to SCM, EDM is based on the principle of state-space reconstruction shown in Takens ' theorem of non-equilibrium systems. Granger causality is another causal inference technique based on statistical modeling [22]. Granger causality however as stated by Granger himself, only works with linear and stochastic systems and cannot be applied to a nonlinear dynamical system. Compared to these alternatives, EDM is better suited to find the causal relationships in a nonlinear dynamical system such as the brain. Tajima *et al.* [23] also applied embedding theorems in nonlinear state-space reconstruction to analyze a dynamic system. They also built on the causality inference method from Sugihara *et al.* [12] in their work.

EDM has been successfully applied to diverse research fields [2]. In neuroscience, CCM was applied to identify the effective connectivity between brain areas from magnetoencephalography (MEG) data [4]. In ecology, Grziwotz *et al.* found the causal relationships between the environment and mosquito abundance by using CCM [5]. Environmental factors, such as temperature, precipitation, dew point, air pressure, and mean tide level were identified to causally affect mosquito abundance. Ma *et al.* applied simplex projection to forecast wind generation [6]. In [7], an EDM algorithm called S-Map [24] was used to find the relationship between harvested and unharvested fish in terms of size, age, and others. Luo *et al.* applied CCM to estimate the causal relationships of user behavior in an online social network [25]. These use cases demonstrate the wide applicability of EDM to analyze nonlinear dynamical systems.

2.2.3 cppEDM

cppEDM [26] is the latest implementation of the EDM framework. cppEDM is a general purpose C++ library used as a backend by rEDM [27] and pyEDM [28], which are EDM implementations for the R and Python language, respectively.

Two major issues have been identified in cppEDM that hinder large-scale analysis on HPC systems: redundant computation and lack of GPU support. Since cppEDM is a general purpose library, it provides a one-to-one cross mapping function to identify the causality between a selected combinations of time series variables. The all-to-all cross mapping function is implemented by reusing the one-to-one cross mapping function. This results in redundant computation. Additionally, cppEDM is a reference implementation of EDM; therefore, it is not optimized for a specific hardware architecture such as GPUs. Furthermore, cppEDM suffers from significant load imbalance among workers because it performs static decomposition of the problem. In fact, a performance evaluation in a previous work showed that the runtime of workers varied greatly from 5 hours to 8.5 hours [13].

2.3 mpEDM

In this section, I first outline the original causal inference algorithm in cppEDM. Then, I describe the algorithmic improvement and the design of the inter-node and intra-node parallelization in mpEDM.

2.3.1 Original Algorithm

Algorithm 1 outlines the causal inference algorithm in cppEDM. The input to the algorithm is an $L \times N$ array ts, where L is the number of time steps within a time series and N is the number of time series. In addition to the input dataset, maximum embedding dimension E_{max} and time lag τ need to be supplied. The output is an $N \times N$ casual map ρ. The algorithm consists of two phases: (1) simplex projection and (2) CCM. Simplex projection finds the optimal embedding dimension for each time series. CCM estimates the causal relationship between two time series using the optimal embedded dimension obtained in the first phase. Note that in the original definition of CCM, predictions are made multiple times using randomly subsampled library sets of different sizes and it is tested whether increasing the library set size improves the prediction accuracy. This research excluded this step since the convergence test passes in most cases if the prediction using the full library set achieves high accuracy.

In the first phase, simplex projection (line 1–11) takes a time series in the dataset and splits into $library$, the first half, and $target$, the second half (line 3–4). Next, both library and target are embedded into E-dimensional space using time delayed embeddings. A k-nearest neighbors (kNN) search is performed in the state space to find the $E + 1$ nearest target points from each library point (line 5). The search results are stored in two lookup tables $indices$ and $distances$, both of which are two-dimensional arrays of shape $L \times (E + 1)$. Element (i, j) in the indices array is the index of the j-th nearest target point from library point i, whereas element (i, j) in the $distances$ array is the Euclidean distance between the library point i and its j-th nearest target point. The $distances$ array is then

Algorithm 1: Causal Inference in cppEDM

```
Input: Dataset $ts$ ($N$ time series of length $L$), maximum embedding
       dimension $E_{max}$
Output: $N \times N$ causal map $\rho$
// Phase 1: Simplex projection
1  for $i \leftarrow 1$ to $N$ do
2      for $E \leftarrow 1$ to $E_{max}$ do
3          $library \leftarrow$ First half of $ts[i]$
4          $target \leftarrow$ Second half of $ts[i]$
5          $indices, distances \leftarrow$ kNN($library, target, E$)
6          $distances \leftarrow$normalize($distances$)
7          $prediction \leftarrow$ lookup($indices, distances, library, E$)
8          $\rho[E] \leftarrow$ corrcoef($target, prediction$)
9      end
10     $optE[i] \leftarrow \underset{E}{\text{argmax}}\ \rho[E]$
11 end
// Phase 2: CCM
12 for $i \leftarrow 1$ to $N$ do
13     for $j \leftarrow 1$ to $N$ do
14         $indices, distances \leftarrow$ kNN($ts[i], ts[i], optE[j]$)
15         $distances \leftarrow$normalize($distances$)
16         $prediction \leftarrow$ lookup($indices, distances, ts[j], optE[j]$)
17         $\rho[i, j] \leftarrow$ corrcoef($ts[j], prediction$)
18     end
19 end
```

converted to exponential scale and each row is normalized (line 6). A one step ahead prediction of a target point is made by (1) obtaining the indices of its $E+1$ library neighbors from $\boldsymbol{indices}$, (2) obtaining the one step ahead values of those library points from $\boldsymbol{library}$ and (3) computing a weighted average of the future library points using $\boldsymbol{distances}$ (line 7). Finally, Pearson's correlation coefficient is computed to evaluate the predictive skill of the simplex projection using the prediction results and real observed withheld values (line 8). This is repeated for every E ranging from 1 to E_{max} ($\leq$20 in practice). The E value that achieves the highest accuracy is determined to be the optimal embedding dimension for the time series and stored in $\boldsymbol{optE}$ (line 10).

In the second phase, CCM (line 12–19) works similar to simplex projection but predicts between two different time series. A given $\boldsymbol{library}$ time series is used to cross predict another $\boldsymbol{target}$ time series in the dataset to evaluate whether the latter is the cause of the former. It computes and normalizes the kNN tables from the library time series (line 14–15) and uses the tables to predict the target time series (line 16). Note that simplex projection predicts within the same time series while CCM predicts across two different time series. Therefore, the kNN tables computed in the simplex projection phase cannot be reused in the CCM phase. The correlation between the predicted values and the actual values represents strength of causality (line 17). In this manner, causal inference is performed for all combinations of time series in the dataset.

I have profiled cppEDM and found out that over 97% of the total runtime is spent in the kNN search. In addition, I have discovered that the time delayed embedding in cppEDM replicates the time series $E+1$ times and causes significant memory overhead.

2.3.2 Improved Algorithm

The key observation behind the algorithmic improvement is that the kNN lookup table for CCM is constructed from the library time series only, and the target time series is not used. This suggests that once the kNN lookup table is computed for a particular library time series, the precomputed table can be reused to make predictions for every target time series. This improvement is trivial if N is in the same order as E_{max}, which was the case in previous use cases of EDM. However, in this use case, N is equal to the number of active neurons in a zebrafish brain, which is roughly 10^5. Therefore, the potential speedup becomes significantly large.

Algorithm 2 shows the pseudocode of the improved causal inference algorithm in mpEDM. The simplex projection algorithm is unchanged from cppEDM but its kNN and lookup functions are parallelized and optimized. The CCM algorithm in mpEDM is improved in the following manner. For each library time series, I first compute the kNN lookup tables for every embedding dimension ranging from 1 to E_{max} (line 4–7). Then, I iterate through all $target$ time series and use the precomputed lookup table for the optimal embedding dimension of the $target$ time series to predict the $target$ time series (line 9–10). Finally, I compute the correlation between the prediction and the actual $target$ to estimate the causality (line 11).

Algorithms 3 outlines the kNN function for CPU. I first calculate the all-to-all distances between every library and target point in the state space. Note that I do not explicitly create the time series embeddings on memory but I compute them on-the-fly to reduce memory footprint and increase cache hit. In addition, both $indices$ and $distances$ are stored in row-major format to match the access pattern. Then, each row in the $distances$ and $indices$ arrays is partially sorted in descending order using the distances as sort keys. I use heap sort to implement partial sort. After the sorting, both arrays are trimmed from $L \times L$ to $L \times (E+1)$ and returned. Algorithm 4 shows the kNN function for GPU. In the GPU version, I create time series embeddings on the host and transfer them to the device. The kNN search is executed on the GPU and the resulting kNN tables are returned to the host.

Algorithm 5 outlines the lookup function. It uses the kNN lookup tables $indices$ and $distances$ of the $library$ time series. For each target point, the indices of its $E+1$ neighbors are retrieved from the $indices$ table. Then, those neighbors are accumulated using the weights stored in the $distances$ table. Finally, the function returns the predicted $target$ time series.

The average time complexity of each algorithm is analyzed as follows. The time complexity of the kNN function in Algorithm 3 and 4 is $O(L^2E)$ because the all-to-all distance calculation is $O(L^2E)$ and the sorting is approximately

Algorithm 2: Causal Inference in mpEDM

```
Input: Dataset $ts$ ($N$ time series of length $L$), maximum embedding
       dimension $E_{max}$
Output: $N \times N$ causal map $\rho$
// Phase 1: Simplex projection
1  for $i \leftarrow 1$ to $N$ do
       // Same as cppEDM (Algorithm 1)
2  end
// Phase 2: CCM
3  for $i \leftarrow 1$ to $N$ do
4      for $E \leftarrow 1$ to $E_{max}$ do
5          $indices[E], distances[E] \leftarrow$ kNN($ts[i], ts[i], E$)
6          $distances \leftarrow$normalize($distances$)
7      end
8      for $j \leftarrow 1$ to $N$ do
9          $E_j \leftarrow optE[j]$
10         $prediction \leftarrow$ lookup($indices[E_j], distances[E_j], ts[j], E_j$)
11         $\rho[i, j] \leftarrow$ corrcoef($ts[j], prediction$)
12     end
13 end
```

$O(L^2 \log E)$. The time complexity of the lookup function in Algorithm 5 is $O(LE)$. By combining these results, the time complexity of simplex projection in mpEDM is $O(NL^2E)$, which is the same as cppEDM. The time complexity of CCM in mpEDM, on the other hand, is $O(NL^2E^2 + N^2LE)$. In cppEDM, the time complexity of CCM is $O(N^2L^2E)$. As a result, the time complexity of the whole causal inference algorithm in mpEDM is $O(NL^2E^2 + N^2LE)$.

Algorithm 3: kNN for CPU

```
Input: library and target time series, embedding dimension $E$, time lag $\tau$
Output: Arrays distances and indices for lookup
// All-to-all distance calculation
1  for $i \leftarrow 1$ to $L$ do
2      for $k \leftarrow 1$ to $E$ do
3          $distances[i, :] \leftarrow 0$
4          for $j \leftarrow 1$ to $L$ do
5              $indices[i, j] \leftarrow j$
6              $distances[i, j] \leftarrow$
                 $distances[i, j] + (target[k\tau+i] - library[k\tau+j])^2$
7          end
8      end
9  end
// Sorting
10 $top\_k \leftarrow$ E+1
11 for $i \leftarrow 1$ to $L$ do
12     $indices[i, :] \leftarrow$ partialSort($indices, distances, top\_k$)
13 end
```

2.3.3 Inter-Node Parallelism

To distribute the work across multiple compute nodes, I naturally choose the loops with the highest granularity. That is, the two outermost loops that iterate over the time series (line 1–2 and 3–13 in Algorithm 2). I implement a

Algorithm 4: kNN for GPU

```
Input: library and target time series, embedding dimension $E$, time lag $\tau$
Output: Arrays distances and indices for lookup
// Embedding
1  for $i \leftarrow 1$ to $E$ do
2      for $j \leftarrow 1$ to $L$ do
3          $libraryBlock[i, j] \leftarrow library[i\tau+j]$
4          $targetBlock[i, j] \leftarrow target[i\tau+j]$
5      end
6  end
// All-to-all distance calculation and sorting
7  $top\_k \leftarrow$ E+1
8  Copy $libraryBlock$ and $targetBlock$ to device
9  $indices, distances \leftarrow$
     nearestNeighbour($libraryBlock, targetBlock, top\_k$)
10 Copy $indices$ and $distances$ to host
```

Algorithm 5: Lookup

```
Input: Array of indices and distances, target time series, embedding
       dimension $E$ of target
Output: Prediction of the time series prediction
1  for $i \leftarrow 1$ to $L$ do
2      $prediction[i] \leftarrow 0$
3      for $j \leftarrow 1$ to $E + 1$ do
4          $idx \leftarrow indices[i, j]$
5          $dist \leftarrow distances[i, j]$
6          $prediction[i] \leftarrow prediction[i] + target[idx] \cdot dist$
7      end
8  end
```

simple master-worker framework based on MPI to distribute these loops. To dynamically distribute work and mitigate load imbalance among workers, I adopt *self-scheduling* in a master-worker framework. In self-scheduling, the master accounts and dispatches tasks to workers. Each worker performs assigned tasks, and once it completes, the worker asks the master for a new task.

The high-level organization of the inter-node parallelism is as follows. First, the workers execute the simplex projection phase. The optimal embedding dimension for each time series is reported back to the master. Once the first phase is complete, the master broadcasts ***optE*** to all workers. Subsequently, the workers execute the all-to-all CCM phase. The final results are written to the file system by each worker to alleviate the load on the master.

Both the input dataset and the inferred causal map are stored as HDF5 [29] files for easy integration with the pre/post processing workflow. The workers read the input HDF5 file in parallel and keep the entire dataset on memory during the execution. Every time a worker completes a cross map, the worker writes an element of the causal map asynchronously to the output HDF5 file. This small random write pattern, however, is known to be slow on parallel file systems. In fact, I observed that write I/O becomes a significant bottleneck of the application on GPFS. I therefore take advantage of BeeOND (BeeGFS On Demand) [30], the burst buffer deployed on ABCI. BeeOND combines local SSDs installed on the compute nodes and provides an on-demand parallel file system to a job. The workers write the results to BeeOND to minimize I/O overhead.

2.3.4 Intra-Node Parallelism

I focus the efforts to parallelize and optimize the kNN kernel since it is the primary bottleneck in cppEDM as discussed in section 2.3.1. I design and implement kNN kernels for both CPU and GPU architecture to ensure that mpEDM can efficiently run on a wide variety of computing platforms. In the kNN kernel for CPU shown in Algorithm 3, the two loops that iterate over the time steps within a time series are parallelized using OpenMP (line 1–9 and 10–13 in Algorithm 3). I also utilize OpenMP 4.0 SIMD directives to vectorize the innermost loop explicitly. Note that the nested loops are ordered such that the memory accesses in the innermost loop are contiguous.

In the kNN kernel for GPU shown in Algorithm 4, I take advantage of ArrayFire [31], a highly optimized library for GPU-accelerated computing. ArrayFire provides backends for CUDA, OpenCL and CPU, but in this paper I only use the CUDA backend since ABCI is installed with Tesla V100 GPUs. The kNN algorithm implemented in ArrayFire is essentially the same as the CPU implementation. ArrayFire uses a block-wide parallel radix sort implementation in the CUDA UnBound (CUB) template library. Since each ABCI compute node is equipped with four GPUs, I also distribute the work across multiple GPUs. To achieve this, the loop that iterates over $\boldsymbol{E}$ (line 4–7 in Algorithm 2) is parallelized such that each GPU computes lookup tables for one or more $\boldsymbol{E}$. I dynamically schedule this loop to ensure load balancing across GPUs because the runtime of the kNN kernel depends on $\boldsymbol{E}$ as discussed in section 2.3.2.

For the lookup kernel shown in Algorithm 5, I currently only have a CPU version of this kernel. The time step loop is parallelized using OpenMP (line 1–8 in Algorithm 5). This kernel is heavily memory bandwidth bound since it requires random memory access.

2.4 Evaluation

The computational performance of mpEDM was evaluated on ABCI. Furthermore, I present the scientific outcomes obtained using mpEDM.

2.4.1 Evaluation Environment

ABCI [32] is the world's first large-scale Open AI Computing Infrastructure. It is constructed and operated by the National Institute of Advanced Industrial Science and Technology (AIST). According to the latest TOP500 list published in November 2019 [33], ABCI is the most powerful supercomputer in Japan and the 8th in the world. ABCI has 1,088 compute nodes, each equipped with two 20-core Intel Xeon Gold 6148 CPUs, four NVIDIA Tesla V100 SXM2 (16GB) GPUs, 384GB of RAM and 1.6TB of local NVMe SSD. The parallel file system is based on GPFS with a total capacity of 22PB.

2.4.2 Performance Evaluation

I compared mpEDM with cppEDM from the following three aspects: total runtime, parallel scalability and impact of dataset size on the runtime. I used three real-world datasets recorded from larval zebrafish under different conditions. Table 1 shows the list of datasets used in the evaluation.

Table 1: Datasets used in the evaluation

Dataset	# of Time Steps	# of Time Series	Size
Fish1_Normo	1,450	53,053	0.7 GB
Subject6	3,780	92,538	3.0 GB
Subject11	8,528	101,729	9.5 GB

Total Runtime mpEDM shows significantly higher performance compared to cppEDM. Table 2 shows the performance comparison between cppEDM and mpEDM. cppEDM took 8.5 hours to analyze the Fish1_Normo dataset using 512 ABCI nodes [13], whereas mpEDM took only 20 seconds to analyze the same dataset using 512 ABCI nodes with GPU architecture. The result shows that mpEDM is 1,530× faster than cppEDM. Moreover, mpEDM finished the causal inference of two larger datasets: Subject6 in 101 seconds and Subject11 [15] in 199 seconds. The reason for the missing cppEDM runtimes for the Subject 6 and Subject 11 datasets is that those datasets are too large for cppEDM to complete the calculations in a realistic time.

Table 2: Performance comparison between cppEDM and mpEDM

	cppEDM	mpEDM	
Dataset	512 Nodes	1 Node	512 Nodes
Fish1_Normo	8.5h	1,973s	20s
Subject6	N/A	13,953s	101s
Subject11	N/A	39,572s	199s

Parallel Scalability I measured the parallel scalability of mpEDM by varying the number of workers and measuring the runtime of mpEDM with and without GPU. I used the largest Subject11 dataset in this evaluation.

Figure 5 shows the strong scaling performance of mpEDM. In the *Single Node* setup, mpEDM is executed on a single node without MPI. In the *X Workers* setup, mpEDM is executed with MPI using the specified number of workers. I measured up to 511 workers since ABCI allows a maximum of 512 nodes per job (except for jobs running under the ABCI grand challenge program, which can use the full 1,088 nodes). The result shows that the GPU version runs as twice as fast as the CPU version in every case. I noticed that the CPU version ran in the single worker setup 10% slower than the single node setup. I believe this slowdown is caused from the interference between the background tasks performed by the BeeOND daemon and the computation in mpEDM. This does not happen with the GPU version because the average CPU utilization is lower than the CPU version.

Figure 6 shows the relative speedup of the multi-node setup in relation to the single node setup. It reveals that the speedup is nearly linear with both GPU and CPU. However, the speedup of the GPU version drops when the number of nodes is 64 or more.

I measured the breakdown of each phase to investigate the cause behind the scalability decline. I compared 32 workers and 128 workers since the GPU version declines beyond 64 nodes. Figures 7 and 8 show the breakdown of average runtime for processing a single time series in simplex projection and CCM. The two figures clearly indicate that memory copy, MPI communication and I/O are not bottlenecks and do not significantly increase with the number of workers. However, the kNN function becomes slower when the number of workers increases. I found out that the kNN search for the first time series processed on a worker is significantly slower (ranging from 3.3 seconds to 16.4 seconds) than the subsequent ones. I believe this is caused by the initialization process of the GPUs. This issue could come from job distribution and initialization on each worker node in the supercomputer. Some nodes might need an extra time to initialize, such as mapping a unified memory of CUDA, clearing memory from previous job, etc. which needs some extra initialization time to make the GPU be ready to run the

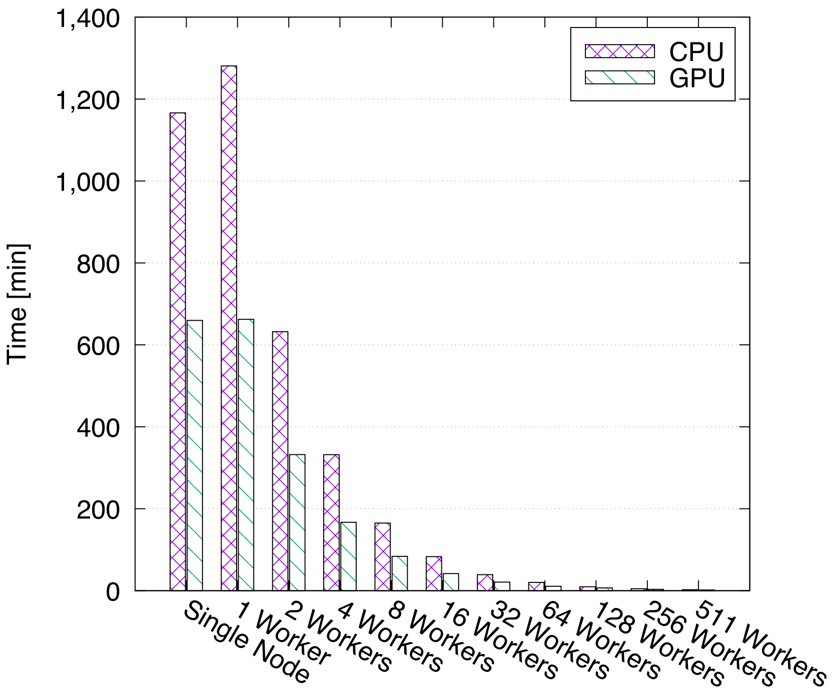


Figure 5: Strong scaling performance (absolute runtime)

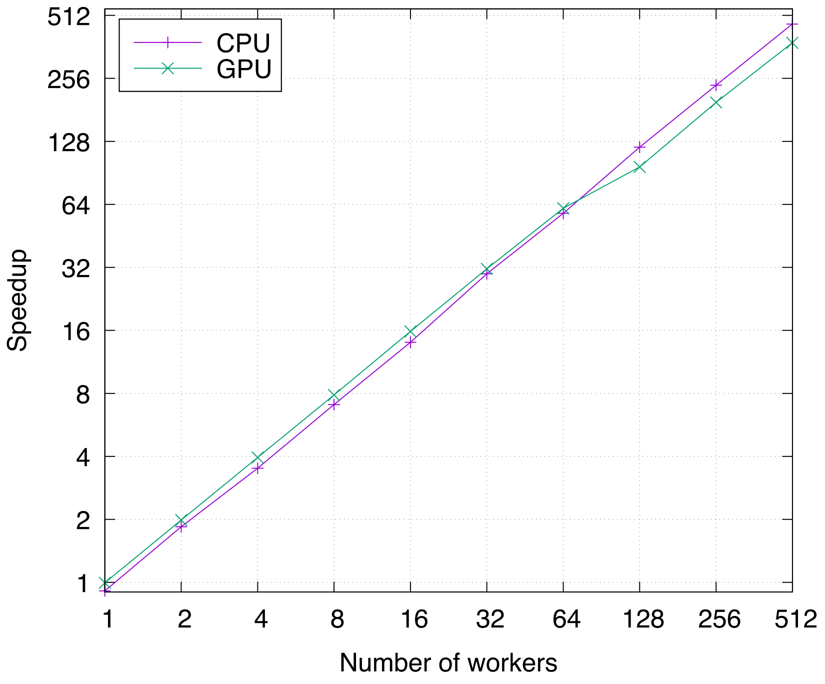


Figure 6: Strong scaling performance (relative speedup)

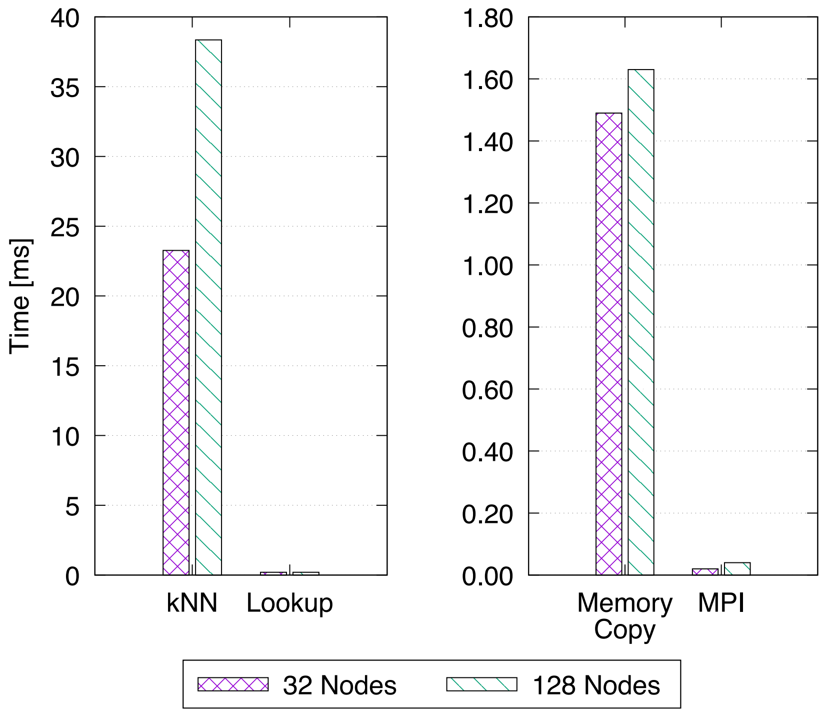


Figure 7: Breakdown of simplex projection (average runtime per time series)

job.

