Development of Lightweight and Accurate Intrusion Detection on Programmable Data Plane

Thi-Nga Dao, Van-Phuc Hoang, Chi Hieu Ta, Van Son Vu
Le Quy Don Technical University
236 Hoang Quoc Viet, Hanoi, Vietnam
daothinga.mta@gmail.com

***Abstract*—With the aim of developing a lightweight yet accurate network security method for Internet of Things, this paper presents the neural-network-based intrusion detection model that incorporates a parameter trimming method. The intrusion detection and classification function is implemented on programmable data plane, thus significantly reducing the detection time. Moreover, by using the neuron pruning approach, the proposed architecture requires a much lower delay for traffic classification with a slight reduction in classification accuracy. We conduct experiments using a P4 programming language and the collected results show that the pruned intrusion detection model with low model complexity is more feasible for edge devices with constrained computing and memory resources than the fully-connected model.**

***Index Terms*—Intrusion detection, programmable data plane**

I. Introduction

With the recent exponential increase in the number of connected Internet of Things (IoT) devices, network security becomes an integral part of IoT networks. More specifically, the IoT traffic should be monitored and classified into different traffic classes (e.g., normal or an attack type). Then, the networking devices (i.e., switches) take an appropriate action based on the output of the traffic classification model. There exist two contrasting requirements for the network security system: high accuracy and low detection delay. In this work, we propose an intrusion detection model that can achieve both the above-mentioned demands.

To produce the low detection latency, the classification model is usually implemented on edge devices (e.g., switches) instead of sending traffic to external devices for classification. One big difficulty of embedding the classification function on edge devices is that the classification model should have a lightweight architecture with low model complexity since the edge devices are usually equipped with limited computing and memory resources. Note that high accurate classification models are constructed based on advanced machine learning techniques (e.g., neural network) with high model complexity. Therefore, a simplification method is needed to reduce the model complexity of the classification models.

There are multiple simplification methods applied for the neural network: neuron pruning [1]–[5], memory reduction [6]–[8], operations simplification [9]–[11]. Neuron pruning or parameter trimming usually consists of three steps: training the whole fully-connected network from the scratch, removing the least important connections from the network, fine-tuning the pruned network. Memory reduction can be achieved by sharing the same memory buffer among different weights, thus allowing less memory to store network parameters. Meanwhile, operations simplification can be done by converting the complex to simpler operations (e.g., from float to integer or binary operations). In this paper, we apply the simple neuron pruning method [2] to reduce the complexity of the classification model, thus making it suitable for edge devices with constrained computing resources.

We first present how to construct the pruned model using the three above-mentioned steps for intrusion detection and classification that can be implemented on a programmable switch. Note that the programming language (i.e., P4) for data plane only supports a limited set of arithmetic operations. Therefore, a suitable neural network structure should be selected. More particularly, we use the ReLU activation function for the hidden layer due to its simplicity. In addition, network parameters such as weights and biases are converted to integer numbers for integer operations since P4 does not consider floating-point operations.

Then, the implementation of the proposed model on programmable data plane is described in detail. More specifically, we explain the network topology used to evaluate the performance of the proposed architecture. Next, we present the structure of programmable switches with four main blocks: parser, ingress control, outgress control, and deparser. The position of the classification function is also stated. We also demonstrate the flow of the intrusion detection model on the P4 language.

The main contributions of our work is summarized as follows.

- We construct a lightweight intrusion detection architecture based on a network simplification method, i.e., neuron pruning.
- The proposed method is evaluated with different pruning rates and compared with the fully-connected architecture in terms of classification accuracy and detection delay on programmable switches.
- We present how to implement the network detection and classification function on programmable data plane.

This research is funded by Vietnam National Foundation for Science and Technology Development (NAFOSTED) under grant number 102.02-2020.06.

The rest of the paper is organized as follows. Section II presents the detail architecture of the NN-based intrusion detection pruned model. Then, the implementation of the detection function on programmable switches is described in Section III. Section IV shows the performance evaluation of the proposed model with a variety of network parameters. Finally, we conclude our work in Section V.

