Efficient Attack Detection with Multi-Latency Neural Models on Heterogeneous Network Devices

Guorui Xie[1,2], Qing Li[2], Haolin Yan[1,3], Dan Zhao[2], Gianni Antichi[4,5], Yong Jiang[1,2]
[1] International Graduate School, Tsinghua University, Shenzhen, China [2] Peng Cheng Laboratory, Shenzhen, China
[3] Xidian University, Xi'an, China [4] Politecnico di Milano, Milan, Italy [5] Queen Mary University of London, UK

2023 IEEE 31st International Conference on Network Protocols (ICNP) | 979-8-3503-0322-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICNP59255.2023.10355579

***Abstract*—To achieve fast and accurate attack detection, some works manually tailor neural networks (NNs) for deployment on CPUs of gateways, routers, or even programmable switches. However, with such solutions, NNs must be custom-tailored across different devices to meet the heterogeneous settings (e.g., OS and CPU types). Even worse, a model may require frequent adjustments to adapt to the same device's varying traffic rates. In this paper, we present SOTERIA, an automated multi-latency NN generation and scheduling system for fast and accurate detection against fluctuating traffic rates across heterogeneous hardware. SOTERIA first uses an evolutionary training algorithm to evolve the Pareto front, i.e., the set of NNs with a good spread on accuracy and model size. Then, for each device, SOTERIA filters the optimal multi-latency NNs by non-dominating sorting on the NNs' test latency on the device. Finally, to cope with the dynamic traffic rate, we design a heuristic scheduling scheme that adaptively selects NNs to maintain a balance between the detection accuracy and latency.**

I. INTRODUCTION

Defending the network against attacks is a fundamental topic [1], [2]. Currently, a prevalent way is to regard attack detection as a classification task and use powerful neural networks (NNs) to identify the malicious traffic [3]–[6]. For example, [4] takes the statistical features of each flow as input and proposes a feed-forward NN to accurately find attack flows (e.g., DoS) in the traffic. However, a drawback of NNs is that their intensive and sophisticated computation should be accelerated by strong GPUs. Many simple gateways/routers that conduct attack detection to secure the network only host inexpensive CPUs, which hinder the practical applications of NNs. To help NNs run on simple devices with acceptable latencies, researchers have manually adapted NNs to meet the device characteristics [7], [8]. The authors in [7] design an ensemble model of lightweight autoencoders for low-latency detection on a Raspberry PI-based router. In [8], the widely used convolution neural operation is changed to a more efficient branch way for CPUs of programmable switches. Notably, programmable switches usually handle traffic of Tbps with ASICs [9]. Thus, only a small fraction of the traffic (e.g., the suspicious one) is filtered by rules in the ASIC and then processed by the switch CPU in [8].

Nonetheless, we argue that tailoring a specific NN through artificial expertise is not robust, nor scalable. First, the network consists of heterogeneous network devices that differ in settings like operating systems and CPU types. It is impractical to run a single NN with a consistent and satisfactory latency on all devices. Though one can manually design a customized NN for each device for aligned low latency, such a human-involved process is time-consuming and labor-intensive [10]. Second, even for the same device, the fluctuation in traffic rate over time can also lead to changes in NN's latency. For example, when the traffic rate becomes bursty [11], the NN may fail to process the traffic in time, causing delayed attack detection, reaction, and consequently financial loss.

In this paper, we present SOTERIA[1] to tackle these problems. SOTERIA automatically generates NNs by Neural Architecture Search (NAS) [12], [13], which requires little human involvement. Given heterogeneous devices, SOTERIA recommends NNs of suitable latencies for the deployment and enables elastic scheduling among the deployed NNs, guaranteeing timely and accurate detection against the fluctuating traffic rate. To achieve the above design system, the following challenges need to be addressed: **1)** The NAS efficiency needs to be improved. Previous NAS solutions involve repeatedly training and evaluating NNs for each device [14], [15], consuming huge resources (e.g., 3150 GPU days in [16]). Although the current weight-sharing (i.e., sharing and embedding parameters of NNs into a supernet for training [13], [15]) accelerates the training, it still lacks efficiency: With an exponential search space (aka the set of candidate NNs), only a small fraction of trained NNs are chosen after the evaluation, while most NNs are trained in vain, wasting expensive GPU memory and hours. Besides, training many low-performance NNs together can affect the accuracy of the chosen NNs due to their heavily shared parameters [17]. **2)** A dedicated recommendation scheme is required to choose NNs that suit the latency requirement of each particular device from the trained NNs by NAS. **3)** To schedule NNs within a device, an adaptive detection process is desired, which should intelligently measure the detection overhead caused by the changing traffic rate and then selects a proper NN to maintain a balance between detection latency and accuracy.

We address these challenges by the following key ideas:

- We propose an Evolutionary Training Algorithm (ETA) to progressively train sub-NNs (i.e., NNs in the weight-

Corresponding author: Qing Li (liq@pcl.ac.cn)

[1]In Greek mythology, Soteria is the goddess of preservation and safety.

shared supernet) in the Pareto front [18], [19]. The Pareto front is the set of optimized sub-NNs that has a good spread on both accuracy and model size. Since it is impractical to measure per-device latency in time during the training, we use the latency-related model size instead. In each training iteration, sub-NNs of high accuracy and diverse model sizes are selected. Then, evolutionary crossover and mutation are applied to them, deriving a new generation of sub-NNs for the next evolutionary training. In this way, ETA eliminates the weight-sharing of redundant sub-NNs outside the accuracy-size Pareto front and thus improves the training efficiency.

- We re-sort the trained NNs in ETA to obtain the actual accuracy-latency Pareto front for the device. As the model size can roughly but not exactly reflects the running latency on different devices [20], for a deployment device, we first run tests of latency on it. Then, guided by the objectives of detection accuracy and running latency, the non-dominated sorting filters a new accuracy-latency Pareto front for the device from the tested NNs. The accuracy-latency Pareto front maintains the NNs of best spread on both accuracy and latency for this device, and thus is recommended for the device's deployment.
- We design a heuristic NN scheduling scheme to adaptively select NNs to suit the changing traffic rate during the detection process on a device. As flows are queued for detection, we use the past records of queue sizes to approximate the trend of the flow queue, i.e., increasing or decreasing. By jointly considering the queue trend and length, which implicitly illustrate the current traffic rate, the scheduler selects NNs per flow detection by the following principle: if the previously selected NN results in a growing long queue, it will choose a faster NN to accelerate the detection; Otherwise, it will select a slightly slower NN for higher accuracy.

Thorough experiments on three public datasets [21]–[23] and three commodity devices reveal that: **1)** The detection performance of SOTERIA is higher compared to eight NN solutions [4]–[6], [12], [13], [24]–[26], e.g., accuracy and F1-score are improved by 2.56% and 1.34%, respectively. **2)** When compared with other NAS solutions [13], [24], the required GPU memory and time to train each NN is reduced by 9.49× and 5.20×. **3)** By the heuristic scheduling, the prototyped SOTERIA[2] accelerates the detection time by 9.50× when the traffic rate changes from 15Kpps to 300Kpps (pps indicates the packet per second).

II. BACKGROUND AND MOTIVATION

A. Neural Networks in Attack Detection

Due to their superior performance, NNs have been widely applied to network attack detection [3]–[6]. Generally, NN-based detection can be regarded as a classification task. For collected flows[3], researchers first extract desired features (e.g., packet sizes, IP flag counts) and label classes (e.g., DoS, probe, or benign). Then, these feature-label pairs are fed to train an NN customized from standard NN modules like Convolutional Neural Network (CNN) [27] and Recurrent Neural Network (RNN) [28]. The trained NN is able to infer the categories of new flows with provided features.

Nonetheless, NNs are computation-intensive and require powerful hardware (e.g., GPU-equipped x86 servers) for acceleration, which hinders their further applications in networks, as most inexpensive devices (e.g., routers, or switches) for attack detection may only contain CPUs. As such, several solutions are proposed to lighten the NN for network devices. The authors in [7] propose KitNet, which modifies the neural architecture by incorporating several lightweight autoencoders. By performing experiments on a Raspberry PI, the authors demonstrate KitNet's efficiency and ability to run on a simple router with low latency. In [8], the authors propose Branch Convolution Net (BCN), which divides the convolution operation into several sub-operations (i.e., branches) to reduce the computational complexity. To handle high-speed traffic, the authors offload the BCN to a programmable switch and cooperate with a rule-based decision tree (DT), i.e., the two-phase detection. The DT (represented by a P4 program) works on the switch ASIC [9], handling traffic of hundreds of Gbps to filter the suspicious packets. Then, the filtered packets (only a small fraction, e.g., 5% in their experiments) are redirected to the switch CPU for the BCN-based in-depth detection.