To verify this, I created a simple program that initializes the GPUs and allocates some GPU memory on a single node. I submitted a job that run this program 100 times and measured the initialization time. The result revealed that the initialization time follows a long-tailed distribution: the median was 4.6 seconds while the maximum was 22.9 seconds. This suggests that a few stragglers impact the total runtime and degrade the scalability as the number of workers increases.

Impact of Dataset Size I evaluated how the size of the dataset impacts the runtime of mpEDM using dummy datasets with different sizes. Furthermore, I measured the time spent in each function. I also measured the speedup of the GPU version over the CPU version with varying number of time steps.

Figures 9 and 10 show the runtime of mpEDM when increasing the number of time series and time steps, respectively. I confirmed that the increase of runtime is not bigger than the increase predicted from the time complexity. I also confirmed

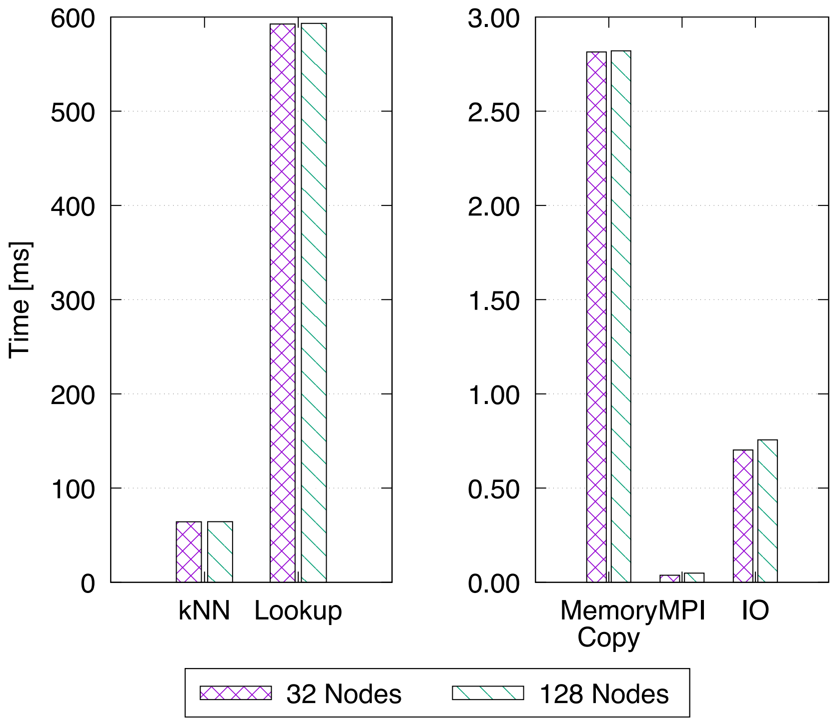


Figure 8: Breakdown of cross mapping (average runtime per time series)

that CCM consumed the majority of the total runtime and other tasks including I/O and MPI communication are ignorable.

Figures 11 and 12 show the runtime breakdown of each function in CCM when increasing the number of time series and time steps, respectively. Figure 11 shows that the runtime of the lookup function becomes dominant when increasing the number of time series. On the other hand, Fig. 12 shows that the runtime of the kNN function becomes dominant when increasing the number of time steps. These trends can be explained from the time complexity analysis of each algorithm described in section 2.3.2.

Figure 13 shows the speedup of the GPU version over the CPU version when varying number of time steps. I compared the performance between a single CPU socket and one or more GPUs to evaluate the GPU speedup. Evidently, the GPU speedup increases with the number of time steps. Single GPU is slower than the CPU if the number of time steps is 2,000 or less. This is because of the overhead inherent to offloading computation to the GPU. However, single GPU

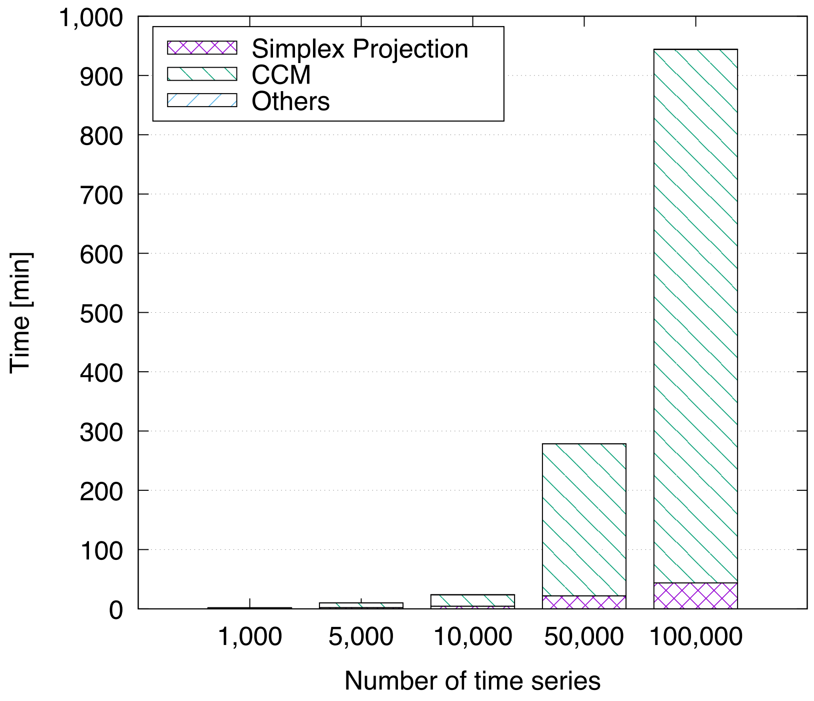


Figure 9: Runtime with varying number of time series (10,000 time steps)

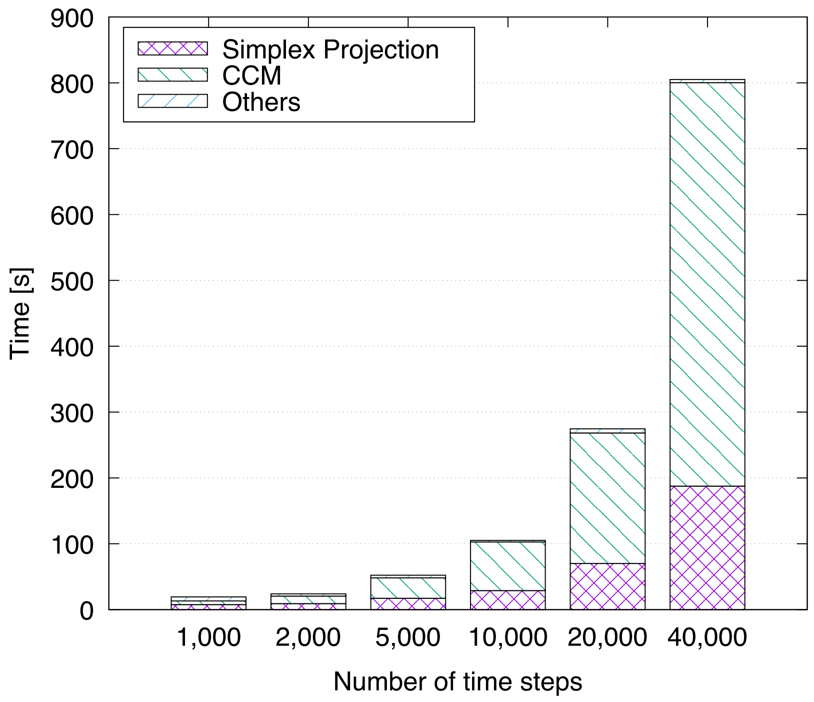


Figure 10: Runtime with varying number of time steps (1,000 time series)

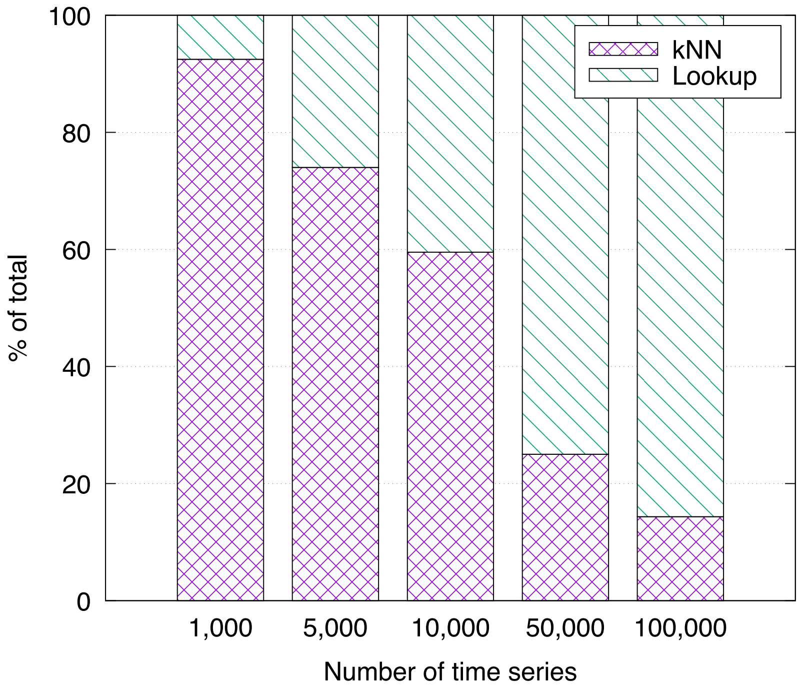


Figure 11: Breakdown of CCM - varying number of time series (10,000 time steps)

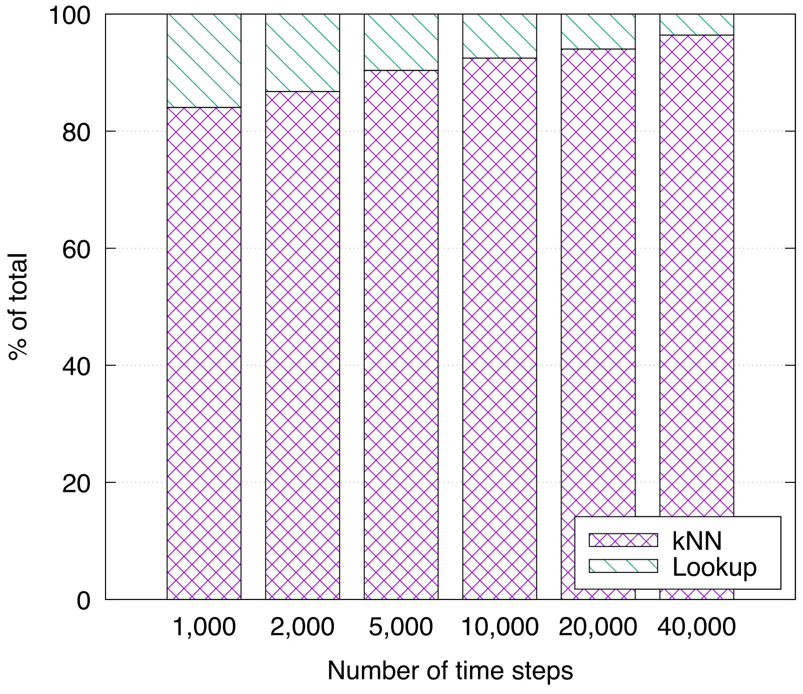


Figure 12: Breakdown of CCM - varying number of time steps (1,000 time series)

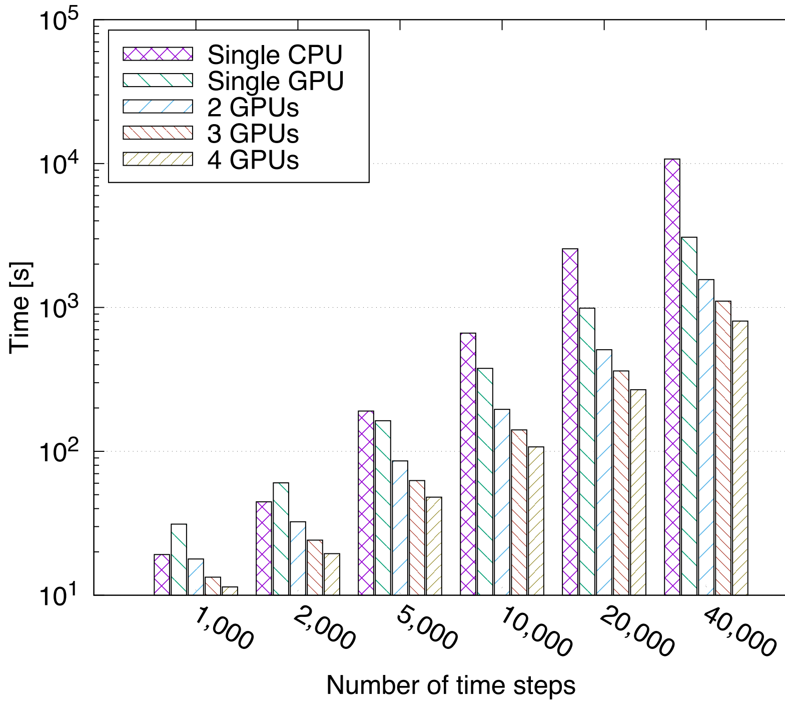


Figure 13: GPU speedup with varying number of time steps (1,000 time series)

consistently surpasses the CPU if the number of time steps if 5,000 or more. If the number of time steps is 40,000, the speedup of a single GPU is 3.5 times compared to CPU. When four GPUs are used, the speed up is 13.4 times.

2.4.3 Scientific Outcomes

Figure 14 shows the scientific outcomes obtained using mpEDM. The results showed that it could determine the causal connectivity across the entire brain across two behaviors. This shows that depending on task, the network of relationships between individual neurons change and become more connected, homogeneous and simplified with a goal directed task. In the resulting network connectivity increased and became simpler. Furthermore, we were able identify individual neurons that integrate signals from multiple other neurons that contain decision making information. These neurons allow the prediction of fish turn behaviors while swimming and generate low dimensional manifold models based on data geometry that are able to predict the fish ’s behavior at least 0.5 seconds

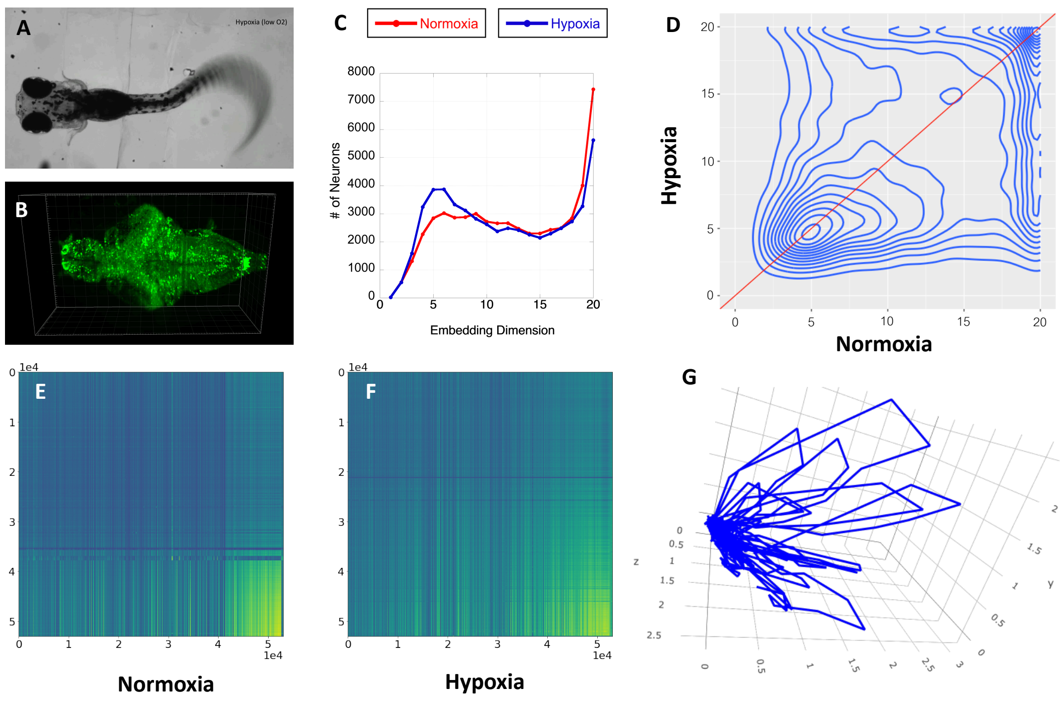


Figure 14: Scientific results

(A) Zebrafish larvae were imaged to study their response to low oxygen.

(B) The day larvae were imaged using a SPIM lightsheet microscope and whole brain calcium activity was recorded at single cell resolution.

(C) The calculation of the dimensionality of the neuronal populations show a decrease under low oxygen (hypoxia) as seen in the distribution.

(D) Measured transitions between normal oxygen concentrations (normoxia) to hypoxia show a bias below to the right of the diagonal line showing that dimensionality decreases as oxygen decreases.

(E, F) Whole brain CCM all vs all causal inference matrix of an all vs all neurons. Results show a more homogeneous map in hypoxia (F) than normoxia (E) indicating a simplification of behavior consistent with the above dimensionality drop.

(G) An identified signal integration manifold capable of predicting turns of the fish at least 0.5 seconds (a single time step) ahead of time. Whenever the neural activity trajectory enters one of the loops of the manifold, the fish will turn.

(a single time step) ahead. A three dimensional projection of one of these manifolds is shown in Fig. 14 (G), where entering the loop predicts turn behavior. Based on the combined activity of two neurons and information on prior states we are able to predict when the fish will turn. Beyond this, this is the first map of causal connectivity of any vertebrate animal at single neuron resolution.

2.5 Conclusion & Future Work

EDM is a nonlinear time series analysis framework proven its applicability in various fields. However, EDM has only been applied to small datasets due to its computational cost. In this paper, I designed and implemented mpEDM, a parallel distributed implementation of EDM optimized for execution on modern GPU-centric supercomputers. mpEDM improves the EDM algorithm to reduce redundant computation and optimizes the implementation to fully utilize hardware resources such as GPUs and SIMD units. mpEDM took only 20 seconds to finish the causal inference of a dataset containing the activity of 53,053 zebrafish neurons on 512 ABCI nodes. This is 1,530× faster than cppEDM, the current standard implementation of EDM. Moreover, mpEDM could analyze a 13× larger dataset in 199 seconds. This is the largest EDM causal inference achieved to date.

I will continue to optimize the performance of mpEDM. As discussed in section 2.4.2, I need to improve the performance of the lookup as it becomes the primary bottleneck when scaling up the number of time series further. NEC vector engine processor is examined for seeking the hardware acceleration of the lookup function. I will also explore other efficient implementations of nearest neighbor search on GPUs. Currently, mpEDM uses the exact kNN search implementation provided by ArrayFire. There exist many studies on efficient Approximate Nearest Neighbor (ANN) search [34, 35]. However, it is unclear how ANN affects the accuracy of EDM predictions. Another well-known approach is to use spatial indices such as KD-trees and Ball-trees to accelerate kNN search [36, 37].

Additionally, EDM algorithms other than simplex projection and CCM will be implemented in mpEDM to expand mpEDM to a standard implementation of EDM on HPC systems. I will make this EDM library widely available to the community with a hope to assist scientists in need to analyze large-scale time series datasets of nonlinear dynamical systems.

3. A Transparent, Low-overhead Monitoring System for OpenFlow Networks

3.1 Introduction

In the current networking architecture, network devices in a network are individually and manually configured by the administrator. This design makes it challenging to manage large and complex networks. Software-Defined Networking (SDN) [38] is an alternative networking architecture that centralizes the control of network devices to a centralized software controller and introduces programmability to the network infrastructure. In current networks, the packet forwarding function (*data plane*) and the routing decision function (*control plane*) are inseparably implemented in the same network device. In SDN, these two are disaggregated. The packet forwarding is handled by SDN switches, whereas the routing decision is handled by a centralized software controller. Each SDN switch maintains a flow table, which is a collection of flow entries. A flow entry contains a set of (1) matching conditions, which specify the packets that the flow matches, and (2) actions, which specify how matched packets are processed. Flow entries are generated by the controller and installed to switches.

The OpenFlow protocol [39] is widely used to communicate between SDN switches and the centralized SDN controller. OpenFlow defines multiple message types for different purposes, such as installing flow entries and collecting switch information and statistics, for example. Some examples of OpenFlow messages are shown in Table 3. Hardware vendors such as Mellanox, Pica8 and NoviFlow produce hardware OpenFlow switches. There are also several software OpenFlow switches including Open vSwitch [40] and Lagopus [41]. Furthermore, software frameworks that facilitate the development of OpenFlow controllers, such as Ryu [42], Faucet [43], Open Network Operating System (ONOS) [44, 45], and OpenDaylight [46], are available.

Figure 15 illustrates how an OpenFlow network delivers a packet. Every time a switch receives a packet from a host (step ① in Fig. 15), the switch searches its flow table for a flow entry that matches the incoming packet (step ②). If a matching flow entry is found, the switch performs the action indicated in the flow

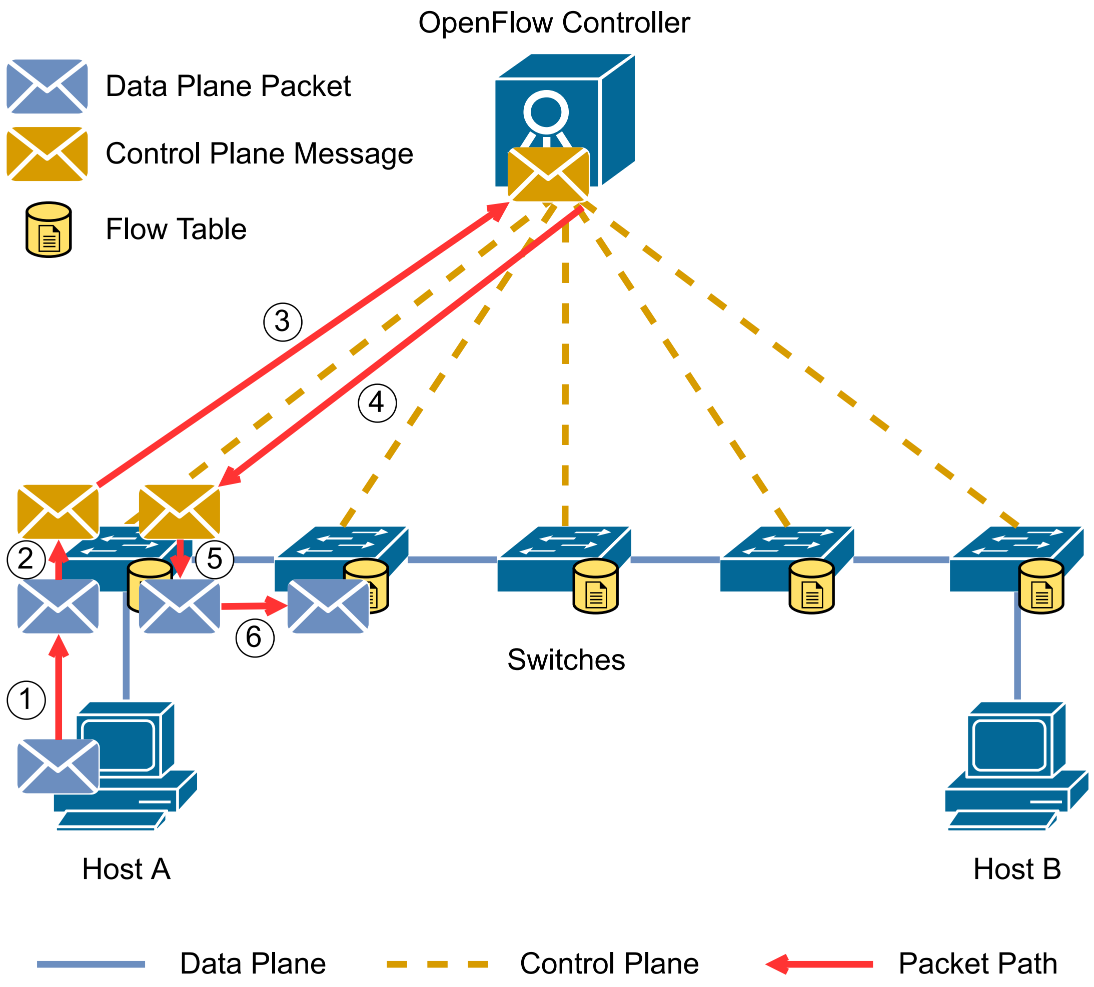


Figure 15: An OpenFlow network

Table 3: Example of OpenFlow Messages Types

Message Type	Direction	Purpose
FlowMod	Controller→Switch	Modifies the flow table of a switch.
FeaturesRequest	Controller→Switch	Requests the supported features of a switch.
FeaturesReply	Switch→Controller	Responds to a controller's Features-Request message.
FlowStatsRequest	Controller→Switch	Requests statistics about individual flows on a switch.
FlowStatsReply	Switch→Controller	Responds to a controller's FlowStat-sReply message.
PortStatsRequest	Controller→Switch	Requests statistics about individual ports on a switch.
PortStatsReply	Switch→Controller	Responds to a controller's PortStat-sReply message.
PacketIn	Switch→Controller	Sends an unmatched packet to the controller.
PacketOut	Controller→Switch	Injects a packet to the data plane of a switch.

entry (step ⑥). If no matching flow entry is found, the switch sends a PacketIn message to the controller (step ③). The controller then examines the PacketIn message and determines where the packet that generated the PacketIn message should be forwarded next. Based on this decision, the controller installs a new flow entry to the switch by sending a FlowMod message (step ④). This procedure is repeated until the packet reaches its destination.