II. A Neural-Network-based Intrusion Detection Model With Neuron Pruning

This paper aims to design a timely and lightweight network intrusion detection model that can be suitable for programmable networking devices with a limited computing resource. Recently, neural networks (NNs) have emerged as an advanced machine learning technique with high accuracy to learn a non-linear mapping from input features to output values. However, NNs suffer from the high model complexity, which leads to high detection delay.

To address the issue of large detection latency, we apply a neuron pruning technique that trims unnecessary connections of the model and only keeps salient weights. The architecture of the pruning-based network intrusion classification model is shown in Figure 1. For simplicity, Figure 1 shows an example where the number of output unit is 1. In fact, the output layer consists of n_y units where n_y is the number of data classes. The architecture consists of three layers: input, hidden, and output. The ReLU activation function is used for the hidden layer since it can be easily implemented using the P4 programming language. The sigmoid and softmax functions are considered for the detection and classification models, respectively. In this paper, since we focus on the classification problem, the softmax function is used at the output layer.

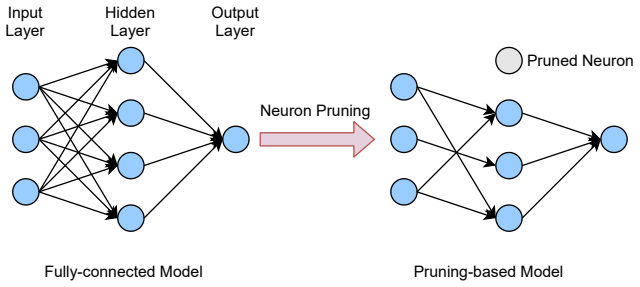

Fig. 1. The pruning-based intrusion detection architecture

There are three steps for the training procedure: learning the fully-connected model, pruning unimportant connections, re-training the pruned network. In the first phase, parameters including weights and biases are trained by minimizing the entropy-based loss function L as follows.

$$L = -\frac{1}{m}\sum_{j=1}^{m}\sum_{i=1}^{n_y} t_i^{(j)} \log(y_i^{(j)}) \tag{1}$$

where m and n_y are the number of samples and the number of data classes, respectively, while $t_i^{(j)}$ and $\log(y_i^{(j)})$ denote the i^{th} true label and predicted output of the j^{th} sample.

In the next phase, the trained weights with the least minimum absolute values are removed from the network since the less value of weight means less important for the network. We define the pruning rate p_{prune} ($0 \leq p_{prune} \leq 1$) as the ratio of the number of removed connections to the total number of connections of the fully connected layer. The percentile p_w of the absolute weight values is calculated. Then, if a weight with the absolute value is greater than p_w, we keep this connection. Otherwise, the connection is trimmed from the network. We use a binary mask matrix M with the same size as the weight matrix to denote the pruning status of weights. For example, for the connection from the input to the hidden layer in Fig. 1, the binary mask is represented as $M = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$. Since the first neuron of the hidden layer is removed, the first column of M is set to 0.

In the final phase, the remaining connections of the pruned network are re-trained. We can call this step is fine-tuning. The entropy-based loss function is still used for re-training. The difference with the 1st phase is that we multiple the connections with the binary mask matrix so that the pruned weights are not trained in this phase.

Note that since the P4 language only supports integer operations, we need to convert the trained network parameters into integer values. Assume that k bits are used to represent the fractional part of parameters. Let W and b denote the trained weight and bias float values, respectively. We define X and h as input features and hidden units. Then, the ReLU hidden units h_{int} in the integer format are derived as below.

$$W_{int} = \text{int}(W \times 2^k) \tag{2}$$

$$X_{int} = \text{int}(X \times 2^k) \tag{3}$$

$$b_{int} = \text{int}(b \times 2^{2k}) \tag{4}$$

$$h_{int} = \begin{cases} (W_{int}X_{int} + b_{int})//2^k, & \text{if } W_{int}X_{int} + b_{int} \geq 0 \\ 0, & \text{otherwise.} \end{cases} \tag{5}$$

After doing the division ($//$), we get the integer part of the output of the division.