These expert-designed NNs, however, are too expensive in time and human resources [10], [29]. In practical networks, the manual design process has to be repeated from one device to another according to heterogeneous configurations of operating systems and CPUs. Moreover, one handcraft NN deployed on the device is of constant running time and may fail to adapt to changing network rates, e.g., delaying the detection due to the network burst [11].

B. Neural Architecture Search

To free the exhaustive labor in expert-designed NNs, [14], [15] proposes to generate NNs by automated NAS. In NAS, researchers first define a search space that specifies the feasible neural operations. An NN in the search space is regarded as two parts: **1)** An architecture vector containing the specific operations; **2)** Trainable parameters in contained operations.

Then, different automated schemes [14], [15] are employed to search the NN architecture and optimize the corresponding NN parameters. In [14], the authors use a Reinforcement Learning (RL)-based agent to automatically generates candidate architecture vectors. Then, each architecture-associated NN is assigned parameters and trained from scratch. The performance of all evaluated NNs is used as the reward to train the RL agent. After the training of the agent is converged, an optimized NN is returned. While human resources are reduced, these solutions train all NNs individually and cause huge computation burdens (e.g., GPU memory and hours). To reduce the computation cost, recent works propose to train NNs in a weight-sharing manner [13], [15]. In the weight-

[2]The code is in https://github.com/xgr19/SOTERIA.

[3]A flow refers to a set of packets from the same 5-tuple (i.e., source/dest. IP, source/dest. ports, and transmission protocol).

sharing schemes, a supernet is constructed, whose parameters can be flexibly shared across NNs to form candidate sub-NNs according to the architecture vectors. In order to train the shared parameters efficiently, researchers may equally sample and train sub-NNs from large-size to small-size, i.e., progressively shrinking [13]. Other NAS approaches for automated NN generation are [12], [25], [26], [30].

Nevertheless, given the numerous sub-NNs (typically exponential) in the supernet, uniformly sampling sub-NNs to train is inefficient. The training of many sub-NNs is futile as only a small number of optimal sub-NNs are returned to the user. Though new schemes [24], [31] are proposed to reduce the sampling of sub-NNs during the training, their NN sampling schemes are empirical, and thus hard to identify the optimal sub-NNs exactly. Therefore, they may sample low-performance sub-NNs during the weight-sharing training and thus degrade the final accuracy of returned sub-NNs [17].

C. Evolutionary Algorithm

Unlike empirical sampling, [19] shows that the evolutionary algorithm can automatically select optimal individuals from a population, which may open up new sampling opportunities for weight-sharing training. When using the evolutionary algorithm to solve problems, a key concept is "domination". Solution A is said to dominate solution B, if

$$\forall i \in \{1, \ldots, k\}, f_i(A) \leq f_i(B), \tag{1}$$

$$\exists i \in \{1, \ldots, k\}, f_i(A) < f_i(B), \tag{2}$$

where $f_1, \ldots, f_k$ are specified minimization objectives. If an objective is to be maximized, we can leverage its negative or inverse instead.

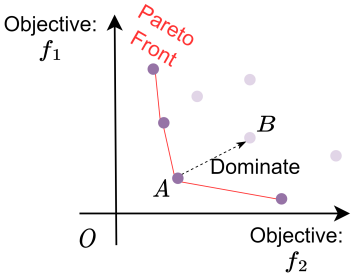


Fig. 1: An example of domination and Pareto front.

As an example, A is shown to dominate B in Fig. 1 as the objective values of A are smaller than those of B on f_1 and f_2. Unlike B, A is a non-dominated solution, i.e., not dominated by any other solution. In multi-objective problems ($k \geq 2$), it is infeasible to select one best solution to minimize all objectives simultaneously. Hence, the effort is devoted to finding all non-dominated solutions through the non-dominated sorting [18]. That is, each solution is compared with every other solution to record its count C of being dominated. Finally, the set of optimal (non-dominated) solutions is composed of solutions whose $C = 0$. This set is also known as the Pareto front (see Fig. 1). To continue the optimization, the Pareto front is sampled to be the parents, producing a new and better generation of offspring population by evolutionary crossover and mutation [32]. If the number of solutions in the Pareto front at hand is not sufficient to produce the offspring, researchers can jointly consider the second Pareto front (solutions of $C = 0$ after removing the original Pareto front), third Pareto front (solutions of $C = 0$ after removing the original and second Pareto front), etc. Then, the non-dominated sorting and the offspring reproducing are conducted iteratively to optimize the solutions.

We argue that, by exploiting specified objectives, the evolutionary algorithm can be introduced to sample the sub-NNs in weight-sharing training and thus boost the NAS performance. Based on this, we propose SOTERIA, which adapts the evolutionary algorithm to the weight-sharing training to generate a set of optimal NNs (Pareto front) with varying latency and accuracy. Compared with solutions [7], [8] to manually tune NN architecture for network devices, SOTERIA is a NAS-based system that automatically generates multiple NNs at once, saving expensive human resources. Also, unlike previous NAS solutions [13], [15], [24] to equally train numerous sub-NNs in the weight-sharing, the evolutionary algorithm helps SOTERIA to focus on the optimal sub-NNs in the Pareto front, reducing the training costs (GPU memory and hours) while boosting the accuracy. Finally, with our NN scheduling on the device, SOTERIA can output accurate detection results in time even if the deployed device is faced with traffic rate changes, which is not considered in NN-based detection like [4]–[8].

III. SOTERIA OVERVIEW

The framework overview of SOTERIA, which consists of two main processes: **NAS Training** and **Detection Process**, is depicted in Fig. 2. Similar to NAS approaches like [13], [15], [25], our NAS training also needs to be executed on the GPU server because the training of sub-NNs on the massive dataset is still computationally expensive and requires the acceleration of GPUs. However, our NAS training is one-shot, and we can redeploy the trained sub-NNs on different network devices.

NAS Training (Section IV). We first specialize the search space and encode its possible NNs' architectures into vectors (detailed in Section IV-A). Each architecture vector consists of $2N+1$ integers, where N is the predefined maximum number of layers. The first integer in the green square represents the valid number of layers ℓ in an NN ($\ell \leq N$). The following integers in the yellow and orange squares respectively maintain the specific setting of kernels and channels for each layer. Similar to other NAS schemes [13], the architecture representing our supernet has N layers, and each layer has the predefined maximum kernel sizes and channels. According to an architecture vector, we can select partial parameters in the supernet to form sub-NNs, i.e., weight-sharing.

With random sub-NNs' architectures as the origin population, we start to run the evolutionary training (detailed in Section IV-B). In the evolution, we utilize two objectives, accuracy and model size, to sample (i.e., non-dominated sorting [18]) and evolve (i.e., crossover and mutation [32]) architectures for the weight-sharing training. As accuracy is to