Investigating and understanding the behavior of an OpenFlow network is challenging [47,48]. This is because, although the control logic is logically centralized in the OpenFlow controller, the state of the network (*e.g.,* flow tables) is distributed across the network. Since conventional network monitoring systems are not designed to cope with OpenFlow networks, researchers have developed various monitoring systems tailored for OpenFlow networks [49–51]. However, existing

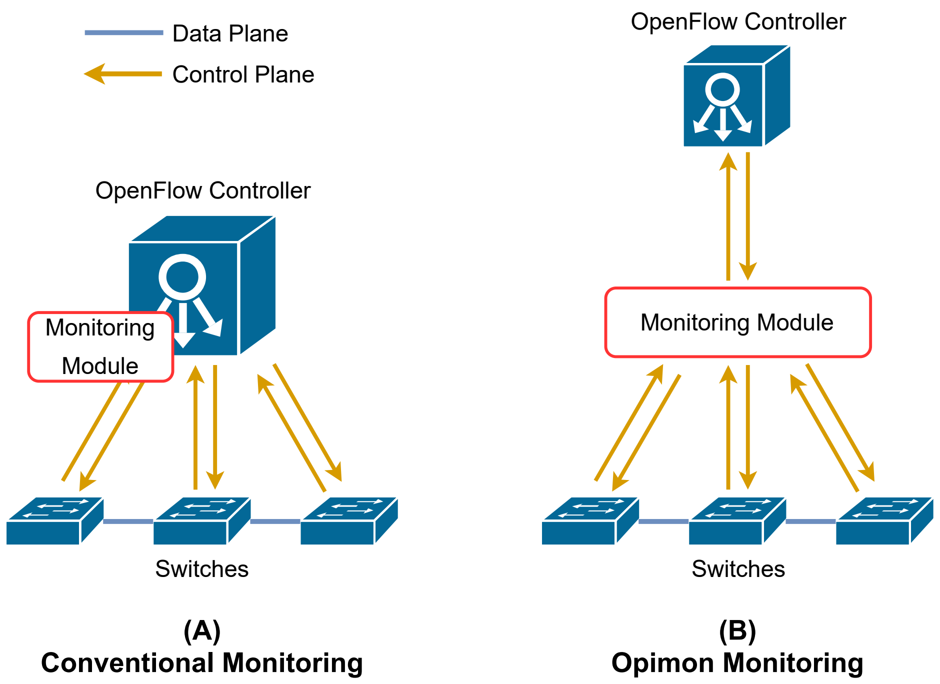


Figure 16: Comparison between conventional and proposed monitoring system

systems either rely on a specific controller framework or require modifications to the controller. This is often unacceptable when monitoring production networks.

Additionally, Security is one of the major issues that the network administrator needs to concern. *Denial of Service (DoS)* is one of the common cyber-attacks, which aims to make the resource or service unavailable for legitimate users. If the attack traffic comes from multiple sources, the attack is called *Distributed Denial of Service (DDoS)*. In DDoS attack, an attacker sends a huge number of network packets from many sources to a victim until the service or resource becomes unavailable. There are many types of packets that can be used in a DDoS attack. Some types of DDoS attacks (*e.g.* HTTP-based DDoS attacks [52]) are difficult to detect or separate from the heavily-loaded production traffic. DDoS attack also causes many problems in an OpenFlow network in terms of performance and reliability.

This section proposes a monitoring system for OpenFlow networks, which I refer to as *Opimon* (OpenFlow Interactive Monitoring)[2]. Opimon is completely

[2]`https://github.com/wassapon-w/opimon`

transparent to the network and works with any OpenFlow switch or controller without requiring any modification. Furthermore, Opimon imposes little overhead to the network performance and can be used in production networks. Opimon collects the topology, flow tables, and switch statistics from the target network, and interactively visualizes the state of the network through a web interface in real-time. Additionally, Opimon has a security analysis module to detect DDoS attack with machine learning. Opimon is based upon previous work [53], but its monitoring module is redesigned to minimize the incurred overhead.

Figure 16 compares the design of a conventional OpenFlow monitoring system and Opimon. In a conventional design, the monitoring system was integrated into the OpenFlow controller as a sub component. Thus, the monitoring system was dependent on the OpenFlow controller or the framework it uses. Opimon, on the other hand, acts as a transparent proxy between the controller and switches, and works with any controller. However, this design causes an unavoidable overhead when forwarding and collecting OpenFlow messages. I minimize the overhead by employing a multi-process architecture that scales with the number of switches. Furthermore, I decouple the message forwarding and collection into different processes so that messages are forwarded with minimum delay.

3.2 Related Work

3.2.1 Network Monitoring

Various monitoring protocols and tools are available in traditional network architectures. Simple Network Management Protocol (SNMP) is one of the most widely used protocols for monitoring networks [54]. SNMP is used to collect information from network devices as well as to configure network devices. sFlow [55] is another popular technology for monitoring the traffic flows in a network. sFlow agents reside on network devices and sample traffic flows from the network, and the sampled traffic is aggregated and analyzed by a sFlow collector. Both SNMP and sFlow are, however, not designed for OpenFlow networks and are unable to obtain OpenFlow-specific information such as the content of flow tables.

Therefore, researchers have designed and implemented monitoring systems tailored for OpenFlow networks. OpenNetMon is a extension module for the

POX [56] OpenFlow controller that provides monitoring capabilities [49]. OpenNetMon polls statistics from switches and calculates the throughput and packet loss of each flow. The polling interval is adaptively controlled to reduce the switch CPU load while ensuring measurement accuracy.

OOFMonitor is a monitoring system for OpenFlow networks that collects the delay, jitter, packet loss rate, and link utilization [50]. Since OOFMonitor relies on the API exposed by the Ryu OpenFlow controller, it is incompatible with other controllers. In addition, OOFMonitor does not provide any feature to visualize the collected network information.

Isolani et al. proposed a modular system for interactive monitoring, visualization and configuration of OpenFlow networks [51]. Their system uses the RESTful API provided by the Floodlight OpenFlow controller to collect the topology of the network and the traffic counter of every flow entry present on switches.

Warraich et al. developed a system to monitor the traffic statistics at Internet eXchange Points (IXPs), called SDX-Manager [57]. It integrates a traditional IXP-Manager with an SDN controller. Grafana is used to visualize the traffic statistics. However, SDX-Manager is build on top of the Faucet OpenFlow controller framework and lacks support for other controllers.

These existing monitoring systems share a common limitation: they depend on a specific controller or API, which are not part of the OpenFlow specification and not standardized. This limitation clearly hinders practicality because network designers or administrators are forced to choose a specific OpenFlow controller that is compatible with the monitoring system. In contrast, Opimon does not rely on a specific controller or API and can be integrated in any OpenFlow networks.

Network hypervisors such as FlowVisor [58] and AutoVFlow [59] enable virtualization of OpenFlow networks by slicing a physical network into multiple isolated virtual networks. Both of them employ a proxy-based design, where a transparent proxy is placed between the OpenFlow controllers and switches. The proxy examines and modifies the exchanged OpenFlow messages to isolate the network slices with one another. This design allows the hypervisors to be compatible with any OpenFlow controllers or switches. However, monitoring capabilities are not provided.

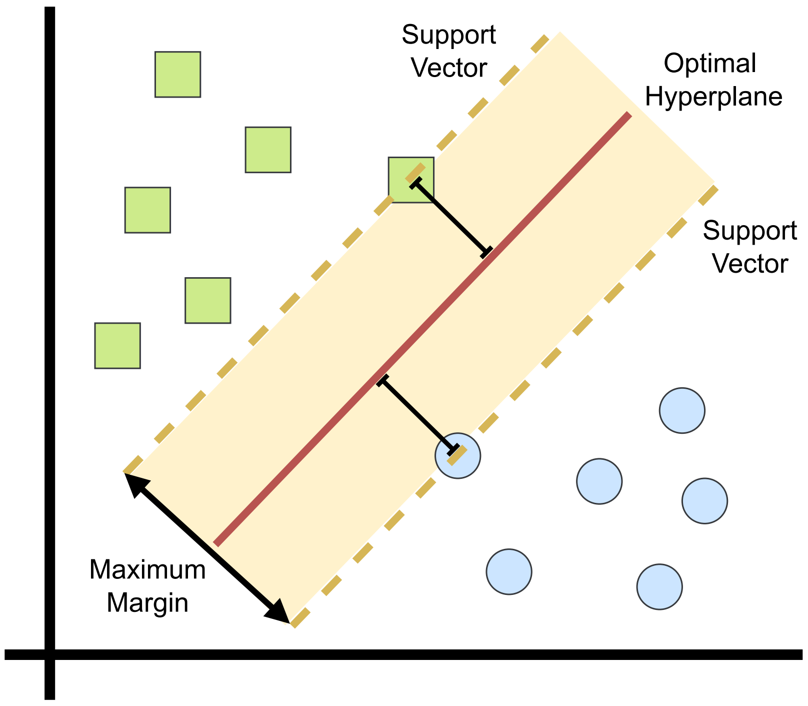


Figure 17: Support Vector Machine (SVM)

3.2.2 Support Vector Machine and Deep Learning

Support Vector Machine (SVM) is one of the most widely used machine learning algorithms for classification problems. Figure 17 illustrate components of SVM. SVM finds a separating line, or called hyperplane, that classifies the data into categories. Support vectors are a data point that nearest to the hyperplane. SVM calculates a margins, or a distance between the hyperplane and support vectors. The goal of SVM is to find a maximum margins for optimal hyperplane. SVM is applied to a wide spectrum of tasks [60].

Meanwhile, deep learning, a machine learning algorithm that utilize deep hierarchical layers of neural networks are gaining much attention from researchers. Much research has been conducted to compare the performance and accuracy of these two algorithms for various applications. For instance, the development of Myocardial Infarction detection also compares the performance of SVM against Artificial Neural Networks (ANN) algorithm [61], a fundamental structure of deep learning. In their study, LIBSVM is used for the SVM classification [62]. The works on a development of a detection tool for SDN use some features from the NSL-KDD dataset in their DDoS attack detection tool [63, 64]. Additionally, the performance of the machine learning library affects the performance of the classification. However, some studies on SVM compared the performance between LIBSVM and ThunderSVM [65, 66], and concluded that ThunderSVM is faster on both CPU and GPU.

In a review of deep learning frameworks [67], many deep learning frameworks, including Theano (with Keras), Torch, Caffe, Tensorflow, and Deeplearning4J, were compared in terms of speed and accuracy of classification. The results showed that Theano (with Keras) used less time for training models when tuning with large epochs and achieved higher accuracy. However, Theano stopped its development since 2017 [68]. Hence, this experiment employs Keras with TensorFlow as the backend engine for handling low-level operations such as tensor products, convolutions, and others. For the model evaluation, precision, recall, accuracy, and F1 score are selected as the metrics based on the studies done in [60, 61]. Some features were determined based on the previous research about the development of DDoS attack detection tools using information from packets [69].

3.2.3 DDoS Attack Detection

There has been much research on SDN and OpenFlow networks. However, very few studies have addressed the security issues in OpenFlow networks. A literature proposed a method to detect DDoS attacks in OpenFlow networks [70]. There is a comprehensive survey about security measures against DDoS attacks in SDN or OpenFlow networks. In this survey, the authors reviewed literatures on DDoS detection and mitigation in SDN and OpenFlow networks. One of the interesting techniques in the survey uses machine learning to detect DDoS attacks in SDN. In this survey, it suggests a research that they use a SVM classifier to detect DDoS attack in SDN [71]. In addition, they compared SVM with other machine learning techniques and they concluded that SVM achieved the best performance. DARPA dataset [72], which contains DDoS traffic in a traditional network, was used in this research.

Deep learning is frequently used to detect DDoS attack in SDN. A review [73] analyzed machine learning techniques for handling the issues of intrusion and DDoS attacks in SDN. This research compared five machine learning techniques, which include Neural Networks, Bayesian Network, Support Vector Machine, Genetic Algorithm, and Fuzzy Logic. They showed the pros and cons of these five machine learning techniques when applied to DDoS detection. They concluded that each machine learning technique has a unique characteristic and provides different results in terms of training time and accuracy. They concluded that neural network is capable to generalize from limited, noisy, and incomplete data. In addition, neural network does not require expert knowledge for classification. However, they mentioned that neural network trains slower than the other machine learning techniques and may not be suitable for real-time detection. They concluded that SVM is better in handling small dataset and provides high decision rate and training rate, insensitiveness to dimension of input data. Based on this review, I compare the performance between neural network and SVM. However, SVM requires long training time and can only be used for binary classification.

A research used deep learning to detect DDoS traffic in SDN [64]. The authors implemented DDoS detection on top of the SDN controller. They showed that their tool was able to classify normal and attack traffic with an accuracy of 99.82% with a very low false-positive rate. However, their DDoS detection system was implemented as a network application uses a Northbound API which may not be compatible with every controller because Northbound API is not standardized. Through the literature of DDoS detection, I aim to provide appropriate information to choose an optimal machine learning technique for DDoS detection algorithms in terms of accuracy and performance. In a preliminary experiment, SVM and deep learning is chosen to compare the effectiveness on DDoS detection in terms of classification accuracy and speed of classification.

3.3 Opimon

This section describes the design and implementation of Opimon. I first describe the high-level architecture of Opimon and then elaborate on each component.

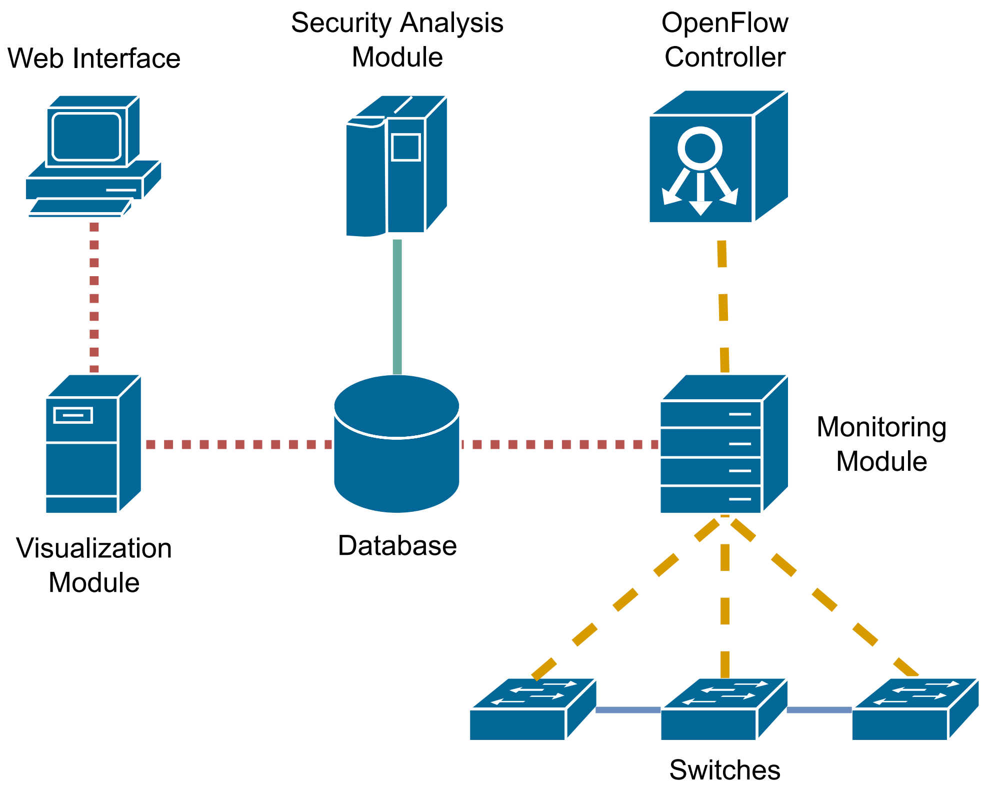


Figure 18: High-level Architecture of Opimon

3.3.1 High-level Architecture

Figure 18 illustrates the high-level architecture of Opimon. Opimon is mainly composed of three modules: (1) the *monitoring module*, (2) the *visualization module* and (3) the *security analysis module*. The monitoring module behaves as a transparent proxy and intercepts every OpenFlow message exchanged between the controller and the switches. The intercepted messages are stored into a database. The current implementation uses MongoDB as a database. The visualization module queries the collected messages from the database and shows various network information via a web interface in real-time.

3.3.2 Monitoring Module

Overall Design The monitoring module is responsible for collecting the OpenFlow messages exchanged in the control plane of an OpenFlow network. Since I

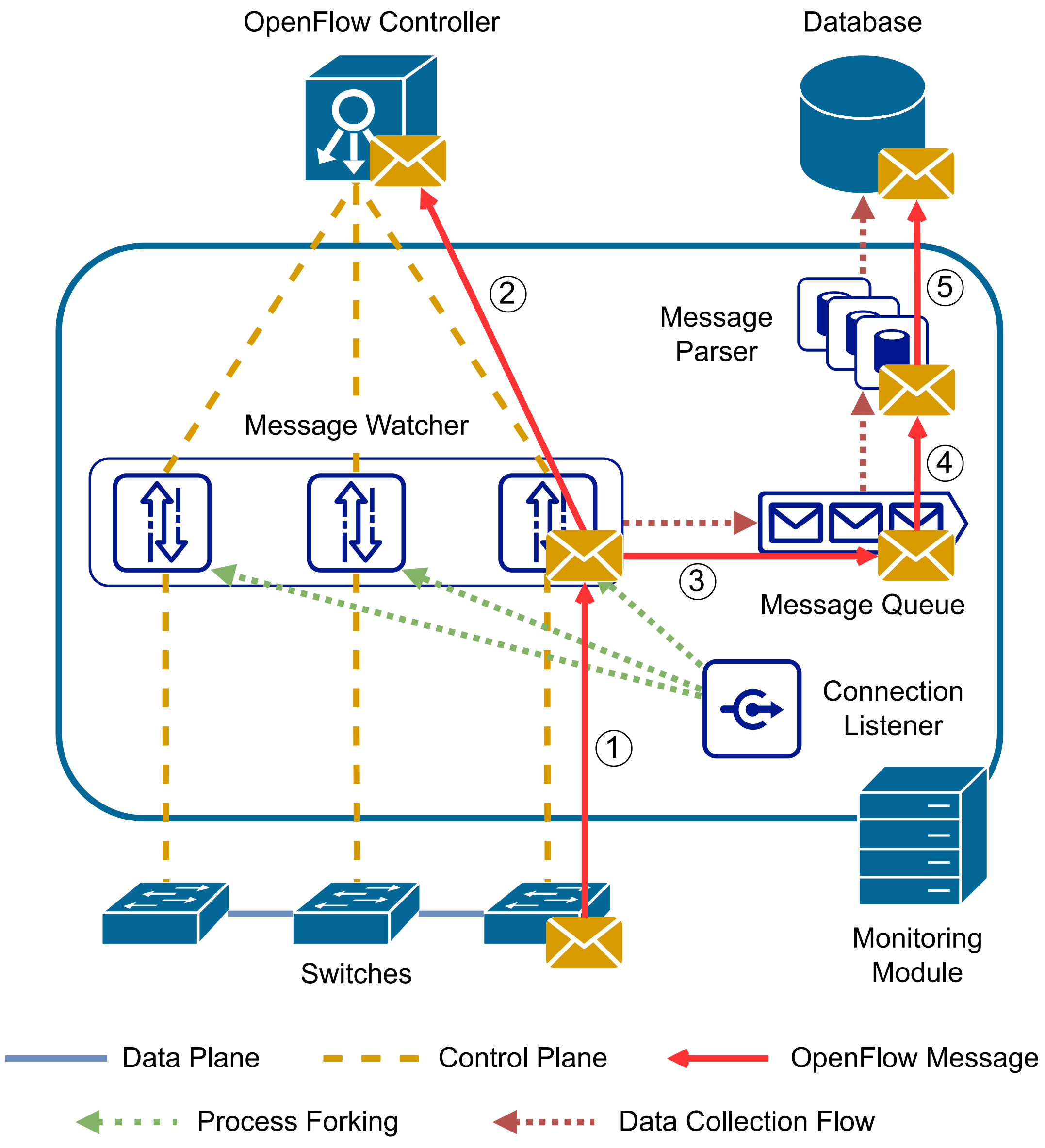


Figure 19: Monitoring Module

found out that message parsing is the primary bottleneck in collecting OpenFlow messages, I decouple message forwarding and parsing into different processes so that OpenFlow messages can be forwarded with minimal delay. The monitoring module is implemented in Python. This module runs the following three types of processes: connection listener process, message watcher process, and message parser process.

- *Connection listener process*: This process is responsible for handling new connections from switches and coordinating other processes. The connection listener waits for incoming connections from switches and forks a new message watcher process every time a switch is connected. The connection listener also creates a set of message parser processes.
- *Message watcher process*: This process is responsible for forwarding and collecting messages exchanged between the switches and the controller. A message watcher process is created for each switch. Every time a message watcher receives a new message from a switch or a controller, it pushes a copy of the raw message into the message queue and then forwards the message to the other side.
- *Message parser process*: This process is responsible for parsing each message in the message queue and storing the parsed message into MongoDB. This process uses the Ryu OpenFlow framework to parse the raw message. An example of a FlowMod message stored in MongoDB is shown in Listing 1.

Interaction of Processes Figure 19 shows the interaction of processes inside the monitoring module. When a switch connects to the monitoring module, the connection listener process forks a new message watcher process. The newly forked message watcher process accepts the connection from the switch and opens another connection to the OpenFlow controller. Every time a message watcher receives a message from a switch (step ① in Fig. 19), the message watcher clones the received message and forwards a copy to the controller (step ②). Another copy of the message is pushed into the message queue (step ③). The message parser asynchronously processes pop messages from the message queue (step ④) and store the parsed messages into MongoDB along with the current timestamp

Listing 1: Example of a FlowMod message stored in MongoDB

```
"_id" : ObjectId("5f840827165060224 8ff3b5d"),
"switch" : "0x2",
"message" : {
    "header" : {
        "version" : 1,
        "type" : 14,
        "length" : 80,
        "xid" : 679114503
    },
    "match" : {
        "wildcards" : 4194294,
        "in_port" : 1,
        "dl_src" : "00:00:00:00:00:00",
        "dl_dst" : "80:00:00:00:00:02",
        "dl_vlan" : 0,
        "dl_vlan_pcp" : 0,
        "dl_type" : 0,
        "nw_tos" : 0,
        "nw_proto" : 0,
        "nw_src" : "0.0.0.0",
        "nw_dst" : "0.0.0.0",
        "tp_src" : 0,
        "tp_dst" : 0
    },
    "cookie" : 0,
    "command" : 0,
    "idle_timeout" : 0,
    "hard_timeout" : 0,
    "priority" : 32768,
    "buffer_id" : NumberLong("4294967295"),
    "out_port" : 65535,
    "flags" : 1,
    "actions" : [ {
        "type" : 0,
        "len" : 8,
        "port" : 2,
        "max_len" : 65509
    } ]
},
"timestamp" : ISODate("2020-10-12T07:39:19.017Z")
```

(step ⑤). Messages sent from the controller to switches are handled in the same manner.