After applying the neuron pruning method, the model complexity can be considerably reduced. We now compare the number of floating-point operations (FLOPS) between the fully-connected and pruned models in the case of a hidden layer. We define n_x and n_h as the number of input and hidden units, respectively. Then, the number of FLOPS can be reduced by $(n_x n_h + n_h n_y)p_{prune}$. In our case, the number of input features and output units are 6 and 5, respectively. If the number of hidden features is 20 and $p_{prune} = 0.5$, the

proposed pruning-based architecture can reduce 110 multiplication operations compared to the fully-connected model.

After fine-training the pruned network, the parameters are sent to programmable switches. Each switch computes the output values y that represent the probability of traffic classes for an incoming packet. Then, the packet is classified into label with argmax(y). Depending on the classified label, we can take different actions for this packet. For example, the packet can be forwarded normally or dropped at the switch.

III. Implementation of The Intrusion Detection Function on Programmable Data Plane

In this section, we describe how to implement the intrusion detection function on programmable data plane. Figure 2 shows the network configuration with two hosts (i.e., h1 and h2) and a programmable switch that connects these hosts. The host h1 is a sender that generates data traffic following a given trace and then sends the packets to the host h2 via the switch. We use the dataset [12] for data generation. Upon packet arrival, the switch analyzes the packet header and extracts input features. Then, the switch selects an appropriate action based on the probability of traffic classes. For our simulation, we assume that all packets are forwarded to the host h2 to measure the detection delay at the switch. The Mininet [13] network simulation is used to set up the network topology including hosts and links connecting hosts. Note that the link delay is ignored and the link bandwidth is set to 30 Mbps. For packet generation and reception, the Python-based scapy library is used.

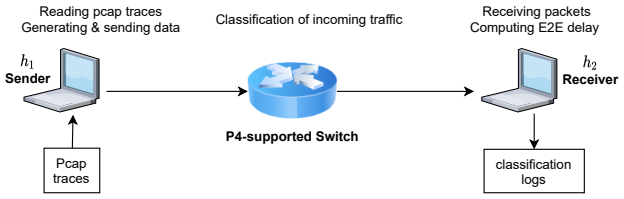

Fig. 2. The network topology for performance evaluation

Figure 3 shows a common architecture of the p4-supported switch consisting of four main programmable blocks: parser, ingress control, outgress control, and deparser. When an incoming packet arrives at the switch, the Ethernet, IP, and TCP headers are analyzed to extract necessary information (e.g., source and destination IP addresses, source and destination port numbers, and arriving time). Then, the packet is processed at the incoming and outgoing ports by ingress and outgress control blocks, respectively. Look-up tables are used to find appropriate actions (e.g., sending data to a specific port and updating the time-to-live parameter) for the given packet. The switch can select one of the possible actions including forwarding, dropping packets, and adding the alarm field to the packet header.

To embed the intrusion detection function on the programmable switch, we use the P4 programming language.

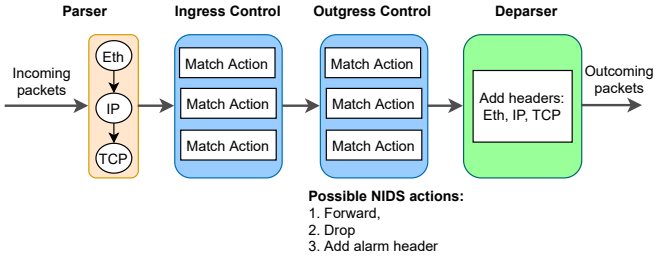

Fig. 3. The implementation of intrusion detection on the programmable switch

Note that this language only supports a limited number of operations on binary and integer numbers. More specifically, arithmetic operations including addition, subtraction, and multiplication are supported while no division/modulo operation can be used. Bit shift and element-wise comparison operations are also supported. We apply the right bit shift instead of division operation to control the number of bits used for units in the network.