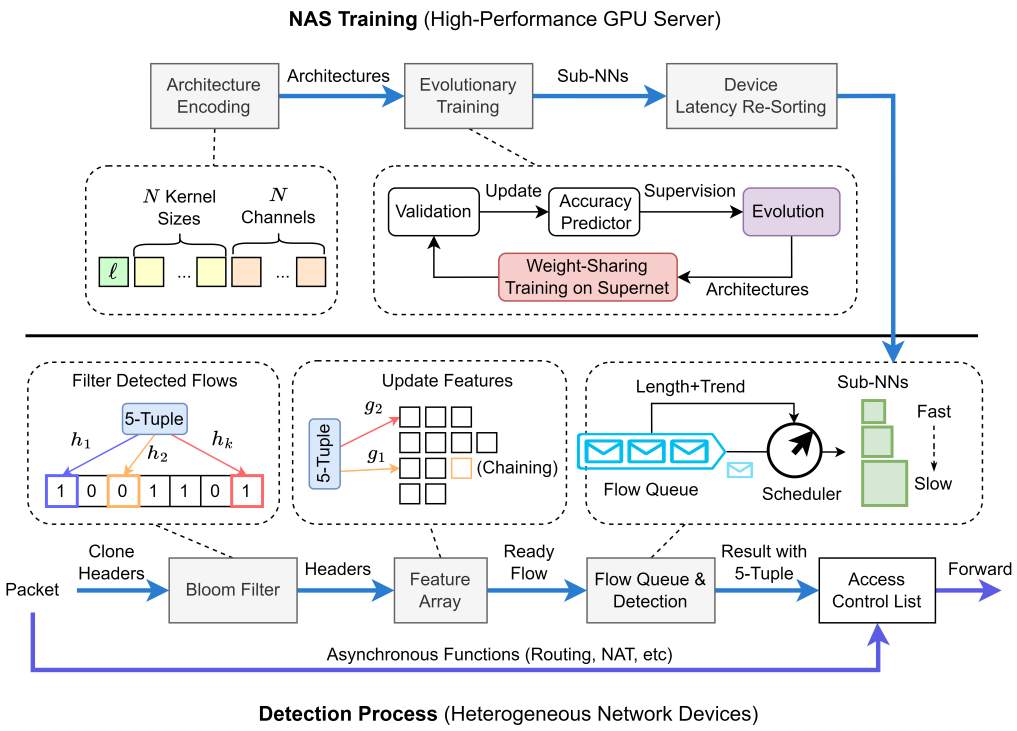


Fig. 2: The framework of SOTERIA.

be maximized, we actually use its negative: $100\% - accuracy$ instead according to Equations (1) and (2). Generally, the accuracy is only obtained after training sub-NNs associated with the whole architecture population, which is computationally expensive. To efficiently estimate the accuracy, we leverage an accuracy predictor [33], which takes in sub-NNs' architecture vectors and returns their accuracy. Besides, as interrupting the training to test latency on each individual device per NN is slow and impractical, the model size is used instead, which can be quickly computed through the architecture vector. Once the optimal architectures are obtained in an iteration by the non-dominated sorting, we construct their corresponding sub-NNs for weight-sharing training and validation. Then, the (accuracy, architecture) pairs of these sub-NNs are used to update the lightweight accuracy predictor for the next evolution.

After the training, a set of sub-NNs that has a good spread on the accuracy and model size, i.e., the accuracy-size Pareto front, is obtained. However, [20] reveals that the model size is not strictly correlated with the running latency on different devices. Thus, we recommend re-sorting the NNs according to their accuracy and tested latency on the target device, getting the accuracy-latency Pareto front (detailed in Section IV-C).

Detection Process (Section V). Each deployment device is equipped with the NNs along with three data structures: bloom filter, feature array, and flow queue (data structures are detailed in Section V-A). As headers maintain information like packet size and 5-tuple for feature computation [34], [35], we only clone the packet headers to save memory, while sending the original packets for asynchronous network functionalities. The headers are first sent to the bloom filter for membership queries [36]. That is, the 5-tuple is hashed $(h_1, \ldots, h_k)$ to quickly determine if its corresponding flow has been detected. If so (i.e., all hashed positions are "1"), the following processes are skipped to mitigate the detection overhead. Otherwise, the 5-tuple is hashed (i.e., g_1, g_2) into the feature array for updating the corresponding features (e.g., packet sizes, IP/TCP flag counts). Notably, we use chaining to solve hash collision in the feature array. We collect M packets for each 5-tuple defined flow. Once a flow is ready (i.e., its M packets have been collected), its hashed positions in the bloom filter are set to 1 and its features are dequeued.

The ready flow (actually its features) is again lined in the flow queue, waiting for the scheduler to run a suitable NN (detailed in Section V-B). The green squares of different sizes indicate sub-NNs of varying running latencies and accuracies. Through a heuristic scheme, the scheduler jointly considers the current length and the growing trend of the flow queue to decide a suitable sub-NN for each flow detection, mitigating the overall detection pressure. During the detection, the selected sub-NN outputs the class (benign or a specific attack) of a flow. Last, the 5-tuple along with its detected class is written to the access control list to instruct further reactions (forwarding/blocking) for the asynchronously passing packets.

IV. NAS TRAINING

A. Architecture Encoding

We consider building NNs based on a series of 1D convolutions that yield high accuracy in the network literature [5], [8],

[37], [38]. In a 1D convolution layer, two important settings are the kernel size and the number of channels. The kernel size decides how many trainable parameters are in a convolution kernel. The number of channels is in fact the number of kernels in a layer. Hence, by setting the number of convolution layers, along with kernel sizes and channels per layer, we can obtain an NN for the detection task.

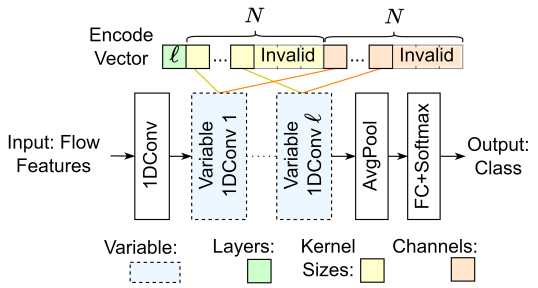


Fig. 3: The architecture encoding.

As such, we propose the architecture encoding in Fig. 3 to obtain NNs of variable settings. The architecture maintains fixed layers and variable layers. The fixed layer right near the input is also a 1D convolution. The other two fixed layers (i.e., average pooling, fully connected layer (FC) with softmax) are placed after the variable layers to output the classification results [37], [39]. Apart from the fixed layers, there are ℓ variable 1D convolution layers, where $3 \leq \ell \leq N = 9$. These variable layers are encoded in a fixed-length integer vector (i.e., the architecture vector of colored squares). Each of the variable layers can be changed in the kernel size and number of channels. Similar to [14], [40], the candidate kernel sizes are 3, 5, and 7; the number of channels can be 3×, 4×, or 6× of the input channels. Finally, these varying settings reach a search space with

$$\sum_{\ell=3}^{N} \Big(|KernelSizes| \times |Channels| \Big)^{\ell} \approx 4.35 \times 10^{8} \quad (3)$$

NNs of different model sizes, which is sufficient for the diverse performance of devices.

Among these NNs, the one with the largest number of layers and the largest number of kernels and channels in each layer is the supernet. The rest NNs (i.e., sub-NNs) can be derived from the supernet by their encoded architecture vectors and sharing techniques like the parameter transformation [13]. For example, the central five parameters of a convolution ($KernelSize = 7$) in a supernet can be referenced by a sub-NN whose architecture vector has $KernelSize = 5$ through the parameter transformation.

B. Evolutionary Training Algorithm

To optimize parameters in the weight-sharing, previous researchers (e.g., [13], [15]) minimize the expected loss $\mathcal{L}$ on the training data $\mathcal{D}_{trn}$, i.e.,

$$\min_{w} \mathrm{E}_{\alpha \sim p(\mathcal{A})} \left[\mathcal{L}(w_\alpha; \mathcal{D}_{trn}) \right], \quad (4)$$

where w is the weights (parameters) in the supernet and w_α is the shared weights sampled from the supernet by architecture $\alpha \in \mathcal{A}$. $p(\mathcal{A})$ refers to the uniform sampling distribution over all possible architectures in $\mathcal{A}$. Therefore,

$$p(\alpha) = \frac{1}{|\mathcal{A}|}. \quad (5)$$

As NAS only returns the optimal sub-NNs, a large number of irrelevant architectures, especially the ones of low learning capabilities, are trained in vain, not only costing expensive resources but also disturbing the accuracy of returned NNs as their weights are heavily shared [17].

Intuitively, we want to block the irrelevant architectures before the training, i.e.,

$$p(\alpha) = \frac{1}{|\mathcal{A}_{Pareto}|} \mathbb{1}_{\mathcal{A}_{Pareto}}(\alpha), \quad (6)$$

where $\mathbb{1}_{\mathcal{A}_{Pareto}}(\alpha) = \begin{cases} 1, & \alpha \in \mathcal{A}_{Pareto}, \\ 0, & otherwise, \end{cases}$ is an indicator function, and $\mathcal{A}_{Pareto}$ is the Pareto front. However, $\mathcal{A}_{Pareto}$ is obtained after the non-dominated sorting of the trained sub-NNs with respect to accuracy and model sizes. It is impossible to know the accuracy before training. Hence, we propose Algorithm 1 to progressively approximate $\mathcal{A}_{Pareto}$.