The previous version of Opimon [53] employed a single-process and multi-threaded design using Python's threading[3] module, where the monitoring module

[3]`https://docs.python.org/3/library/threading.html`

launched multiple threads each responsible for receiving, parsing, and forwarding of messages. However, this design suffered from low forwarding performance caused by the Global Interpreter Lock (GIL)[4] of Python. GIL is a mutex that ensures only a single Python interpreter thread can execute at a time. Although GIL simplifies the handling of thread-safety, CPU-intensive multi-threaded programs cannot benefit from multi-core CPUs. Using profilers, I found out that message parsing in Opimon is CPU-intensive and blocks the receiving and forwarding of messages. This induced prohibitive latency and packet drops at high traffic load.

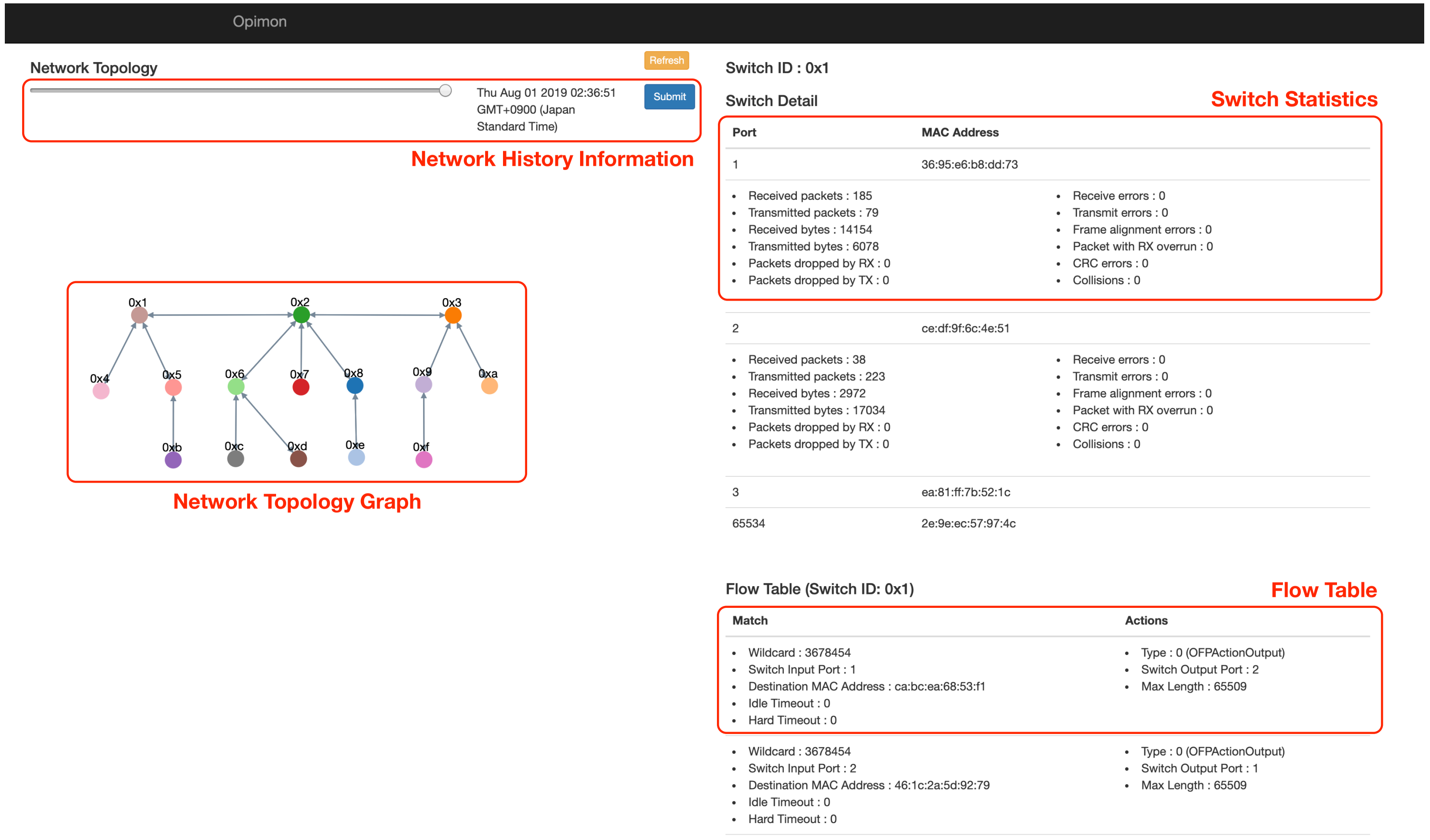
In this version, I redesign the monitoring module based on the Python's multiprocessing[5] module and separate the collection and parsing of messages into different processes. Since multi-processing is not limited by GIL, the new design allows the monitoring module to utilize multiple CPU cores. In addition, the message watcher processes and the message parser processes are loosely coupled through an asynchronous inter-process queue. This design allows the monitoring module to adapt to sudden changes in the message traffic and to scale the message watchers and parsers independently.

Collection of Network Information In addition to passively intercepting the messages exchanged in the control plane, Opimon actively queries the switches to collect more information. This design, however, causes a side effect because the OpenFlow controller will receive replies to queries that it has not issued. This potentially causes unexpected behavior of the controller and violates the goal of being transparent. Thus, Opimon marks injected messages with a special transaction identifier (xid) to distinguish them from OpenFlow messages generated by the controller. Replies from switches carrying the same special xid are filtered out and not forwarded.

The monitoring module collects the three types of network information in the following manner:

- Network Topology: The network topology is detected using the Link Layer Discovery Protocol (LLDP). The monitoring module injects LLDP packets

[4]`https://docs.python.org/3/glossary.html#term-global-interpreter-lock`
[5]`https://docs.python.org/3/library/multiprocessing.html`



Thu Aug 01 2019 02:36:51 GMT+0900 (Japan Standard Time)

Switch ID : 0x1

Switch Detail

Port	MAC Address
1	36:95:e6:b8:dd:73

- Received packets : 185
- Transmitted packets : 79
- Received bytes : 14154
- Transmitted bytes : 6078
- Packets dropped by RX : 0
- Packets dropped by TX : 0

- Receive errors : 0
- Transmit errors : 0
- Frame alignment errors : 0
- Packet with RX overrun : 0
- CRC errors : 0
- Collisions : 0

Port	MAC Address
2	ce:df:9f:6c:4e:51

- Received packets : 38
- Transmitted packets : 223
- Received bytes : 2972
- Transmitted bytes : 17034
- Packets dropped by RX : 0
- Packets dropped by TX : 0

- Receive errors : 0
- Transmit errors : 0
- Frame alignment errors : 0
- Packet with RX overrun : 0
- CRC errors : 0
- Collisions : 0

Port	MAC Address
3	ea:81:ff:7b:52:1c
65534	2e:9e:ec:57:97:4c

Flow Table (Switch ID: 0x1)

Match	Actions
• Wildcard : 3678454 • Switch Input Port : 1 • Destination MAC Address : ca:bc:ea:68:53:f1 • Idle Timeout : 0 • Hard Timeout : 0	• Type : 0 (OFPActionOutput) • Switch Output Port : 2 • Max Length : 65509
• Wildcard : 3678454 • Switch Input Port : 2 • Destination MAC Address : 46:1c:2a:5d:92:79 • Idle Timeout : 0 • Hard Timeout : 0	• Type : 0 (OFPActionOutput) • Switch Output Port : 1 • Max Length : 65509

Figure 20: Visualization of the virtual network using Opimon

into a switch using a PacketOut message. When an adjacent switch receives an LLDP packet, it encapsulates the packet in a PacketIn message and sends to the controller. The monitoring module intercepts and parses this message and records the adjacency between switches.

- Switch Information: The switch ID, number of ports, and port MAC addresses are collected by querying the individual switches using FeaturesRequest messages. Port statistics are obtained using PortStatsRequest messages.

- Flow Table: The flow table of each switch is monitored by intercepting FlowMod messages sent out from the controller, which are used to add, modify or delete flow entries on a switch. The statistics of each flow entry is collected by periodically querying switches using FlowStatsRequest messages.

Opimon can detect topology changes in the network caused by incidents such as switch and link failures. When a switch fails and disconnects from Opimon, Opimon stops monitoring the switch and removes it from the web interface. When a link fails, the failed link is detected by LLDP and removed from the web interface.

3.3.3 Visualization Module

Overall Design The visualization module is responsible for showing the collected network information to the user in real-time. The visualization module is a web application consisting of a front-end and a back-end.

The front-end periodically polls the back-end to retrieve the latest network information and renders the result as a web page. D3.js is used to render the network topology and jQuery is used to show a table of port statistics and a flow table of the selected switch. The back-end exposes a RESTful API that queries MongoDB and returns the latest network information in JSON format. The back-end is built upon the Express web application framework and Node.js JavaScript runtime.

MongoDB is used as a database due to its flexibility, speed, and scalability. The visualization module uses four tables on MongoDB database as follows:

- flow_mods: This table collects data obtained from FlowMod messages such as switch id, matching conditions, action, hard time out, and idle timeout. Table 4 shows the fields in the `flow_mods` table.

- switch_port: This table stores data obtained from the FeaturesReply message including the switch id, number of the ports in a switch, and MAC address of each port. Table 5 shows the fields in the `switch_port` table.

- topology: This table stores data obtained from the LLDP packets. This includes switch id and port numbers of source and destination switch. Table 6 shows the fields in the `topology` table.

- port_stats: This table stores data obtained from PortStatsReply message. It contains statistics of each port on a switch. Table 7 shows the fields in the `port_stats` table.

Web Interface Figure 20 shows the web interface of Opimon. The web interface has three sections (network topology, switch information, and flow table) divided into two columns.

- Network Topology: This section shows the network topology. A node represents a switch in the network and an arrow edge represents as a link with a direction of the data flow. Each node is labeled with the ID of the corresponding switch. The labels can be customized in a configuration file to make them easier to identify. Each node in the graph is clickable to show the switch information and flow table of that switch. On top of the network topology, a slider is available to select the time in the past to investigate the previous status of the network that Opimon collected from the selected time.

- Switch Information: This section shows the details of the selected switch in the network topology view as a table. The table shows the MAC address and statistics of each switch port. This information is collected from FeaturesReply message. Each row in the table shows the port statistic that the monitoring module collects from PortStatsReply of StatReply message.

Table 4: Fields in `flow_mods` table

Field	Description
header	Header of packet
version	Version
type	Type of packet
length	Length of packet
xid	Transaction ID
match	Instance of `OFPMatch`
wildcards	Wildcard fields
in_port	Switch input port
dl_src	Ethernet source address
dl_dst	Ethernet destination address
dl_vlan	Input VLAN ID
dl_vlan_pcp	Input VLAN priority
dl_type	Ethernet frame type
nw_tos	IP ToS (actually DSCP field, 6 bits)
nw_proto	IP protocol or lower 8 bits of ARP opcode
nw_src	IP source address
nw_dst	IP destination address
tp_src	TCP/UDP source port
tp_dst	TCP/UDP destination port
cookie	Opaque controller-issued identifier
command	One of the following values. `OFPFC_AD, OFPFC_MODIFY, OFPFC_MODIFY_STRICT, OFPFC_DELETE, OFPFC_DELETE_STRICT`
idle_timeout	Idle time before discarding (seconds)
hard_timeout	Max time before discarding (seconds)
priority	Priority level of flow entry
buffer_id	Buffered packet to apply to (or `0xffffffff`)

Table 4 continued from previous page

Field	**Description**
out_port	For `OFPFC_DELETE*` commands, require matching entries to include this as an output port.
flags	One of the following values. `OFPFF_SEND_FLOW_REM, OFPFF_CHECK_OVERLAP, OFPFF_EMERG`
actions	List of `OFPAction*` instance

Table 5: Fields in `switch_port` table

Field	**Description**
switch_id	Switch ID
port_no	Port number
hw_addr	MAC address of port

Table 6: Fields in `topology` table

Field	**Description**
switch_dst	Destination switch ID
port_dst	Destination switch port
switch_src	Source switch ID
port_src	Source switch port

- Flow Table: This section shows the active flows in the selected switch. A table shows the match condition and action of each flow. Hard timeout and idle timeout are shown in the table. The information of the flow table is collected from FlowMod and FlowStatsReply of StatReply messages.

Table 7: Fields in `port_stats` table

Field	Description
port_no	Port number
rx_packets	Number of received packets
tx_packets	Number of transmitted packets
rx_bytes	Number of received bytes
tx_bytes	Number of transmitted bytes
rx_dropped	Number of packets dropped by RX
tx_dropped	Number of packets dropped by TX
rx_errors	Number of receive errors
tx_errors	Number of transmit errors
rx_frame_err	Number of frame alignment errors
rx_over_err	Number of packet with RX overrun
rx_crc_err	Number of CRC errors
collisions	Number of collisions

3.3.4 Security Analysis Module

The security analysis module analyzes the network traffic and classifies normal network traffic and a DDoS attack traffic. This module is implemented as a security analysis server, which uses machine learning techniques for analyzing the traffic collected from the data plane and control plane. The security analysis server queries the monitored data from the database. Then, it classifies the traffic from the data using a pre-trained machine learning model. Opimon employs SVM and Deep Learning as classification models since they are commonly used for DDoS attack detection in literature [64, 70, 71, 73].

SVM is often combined with a *kernel function*, which maps the input data into a higher dimensional space to classify linearly inseparable data. In this research, I selected Linear, Polynomial, and Radial Basis Function (RBF) as the kernel functions. The definition of each kernel is shown in Equation (1), (2), and (3).

$$\text{Linear}: k(x, y) = x^{\top} y \tag{1}$$

$$\text{Polynomial}: k(x, y) = \left(\gamma (x^{\top} y) + r\right)^{d} \tag{2}$$

$$\text{RBF}: k(x, y) = e^{-\gamma \| x-y \|^{2}} \tag{3}$$

From Equation (1), (2), (3), the tuning parameters are determined as degree d, gamma γ, and coefficient r. The parameter of degree indicates the dimension. The C-Support Vector Machine Classification (SVC) is used for the SVM model. ThunderSVM Python library is used to implement SVM on the security analysis server.

Deep learning is a class of machine learning model that uses multiple layers of neural networks. In Opimon, the Deep Feed Forward (DFF) network is used [74]. Keras with TensorFlow backend engine [75], a high-level API for machine learning, is used to implement DFF network on the security analysis server. Figure 21 shows the structure of the DFF network.

I use the 2009 DARPA Intrusion Detection dataset to train the machine learning models [76, 77]. This dataset contains 10 days of network traffic including HTTP, SMTP, and DNS background traffic, and consists of 7,000 pcap files with a total size of 6.5TB. DARPA-2009 DDoS Attack-20091105 was assigned as a sample DDoS attack dataset [78]. This dataset contains about 6 minutes of SYN flood DDoS attack in three pcap files. The DDoS attack traffic is sent from 100 different IPs. For the normal packet dataset, I selected 1,000 pcap files from the original dataset that were collected from the same day, based on information from the ground truth table. The ground truth table shows dates, start-end times and traffic descriptions for each attack traffic such as packet type, source and destination IP, source and destination port. Before training the models, I pre-processed the dataset as shown in Figure 22. After pre-processing the datasets, those files are converted into a file in Comma-Separated Values (CSV) format. Finally, the CSV file is converted into SVM-Light format [79] for using with ThunderSVM.

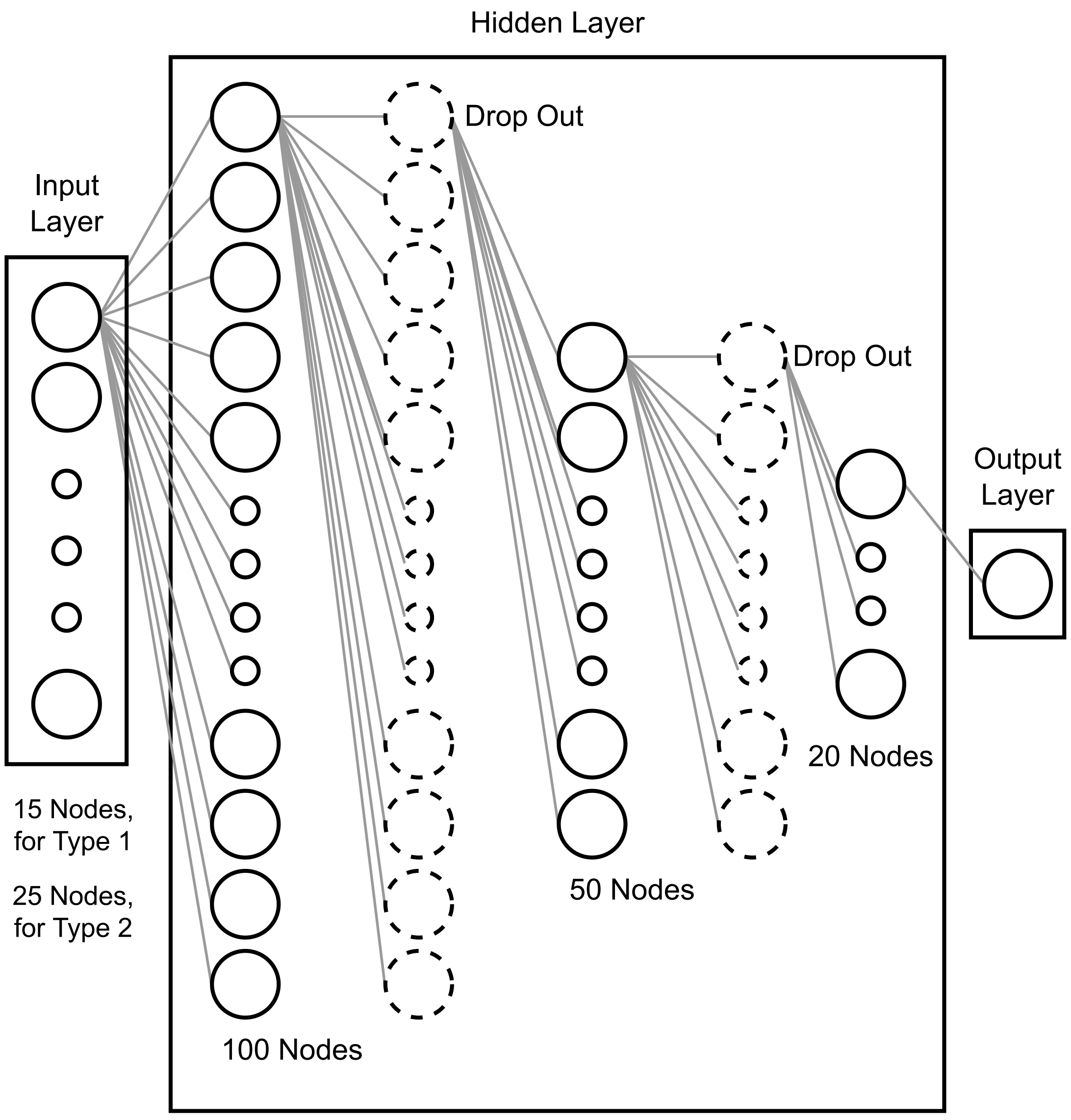


Figure 21: Simple deep feed forward (DFF) neural network

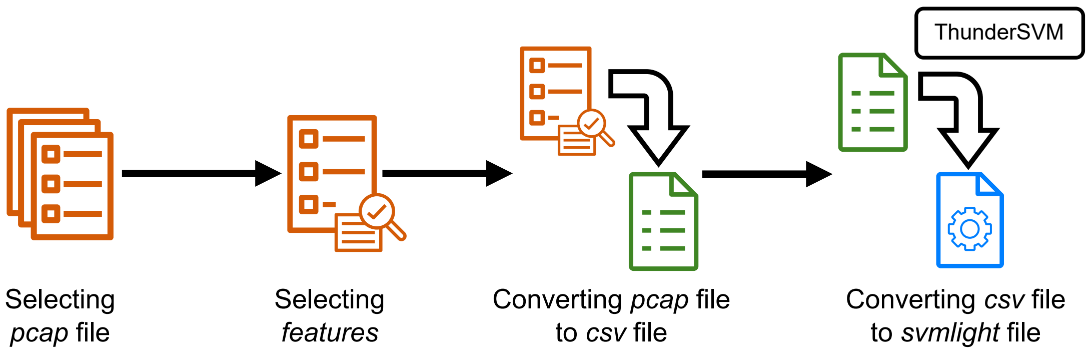

ThunderSVM
Selecting *pcap* file
Selecting *features*
Converting *pcap* file to *csv* file
Converting *csv* file to *svmlight* file

Figure 22: Overview of data pre-processing

I aggregate packet information for a certain time window, which in this case is one second. The machine learning model analyzes the patterns on the number of transferred packets, the number of observed IP addresses and so on over the time windows, and classifies each time window into the time period where a DDoS attack is underway or the time period where no attack is detected.

In this work, I have two types of features I am interested in. Table 8 and Table 9 present the information for these two types of features respect

Table 8 shows time window aggregated features, the first type of network features are aggregated for each time window. Table 9 shows packet specific features, the second type of features on each packet are added with the previous time window aggregated information. The total numbers of time window aggregated features and packet specific features are 16 and 26, respectively.

In this experiment, I generated the training dataset and testing dataset by combining the data of the time windows under the DDoS attack situation and normal situation. Table 10 shows the number of samples in each dataset. Time window aggregated dataset denotes the dataset in the type of Table 8. Packet basis dataset denotes the dataset in the type of Table 9. I have two size datasets with different sizes for packet specific data, packet specific (S) and (L). Packet specific (S) is prepared for the purpose of comparing the performance with time window aggregated dataset so that it has the same sample size. Packet specific (L) is a prepared to evaluate the performance of SVM and DFF with a huge size of dataset.

Table 8: Time Window Aggregated Features

Feature	**Description**
Status	Whether being attacked by DDoS attack or not
All Packets	Number of arrived packets within a time window
Num_IPpair	Number of unique IP source and IP destination address pairs within a time window
Num_IPsrc	Number of unique IP source addresses within a time window
Num_IPdst	Number of unique IP destination addresses within a time window
Num_Portpair	Number of unique source and destination port pairs within a time window
Num_Portsrc	Number of unique source ports within a time window
Num_Portdst	Number of unique destination ports within a time window
Num_Ether	Number of Ethernet packets within a time window
Num_Dot3	Number of IEEE 802.3 packets within a time window
Num_TCP	Number of Transmission Control Protocol packets within a time window
Num_UDP	Number of User Datagram Protocol packets within a time window
Num_ARP	Number of Address Resolution Protocol packets within a time window
Num_ICMP	Number of Internet Control Message Protocol packets within a time window
Num_LLC	Number of logical link control packets within a time window
Num_Len	Number of unique packet lengths within a time window

Table 9: Packet Specific Features

Feature	**Description**
Ether_or_Dot3	Packet is either Ethernet or IEEE 802.3
MAC_src	MAC address of the source device
MAC_dst	MAC address of the destination device
Ether_type	Type of Ethernet packet (*e.g.* 0x0800 is Internet Protocol version 4 (IPv4))
LLC	Packet is logical link control (LLC)
LLC_ssap	Source logical address of LLC packet
LLC_dsap	Destination logical address of LLC packet
IP_ttl	Time to live value of the packet
IP_version	The version of IP, such as IPv4
TCP	Packet is TCP
UDP	Packet is UDP
ARP	Packet is ARP
ICMP	Packet is ICMP
pLen	Packet size
Status	Whether the network is attacked by DDoS attack or not
num_ip_pair	Number of source and destination IP paris
all_packets	Total number of packets arrival within a window
ratio_ip	Number of IP sources divided by the number of IP destinations
num_ip_src	Number of source IPs
num_ip_dst	Number of destination IPs
num_port_pair	Number of source port pairs

Table 9 continued from previous page

Feature	**Description**
ratio_port	Number of source ports divided by the number of destination ports
num_port_src	Number of source ports
num_port_dst	Number of source ports
weight_ip	Weight of the number of top three IP pairs
weight_port	Weight of the number of top three port pairs

Table 10: Number of samples in each dataset

Dataset	DDoS attack	Normal	Total
Time window aggregated	335	365	700
Packet specific (S)	331	369	700
Packet specific (L)	481,903	518,097	1,000,000

3.4 Evaluation

I evaluated Opimon from two aspects. First, I deployed Opimon to a virtual network and tested if Opimon can correctly detect the network topology and the flow table of each switch. I conducted the same test on a large-scale international OpenFlow testbed, PRAGMA-ENT. Second, I measured the performance of a controller with and without Opimon and quantified the overhead imposed by Opimon. In addition, I investigate machine learning techniques for applying with DDoS attack detection. Support Vector Machine (SVM) and Deep Feed Forward (DFF) are compared in this preliminary investigation.