To add a customized packet processing function on programmable data plane, we can edit the ingress or outgress control blocks. Figure 4 shows the main components of the NN-based intrusion detection model implemented in the outgress control. We first declare variables for hidden units (e.g., h_0 and h_1) using 32-bit signed binary numbers. Then, hidden units are computed based on input features and network parameters (weights and biases). We implement the ReLU activation function by comparing the most significant bit of hidden units with 1. For example, if h_0 is a negative number, then $h_0 = 0$. Otherwise, the value of h_0 is divided to 2^{10} since we use 10 bits for the fractional number of weight and bias parameters. Finally, the units at the output layer are derived using the hidden units. The traffic is classified into the label with the highest output value.

```
control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata) {
    apply {
        // Declare variables for hidden units
        bit <32> h0;
        bit <32> h1;

        // Compute hidden units based on input features
        h0 = x0*285 - x1*8 - x2*60 - x3*3 + 20;
        h1 = -x0*6 + x1*2 - x2*5 + x3*9 - 45;

        // Implement ReLU
        if (h0[31:31] == 0x01) {h0 = 0;}
        else {
            h0 = h0>>10;
        }

        // Compute output unit
        y0 =  h0*36 + h1*64 - h2*309 + 15;
    }
}
```

Fig. 4. Flow of the NN-based intrusion detection model

IV. Experimental Results

The experiments are implemented in a desktop PC with Intel Core i7 2.5GHz CPU (with the Radeon R9 M370X 2048 MB and Intel Iris Pro 1536 MB GPU support) and 16 GB RAM. The dataset [12] is used to evaluate the proposed architecture. More specifically, we divide the whole dataset into training and test sets with a ratio of 7:3. The parameters are trained using the training set while the evaluation performance is collected on the test set only. There are five different traffic classes: normal, reconnaissance, man-in-the-middle, denial-of-service, and botnet. We select 6 salient input features from the set of features provided in [14]. To derive input features of the classification model, whenever a packet arrives at the switch, the source and destination IP addresses are extracted from the packet header. The selected features include weight, mean, and variance of packet length with the flow id is the source IP address and weight, mean, and variance of the elapsed period between two consecutive arriving times with the flow id is the source and destination IP addresses. Note that these features are computed statistically by switches and presented using a 32-bit binary number.

Figure 5 shows the learning curves of the training process when the pruning rate is set to 0.6 and there is a hidden layer of 10 units. In the first phase, while training the fully-connected (FC) model, the loss value on training and validation sets is reduced over time. When there is no improvement in classification accuracy on the validation set for the most recent 20 epochs, the training procedure of phase 1 stops. Then, the least important trained weights with the smallest absolute values are pruned in the second phase. Lastly, we re-trains the pruned model by fine-tuning the remaining parameters to minimize the loss function. During this phase, the loss value gradually gets smaller. Similar to phase 1, we terminate the re-training procedure when the classification accuracy is not improved for the last 20 epochs. After phase 3, the traffic classification accuracy of the pruned model (around 94%) is slightly lower than that of the FC model (around 94.3%). Since we remove 60% of connections in the FC model, the loss function of the pruned model is considerably higher than that of the FC architecture.

TABLE I
Effects of the number of bits for parameters

Number of bits	Accuracy (%)	Precision	Recall
4	75.43	0.825	0.866
6	83.03	0.839	0.855
8	90.79	0.875	0.907
10	92.83	0.899	0.932
12	93.15	0.902	0.936
16	93.19	0.903	0.937
24	93.19	0.903	0.937

We now investigate the effects of the number of bits k used to present the fractional number of weights. For example, if weight is $w = 1.25$ and we use 4 bits for the fractional part, then the weight value after rounding is $1.25 \times 2^4 = 20$. The pruning rate is set to 0.6 and there are 10 hidden units. Table I shows the performance of the pruned model including the classification accuracy, precision, and recall with different k values from 4 to 24. Note that precision and recall are computed for each label and the weighted average values are presented in the table. If k increases, the performance of the pruned model is improved significantly and starts to saturate when k is greater than 10. Therefore, we use $k = 10$ bits as the default value. In cases of $k \geq 12$, the classification accuracy, precision, and recall become stable at around 93.19%, 0.903, and 0.937, respectively.