Algorithm 1 Evolutionary Training Algorithm (ETA)

Input: Data $\mathcal{D}_{trn}$, $\mathcal{D}_{val}$, and model size metric $\mathcal{S}$.
Output: Trained $sub\text{-}NNs$.

1: Let architecture population $child = \emptyset$.
2: **for** $space$ in Crossover & Mutation Space **do**
3: **for** training epoch $\leq MAX$ **do**
4: **if** $child$ is $\emptyset$ **then**
5: $child = \texttt{RandomSample}(space)$. # Initialize
6: **end if**
7: $sub\text{-}NNs = supernet(child)$. # Weight-sharing
8: $acc = \texttt{TrainAndEval}(sub\text{-}NNs, \mathcal{D}_{trn}, \mathcal{D}_{val})$.
9: Update predictor CART by acc and $child$.
10: $state = \texttt{Expect}(acc) + \texttt{Variance}(\mathcal{S}(child))$.
11: **if** $state$ is NOT improved **then**
12: Break the training in $space$. # Early stopping
13: **else**
14: $child = \texttt{Evolute}(child, space, \mathrm{CART}, \mathcal{S})$.
15: **end if**
16: **end for**
17: $\mathcal{A}_{Pareto} = \mathcal{A}_{Pareto} \cup child$.
18: **end for**
19: $\mathcal{A}_{Pareto} = \texttt{NoDominateSort}(\mathcal{A}_{Pareto}, acc, \mathcal{S})$.
20: $sub\text{-}NNs = supernet(\mathcal{A}_{Pareto})$. # Trained sub-NNs

In Line 2, we iterate the crossover & mutation space progressively. In light of Fig. 3, we first allow the evolution happens in the kernel sizes, while the channels and layers are at their maximum values (6× and 9, respectively). Next, the space expands to kernel sizes and channels, but the number of layers still remains at 9. Last, the layers are also varied from 3 to 9. This is because randomly training NNs of different sizes

can result in lower accuracy. Our space expansion roughly arranges the training order of NNs from large to small, which avoids the interference between small and large NNs, and boosts the weight-sharing effectiveness [13].

In Lines 4∼6, we initialize the empty architecture population $child$ by uniformly sampling architecture vectors in the beginning crossover & mutation space.

In Lines 7∼9, we first train and evaluate the $sub\text{-}NNs$. By utilizing the weight-sharing, sub-NNs associated with architectures in $child$ are assigned shared parameters from $supernet$ via a transformation in [13]. Then, the validation accuracies acc and corresponding architecture vectors $child$ of trained sub-NNs are used to update the accuracy predictor, **CART**, the Classification And Regression Tree model [33]. After the training, CART is expected to establish the mapping from architectures to accuracies. It is common to employ lightweight tree models as predictors [40], [41], as their training is swift and well-implemented in the off-the-shelf scikit-learn library [42].

In Lines 10∼15, we decide whether to early stop the evolution in the current crossover & mutation space before reaching the MAX training epoch. First, we compute the accuracy expectation and the model size variance of the population $child$. The variance measures the NN dispersion on model size (i.e., indicating the possible richness of NNs under different latency requirements). If $state$ is not improved with respect to the previous epochs, we stop the training in the current $space$. Otherwise, `Evolute(.)` conducts crossover and mutation on the architecture vectors $child$ to generate new architectures, and then the optimal architectures are returned as new $child$ by the sorting with objectives of predicted accuracy (CART) and model size ($\mathcal{S}$). The **crossover and mutation** derive new architectures by inputting the architectures in $child$. We leverage the simulated binary crossover (SBX) operator [32] to compute the probability density of offspring and randomly sample the new architectures, which has been implemented in pymoo library [43]. The **sorting** is detailed by Algorithm 2 in Section IV-C.

In Lines 19∼20, we return the trained sub-NNs. First, we apply the non-dominated sorting on the collected architectures $\mathcal{A}_{Pareto}$, and select the architectures in the Pareto front. Note that the current objectives are trained accuracy acc and model sizes $\mathcal{S}$ of architectures. Finally, we assign the trained parameters in the supernet to each sub-NN in sorted $\mathcal{A}_{Pareto}$, according to the sub-NN's architecture and the parameter transformation.

C. Device Latency Re-Sorting

The NAS training is on GPUs due to the computational optimization of multiple sub-NNs. Thus, it is time-consuming to frequently perform cross-hardware (between GPUs and network devices) latency measurements [44]. Following the previous success [24], [31], it is feasible to replace the latency with the more convenient model size in the training. Generally, models of big sizes are more complex and thus slower. But the model size may not be representative of all devices due to the various hardware factors [20]. As such, we recommend an extra non-dominated sorting to re-sort the trained sub-NNs by their test latencies on the target device. In Algorithm 1 Line 20, we generate a set of NNs instead of only one, so it is highly likely to filter out a new Pareto front according to the latency again.

To record running latencies, we implement an automated script to load and run sub-NNs on devices. Then, for each device with collected latencies, we run Algorithm 2 with $f_1 = 100\% - accuracy$ and $f_2 = latency$ to obtain its new accuracy-latency Pareto front. We iterate through each sub-NN and check whether it is dominated by any other sub-NNs (Lines 3∼7). Then, the non-dominated sub-NNs are collected as the new Pareto front (Lines 8∼10). Finally, only sub-NNs in this new Pareto front are recommended for the device. One may replace the model size metric in ETA with latency to omit Algorithm 2. However, due to different device settings, an NN may run with varying latencies on devices [20] and thus results in repeated ETA, which is more expensive than repeating Algorithm 2.

Algorithm 2 Non-Dominated Sorting

Input: The sub-NNs $sub\text{-}NNs$, and objectives f_1, f_2.
Output: Pareto front $\mathcal{F}$.

```
1:  for NN ∈ sub-NNs do
2:      c_NN = 0.          # Count of being dominated for NN
3:      for NN' ∈ sub-NNs do
4:          if NN' dominates NN then
5:              c_NN += 1.     # Equation (1) and (2) with f1, f2
6:          end if
7:      end for
8:      if c_NN = 0 then
9:          F ∪ NN.
10:     end if
11: end for
```

V. Detection Process

A. Data Structures

In Fig. 2, we use three data structures: a bloom filter, a feature array, and a flow queue, in the detection process.

The bloom filter [36] is a space-efficient data structure consisting of k hash functions and an m-bit array. In the detection process, the bloom filter filters the packets by their 5-tuples. Initially, the m bits are zeros. After a flow has enough packets for feature computation and detection, its 5-tuple is hashed in the bloom filter by k hash functions, and the located k bits are set to ones. Then, future packets whose hashed bits are ones in the bloom filter will skip the subsequent processes and directly obtain their class labels in the access control list to further reduce the detection pressure. Although the bloom filter accelerates the packet filter to a time complexity of constant k, its hash-based query may result in false positives. That is, packets of an undetected flow could be regarded as "detected" if a hash collision happens. Fortunately, a sufficiently small false positive rate (i.e., 10^{-6} in SOTERIA)

TABLE I: Features derived from M packets of a flow.

Name	Description
Packet sizes	The max, min, average, variance, and sum of M packets
IP flag counts	MF and DF of M packets
TCP flag counts	SYN, ACK, PSH, FIN, RST, and ECN of M packets
TCP window sizes	The max, min, average of M packets
The Mth packet	Header length, TTL, transmission protocol, source and destination ports

can be mathematically guaranteed by adjusting the k and m of the bloom filter according to [36].

The feature array consists of hash arrays for storing the flow features. Following [34], [35], we use features derived from packet headers (Table I). As such, the feature array does not need to hold packet payload, which significantly reduces the memory requirement. To insert a packet, two hash functions g_1, g_2 are applied on the packet's 5-tuple, locating the positions that potentially hold the flow defined by the hashed 5-tuple. If no specific flow is found in the hashed positions, we use the chaining technique to insert the packet. That is, a new cell is allocated in the array to store packet headers from the hashed 5-tuple.