3.4.1 Correctness of Monitoring Results

A virtual network was used to verify if Opimon is able to correctly detect the topology of the network and the flow entries installed on each switch. I used Mininet [80], a network emulator that creates virtual networks comprising many

Listing 2: Mininet script to create the virtual network

```
from mininet.topo import Topo

class MyTopo(Topo):
  def __init__(self):
    # Initialize topology
    Topo.__init__(self)

    # Create Switch1 to Switch15
    ...

    # Add links between core switches
    self.addLink(Switch1, Switch2)
    self.addLink(Switch2, Switch3)

    # Add links between core and edge switches
    self.addLink(Switch1, Switch4)
    self.addLink(Switch1, Switch5)
    self.addLink(Switch2, Switch6)
    self.addLink(Switch2, Switch7)
    self.addLink(Switch2, Switch8)
    self.addLink(Switch3, Switch9)
    self.addLink(Switch3, Switch10)

    self.addLink(Switch5, Switch11)
    self.addLink(Switch6, Switch12)
    self.addLink(Switch6, Switch13)
    self.addLink(Switch8, Switch14)
    self.addLink(Switch9, Switch15)

topos = { 'mytopo': ( lambda: MyTopo() ) }
```

hosts and switches on a single computer, to create a virtual network. Listing 2 shows the Mininet script to create the virtual network. The virtual network comprises 15 switches forming a tree topology. I used Ryu's builtin L2 learning switch (`ryu.ryu.app.simple_switch`) as the OpenFlow controller.

Figure 20 is a screenshot of Opimon's web interface when monitoring the virtual network. I verified that the network topology and the flow entries in each switch are correct.

Opimon was also deployed to a large-scale international OpenFlow network testbed referred to as the PRAGMA Experimental Networking Testbed (PRAGMA-ENT) [81]. This testbed is maintained and used by researchers participating

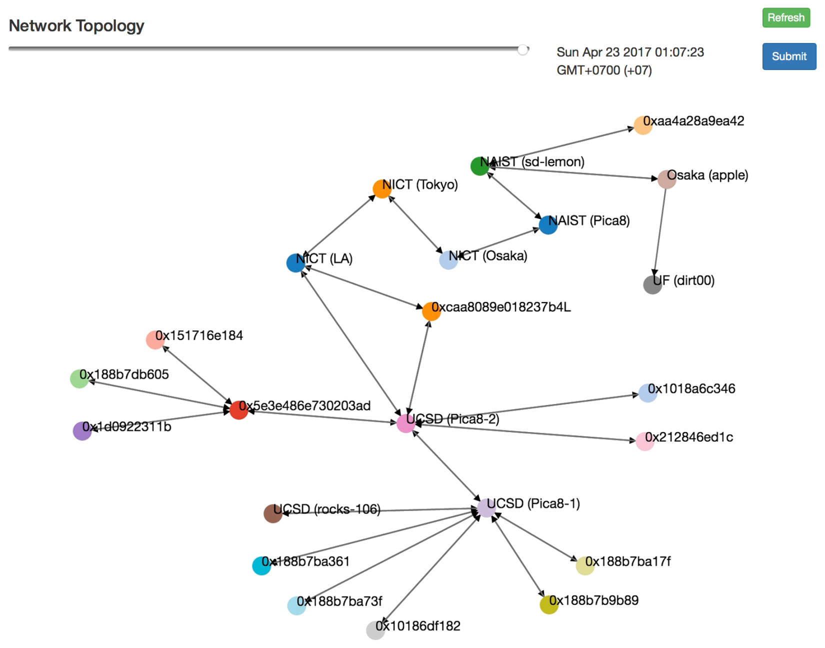


Figure 23: Visualization of the PRAGMA-ENT network using Opimon

in the Pacific Rim Application and Grid Middleware Assembly (PRAGMA). The OpenFlow switches in this network, including both hardware and software switches, are deployed at multiple PRAGMA partner institutions in Japan, the United States, and Taiwan. The switches are connected via VLANs and Generic Routing Encapsulation (GRE) tunnels. PRAGMA-ENT uses a controller implementation based on a routing switch of Trema OpenFlow framework to emulates a layer 2 switch [82]. Figure 23 shows the network topology of PRAGMA-ENT. It shows the switches deployed at the Nara Institute of Science and Technology (NAIST), Osaka University, National Institute of Information and Communications Technology (NICT), University of California San Diego (UCSD), and University of Florida (UF). I confirmed that Opimon was able to correctly monitor the PRAGMA-ENT network in real-time.

3.4.2 Overhead Imposed by Opimon

I measured the latency and throughput of Ryu's L2 learning switch controller with and without Opimon to quantify the overhead imposed by Opimon. A

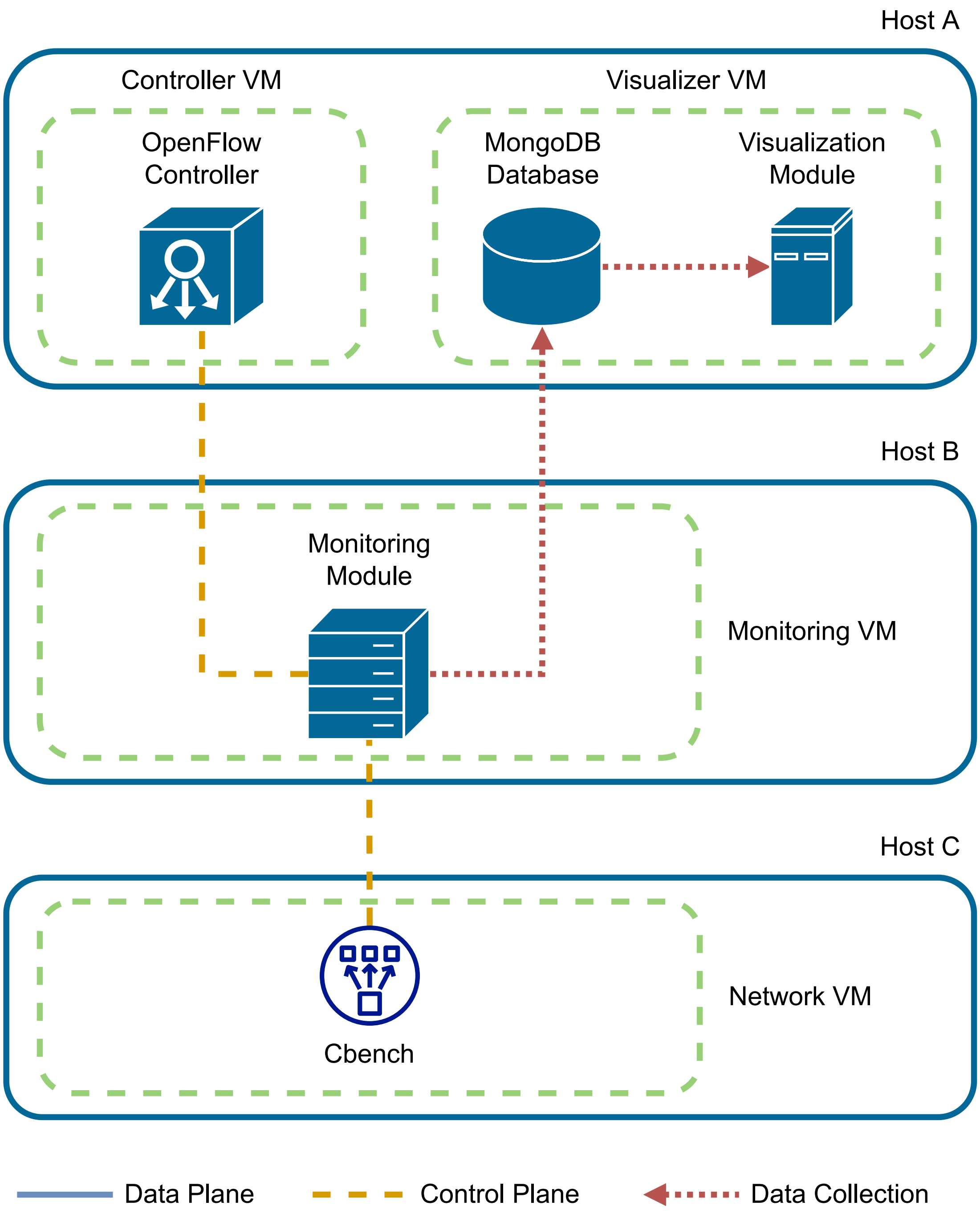


Figure 24: Experimental Environment

Table 11: Virtual machines used for evaluation

VM	vCPU	RAM	Software
Controller VM	4	8 GB	Ryu L2 Learning Switch
Visualizer VM	8	16 GB	Visualization Module & MongoDB
Monitoring VM	16	16 GB	Monitoring Module
Network VM	16	16 GB	Cbench

benchmark tool for OpenFlow controllers called Cbench [83, 84] was used in this evaluation. Cbench simulates a number of OpenFlow switches by opening multiple connections to the controller and concurrently sending PacketIn messages to simulate the arrival of packets at switches. In the latency mode, Cbench sends a PacketIn message and waits for the controller to reply with a FlowMod message. In the throughput mode, Cbench sends a large number of PacketIn messages and counts the number of FlowMod messages received from the controller.

I set up four VMs on three hosts for this evaluation as shown in Fig. 24. On host A, I deployed a VM that ran the OpenFlow controller and another VM that ran Opimon's visualization module and MongoDB. I deployed a VM for Opimon's monitoring module on host B and a VM for Cbench on host C. All hosts were equipped with two Intel Xeon Silver 4208 CPUs and 96 GB of RAM. Table 11 shows the resource allocation to each VM.

I used Python 3.8.5 and the latest master version of Ryu [42] (git commit `a394673`). The visualization module was executed with Node.js 10.19.0 and Express 4.17.1. MongoDB 3.6 was used as the database. I used Cbench included in the latest master version of Oflops [84] (git commit `762d517`) and built it with the reference OpenFlow implementation [85] (git commit `82ad07d`). All VMs ran Ubuntu Server 18.04.

Using Cbench, I measured the latency and throughput of the controller while varying the number of simulated switches from 16 to 256. I compared the latency and throughput with and without using Opimon in each case. Each measurement was repeated 10 times to quantify the performance variability.

Figure 25 shows a comparison of latency. Here the error bars represent the standard deviation. As expected, the latency of the controller becomes higher

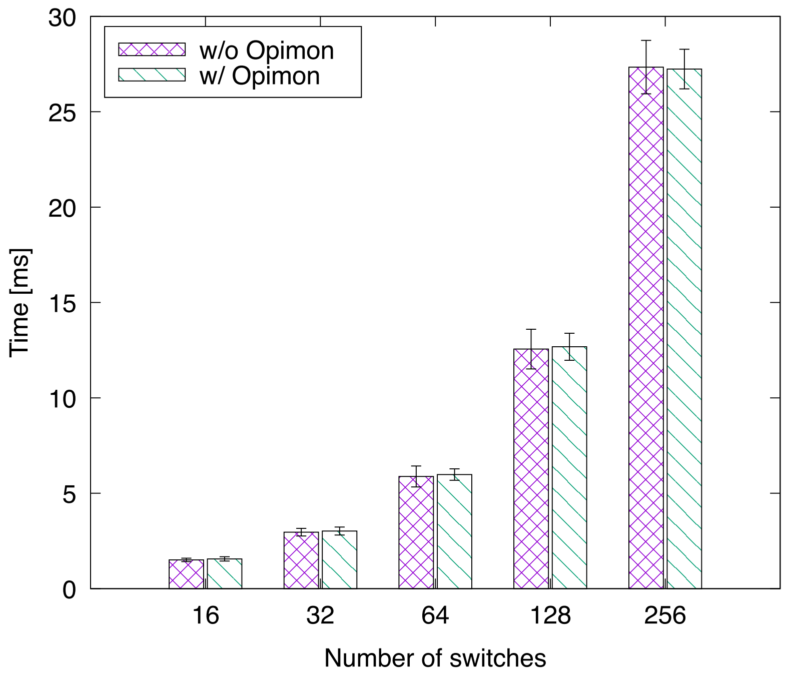


Figure 25: Controller latency (Ryu L2 learning switch)

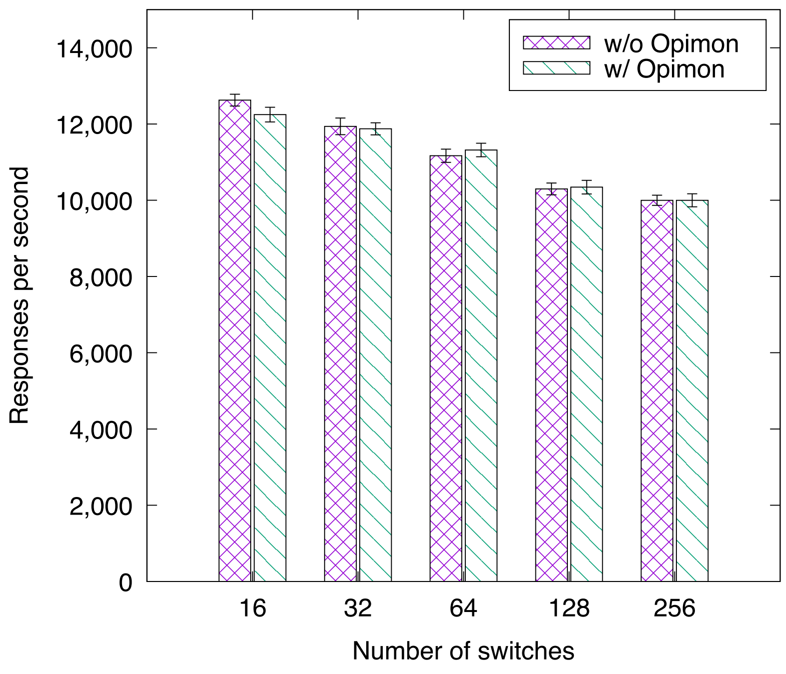


Figure 26: Controller throughput (Ryu L2 learning switch)

when the number of switches increases. The results indicate that Opimon introduces an overhead of approximately 0.5 μs at maximum. This is only 3% of the latency without Opimon, even when handling 256 switches. Figure 26 shows a comparison of throughput. The results shows that Opimon decreases the throughput of the controller for 5% at most.

3.4.3 Performance Comparison to OpenNetMon

In this experiment, I compared the overhead of Opimon to an existing OpenFlow monitoring system, OpenNetMon [49]. I used the same environment as the previous experiment (Fig. 24), and deployed OpenNetMon on the controller VM. Since OpenNetMon only works with its built-in routing switch controller based on POX, I measured the overhead caused by Opimon and OpenNetMon using this routing switch controller.

Figures 27 and 28 show the latency and throughput measured using Cbench. These plots indicate that the performance difference between OpenNetMon and Opimon is marginal. Furthermore, the fact that Opimon worked with OpenNetMon's routing switch controller based on POX demonstrates that Opimon is transparent to the OpenFlow controller and controller framework. This evaluation shows the advantage of Opimon over OpenNetMon in terms of compatibility with any OpenFlow controllers and controller frameworks.

3.4.4 Accuracy of DDoS Attack Detection

The experiments were conducted on a server running Ubuntu 18.04 equipped with an NVIDIA GTX 1080 GPU. The experiments were performed using Keras with TensorFlow backend for deep learning and ThunderSVM for SVM. The testing dataset is 10% of dataset randomly selected based on K-Folds cross validation [86]. The remaining data is used as the training dataset. As shown in Table 12, I classify the prediction results into four categories: True Positive (T_p), True Negative (T_n), False Positive (F_p), and False Negative (F_n). These four categories are used to calculate the performance metrics including with accuracy, recall, precision, and F1 score. These values are calculated with K-Folds cross validation step to reduce the possible overfitting in the model [87].

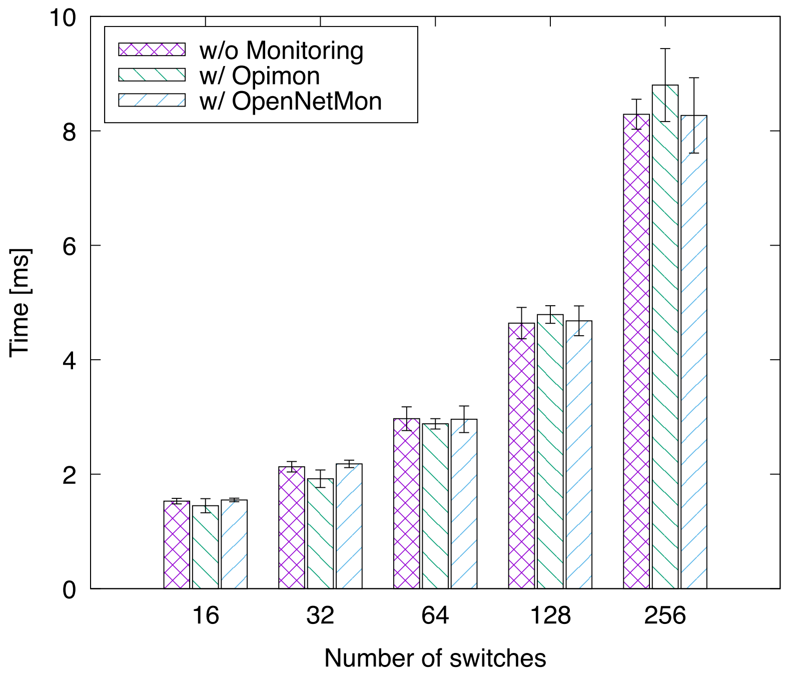


Figure 27: Controller latency (OpenNetMon routing switch)

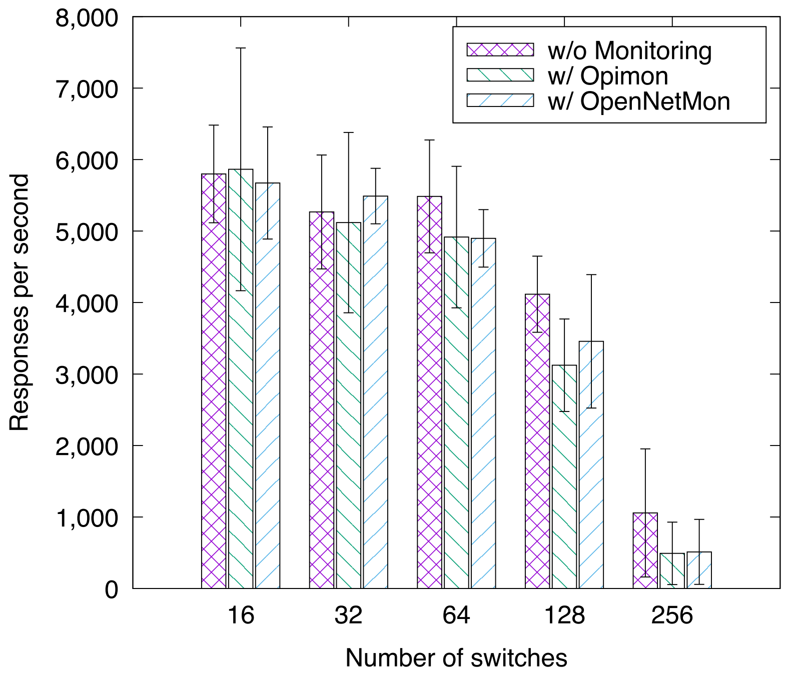


Figure 28: Controller throughput (OpenNetMon routing switch)

Table 12: Confusion matrix

		Actual	
		Positive	Negative
Predicted	Positive	True Positive (T_p)	False Positive (F_p)
	Negative	False Negative (F_n)	True Negative (T_n)

- Accuracy: A ratio of the correct prediction result from the total prediction result. This metric is commonly used to evaluated the performance a machine learning model.

$$\text{Accuracy} = \frac{T_p + T_n}{T_p + T_n + F_p + F_n} \tag{4}$$

- Recall (R): A ratio of the correct prediction result from the total positive of the actual results.

$$R = \frac{T_p}{T_p + F_n} \tag{5}$$

- Precision (P): A ratio of the corect prediction result from the total predicted positive results.

$$P = \frac{T_p}{T_p + F_p} \tag{6}$$

- F1 score: A measure of a test's accuracy by using recall R and precision P.

$$\text{F1 score} = \frac{2 \times (P \times R)}{P + R} \tag{7}$$

For SVM, I compared several SVM kernels, which are Linear, Polynomial, and Radial Basis Function (RBF) kernel, to find the best suited kernel for DDoS attack detection. The results are shown in Table 13 and Table 14. In these tables, w denotes the time window aggregated dataset and p denotes the packet specific dataset.

Polynomial kernel has achieved the highest accuracy for overall. The different of pre-processing dataset affect some kernels by increase the training time and detection accuracy, especially RBF and Linear kernel. In the experiments of

Table 13: Best accuracy from each kernels

Kernel	**Accuracy**	**Recall**	**Precision**	**F1-score**
Linear (w)	76.00%	0.765	0.744	0.754
Linear (p)	90.28%	0.925	0.883	0.902
Polynomial (w)	93.01%	0.922	0.933	0.927
Polynomial (p)	92.58%	0.894	0.933	0.906
RBF (w)	60.05%	0.912	0.599	0.661
RBF (p)	48.43%	1.000	0.479	0.648

Table 14: Time used for each kernel

Kernel	**Training Time (s)**	**Testing Time (s)**
Linear (w)	331.6156	0.0033
Linear (p)	30.6892	0.0030
Polynomial (w)	371.1180	0.0029
Polynomial (p)	379.4168	0.0031
RBF (w)	5.9043	0.0038
RBF (p)	172.5352	0.0034

polynomial kernel, I also plotted graphs to find relationships between the degree of the polynomial function and accuracy and calculation time. The graphs are shown below in Figure 29, 30, 31. In these graph, the x-axis represents the value of γ and the y-axis shows the a different measurement value on each figure. The y-axis of Figure 29 shows accuracy of the model, Figure 30 shows the time in second that uses for training the model, and Figure 31 shows the time in second that uses for classification.

Table 14 shows that RBF kernel takes the least time to train the model. However, the model performance is lower compared to the other kernels. From Figure 29, 30, 31, I found that when degree increased, training time and accuracy tend to increase while predict time tends to decrease.