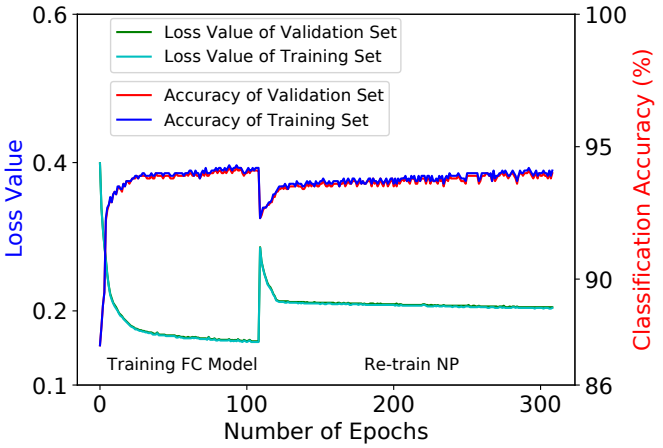

Fig. 5. Learning curves of the pruning-based network with the pruning rate 0.6

TABLE II
Effects of the number of hidden units on performance of the pruned model

Number of hidden units	Accuracy (%)	Precision	Recall
5	92.99	0.880	0.917
7	93.33	0.881	0.913
10	93.34	0.899	0.932
20	93.42	0.903	0.935
30	93.93	0.927	0.937

We also evaluate the impacts of the number of hidden units on the classification accuracy, precision, and recall of the pruned model as shown in Table II. The number of hidden units varies from 5 to 30 while the pruning rate is 0.6. For example, with 10 hidden units, accuracy, precision, and recall are 93.34%, 0.899 and 0.932, respectively. That means the classification model achieves 93.34% accuracy for the whole samples in the test set, 89.9% accuracy for samples that are predicted as positive and 93.2% accuracy for samples with actual positive labels. Generally, the performance of the proposed model is slightly improved with the increase in the number of hidden units. Since the model complexity and the detection latency become larger when using more hidden

units, we select 10 hidden units as the default value for other experiments.

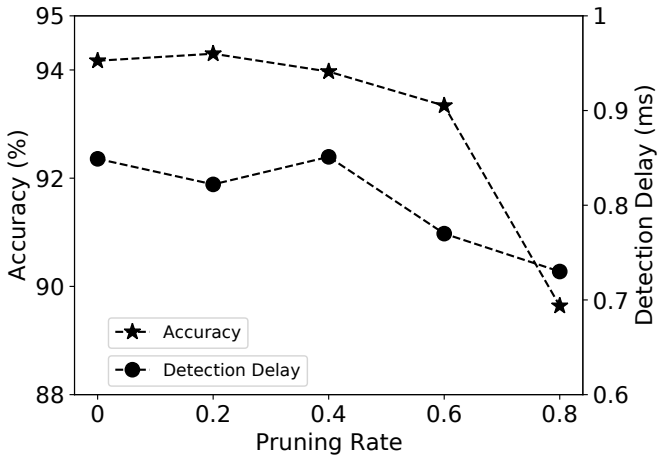

Fig. 6. The effects of pruning rate on performance of the pruned model

To evaluate the impacts of pruning rate p_{prune} on the performance of the pruned model, we quantify the classification accuracy and detection delay with different p_{prune} values from 0 to 0.8. The number of bits for the fraction part of parameters is set to 10. The FC model is considered as an existing model with $p_{prune} = 0$. The pruned model is trained using five different learning rates (i.e., 0.001, 0.005, 0.01, 0.03, 0.05) and the parameters are selected when the highest performance on the validation set is recorded. As shown in Figure 6, the pruned model and the FC model have similar performance when $p_{prune} < 0.4$. This is because there are enough connections to accurately classify traffic in cases of a low pruning rate. When we increase the pruning rate, the accuracy keeps decreasing gradually. For example, when $p_{prune} = 0.8$, accuracy is reduced by 4.3% compared to the pruning rate of 0.4. Meanwhile, thanks to the reduction of the number of parameters, the detection delay can be improved from 0.8511 to 0.73 (ms) (i.e., a reduction of 14.2%).