The flow queue dynamically allocates memory to store the flows waiting for detection. If a flow with a specific 5-tuple is ready (i.e., M packets are collected), we dequeue this flow from the feature array, compute its features according to Table I, and then enqueue it into the flow queue. This flow queue is First-In-First-Out (FIFO), i.e., the flows located in the queue head will be first dequeued to the scheduler to select a proper sub-NN for the detection.

B. Heuristic Scheduling

After receiving a flow's features, the scheduler is to select a suitable sub-NN for the detection inference, meeting the current traffic rate. To this end, the scheduler first considers the growing trend of the flow queue. Given observation points $\{(x_1, y_1), \ldots, (x_k, y_k)\}$ of the flow queue where x_i and y_i are the observed time and queue length, respectively, the growing trend T is approximated by the least-squares estimation [45]:

$$T = \frac{\sum_{i=1}^{k}(x_i - \overline{x})(y_i - \overline{y})}{\sum_{i=1}^{k}(x_i - \overline{x})^2}, \tag{7}$$

where $\overline{x}$ and $\overline{y}$ are the average values of x_i and y_i, respectively. The trend T approximately reflects whether the queue length is increasing ($T > 0$) or decreasing ($T < 0$). If the previously selected sub-NN results in an increased (decreased) trend, the scheduler may now select a sub-NN of a lower (higher) latency.

However, scheduling sub-NNs merely based on T is not sufficient. While $T < 0$ indicates the queue length is decreasing, the queue could also be very long at the same time, resulting in a high waiting time of tailed flows. As such, we consider both the trend and the queue length in Algorithm 3 when scheduling sub-NNs. In Line 2, we compute the growing trend T by Equation (7). Then, we select an NN according to the following conditions:

Algorithm 3 Heuristic Scheduling

Input: Sub-NNs in the accuracy-latency Pareto front, observations $\{(x_1, y_1), \ldots, (x_k, y_k)\}$, and queue length L.
Output: The selected NN.

1: Let i denote the index of the last selected NN.
2: $T = \texttt{TrendCompute}((x_1, y_1), \ldots, (x_k, y_k))$.
3: **if** $T < 0$ **then**
4: $\quad allowed = \texttt{max}(0, |sub\text{-}NNs| - \frac{L}{100})$.
5: $\quad i = \texttt{min}(i + 1, allowed)$. # Latency↑, accuracy↑
6: **else if** $T > 0$ **then**
7: $\quad i = \texttt{max}(0, i - 1)$. # Latency↓, accuracy↓
8: **else**
9: $\quad i = i$.
10: **end if**
11: $NN = sub\text{-}NNs[i]$

$T < 0$ (Lines 3∼5) indicates the flow queue is decreasing and thus the detection pressure is reduced. Hence, we select an NN with a higher latency (i.e., increment the index i by one) to improve the accuracy. To consider the wait time of flows in the queue tail, we further limit the index increment by the queue length L. That is, the maximum allowed index is reduced by one when L increases by one hundred.

$T > 0$ (Lines 6∼7) indicates the flow queue is increasing. As more flows are queued for the detection, we speed up the inference process by selecting an NN with a lower latency (i.e. index of $i - 1$). Of course, this speedup is achieved by sacrificing the detection accuracy.

$T = 0$ (Lines 8∼9) means the length of the flow queue is stable, indicating that the last selected NN is suitable for the current network state. So we keep the selected NN unchanged.

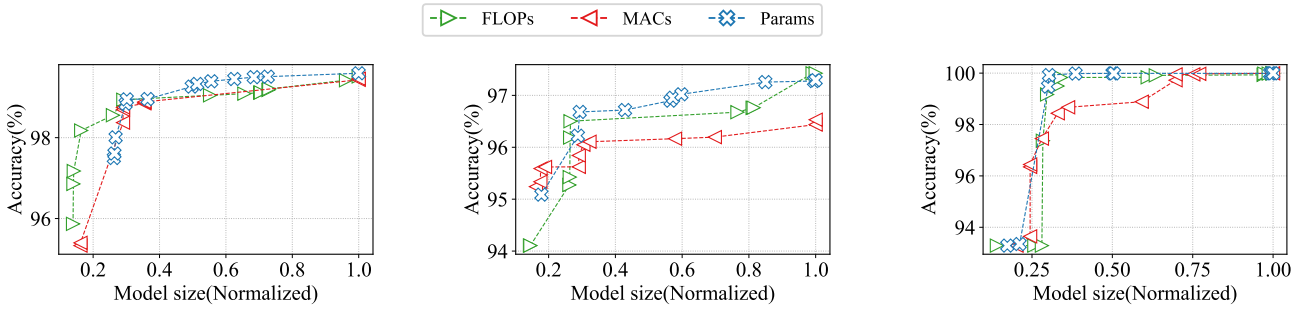
TABLE II: Deployed devices.

	System	CPU			RAM
		Type	Clock	Cores	
PI (Raspberry PI 4B)	Raspbian GNU/ Linux 11	Broadcom BCM2711	1.5 GHz	4	4 GB
EdgeCore (Wedge 100BF-65X)	Open Network 4.14	Intel D-1517	1.6 GHz	8	8 GB
H3C (S9850-32H)	Open Network 4.14	Intel D-1527	2.2 GHz	8	8 GB

VI. Evaluation

A. Experimental Settings

We utilize three public datasets: UNSW-NB15 [22], Bot-IoT [21], and CICIDS [23]. Each of these datasets is randomly divided into three parts, with 80% of the data used for training, 10% for validation, and 10% for testing. SOTERIA is compared



(a) BoT-IoT (b) UNSW-NB15 (c) CICIDS

Fig. 4: The Pareto fronts on objectives of detection accuracy (f_1) and different model size metrics (f_2) on three datasets.

with eights NAS and manually configured NNs: AttentiveNAS [24], OFA [13], ProxylessNAS [25], ENAS [26], and DARTS [12] are NAS solutions; LuNet [5], DeepLSTM [6], FFDNN [4] are handcrafted NNs. All NAS schemes use the search space similar to ours in Section IV-A. All model training, validation, and comparison are on a server with CPU of Intel(R) Xeon(R) E5-2643 v4 @3.40GHz, GPU of Tesla M60 (8GB×2), Python 3.9, and PyTorch 1.12.0.

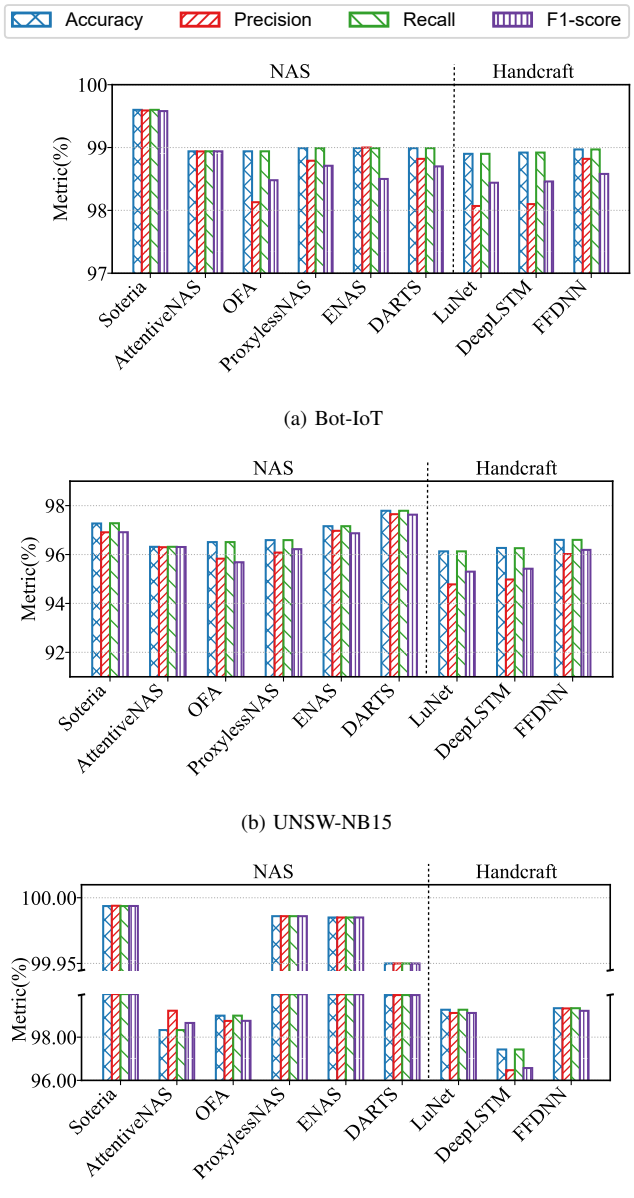
We also prototype SOTERIA on three devices (Table II): PI, EdgeCore, and H3C with C++ and MNN [46]. Like [7], PI is used to act as a low-performance router. EdgeCore and H3C are programmable switches with CPUs and ASICs. Here we assume that the traffic goes through the CPU directly, and further discussion about the ASIC-based enhancement is included in Section VII. The main settings of SOTERIA are: the epochs for each evolution space $MAX = 20$, the collected packets per flow $M = 4$, and the number of observation points in the scheduling is 3. The evolutionary techniques in SOTERIA (i.e., crossover, mutation, and non-dominated sorting) are based on the pymoo library [43], [47].