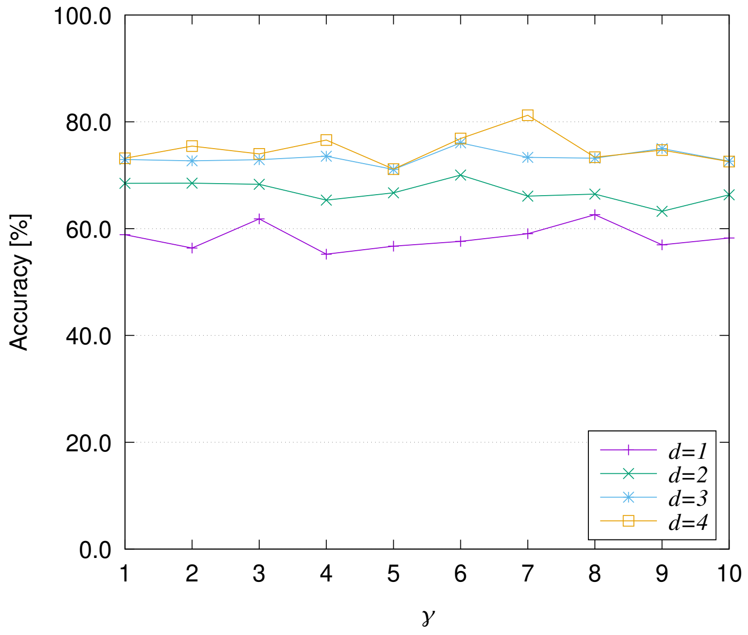


Figure 29: Relationship between γ, d, and accuracy

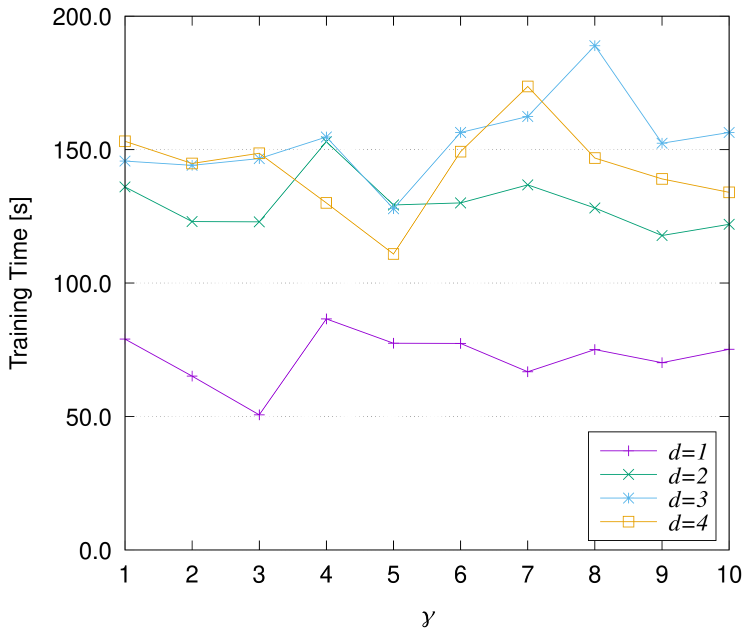


Figure 30: Relationship between γ, d, and calculation time for training the model

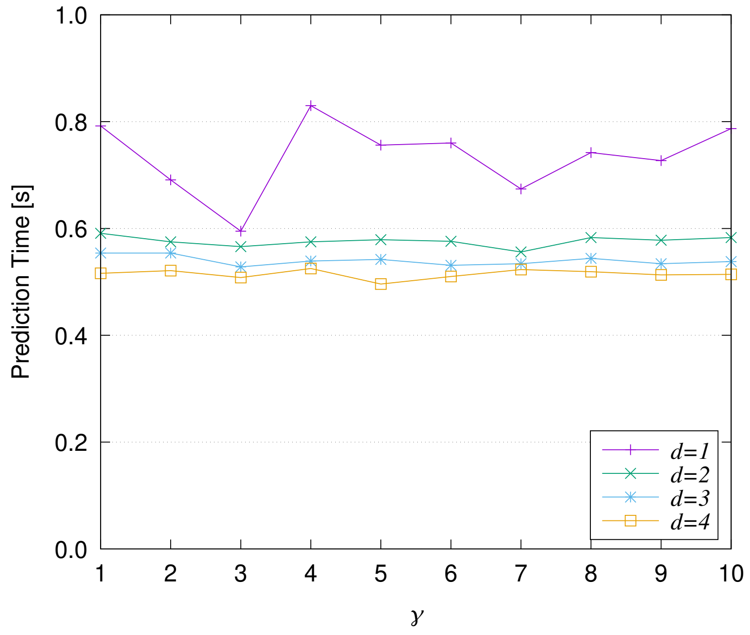


Figure 31: Relationship between γ, d, and calculation time for testing the model

For deep learning, Keras was used in the experiment. I used the same dataset as the SVM experiment. Keras is used to builds the deep learning model. I ran the test with trial and error by adding the model layer to find the best accuracy. Accordingly, the best results in terms of accuracy from each algorithm are shown in Table 15.

The result of the experiment that run with packet specific (L) dataset is used to evaluate the result for evaluating with huge traffic. Deep learning achieved the highest accuracy of 99.63%, recall of 0.994, precision with 0.998 and F1-score is 0.996. On the other hand, SVM achieved an accuracy rate of 81.23%, recall of 0.927, precision of 0.756 and F1-score is 0.826 when considered with the same data size and data type. Conversely, SVM took 138.260 seconds for training and 0.500 seconds for classification, whereas deep learning took 239.614 seconds for training and 14.651 for classification. Thus, I conclude that deep learning performs better than SVM in terms of accuracy of classification, whereas SVM performs better than deep learning in terms of classification time.

Table 15: Best results in terms of accuracy for each algorithm

SVM	Model Performance				Time Used (s)	
	Accuracy	Recall	Precision	F1-score	Train	Test
Time window aggregated	93.01%	0.922	0.933	0.927	371.118	0.003
Packet specific (S)	92.58%	0.894	0.933	0.906	379.417	0.003
Packet specific (L)	81.23%	0.927	0.756	0.826	138.260	0.500

Deep Learning	Model Performance				Time Used (s)	
	Accuracy	Recall	Precision	F1-score	Train	Test
Time window aggregated	61.30%	0.240	0.452	0.295	3.314	0.117
Packet specific (S)	68.30%	0.366	0.922	0.504	3.438	0.115
Packet specific (L)	99.63%	0.994	0.998	0.996	239.614	14.651

3.5 Conclusion & Future Work

I proposed Opimon, a monitoring system for OpenFlow networks. Opimon collects the topology, flow tables, and switch statistics from the target network, and interactively visualizes the state of the network through a web interface in real-time. Opimon is completely transparent to the network and works with any OpenFlow switch or controller without any modification required. Furthermore, Opimon imposes little overhead to the network performance and can be used in production networks. Using Cbench, I simulated up to 256 virtual switches and measured the latency and throughput of the controller with and without using Opimon. The results indicated that the overhead to latency introduced by Opimon is less than 0.5μs (or 3%). In addition, the overhead in terms of throughput was less than 5%.

The preliminary result of machine learning algorithm for the problem of DDoS attack detection has been addressed. Two algorithms, Support Vector Machine (SVM) and Deep Feed Forward (DFF) were compared in terms of classification accuracy and computing time. It was found that DFF can classify the data with a higher accuracy compared to SVM. Therefore, deep learning is a useful choice for the classification of DDoS attack packets in terms of accuracy. However, SVM is an appropriate choice for faster classification.

As future work, I are planning to implement a new module for detecting anomalous traffic in the network, such as DDoS attacks, using machine learning algorithms. In contrast to existing Intrusion Detection Systems (IDS) which require traffic probes to be installed in the data plane, this module will not require additional traffic probes because it will analyze the OpenFlow messages collected using Opimon. I will also extend the visualization module to report the detection results in real-time.

For the next step of network traffic classification, I will apply a deep learning on the security analysis server in Opimon for detecting the attack traffic in the OpenFlow network. However, the DARPA dataset, which is used during preliminary experiment, contains traffic from in the data plane only. I believe that using data plane traffic only is not enough to detect DDoS attack in some cases for OpenFlow network such as a traffic that aims to flood the OpenFlow messages to the controller. I plan to use the data plane dataset and control plane dataset to fully detect DDoS attack in the OpenFlow network. The control plane traffic dataset only could be used for lightweight DDoS attack detection due to the size of dataset is smaller compare to the size of data plane traffic dataset. For example, I can use the number of the PacketIn messages in the control plane to detect DDoS attacks in the data plane. Since a DDoS attack is composed of many different packet types in the flow and the switch might send the packet to the controller for getting the action of the flow, a huge number of OpenFlow messages will be generated in the control plane. I can use this fact and other behaviors of the control plane to detect a DDoS attack in the data plane. In the next step, I will simulate the DARPA dataset, which include normal traffic and DDoS attack traffic obtained from the data plane of a traditional network, in OpenFlow network. Opimon monitoring tool is used for capturing the packet to create a control plane traffic dataset. Due to the fact that each controller may provide different behavior of control plane traffic, multiple of control plane traffic dataset might be collected from different OpenFlow controller. I will apply the deep feed forward on the control plane dataset only for detecting DDoS attack traffic in data plane. Then, the combination of two plane traffic datasets are used for improve the DDoS attack detection in term of speed of classification and classification accuracy. I will evaluate the DDoS detection with each dataset for finding the best accuracy and nearly real-time DDoS attack detection. Additionally, an automation script will be developed for periodically running machine learning models to detect DDoS attack. The result of the DDoS attack detection will be shown in the web interface for alerting the network administrator.

4. Network Traffic Time Series Analysis with Empirical Dynamic Modeling (EDM)

4.1 Introduction

Denial of Service (DoS) attack is one of the most common cyber-attacks in computer networks. DoS attack floods the victim with massive traffic and makes the target service unavailable to legitimate users. If the attack traffic originates from multiple sources, the attack is called Distributed Denial of Service (DDoS) attack. The scale of DDoS attacks can be easily expanded by using computers distributed across the internet. Figure 32 illustrates how the DDoS attack works.

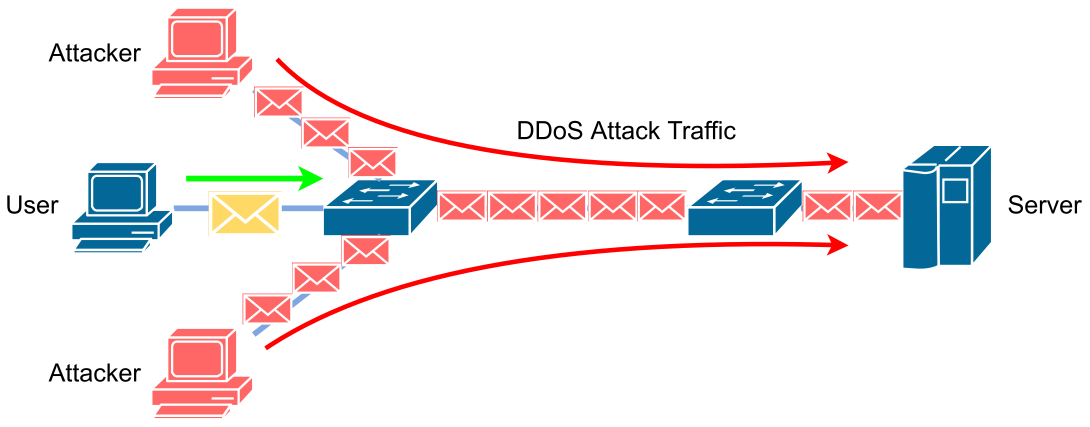


Figure 32: Distributed Denial of Service (DDoS) attack

DDoS attack detection has been a long-standing research topic in the network security field. The difficulty in detecting DDoS attacks stems from the fact that a variety of packet types and techniques can be employed. As mentioned in Section 3.2.3, machine learning has been successfully applied to detect DDoS attacks in previous studies. However, machine learning based models require a large amount of computational resources and datasets to train, they are not suitable for building practical and real time detection systems. Therefore, in this chapter, I propose to use EDM to detect DDoS attacks in networks by predicting the time series behavior of network traffic.

4.2 Related Work

4.2.1 kEDM: a performance-portable implementation of EDM

In this research, another EDM implementation, kEDM, was used. kEDM is a performance-portable implementation of EDM based on the Kokkos framework [88]. kEDM improves and solves several remaining issues in mpEDM, such as performance portability and optimization of the kernels. mpEDM uses the ArrayFire [89] library to implement k-nearest neighbors search and lookup functions, and it performs well across diverse platforms supported by ArrayFire without reimplementing low level functions. However, it lacks the ability to further optimize or modify the kernel to increase additional performance.

Kokkos is a performance portability framework that aims to provides abstractions for both parallel execution of code and data management. Kokkos supports multiple low-level programming models, such as OpenMP, OpenCL, and CUDA. It helps developers to manage and support multiple memory patterns on low-level programming models. Kokkos allows kEDM to remove several inefficiencies from mpEDM and implement custom-tailored kernels. As a result, kEDM achieves up to 5.5× higher performance than mpEDM. Additionally, kEDM also provides a Python binding that allows users to use kEDM with Python programming language [90] instead of C++ programming language.

4.2.2 Other Time Series Prediction Methods

In this research, I have compared time series prediction with EDM to other typical machine learning methods, such as Auto Regression and Long Short-Term Memory. This section introduces these methods.

Auto Regression (AR) is a basic technique to predict the future values of the time series. Equation (8) represents an AR model. It is based on a linear regression with its own past values. AR has a time lag p to represent the number of past values. It uses past p time steps values to predict a current value of the time series. AR model is successfully applied in the multiple works of DDoS attack detection [91–93].

$$x_t = c + \sum_{i=1}^{p} \phi_i x_{t-i} + \varepsilon_t \tag{8}$$

where:

t	: Time step
x_t	: Time series at time t
c	: Constant
p	: Number of past values
ϕ_i	: Parameters of the model
ε_t	: White noise

Meanwhile, Long short-term memory (LSTM) is one of deep machine learning techniques. The structure of the LSTM is improved from Recurrent Neural Network (RNN). RNN is a neural network that has a loop back to persist the information. The loop back can change to connect to the other neural network for creating the chain. Figure 33 shows the structure of an RNN cell and the equations are shown in Eq. (9). RNN has a simple structure, which contains only a single activation layer, such as the hyperbolic tangent (tanh) layer. However, RNN encounters a long-term dependencies issue that causes RNN unable to learn the connection between past and future information. LSTM is proposed to solve this issue.

LSTM was introduced by Hochreiter et al. in 1997 [94]. LSTM is designed to avoid long-term dependencies. Figure 34 shows the structure of a LSTM cell. Each layer of LSTM contains input gate i, output gate o, and forget gate f. The compact forms of the equations for the forward pass of LSTM is shown in Eq. (10). Unlike RNN, each layer of the LSTM has a forget gate f for selecting which information to keep or remove from cell state C. As shown in the Fig. 34, hyperbolic tangent (tanh) functions are commonly used as activation functions for cell output or cell state, and sigmoid functions are used as activation functions for the recurrent steps. This design allows LSTM to hold the information in long-term. LSTM is suitable for classifications and predictions of time series. LSTM is applied in many researches and applications in various fields [95–97].

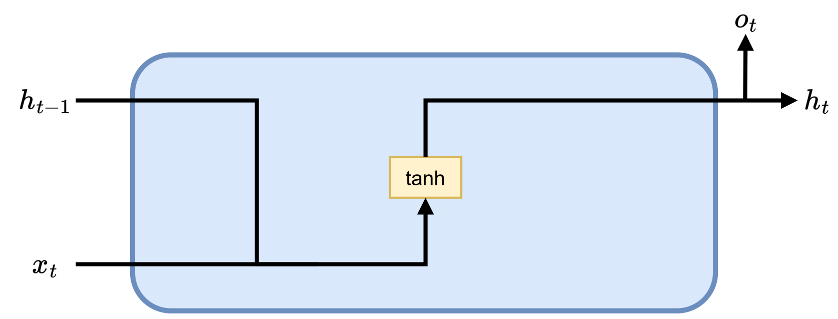


Figure 33: The structure of Recurrent Neural Network (RNN)

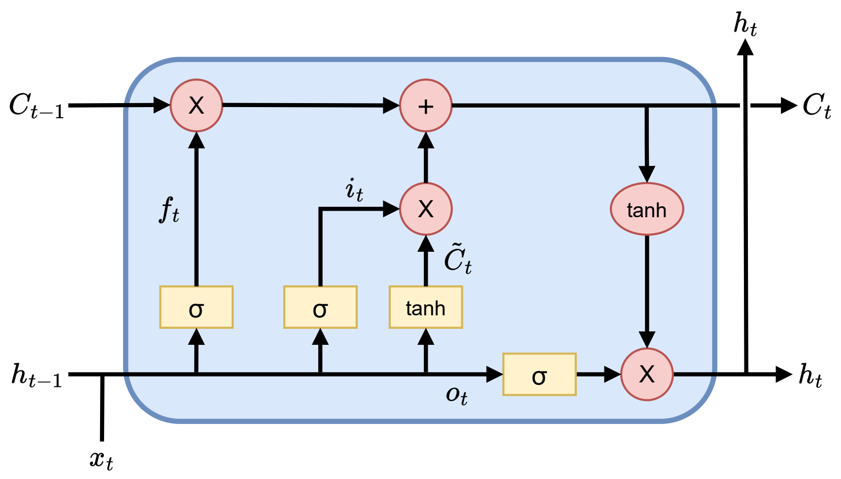


Figure 34: The structure of Long Short-Term Memory (LSTM)

$$
\begin{aligned}
h_t &= \tanh(Ux_t + Wh_{t-1} + b) \\
o_t &= \sigma(Vh_t + c)
\end{aligned} \tag{9}
$$

where:

$$
\begin{aligned}
t &: \text{Time step} \\
\sigma &: \text{Activation function} \\
x_t &: \text{Input vector at time } t \\
h_t &: \text{Hidden state vector} \\
o_t &: \text{Output gate's activation vector} \\
U, V, W &: \text{Weight matrices} \\
b, c &: \text{Bias vectors}
\end{aligned}
$$

$$
\begin{aligned}
f_t &= \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \\
i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\
o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \\
\tilde{C}_t &= \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \\
C_t &= f_t \circ C_{t-1} + i_t \circ \tilde{C}_t \\
h_t &= o_t \circ \tanh(C_t)
\end{aligned} \tag{10}
$$

where:

$$
\begin{aligned}
t &: \text{Time step} \\
\sigma &: \text{Activation function} \\
x_t &: \text{Input vector at time } t \\
f_t &: \text{Forgot gate's activation vector} \\
i_t &: \text{Input gate's activation vector} \\
o_t &: \text{Output gate's activation vector} \\
h_t &: \text{Hidden state vector} \\
\tilde{C}_t &: \text{Cell input activation vector} \\
C_t &: \text{Cell state vector} \\
W_f, W_i, W_o, W_C &: \text{Weight matrices} \\
b_f, b_i, b_o, b_C &: \text{Bias vector}
\end{aligned}
$$

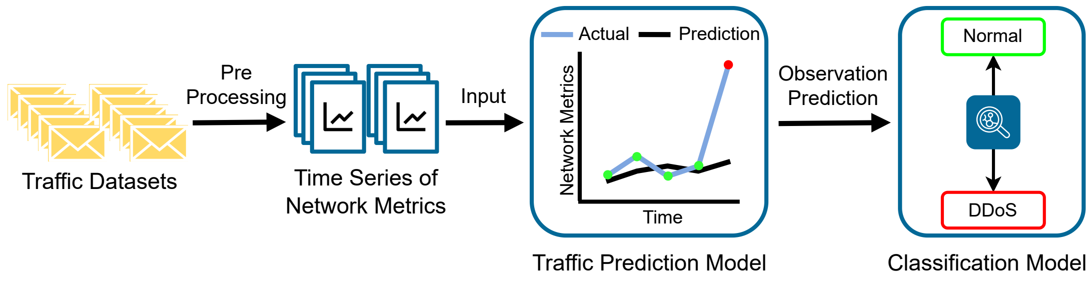


Figure 35: Overview to apply EDM and ML for DDoS attack classification

4.3 Implementation

I conduct a preliminary experiment to evaluate the performance of EDM in network prediction and compare it with AR and LSTM. Figure 35 shows the overview of the process to apply EDM to the network traffic dataset. In the first step, I pre-process the raw packet trace and convert it into multiple time series. I extract the features from the header of each packet and aggregate them into a time series. The temporal resolution of the generated time series is one second. Additionally, the values in the time series are normalized between zero and one. Table 16 lists the features that are aggregated to time series. After that, the time series is used as an input to train and test the machine learning models to evaluate the results. Three machine learning techniques are used to build a DDoS attack detection model: Simplex projection in EDM, AR, and LSTM. Finally, classification models are trained to classify normal and DDoS attack traffic.

The DARPA Scalable Network Monitoring 20091103 dataset [76] is used to evaluate the results. This dataset is a synthetic dataset that imitates 10 days of benign and DDoS attack traffic. The attack recorded in this dataset is a SYN flood attack towards a single destination. The dataset is composed of 10 sets, one set for each day. A set includes 600 to 722 pcap files, each of which contains raw packets. In this evaluation, set 1 and 2 of the DARPA dataset are used. Set 1 contains monitored network traffic from November, 3rd 2009 at 01:23:36 PM (GMT) to November 4th, 2009 at 05:01:46 AM (GMT). Set 2 contains monitored network traffic from November 4th, 2009 at 05:01:47 AM (GMT) to November 5th, 2009 at 5:04:48 AM (GMT). These two sets contain 600 and 720 pcap files, respectively.

Table 16: Features of time series

Feature	**Description**
throughput	Number of bytes
packets_count	Number of packets
avg_size	Average size of packets
proto_set	Number of unique packet protocols
proto_count_TCP	Number of TCP packets
proto_count_UDP	Number of UDP packets
proto_count_ICMP	Number of ICMP packets
flags_set	Number of unique TCP flags
flags_count_PA	Number of packets that have PSH-ACK flags
flags_count_FPA	Number of packets that have FIN-PSH-ACK flags
flags_count_S	Number of packets that have SYN flag
flags_count_SA	Number of packets that have SYN-ACK flags
flags_count_A	Number of packets that have ACK flag
flags_count_FA	Number of packets that have FIN-ACK flags
IP_src_set	Number of unique source IP addresses
IP_dst_set	Number of unique destination IP addresses
IP_sport_set	Number of unique TCP/UDP source ports
IP_dport_set	Number of unique TCP/UDP destination ports
ddos_flag	DDoS attack label

Table 17: Time series dataset

Name	Type of Traffic	# of Time Steps	# of DDoS Time Steps
set1_normal	Normal	56,234	0
set1_ddos	DDoS & Normal	56,234	3,889
set2_normal	Normal	86,576	0
set2_ddos	DDoS & Normal	86,576	4,453

During pre-processing, two datasets are created: the DDoS attack time series dataset and the normal traffic time series dataset. The DDoS attack time series is created by aggregating all packets in the pcap files. The normal traffic time series are created by removing the DDoS attack packets using the ground truth labels. Both datasets have the same number of time series and number of time steps, but the time series differ at the time steps where the DDoS attack is occurring. Table 17 shows the time series dataset after packet aggregation. The normal traffic in set 1 (`set1_normal`) is used to train the model. The normal traffic time series and DDoS attack time series in set 2 (`set2_normal` and `set2_ddos`) are used to test the model. However, the load of traffic of each period in a day is different. To match the training and test dataset period, set 2 time series for testing are trimmed to have the same time of the day as set 1, which is from 01:23:36 PM (GMT) to 05:01:46 AM (GMT) on the next day.

Three prediction models are created by using various Python libraries. The model predicts the future values of time series for each individually features. First, the EDM Simplex projection model is implemented using `kEDM`[6]. The Simplex projection function has three hyperparameters: embedding dimension (E), time delay (τ), and prediction interval (T_p). In this experiment, E and τ are varied to find the best combination for predictions. Second, the AR model is implemented using the linear regression function from `scikit-learn`. Third, the LSTM model is implemented using `keras`. The LSTM model contains 50 LSTM units and uses the ReLu activation function. To the inputs to the AR and LSTM models, a time series splitter is needed to split a time series into a small chunk for X and Y. Figure 36 illustrates the time series splitter when $E = 3$ and $\tau = 2$. X represents input time series, which contains past E values for predicting a value at time step t. X is used during training the model and predicting the time series. Y represents as output results, which contains an actual value at time step t. Y is used during training the model only. This function allows the AR and LSTM model to accept the same hyperparameters with EDM. Due to the time shift of the embedding dimension, the first predicted time steps is shifted and start at step $(E-1)\cdot\tau+1$. In this experiment, the prediction model is capable of predicting network traffic a one time step ahead, which is 1 second in this experiment. The

[6]`https://github.com/keichi/kEDM`

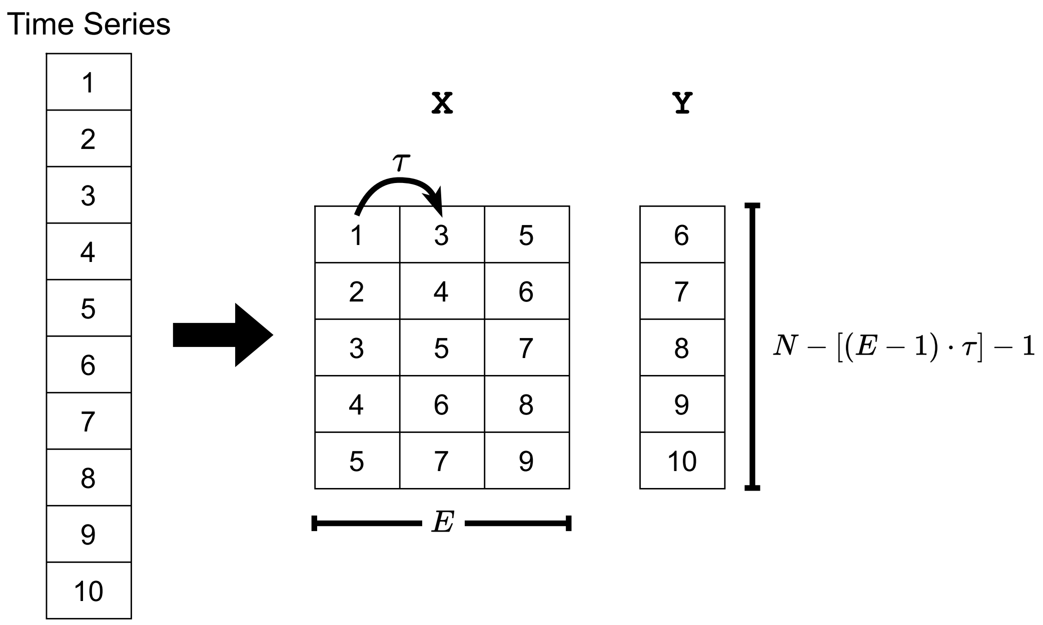


Figure 36: Time series splitting for inputs of AR and LSTM when $E = 3$ and $\tau = 2$

past E observation values are needed to predict the next time step.