From Figure 6, we include that using the simple neuron pruning method can achieve significantly lower detection latency than the FC model with a small sacrifice in classification accuracy. By reducing the model complexity, the pruned model of intrusion detection becomes relevant to programmable switches with constrained computing resources. By measuring the detection delay of the pruned model, we can infer the maximum data rate that can be classified by a switch. For example, if the detection delay is 0.8 (ms) per packet, the switch can classify up to $\frac{1}{0.8} \times 1000 = 1,250$ packets per second. If each packet has 1,000 bytes, the maximum data rate is around 10 Mb/s. Note that the measured detection delay is based on the desktop computer and the real delay should greatly depend on the hardware configuration of the programmable switch. Therefore, the real maximum data rate that can be processed by a switch can be much larger than 10 Mb/s. In our simulation, we aim to compare the detection latency of the pruned model with different parameters and the collected performance can be used to understand the impacts of network parameters on the pruned classification architecture as well as the performance of fully-connected model.

V. Conclusion

This paper aims to achieve the balance between low detection delay and high accuracy for the intrusion detection function by incorporating the neural network detection model with the neuron pruning method. Using a high pruning rate value, the parameter trimming approach can reduce a considerably large number of network connections, thus leading to low model complexity and making the pruned model suitable for programmable data plane. By reducing the detection delay, the proposed architecture allows the network to manage more data traffic, which is an important requirement of the recent Internet of Things.

References

[1] T.-N. Dao and H. J. Lee, "Stacked autoencoder-based probabilistic feature extraction for on-device network intrusion detection," *IEEE Internet of Things Journal*, 2021.

[2] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[3] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient transfer learning," *CoRR*, vol. abs/1611.06440, 2016.

[4] R. Yu, A. Li, C. Chen, J. Lai, V. I. Morariu, X. Han, M. Gao, C. Lin, and L. S. Davis, "NISP: pruning networks using neuron importance score propagation," *CoRR*, vol. abs/1711.05908, 2017.

[5] B. Hassibi and D. G. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," in *Advances in Neural Information Processing Systems 5*, S. J. Hanson, J. D. Cowan, and C. L. Giles, Eds. Morgan-Kaufmann, 1993, pp. 164–171.

[6] S. Wiedemann, K.-R. Müller, and W. Samek, "Compact and computationally efficient representation of deep neural networks," *IEEE transactions on neural networks and learning systems*, vol. 31, no. 3, pp. 772–785, 2019.

[7] K. Shirahata, Y. Tomita, and A. Ike, "Memory reduction method for deep neural network training," in *2016 IEEE 26th International Workshop on Machine Learning for Signal Processing (MLSP)*. IEEE, 2016, pp. 1–6.

[8] Y. Pisarchyk and J. Lee, "Efficient memory management for deep neural net inference," *arXiv preprint arXiv:2001.03288*, 2020.

[9] M. Rusci, L. Cavigelli, and L. Benini, "Design automation for binarized neural networks: A quantum leap opportunity?" in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2018, pp. 1–5.

[10] T. Simons and D.-J. Lee, "A review of binarized neural networks," *Electronics*, vol. 8, no. 6, p. 661, 2019.

[11] H. Qin, R. Gong, X. Liu, X. Bai, J. Song, and N. Sebe, "Binary neural networks: A survey," *Pattern Recognition*, vol. 105, p. 107281, 2020.

[12] K. Hyunjae, A. Dong Hyun, L. Gyung Min, Y. Jeong Do, P. Kyung Ho, and K. Huy Kang, "Iot network intrusion dataset," 2019.

[13] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: Association for Computing Machinery, 2010.

[14] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai, "Kitsune: An ensemble of autoencoders for online network intrusion detection," in *Network and Distributed Systems Security (NDSS) Symposium 2018*, 2018.