B. Model Size Metric Selection

In Algorithm 1, a model size metric $\mathcal{S}$ should be specified. Typically, there are three metrics, i.e., the number of floating-point operations (FLOPs), multiply-accumulate operations (MACs), and trainable parameters (Params), that can be used to evaluate the size of an NN [24], [48]. Fig. 4 shows the Pareto fronts on detection accuracy with these metrics. To make metrics comparable to each other, we normalize them by the min-max normalization [49]. As shown, while the Pareto fronts of the three metrics all have good spreads on the size, the Pareto front of Params has the best accuracy. Thus, we set $\mathcal{S} = Params$ in the following experiments.

C. Comparison with Existing Schemes

Classification performance. As stated in Section II-A, NN-based detection is usually treated as a classification problem. Thus, we utilize four classification metrics: accuracy, precision, recall, and F1-score, to evaluate all compared NNs in Fig. 5. If there are several generated NNs (e.g., Pareto fronts of NAS solutions), we only report the results of the best



(a) Bot-IoT

(b) UNSW-NB15

(c) CICIDS

Fig. 5: The classification performance on three datasets.

NN. As depicted, SOTERIA has a competitive performance on all datasets. For example, in the CICIDS dataset of Fig. 5c, SOTERIA improves the F1-score by 1.34% (compared with AttentiveNAS, 99.99% vs. 98.65%) and the accuracy by 2.56% (compared with DeepLSTM, 99.99% vs. 97.43%).

Training costs. The costs of different approaches are depicted in Fig. 6. As NAS solutions can train several NNs, we here consider the average cost $\overline{C}$ of generating one NN, where

$$\overline{C} = \frac{\text{total cost(memory/hours)}}{\#\ \text{NNs}}. \tag{8}$$

As shown in Fig. 6, SOTERIA is more efficient in training than other schemes. On average, SOTERIA reduces the time of AttentiveNAS by 5.20× (1.00h vs. 5.20h), and the GPU memory of DARTS by 9.49× (126.69MiB vs. 1203.00MiB). It seems that SOTERIA only has limited improvement over OFA and AttentiveNAS, e.g., only saving hundreds of MiB and several hours. But Fig. 6 is the average cost per NN. Such improvement will scale linearly as the number of NNs increases, which is the true case in these NAS solutions that consider numerous NNs during the training.

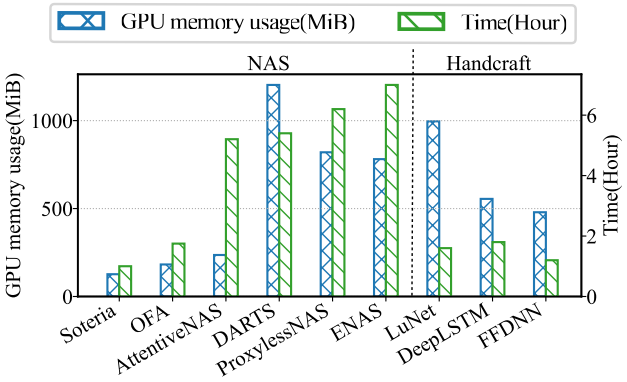


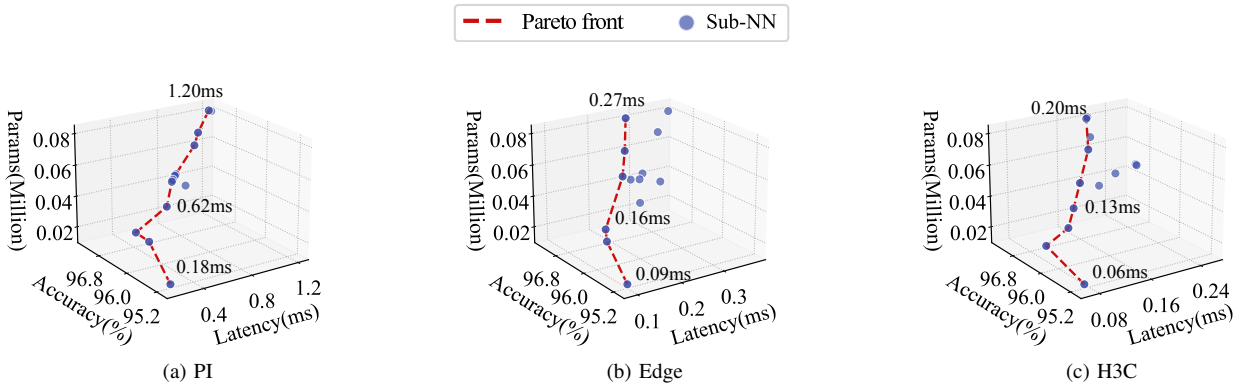
Fig. 6: The average costs of GPU memory and time per NN.

D. Latency Re-Sorting

After we obtain sub-NNs that have a good spread on the accuracy and Params, we should re-sort these sub-NNs according to their test latencies on devices. Fig. 7 shows the resorting. We note that some sub-NNs originally in the accuracy-param Pareto front are outside of the accuracy-latency front. As mentioned earlier, it is hard for model size to represent different device factors well [20]. After the re-sorting in Fig. 7, we finally obtain optimal NNs whose running latencies range from 0.06ms to 1.20ms.

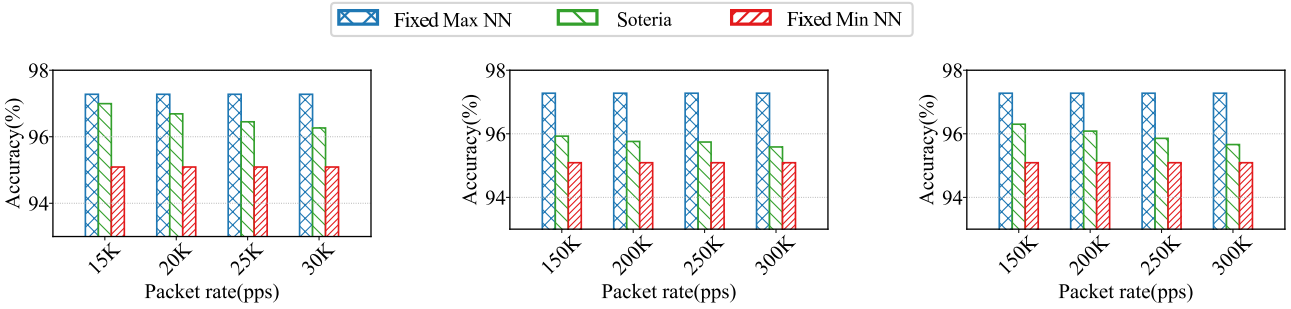
E. Prototype Analysis

We now compare SOTERIA (Algorithm 3) with two simple scheduling schemes: only using the slowest sub-NN of highest accuracy ("fixed max NN"), and only using the fastest NN



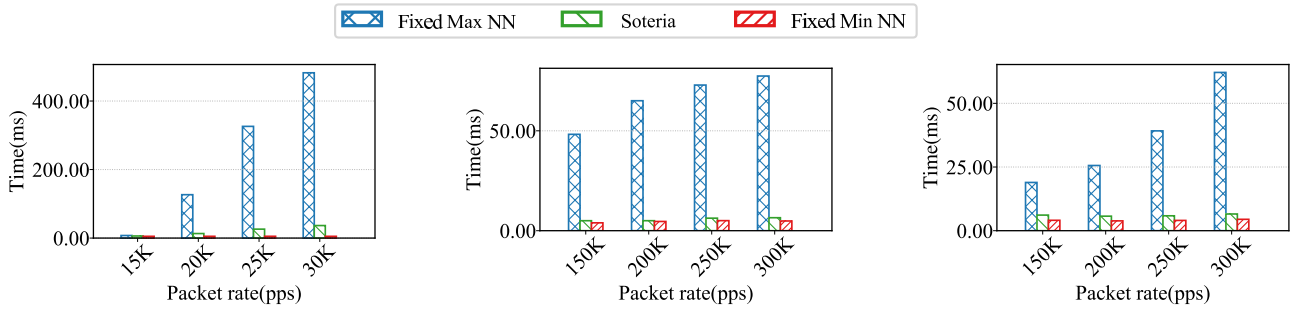
(a) PI (b) Edge (c) H3C

Fig. 7: The re-sorted Pareto front of UNSW-NB15 according to detection accuracy and running latency on the three devices.