The classification model is used to classify whether the current network state is under DDoS attacks or not. The observed and predicted time series are used as inputs to the classification model. A random forest classifier are used to classify between normal state and DDoS attack state. Multiple classification methods were tested with `lazypredict` Python library [98] to select the best classification model and the random forest shows the best classification accuracy. Stratified K-folds cross-validation is applied to split the data for train and test the classification model. The accuracy and F1-score are measured.

4.4 Evaluation

I evaluated EDM with the network traffic prediction from two aspects. First, I compared EDM, LSTM, and AR for the incoming network traffic prediction and measured the error of the prediction to compare with observation. Second, I compared the prediction results between three models for classifying the DDoS attack.

The cluster nodes of the high-performance computing system operated by the Information Initiative Center at the Nara Institute of Science and Technology, were used for evaluation. Each cluster node is equipped with two 12-core Intel Xeon E5 2650v4 CPUs, one NVDIA Quadro P4000 (8GB) GPU, 256GB of RAM and 240GB of local SSD. For the software, I used Python 3.8.1 and the following libraries for building each model. `kedm` 0.3.1 was used to perform simplex projection function. `scikit-learn` 1.1.1 was used to create an AR model, calculate the Root Mean Square Error (RMSE), and train a random forest classifier. `keras` 2.9.0 was used to build an LSTM model.

4.4.1 Incoming Network Traffic Prediction

The Root Mean Square Error (RMSE) is a metric to measure a difference between a pair of observations in the system and prediction value from the model at the same time steps. Equation (11) is the definition of RMSE where x_i is the observation and $\widehat{x_i}$ is the prediction at time steps i. N is the total number of time steps. A smaller RMSE indicates that the predicted time series is similar to the observed time series. To compare the prediction performance of each model in predicting incoming network traffic, I used the RMSE.

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \hat{x}_i)^2}{N}} \tag{11}$$

To find the best prediction result, finding the best combination of hyperparameters was needed. I ran the three models while varying E from 1 to 100 and fixing τ to 1 to predict every 18 features that listed in Table 16. As mentioned in section 4.3, E refers to the number of past values used for prediction, and τrefers to the time interval between past values. This experiment used up to consecutive past 100 seconds to predict the next value. The `set1_normal` was used to train the model for predicting `set2_normal`. As mentioned in section 4.3, the DARPA dataset contains a SYN flood attack. Therefore, features that are related to the DDoS attacks were selected to evaluate the RMSE, which are `flags_count_S`, `packets_count` and `avg_size`.

Table 18 compares the RMSE and runtime, including model training time

Table 18: Hyperparameters for predicting important features of `set2_normal`

Feature	Model	E	τ	RMSE	Runtime [s]
flags_count_S	EDM	8	1	0.010162145	4.985092640
	AR	22	1	0.011150189	0.100373745
	LSTM	3	1	0.011257038	8.603404999
throughput	AR	32	1	0.035387852	0.189230442
	LSTM	69	1	0.035921140	65.61778855
	EDM	59	1	0.037669422	14.20452595
packets_count	AR	51	1	0.035663585	0.355469465
	LSTM	60	1	0.035795853	70.09596133
	EDM	59	1	0.037542822	13.56930447
avg_size	LSTM	93	1	0.029716772	89.16635299
	AR	66	1	0.030173687	0.732991695
	EDM	16	1	0.032064749	6.313738823

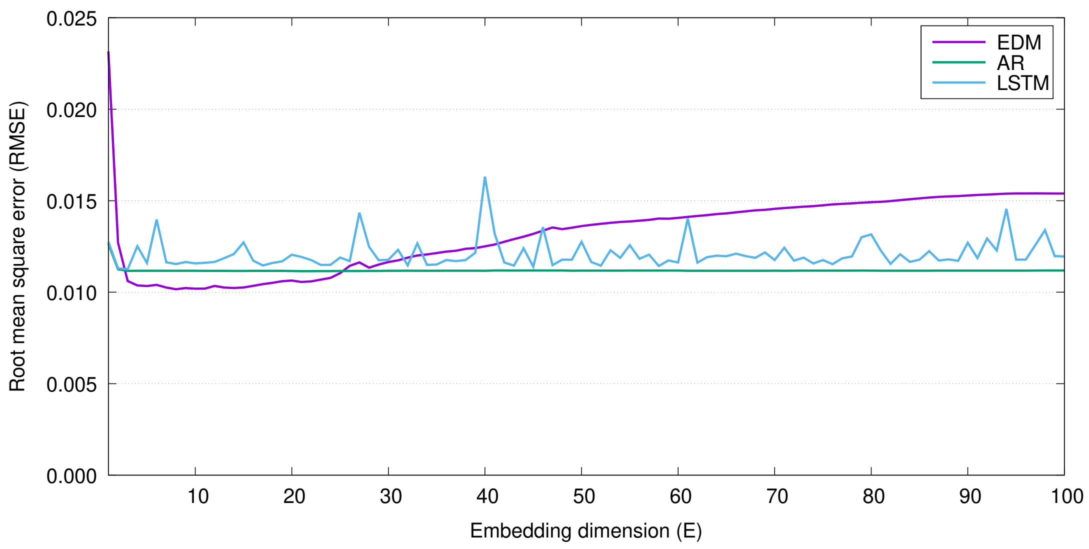


Figure 37: Comparison of RMSE when predicting `flags_count_S` with τ=1

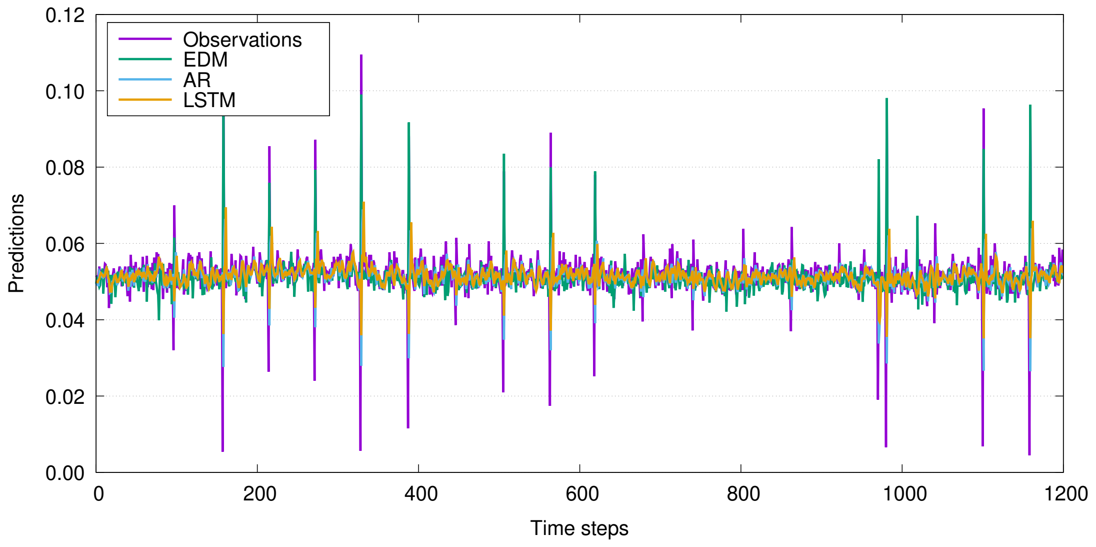


Figure 38: Comparison of prediction value when predicting `flags_count_S` between three models and observation (1,200 seconds)

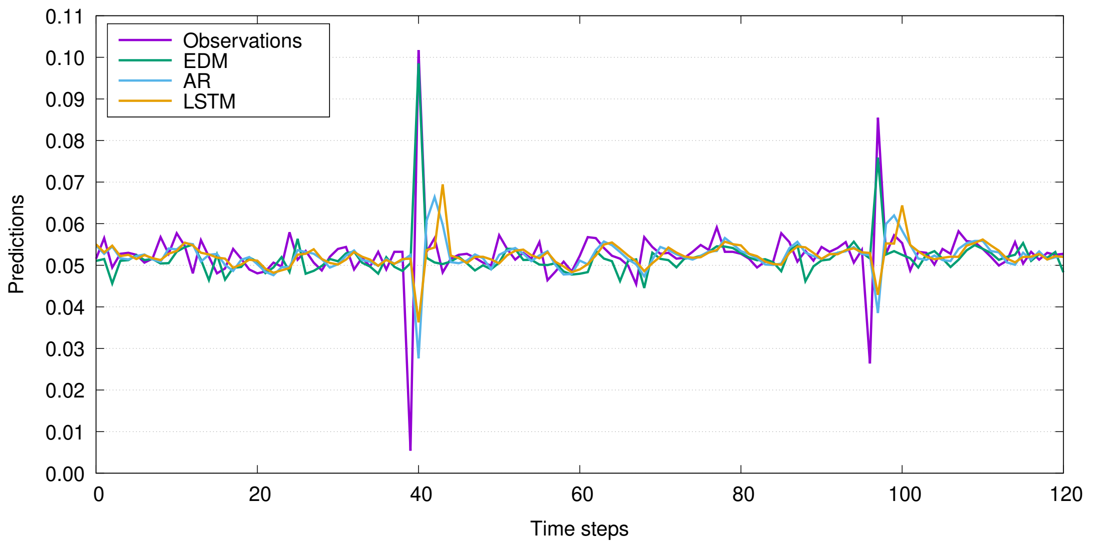


Figure 39: Comparison of prediction value when predicting `flags_count_S` between three models and observation (120 seconds)

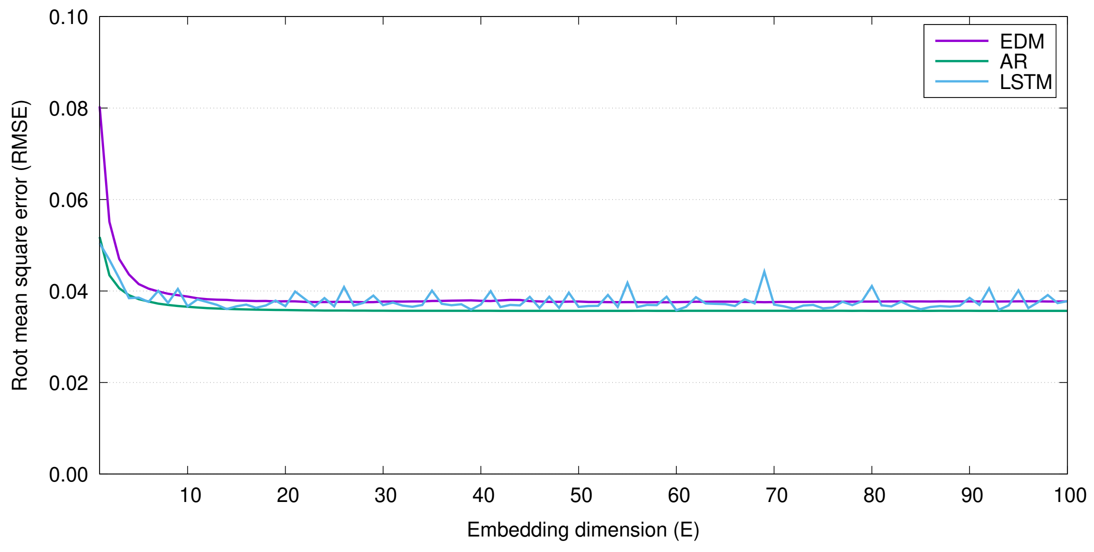


Figure 40: Comparison of RMSE when predicting `packets_count` with τ=1

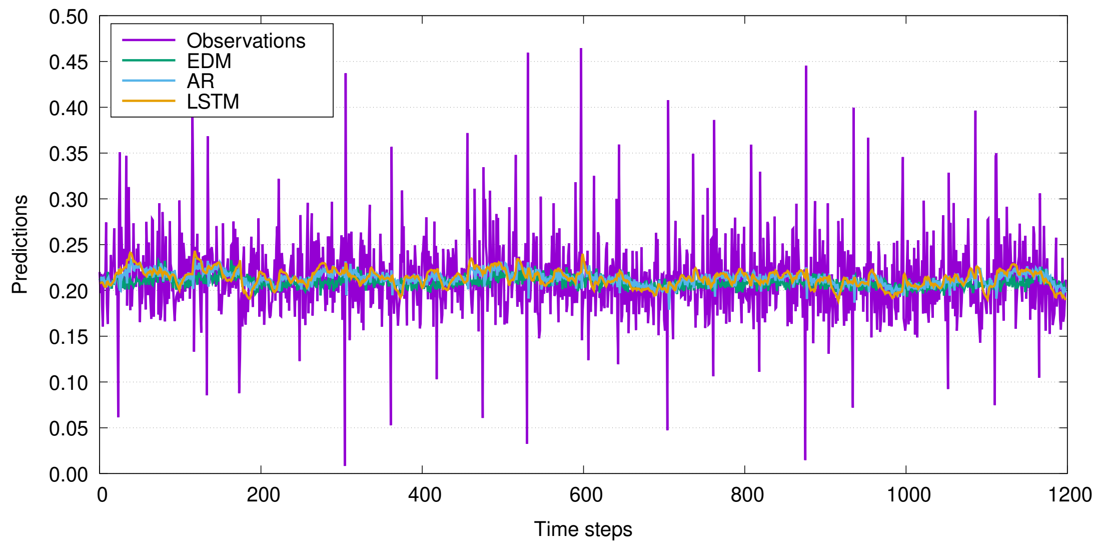


Figure 41: Comparison of prediction value when predicting `packet_counts` between three models and observation

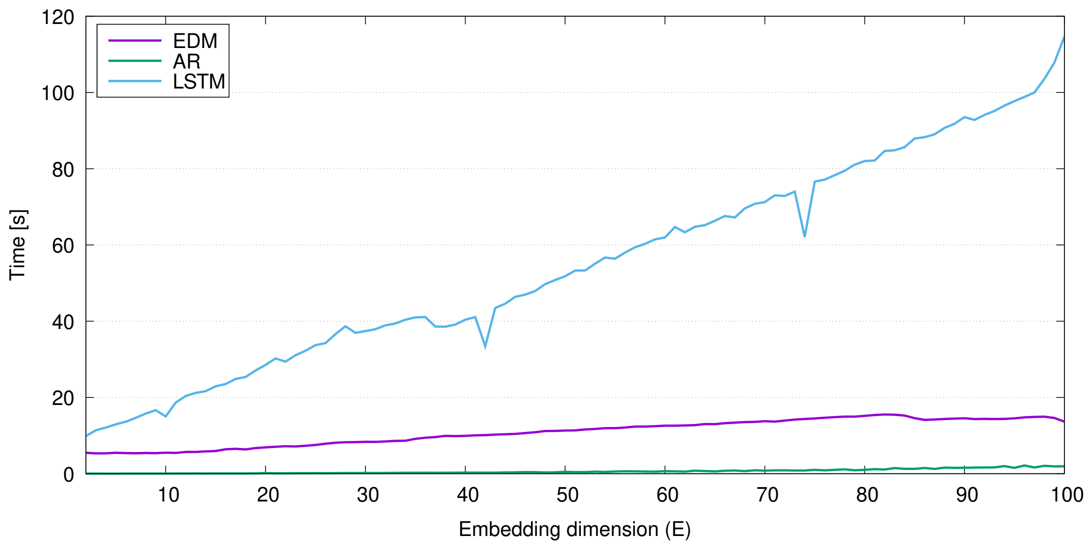


Figure 42: Comparison of average runtime of every prediction target

and time series prediction times, of each model. The results show that EDM is capable of correctly predicting the `flags_count_S` feature, better than other models. Figure 37 shows the RMSE when varying E and fixing τ to 1. The results indicates that the prediction error of EDM increased when E is too large. On the other hand, LSTM and AR were able to make more accurate predictions than EDM. Figure 38 and 39 show the normalized predictions of the three models compared along with the observation for 20 minutes and 2 minutes, respectively. These results show that EDM is capable of predicting stochastic spikes in the time series. However, EDM is not adequate for predicting features of which values change dramatically and continuously over the time compared to AR and LSTM as shown in Fig. 41. Fig. 42 plots the average runtime to predict all 18 features with each model when varying E from 1 to 100 and fixing τ to 1. AR, the most basic time series prediction model, performs the best in terms of prediction time. EDM is not as fast as AR, but it is up to 8× faster than LSTM when using 100 past values to predict the next time step. The difference of runtime between EDM and LSTM increases when using more past values.

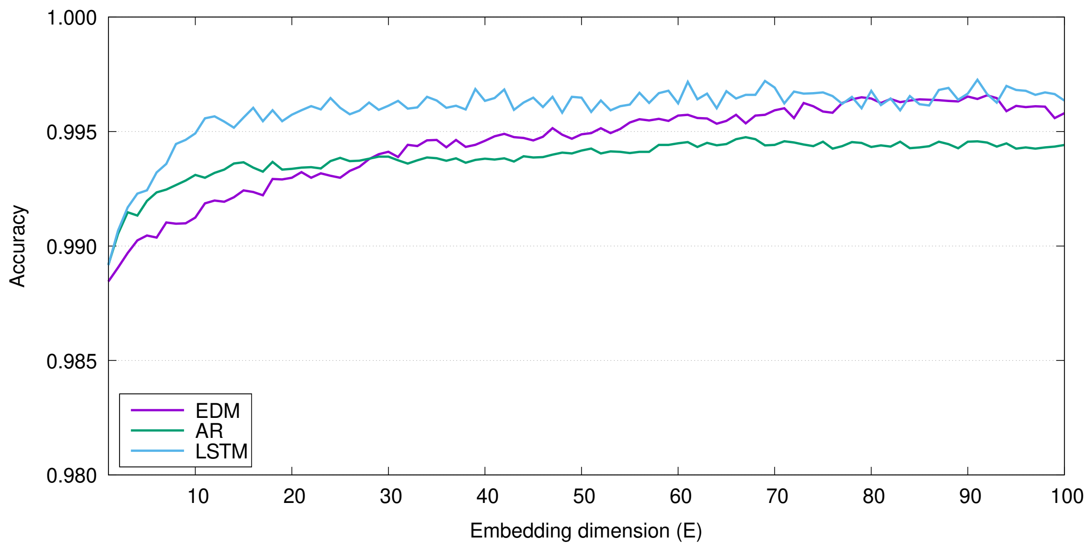


Figure 43: Comparison of DDoS attack classification accuracy with τ=1

Table 19: The highest classification accuracy and F1-scores of the classification results

Model	E	τ	Accuracy	F1-score
Observation only	-	-	0.988771	0.847440
EDM	92	1	0.996583	0.953906
AR	67	1	0.994753	0.927983
LSTM	91	1	0.997260	0.962595

Table 20: Confusion matrix for Observation only

		Predicted	
		Positive	Negative
Actual	Positive	53,808	309
	Negative	322	1,754

Table 21: Confusion matrix for EDM

		Predicted	
		Positive	Negative
Actual	Positive	54,015	102
	Negative	90	1,986

Table 22: Confusion matrix for AR

		Predicted	
		Positive	Negative
Actual	Positive	54,023	119
	Negative	176	1,900

Table 23: Confusion matrix for LSTM

		Predicted	
		Positive	Negative
Actual	Positive	54,059	59
	Negative	95	1,981

4.4.2 DDoS Attack Traffic Classification

In this evaluation, a random forest classifier was trained and evaluated using a DDoS attack traffic dataset, `set2_ddos`. The prediction model was trained on a non-DDoS attack traffic dataset, `set1_normal`, and then it was used to predict the network metrics of another traffic dataset, `set2_ddos`. The classifier was then trained and evaluated using both the observations (`set2_ddos` dataset itself) and the predicted network metrics. The results were evaluated in terms of classification accuracy and F1 score. The definitions of these metrics are shown in Equation (4) and Equation (7), respectively. These metrics were calculated by averaging the results over the stratified 5-folds cross-validation. All random seeds used in the evaluation were fixed to 42 for reproducibility.

I prepared a baseline case where the classifier was trained and evaluated only on the observation dataset without using predicted network metrics. Table 19 shows the comparison between the baseline and the three cases using EDM, AR and LSTM models for prediction. Figure 43 compares the classification accuracy of the three models when increasing $\boldsymbol{E}$, which represents the number of past values to be used to predict the next steps. The results show that all three models achieved better classification accuracy compared to the baseline case using only the observation dataset for classification. The case using the LSTM model provided the best classification accuracy and F1 score. The classification accuracy of EDM increased with increasing $\boldsymbol{E}$. This behavior is because more past values are used to predict a future value. EDM provided better accuracy than the AR model. In conclusion, EDM achieves the same level of classification accuracy as LSTM but requires much shorter runtime.

Table 19 shows that the classification on the observation dataset only also receives high accuracy and F1 score. The confusion matrices in Tables 20 to 23 summarize the numbers of actual classification results over the stratified 5-folds cross-validation tests, indicating that the dataset may be imbalanced. The DDoS attack traffic in the dataset accounts for less than only 5% of the total traffic. This issue may be degrading the accuracy of the models.

4.5 Conclusion & Future Work

This experiment explored the feasibility of applying EDM to network time series predictions as an application of EDM to the computer science field. In this experiment, I used the simplex projection algorithm in EDM to predict 18 network traffic features individually and compared the performance with two popular time series prediction methods, AR and LSTM. AR is the most simplest method for time series prediction and LSTM is more advanced neural network based time series prediction method. The preliminary results of this experiment show that EDM is capable of predicting the time series faster than LSTM and the classification result using EDM prediction provides higher classification accuracy than the AR-based model.

This experiment used univariate simplex projection to forecast a single time series from the time series only. EDM also includes multivariate forecast algorithms to predict a time series from multiple time series. Further experiments are needed to investigate if multivariate prediction can improve the accuracy. In addition, this experiment only covered a limited combination of hyperparameters. Investigating other parameters is also needed. T_p parameters is used to declare the number of time steps to prediction ahead and it needs to be varied for improving the prediction capability. Furthermore, 18 features are currently used to classify DDoS attacks, but these metrics represent a high-level overview of the network state, such as network throughput, number of packets, and other packet header information. To improve the classification accuracy, more specific network metrics, such as the number of packets per second for individual TCP/UDP ports, should be investigated.

5. Conclusion

5.1 Summary

In this dissertation, I have three contributions: accelerating EDM computation to enable real-time analysis of network traffic, capturing network traffic in an SDN in real-time with low-overhead, and applying EDM for network traffic prediction.

First, accelerating EDM computation to enable real-time analysis of network traffic, mpEDM is developed as a massive parallel library of Empirical Dynamic Modeling (EDM) to support modern supercomputer architecture. mpEDM improves the EDM algorithm from the existing EDM implementation, cppEDM. Additionally, it also supports GPU to accelerate the computation. From the result of the first application, Causal Map of the Zebrafish Brain at Single Neuron Resolution, mpEDM can finish the computation within 20 seconds on ABCI, which is 1,530× faster than cppEDM with the same dataset and hardware resources. mpEDM also analyzes 13× larger dataset under 200 seconds. This is the largest EDM causal inference achieved to date.

Second, capturing network traffic in an SDN in real-time with low-overhead, Opimon has been developed as a monitoring system for OpenFlow networks. Opimon monitors and visualizes the network status to the user via a web interface. Opimon is completely transparent to the network and monitors the network in real-time. Opimon is also optimized to reduce the monitoring overhead. I used Cbench to evaluate the Opimon capabilities. I simulated up to 256 virtual switches and measured the latency and throughput of the controller with and without using Opimon. The results indicated that the overhead to latency introduced by Opimon is less than 0.5μs (or 3%). In addition, the overhead in terms of throughput was less than 5%. Opimon also includes a security analysis module for analyzing and processing monitored network traffic. This module is used to convert raw network packets to a time series and feed them as input of the EDM function. Furthermore, it is also used to detect a DDoS attack with machine learning techniques. I conducted a preliminary experiment of DDoS attack detection comparison between Support Vector Machine (SVM) and Deep Feed Forward (DFF) in terms of classification accuracy and computing time. It was found that DFF can classify the data with higher accuracy compared to SVM.

Therefore, deep learning is a useful choice for the classification of DDoS attack packets in terms of accuracy. However, SVM is an appropriate choice for faster classification.

Finally, I used Empirical Dynamic Modeling (EDM) to predict network traffic in Software-Defined Networking (SDN) environments. The achievements of the previous contributions are used in this experiment. Opimon is used to convert raw messages to time series as input of EDM. kEDM, an improved version of mpEDM, is used to execute EDM functions. I used EDM to compare with the popular machine learning techniques, Long short-term memory (LSTM) and auto regression (AR). I compared these three methods in terms of classification accuracy and prediction time. The preliminary experiment shows that EDM is the optimal prediction model for traffic classification with high accuracy and fast prediction.

From the results of this dissertation, EDM shows a potential to apply real-time time series predictions as an application in the computer science field. It is also possible to apply EDM in other applications, which can aggregate the information to the time series such as network traffic. EDM needs, at least, a single time series of the past behavior as the input. However, EDM model is very sensitive to the hyperparameters. Varying hyperparameters and time scale of the time series are recommended to find the best hyperparameter combinations. Then, the best hyperparameters are used to predict the time series and analyze data further.

5.2 Future Work

I aim to integrate the EDM into the security analysis module of the Opimon for automatic network traffic prediction and anomaly traffic detection, especially DDoS attack. Control plane network traffic is used as an input dataset to feed into the model. I will analyze the control plane traffic instead of data plane traffic in SDN. In SDN, events occurring in the network can be grasped to some extent by monitoring the messages exchanged in the control plane, without monitoring the actual network traffic in the data plane. Since the amount of control plane messages is much smaller than the actual traffic in the data plane, this method will greatly reduce the data size and allow detecting DDoS attacks in real-time.

Then, the information of the network predictions will be visualized on the web interface of Opimon for alerting the users and network administrators.

Acknowledgements

I would like to thank the following people for their wisdom, guidance, and support. Without their help, this work would never have been possible.

First and foremost, I would like to express my gratitude to Professor Hajimu Iida for providing a great research environment. His laboratory, Laboratory for Software Design and Analysis, is a great place to pursue research.

To Professor Kazutoshi Fujikawa, I appreciate for his constructive comments and feedback made my work come this far. Without him, my research work and the dissertation would not have been possible.

I would like to express deep appreciation to my supervisors, Associate Professor Kohei Ichikawa and Assistant Professor Keichi Takahashi for their continuous support and guidance in my research work as well as my life in Japan. Their valuable suggestions and comments brought this research to fruition. Without them, I would not have successfully accomplished the doctoral course.

To Assistant Professor Gerald Pao, I would like to thank for his guidance on my research. Additionally, I appreciate his support during my visiting University of California San Diego in 2019.

To Assistant Professor Putchong Uthayopas, who was also my advisor during my time as an undergraduate student at Kasetsart University. He gave me invaluable knowledge in research methodology and widened my vision in the area of high-performance computing. His insightful suggestion helped shape this research in its initial stage. Without him, I could not come this far. Deep in my mind, I will always keep his image and he will always be remembered forever.

To Assistant Professor Chawanat Nakasan, I appreciate for his informative feedback and suggestions always helps raise the quality of this research.

I would like to acknowledge my dissertation committee. Thank you so much for reviewing my dissertation and for the insightful comments and suggestions that helped me to improve the overall quality of this dissertation.

To PRAGMA, Dr. Peter Arzberger, Ms. Shava Smallen, and Ms. Nadya Williams, I appreciate for providing me with a lot of assistance, and advice in organizing PRAGMA workshops and mentoring PRAGMA Students. I also express my gratitude towards Dr. Jason Haga and Dr. Prapaporn Rattanatamrong for their guidance to PRAGMA Students Steering Committee.

To Pananus Potisat, my best friend, I have no words that can express my gratitude for your support in these past few years. Thank you for taking care of me and cheering me up every time when I am depressed. Without you, I might not overcome multiple issues like this.

To Friendship, I would like to express my thanks to all of my Thai and international friends, especially to Sukrit Sriratanawilai, Patawat Watakul, Varunyu Fuvattanasilp, Prom Kantuptim, Kundjanasith Thonglek, Pongsakorn Sihapitak, Kamolchanok Sarisuta, Norapat Nitaramorn, Setthawut Chotchaicharin, and Panyawut Sriiesaranusorn, for their kindness and supportive in every aspects. I feel joyful every time when we stay and travel together. Without them, my daily life could not be enjoyed like this.

Last but not least, I wish to express my highest gratitude to my dearest family for the support, raising and educating me with great care since my youth. No amount of words would sufficiently express my gratitude.

Finally, I would like to thank the Ministry of Education, Culture, Sports, Science and Technology (MEXT) Scholarship for the monetary support. This scholarship enables me to live and pursue my research in Japan comfortably. It is a huge honor to be a recipient of this scholarship.

References

[1] Daniel Kaplan and Leon Glass. *Understanding nonlinear dynamics*. Springer Science & Business Media, 1997.

[2] Chun-Wei Chang, Masayuki Ushio, and Chih-hao Hsieh. Empirical dynamic modeling for beginners. *Ecological Research*, 32(6):785–796, 2017.

[3] Edward L Ionides, Carles Bretó, and Aaron A King. Inference for nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 103(49):18438–18443, 2006.

[4] Hiroaki Natsukawa and Koji Koyamada. Visual analytics of brain effective connectivity using convergent cross mapping. In *SIGGRAPH Asia 2017 Symposium on Visualization*, pages 1–9, 2017.

[5] Florian Grziwotz, Jakob Friedrich Strauß, Chih-hao Hsieh, and Arndt Telschow. Empirical dynamic modelling identifies different responses of Aedes Polynesiensis Subpopulations to Natural Environmental Variables. *Scientific Reports*, 8(1):1–10, 2018.

[6] Jiayi Ma, Ming Yang, Xueshan Han, and Zhi Li. Ultra-short-term wind generation forecast based on multivariate empirical dynamic modeling. *IEEE Transactions on Industry Applications*, 54(2):1029–1038, 2017.

[7] Christian NK Anderson, Chih-hao Hsieh, Stuart A Sandin, Roger Hewitt, Anne Hollowed, John Beddington, Robert M May, and George Sugihara. Why fishing magnifies fluctuations in fish abundance. *Nature*, 452(7189):835–839, 2008.

[8] David Crow, Scott Graham, Brett Borghetti, and Patrick Sweeney. Engaging empirical dynamic modeling to detect intrusions in cyber-physical systems. In *International Conference on Critical Infrastructure Protection*, pages 111–133, 2020.

[9] Zhongyang Han, Jun Zhao, Henry Leung, King Fai Ma, and Wei Wang. A review of deep learning models for time series prediction. *IEEE Sensors Journal*, 21(6):7833–7848, 2019.

[10] Răzvan Gămănuţ, Henry Kennedy, Zoltán Toroczkai, Mária Ercsey-Ravasz, David C Van Essen, Kenneth Knoblauch, and Andreas Burkhalter. The mouse cortical connectome, characterized by an ultra-dense cortical graph, maintains specificity by distinct connectivity profiles. *Neuron*, 97(3):698–715, 2018.

[11] Ethan R Deyle and George Sugihara. Generalized theorems for nonlinear state space reconstruction. *PLoS One*, 6(3), 2011.

[12] George Sugihara, Robert May, Hao Ye, Chih-hao Hsieh, Ethan Deyle, Michael Fogarty, and Stephan Munch. Detecting causality in complex ecosystems. *Science*, 338(6106):496–500, 2012.

[13] Joseph Park, Gerald M Pao, Erik Saberski, Cameron Smith, Jason Haga, Ryousei Takano, Chen Min Yeh, Sreekanth Chalasani, and George Sugihara. Massively parallel empirical dynamic cross mapping. In *37th Meeting of the Pacific Rim Applications and Grid Middleware Assembly (PRAGMA37)*, 2019.

[14] Misha B Ahrens, Michael B Orger, Drew N Robson, Jennifer M Li, and Philipp J Keller. Whole-brain functional imaging at cellular resolution using light-sheet microscopy. *Nature Methods*, 10(5):413, 2013.

[15] Xiuye Chen, Yu Mu, Yu Hu, Aaron T Kuan, Maxim Nikitchenko, Owen Randlett, Alex B Chen, Jeffery P Gavornik, Haim Sompolinsky, Florian Engert, et al. Brain-wide organization of neuronal activity and convergent sensorimotor transformations in larval zebrafish. *Neuron*, 100(4):876–890, 2018.

[16] Adam Thomas Clark, Hao Ye, Forest Isbell, Ethan R Deyle, Jane Cowles, G David Tilman, and George Sugihara. Spatial convergent cross mapping to detect causal relationships from short time series. *Ecology*, 96(5):1174–1181, 2015.

[17] Hao Ye, Ethan R Deyle, Luis J Gilarranz, and George Sugihara. Distinguishing time-delayed causal interactions using convergent cross mapping. *Scientific Reports*, 5(14750):1–9, 2015.

[18] George Sugihara and Robert M May. Nonlinear forecasting as a way of distinguishing chaos from measurement error in time series. *Nature*, 344(6268):734–741, 1990.

[19] Floris Takens. Detecting strange attractors in turbulence. In *Dynamical systems and turbulence, Warwick 1980*, pages 366–381. Springer, 1981.

[20] Karin Schiecke, Britta Pester, Martha Feucht, Lutz Leistritz, and Herbert Witte. Convergent cross mapping: Basic concept, influence of estimation parameters and practical application. In *37th Annual International Conference of the Engineering in Medicine and Biology Society (EMBC)*, pages 7418–7421, 2015.

[21] Judea Pearl. Causal inference in statistics: An overview. *Statistics surveys*, 3:96–146, 2009.

[22] Clive WJ Granger. Investigating causal relations by econometric models and cross-spectral methods. *Econometrica: journal of the Econometric Society*, pages 424–438, 1969.

[23] Satohiro Tajima, Toru Yanagawa, Naotaka Fujii, and Taro Toyoizumi. Untangling brain-wide dynamics in consciousness by cross-embedding. *PLoS computational biology*, 11(11), 2015.

[24] George Sugihara. Nonlinear forecasting for the classification of natural time series. *Philosophical Transactions of the Royal Society of London. Series A: Physical and Engineering Sciences*, 348(1688):477–495, 1994.

[25] Chuan Luo, Xiaolong Zheng, and Daniel Zeng. Causal inference in social media using convergent cross mapping. In *2014 Joint Intelligence and Security Informatics Conference*, pages 260–263, 2014.

[26] cppEDM library. `https://github.com/SugiharaLab/cppEDM`.

[27] H Ye, A Clark, E Deyle, and G Sugihara. rEDM: an R package for empirical dynamic modeling and convergent cross-mapping. `https://github.com/SugiharaLab/rEDM`, 2016.

[28] pyEDM library. `https://github.com/SugiharaLab/pyEDM`.

[29] Mike Folk, Albert Cheng, and Kim Yates. HDF5: A file format and I/O library for high performance computing applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, volume 99, pages 5–33, 1999.

[30] BeeOND: BeeGFS On Demand. `https://www.beegfs.io/wiki/BeeOND`.

[31] James Malcolm, Pavan Yalamanchili, Chris McClanahan, Vishwanath Venugopalakrishnan, Krunal Patel, and John Melonakos. ArrayFire: a GPU acceleration platform. In *Modeling and Simulation for Defense Systems and Applications VII*, volume 8403, pages 49–56, 2012.

[32] ABCI official website. `https://abci.ai/`.

[33] TOP500 list, November 2019. `https://top500.org/lists/2019/11/`.

[34] Wei Chen, Jincai Chen, Fuhao Zou, Yuan-Fang Li, Ping Lu, and Wei Zhao. RobustiQ: A robust ANN search method for billion-scale similarity search on GPUs. In *Proceedings of the 2019 on International Conference on Multimedia Retrieval*, pages 132–140, 2019.

[35] Jia Pan and Dinesh Manocha. Fast GPU-based locality sensitive hashing for k-nearest neighbor computation. In *Proceedings of the 19th ACM SIGSPATIAL international conference on advances in geographic information systems*, pages 211–220, 2011.

[36] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using GPU. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–6, 2008.

[37] Mohammad Reza Abbasifard, Bijan Ghahremani, and Hassan Naderi. A survey on nearest neighbor search methods. *International Journal of Computer Applications*, 95(25):39–52, 2014.

[38] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software defined networks. In *10th USENIX*

Symposium on Networked Systems Design and Implementation (NSDI 13), pages 1–13, 2013.

[39] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[40] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, and Pravin Shelar. The design and implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, 2015.

[41] Lagopus switch and router. `http://www.lagopus.org`.

[42] Ryu SDN framework. `https://github.com/faucetsdn/ryu`.

[43] Faucet: Open source SDN controller for production networks. `https://faucet.nz`.

[44] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. ONOS: Towards an open, distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, pages 1–6, 2014.

[45] Open network operating system (ONOS). `https://www.opennetworking.org/onos/`.

[46] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. OpenDaylight: Towards a model-driven SDN controller architecture. In *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*, pages 1–6, 2014.

[47] Kazuya Suzuki, Kentaro Sonoda, Nobuyuki Tomizawa, Yutaka Yakuwa, Terutaka Uchida, Yuta Higuchi, Toshio Tonouchi, and Hideyuki Shimon-

ishi. A survey on openflow technologies. *IEICE Transactions on Communications*, 97(2):375–386, 2014.

[48] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Maziéres, and Nick McKeown. Where is the debugger for my software-defined network? In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 55–60, 2012.

[49] Niels L.M. Van Adrichem, Christian Doerr, and Fernando A. Kuipers. OpenNetMon: Network monitoring in OpenFlow software-defined networks. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–8, 2014.

[50] Renato B. Santos, Thiago R. Ribeiro, and Cecília de A. C. César. A network monitor and controller using only OpenFlow. In *2015 Latin American Network Operations and Management Symposium (LANOMS)*, pages 9–16, 2015.

[51] Pedro Heleno Isolani, Juliano Araujo Wickboldt, Cristiano Bonato Both, Juergen Rochol, and Lisandro Zambenedetti Granville. Interactive monitoring, visualization, and configuration of OpenFlow-based SDN. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 207–215, 2015.

[52] Karanpreet Singh, Paramvir Singh, and Krishan Kumar. Application layer HTTP-GET flood DDoS attacks: Research landscape and challenges. *Computers & security*, 65:344–372, 2017.

[53] Watanakeesuntorn Wassapon, Putchong Uthayopas, Chantana Chantrapornchai, and Kohei Ichikawa. Real-time monitoring and visualization software for openflow network. In *2017 15th International Conference on ICT and Knowledge Engineering (ICT&KE)*, pages 1–5, 2017.

[54] JD Case, Mark Fedor, Martin Lee Schoffstall, and James Davin. RFC1157: Simple network management protocol (SNMP), 1990.

[55] Peter Phaal, Sonia Panchen, and Neil McKee. RFC3176: InMon Corporation's sFlow: A method for monitoring traffic in switched and routed networks, 2001.

[56] Sukhveer Kaur, Japinder Singh, and Navtej Singh Ghumman. Network programmability using POX controller. In *ICCCS International Conference on Communication, Computing & Systems*, volume 138, pages 134–138, 2014.

[57] Sayyaf Haider Warraich, Zeeshan Aziz, Hasnat Khurshid, Rashid Hameed, Abdul Saboor, and Muhammad Awais. SDN enabled and OpenFlow compatible network performance monitoring system. *arXiv preprint arXiv:2005.07765*, 2020.

[58] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. FlowVisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep*, 1:132, 2009.

[59] Hiroaki Yamanaka, Eiji Kawai, Shuji Ishii, and Shinji Shimojo. AutoVFlow: Autonomous virtualization for wide-area openflow networks. In *2014 third European workshop on software defined networks*, pages 67–72, 2014.

[60] Aditya Nur Cahyo, Risanuri Hidayat, and Dani Adhipta. Performance comparison of intrusion detection system based anomaly detection using artificial neural network and support vector machine. In *AIP Conference Proceedings*, volume 1755, pages 1–7, 2016.

[61] Nitin Aji Bhaskar. Performance analysis of support vector machine and neural networks in detection of myocardial infarction. *Procedia Computer Science*, 46:20–30, 2015.

[62] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):27, 2011.

[63] Tuan A Tang, Lotfi Mhamdi, Des McLernon, Syed Ali Raza Zaidi, and Mounir Ghogho. Deep learning approach for network intrusion detection in

software-defined networking. In *2016 International Conference on Wireless Networks and Mobile Communications (WINCOM)*, pages 258–263, 2016.

[64] Quamar Niyaz, Weiqing Sun, and Ahmad Y Javaid. A deep learning based DDoS detection system in software-defined networking (SDN). *arXiv preprint arXiv:1611.07400*, 2016.

[65] Sigurour Pall Behrend. *Design, implementation, and optimization of an advanced I/O Framework for Parallel Support Vector Machines.* PhD thesis, University of Iceland, 2018.

[66] Zeyi Wen, Jiashuai Shi, Qinbin Li, Bingsheng He, and Jian Chen. ThunderSVM: A fast SVM library on GPUs and CPUs. *The Journal of Machine Learning Research*, 19(1):797–801, 2018.

[67] Vassili Kovalev, Alexander Kalinovsky, and Sergey Kovalev. Deep learning with Theano, Torch, Caffe, TensorFlow, and Deeplearning4J: Which one is the best in speed and accuracy? *2016 Pattern Recognition and Information Processing (PRIP 2016)*, pages 99–103, 2016.

[68] Theano GitHub repository. `https://github.com/Theano/Theano`.

[69] Nour Moustaf and Jill Slay. Creating novel features to anomaly network detection using DARPA-2009 data set. In *14th European Conference on Cyber Warfare and Security*, pages 204–212, 2015.

[70] Kubra Kalkan, Gurkan Gur, and Fatih Alagoz. Defense mechanisms against DDoS attacks in SDN environment. *IEEE Communications Magazine*, 55(9):175–179, 2017.

[71] RT Kokila, S. Thamarai Selvi, and Kannan Govindarajan. DDoS detection and analysis in SDN-based environment using support vector machine classifier. In *2014 Sixth International Conference on Advanced Computing (ICoAC)*, pages 205–210, 2014.

[72] DARPA intrusion detection data sets. `https://archive.ll.mit.edu/ideval/data/`.

[73] Javed Ashraf and Seemab Latif. Handling intrusion and DDoS attacks in software-defined networks using machine learning techniques. In *2014 National Software Engineering Conference*, pages 55–60, 2014.

[74] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[75] Keras: the Python deep learning API. `https://keras.io`.

[76] DARPA Scalable Network Monitoring (SNM) Program Traffic, IMPACT ID: USC-LANDER/DARPA_Scalable_Network_Monitoring-20091103/rev8431. Traces taken 2009-11-03 to 2009-11-12. Provided by the USC/LANDER project. `http://www.isi.edu/ant/lander`.

[77] Manaf Gharaibeh and Christos Papadopoulos. DARPA-2009 intrusion detection dataset report. *Technical Report*, 2014.

[78] DARPA Scalable Network Monitoring (SNM) Program Traffic, IMPACT ID: USC-LANDER/DARPA_2009_DDoS_attack-20091105/rev4383. Traces taken 2009-11-05 to 2009-11-05. Traces taken 2009-11-05 to 2009-11-05. Provided by the USCLANDER project. `http://www.isi.edu/ant/lander`.

[79] Thorsten Joachims. SVMlight: Support vector machine. `http://svmlight.joachims.org/`, 1999.

[80] Rogério Leão Santos De Oliveira, Christiane Marie Schweitzer, Ailton Akira Shinoda, and Ligia Rodrigues Prete. Using Mininet for emulation and prototyping software-defined networks. In *2014 IEEE Colombian Conference on Communications and Computing (COLCOM)*, pages 1–6, 2014.

[81] Kohei Ichikawa, Pongsakorn U-Chupala, Che Huang, Chawanat Nakasan, Te-Lung Liu, Jo-Yu Chang, Li-Chi Ku, Whey-Fone Tsai, Jason Haga, and Hiroaki Yamanaka. PRAGMA-ENT: An international SDN testbed for cyberinfrastructure in the Pacific Rim. *Concurrency and Computation: Practice and Experience*, 29(13):e4138, 2017.

[82] Routing switch. `https://github.com/trema/apps/tree/master/routing_switch`.

[83] Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W Moore. OFLOPS: An open framework for OpenFlow switch evaluation. In *International Conference on Passive and Active Network Measurement*, pages 85–95, 2012.

[84] Cbench: A benchmarking tool for OpenFlow controller. `https://github.com/mininet/oflops/tree/master/cbench`.

[85] Stanford OpenFlow 1.0 reference switch/controller. `https://github.com/mininet/openflow`.

[86] Tzu-Tsung Wong. Performance evaluation of classification algorithms by k-fold and leave-one-out cross validation. *Pattern Recognition*, 48(9):2839–2846, 2015.

[87] David Martin Powers. Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. *International Journal of Machine Learning Technology*, 2(1):37–63, 2011.

[88] Keichi Takahashi, Wassapon Watanakeesuntorn, Kohei Ichikawa, Joseph Park, Ryousei Takano, Jason Haga, George Sugihara, and Gerald M Pao. kedm: A performance-portable implementation of empirical dynamic modeling using kokkos. In *Practice and Experience in Advanced Research Computing*, pages 1–8. 2021.

[89] ArrayFire library. `https://arrayfire.com/`.

[90] kEDM documentation. `https://kedm.readthedocs.io/en/latest/index.html/`.

[91] Thanasis Vafeiadis, Alexandros Papanikolaou, Christos Ilioudis, and Stefanos Charchalakis. Real-time network data analysis using time series models. *Simulation Modelling Practice and Theory*, 29:173–180, 2012.

[92] Yuichi Uchiyama, Yuji Waizumi, Nei Kato, and Yoshiaki Nemoto. Detecting and tracing DDoS attacks in the traffic analysis using auto regressive model. *IEICE Transactions on Information and Systems*, 87(12):2635–2643, 2004.

[93] Mbulelo Brenwen Ntlangu and Alireza Baghai-Wadji. Modelling network traffic using time series analysis: A review. In *Proceedings of the International Conference on Big Data and Internet of Thing*, pages 209–215, 2017.

[94] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[95] Jinyu Li, Abdelrahman Mohamed, Geoffrey Zweig, and Yifan Gong. LSTM time and frequency recurrence for automatic speech recognition. In *2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*, pages 187–191, 2015.

[96] Kundjanasith Thonglek, Kohei Ichikawa, Keichi Takahashi, Hajimu Iida, and Chawanat Nakasan. Improving resource utilization in data centers using an LSTM-based prediction model. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–8, 2019.

[97] Chunting Zhou, Chonglin Sun, Zhiyuan Liu, and Francis Lau. A C-LSTM neural network for text classification. *arXiv preprint arXiv:1511.08630*, 2015.

[98] Lazy Predict documentation. `https://lazypredict.readthedocs.io/en/latest/`.

List of Publication

[1] Wassapon Watanakeesuntorn, Keichi Takahashi, Chawanat Nakasan, Kohei Ichikawa, and Hajimu Iida. Opimon: A transparent, low-overhead monitoring system for OpenFlow networks. *IEICE Transactions on Communications*, E105.B(4):485–493, 2022.

[2] Keichi Takahashi, Wassapon Watanakeesuntorn, Kohei Ichikawa, Joseph Park, Ryousei Takano, Jason Haga, George Sugihara, and Gerald M Pao. kEDM: A performance-portable implementation of empirical dynamic modeling using Kokkos. In *2021 ACM Practice and Experience in Advanced Research Computing (PEARC)*, pages 1–8, 2021.

[3] Wassapon Watanakeesuntorn, Keichi Takahashi, Kohei Ichikawa, Joseph Park, George Sugihara, Ryousei Takano, Jason Haga, and Gerald M Pao. Massively parallel causal inference of whole brain dynamics at single neuron resolution. In *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 196–205, 2020.

[4] Panida Khuphiran, Pattara Leelaprute, Putchong Uthayopas, Kohei Ichikawa, and Wassapon Watanakeesuntorn. Performance comparison of machine learning models for DDoS attacks detection. In *2018 IEEE 22nd International Computer Science and Engineering Conference (ICSEC)*, pages 1–4. IEEE, 2018.

[5] Wassapon Watanakeesuntorn, Putchong Uthayopas, Chantana Chantrapornchai, and Kohei Ichikawa. Real-time monitoring and visualization software for Open-Flow network. In *2017 15th International Conference on ICT and Knowledge Engineering (ICT&KE), Bangkok, Thailand*, pages 1–5, 2017.