(a) PI (b) Edge (c) H3C

Fig. 8: The average accuracy of test flows from UNSW-NB15 under different packet rates (packet per second, pps).



(a) PI (b) Edge (c) H3C

Fig. 9: The average detection time of test flows from UNSW-NB15 under different packet rates (packet per second, pps).

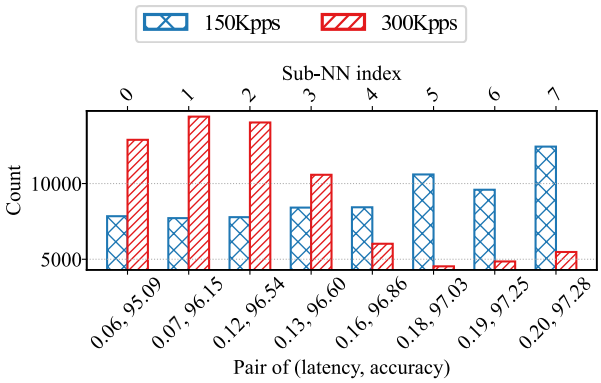


Fig. 10: Selection counts on H3C. The bottom x-axis shows the latency and accuracy of each NN (NNs indexed $0 \sim 7$ in the top x-axis).

of lowest accuracy ("fixed min NN"). Notably, all sub-NNs considered here are generated by our NAS training.

Fig. 8 and 9 show the detection accuracy and time under different packet rates. As depicted, SOTERIA has a competitively high accuracy compared to the fixed max NN while only incurring detection time as low as that of the fixed min NN. For example, in Fig. 8a when pps = 15K, the accuracy of SOTERIA and the fixed max NN are 96.99% vs. 97.27%. In Fig. 9c when pps = 300K, SOTERIA reduces the detection latency by $9.50\times$ when compared with the fixed max NN (6.55ms vs. 62.13ms). In our opinion, reducing the detection latency by tens of microseconds is vital as it allows the detection system more swift to stop the malicious traffic and control the damage as little as possible. Though SOTERIA shows a small improvement (around $1 \sim 2\%$ on accuracy) over the fixed min NN in Fig. 8, we believe that any improved accuracy in defense against attacks is worthy, especially for large enterprises, 1% of undetected attacks can cause severe financial loss [50].

Fig. 10 shows the selected counts of each NN in SOTERIA. When the traffic rate is slow (150Kpps), SOTERIA tends to select NNs of higher latency to produce higher accuracy. For example, the NN of 0.20ms (index 7) is chosen 12.43K times. In contrast, when the traffic rate is high (300Kpps), SOTERIA prefers low-latency NNs to reduce the detection time, e.g., NN of index 1 (0.07ms) is selected 14.41K times. In other words, Fig. 10 is the reason for Fig. 8 and 9, which helps SOTERIA to adapt to changing traffic rates.

VII. Conclusion and Further Discussion

This paper presents SOTERIA to efficiently train and schedule NNs for attack detection on heterogeneous network devices. We first use the evolutionary training and non-dominated sorting to obtain a set of optimal NNs with high accuracy and latency diversity. Finally, the optimal NNs on the device, are heuristically scheduled according to the traffic rates, making a trade-off between detection latency and accuracy.

Though extensive experiments demonstrate the superiority of SOTERIA, its detection time is still a main concern. Fig. 7 shows that the latency of the fastest NN can be 0.06ms. But with the traffic rate increases, multiple flows will be queued and lead to a longer detection time (e.g., 6.55ms when 300Kpps in Fig. 9c). In other words, SOTERIA may be more suitable for simple home routers or small LANs with low traffic rates. To accelerate SOTERIA, a feasible solution is the two-phase detection on programmable switches [8], [51] (Section II-A). We can first train a rule-based decision tree and then install it on the switch ASIC as match-action P4 tables for high-speed packet classification. Packets classified as malicious will then trigger a redirect action in the P4 table and be forwarded to the NN-based secondary detection on the switch CPU. Experiments in [8] show that only 5% of the traffic requires a secondary detection, which is expected to significantly speed up SOTERIA (even to line rate).

VIII. Acknowledgment

We thank our shepherd Prof. Marco Brocanelli and anonymous reviewers. This work is supported in part by the National Key R&D Program of China under Grant No. 2022YFB3105000, the National Natural Science Foundation of China under Grant No. 61972189, the Major Key Project of PCL under Grant No. PCL2023AS5-1, the Shenzhen Key Lab of Software Defined Networking under Grant No. ZDSYS20140509172959989, and the China Scholarship Council (CSC202306210169).

REFERENCES

[1] J. Raiyn *et al.*, "A survey of cyber attack detection strategies," *International Journal of Security and Its Applications*, vol. 8, no. 1, pp. 247–256, 2014.

[2] S. Tan, J. M. Guerrero, P. Xie, R. Han, and J. C. Vasquez, "Brief survey on attack detection methods for cyber-physical systems," *IEEE Systems Journal*, vol. 14, no. 4, pp. 5329–5339, 2020.

[3] J. Zhang, L. Pan, Q. Han, C. Chen, S. Wen, and Y. Xiang, "Deep learning based attack detection for cyber-physical system cybersecurity: A survey," *IEEE CAA Journal of Automatica Sinica*, vol. 9, no. 3, pp. 377–391, 2022.

[4] S. M. Kasongo and Y. Sun, "A deep learning method with wrapper based feature extraction for wireless intrusion detection system," *Computers & Security*, vol. 92, p. 101752, 2020.

[5] P. Wu and H. Guo, "Lunet: A deep neural network for network intrusion detection," in *Proceedings of the Symposium Series on Computational Intelligence*. IEEE, 2019, pp. 617–624.

[6] J. Ashraf, A. D. Bakhshi, N. Moustafa, H. Khurshid, A. Javed, and A. Beheshti, "Novel deep learning-enabled LSTM autoencoder architecture for discovering anomalous events from intelligent transportation systems," *Transactions on Intelligent Transportation Systems*, vol. 22, no. 7, pp. 4507–4518, 2021.

[7] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai, "Kitsune: An ensemble of autoencoders for online network intrusion detection," in *Proceedings of the 25th Annual Network and Distributed System Security Symposium*. The Internet Society, 2018.

[8] G. Xie, Q. Li, C. Cui, P. Zhu, D. Zhao, W. Shi, Z. Qi, Y. Jiang, and X. Xiao, "Soter: Deep learning enhanced in-network attack detection based on programmable switches," in *Proceedings of the 41st International Symposium on Reliable Distributed Systems*. IEEE, 2022, pp. 225–236.

[9] I. Corporation, "Intel Tofino: P4-programmable Ethernet switch ASIC that delivers better performance at lower power," https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html, 2019, [Online; accessed 3-May-2023].

[10] B. Arzani, K. Hsieh, and H. Chen, "Interpretable feedback for automl and a proposal for domain-customized automl for networking," in *Proceedings of the 20th Workshop on Hot Topics in Networks*. ACM, 2021, pp. 53–60.

[11] Z. Zhong, S. Yan, Z. Li, D. Tan, T. Yang, and B. Cui, "Burstsketch: Finding bursts in data streams," in *Proceedings of the International Conference on Management of Data*. ACM, 2021, pp. 2375–2383.

[12] H. Liu, K. Simonyan, and Y. Yang, "DARTS: differentiable architecture search," in *Proceedings of the 7th International Conference on Learning Representations*. OpenReview.net, 2019.

[13] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once-for-all: Train one network and specialize it for efficient deployment," in *Proceedings of the 8th International Conference on Learning Representations*. OpenReview.net, 2020.

[14] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *Proceedings of the 5th International Conference on Learning Representations*. OpenReview.net, 2017.

[15] Z. Guo, X. Zhang, H. Mu, W. Heng, Z. Liu, Y. Wei, and J. Sun, "Single path one-shot neural architecture search with uniform sampling," in *Proceedings of the 16th European Conference on Computer Vision*, vol. 12361. Springer, 2020, pp. 544–560.

[16] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*. AAAI Press, 2019, pp. 4780–4789.

[17] S. You, T. Huang, M. Yang, F. Wang, C. Qian, and C. Zhang, "Greedynas: Towards fast one-shot NAS with greedy supernet," in *Proceedings of the Conference on Computer Vision and Pattern Recognition*. Computer Vision Foundation / IEEE, 2020, pp. 1996–2005.

[18] C. Bao, L. Xu, E. D. Goodman, and L. Cao, "A novel non-dominated sorting algorithm for evolutionary multi-objective optimization," *Journal of Computational Science*, vol. 23, pp. 31–43, 2017.

[19] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[20] N. Ma, X. Zhang, H. Zheng, and J. Sun, "Shufflenet V2: practical guidelines for efficient CNN architecture design," in *Proceedings of the 15th European Conference on Computer Vision*. Springer, 2018, pp. 122–138.

[21] N. Koroniotis, N. Moustafa, E. Sitnikova, and B. P. Turnbull, "Towards the development of realistic botnet dataset in the internet of things for network forensic analytics: Bot-iot dataset," *Future Generation Computer Systems*, vol. 100, pp. 779–796, 2019.

[22] N. Moustafa and J. Slay, "UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)," in *Proceedings of the 2015 Military Communications and Information Systems Conference*. IEEE, 2015, pp. 1–6.

[23] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, "Toward generating a new intrusion detection dataset and intrusion traffic characterization," in *Proceedings of the 4th International Conference on Information Systems Security and Privacy*. SciTePress, 2018, pp. 108–116.

[24] D. Wang, M. Li, C. Gong, and V. Chandra, "Attentivenas: Improving neural architecture search via attentive sampling," in *Proceedings of the Conference on Computer Vision and Pattern Recognition*. Computer Vision Foundation / IEEE, 2021, pp. 6418–6427.

[25] H. Cai, L. Zhu, and S. Han, "Proxylessnas: Direct neural architecture search on target task and hardware," in *Proceedings of the 7th International Conference on Learning Representations*. OpenReview.net, 2019.

[26] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient neural architecture search via parameter sharing," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 80. PMLR, 2018, pp. 4092–4101.

[27] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, 1998.

[28] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, pp. 1735–1780, 1997.

[29] J. Holland, P. Schmitt, N. Feamster, and P. Mittal, "New directions in automated traffic analysis," in *Proceedings of the Conference on Computer and Communications Security*. ACM, 2021, pp. 3366–3383.

[30] E. Real, S. Moore, A. Selle, S. Saxena, Y. I. Leon-Suematsu, J. Tan, Q. V. Le, and A. Kurakin, "Large-scale evolution of image classifiers," in *Proceedings of the 34th International Conference on Machine Learning*, vol. 70. PMLR, 2017, pp. 2902–2911.

[31] J. Yu, P. Jin, H. Liu, G. Bender, P. Kindermans, M. Tan, T. S. Huang, X. Song, R. Pang, and Q. Le, "Bignas: Scaling up neural architecture search with big single-stage models," in *Proceedings of the 16th European Conference on Computer Vision*, vol. 12352. Springer, 2020, pp. 702–717.

[32] K. Deb, K. Sindhya, and T. Okabe, "Self-adaptive simulated binary crossover for real-parameter optimization," in *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 2007, pp. 1187–1194.

[33] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Wadsworth, 1984.

[34] Y. Lim, H. Kim, J. Jeong, C. Kim, T. T. Kwon, and Y. Choi, "Internet traffic classification demystified: on the sources of the discriminative power," in *Proceedings of the 2010 Conference on Emerging Networking Experiments and Technology*. ACM, 2010, p. 9.

[35] C. Busse-Grawitz, R. Meier, A. Dietmüller, T. Bühler, and L. Vanbever, "pforest: In-network inference with random forests," *CoRR*, vol. abs/1909.05680, 2019.

[36] A. Z. Broder and M. Mitzenmacher, "Survey: Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, pp. 485–509, 2003.

[37] X. Wang, S. Chen, and J. Su, "App-net: A hybrid neural network for encrypted mobile traffic classification," in *Proceedings of the 39th Conference on Computer Communications, Workshops*. IEEE, 2020, pp. 424–429.

[38] M. Lotfollahi, M. J. Siavoshani, R. S. H. Zade, and M. Saberian, "Deep packet: a novel approach for encrypted traffic classification using deep learning," *Soft Computing*, vol. 24, no. 3, pp. 1999–2012, 2020.

[39] C. Liu, L. He, G. Xiong, Z. Cao, and Z. Li, "Fs-net: A flow sequence network for encrypted traffic classification," in *Proceedings of the 2019 Conference on Computer Communications*. IEEE, 2019, pp. 1171–1179.

[40] Z. Lu, K. Deb, E. D. Goodman, W. Banzhaf, and V. N. Boddeti, "Nsganetv2: Evolutionary multi-objective surrogate-assisted neural ar-

chitecture search," in *Proceedings of the 16th European Conference on Computer Vision*, vol. 12346. Springer, 2020, pp. 35–51.
[41] Y. Sun, H. Wang, B. Xue, Y. Jin, G. G. Yen, and M. Zhang, "Surrogate-assisted evolutionary deep learning using an end-to-end random forest-based performance predictor," *IEEE Transactions on Evolutionary Computation*, vol. 24, no. 2, pp. 350–364, 2020.
[42] scikit learn, *Decision Trees*, 2007 (accessed August 3, 2023). [Online]. Available: https://scikit-learn.org/stable/modules/tree.html#regression
[43] Anyoptimization, *pymoo*, 2018 (accessed August 3, 2023). [Online]. Available: https://github.com/anyoptimization/pymoo
[44] X. Luo, D. Liu, H. Kong, S. Huai, H. Chen, and W. Liu, "Lightnas: On lightweight and scalable neural architecture search for embedded platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 6, pp. 1784–1797, 2023.
[45] Wikipedia contributors, "Least squares," https://en.wikipedia.org/wiki/Least_squares, 2023, [Online; accessed 10-May-2023].
[46] X. Jiang, H. Wang, Y. Chen, Z. Wu, L. Wang, B. Zou, Y. Yang, Z. Cui, Y. Cai, T. Yu, C. Lyu, and Z. Wu, "MNN: A universal and efficient inference engine," in *Proceedings of Machine Learning and Systems*. mlsys.org, 2020, pp. 1–13.
[47] J. Blank and K. Deb, "pymoo: Multi-objective optimization in python," *IEEE Access*, vol. 8, pp. 89 497–89 509, 2020.
[48] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," in *Proceedings of the Conference on Computer Vision and Pattern Recognition*. Computer Vision Foundation / IEEE, 2019, pp. 10 734–10 742.
[49] Wikipedia contributors, "Feature scaling — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Feature_scaling&oldid=1145780045, 2023, [Online; accessed 10-April-2023].
[50] M. H. U. Sharif and M. A. Mohammed, "A literature review of financial losses statistics for cyber security and future trend," *World Journal of Advanced Research and Reviews*, vol. 15, no. 1, pp. 138–156, 2022.
[51] C. Zheng, Z. Xiong, T. T. Bui, S. Kaupmees, R. Bensoussane, A. Bernabeu, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman, "Iisy: Practical in-network classification," *arXiv preprint arXiv:2205.08243*, 2022.