



iGuard: Efficient Isolation Forest Design for Malicious Traffic Detection in Programmable Switches

Sankalp Mittal*

Google

Bangalore, India

sankalpmittal123@gmail.com

Patel Heetkumar†

Indian Institute of Technology

Hyderabad, India

heetp1011@gmail.com

Harikrishnan V.†

Indian Institute of Technology

Hyderabad, India

harikri017@gmail.com

Praveen Tammana

Indian Institute of Technology

Hyderabad, India

praveent@cse.iith.ac.in

Abstract

Deploying machine learning (ML) models in programmable switch data planes facilitates low latency and high throughput traffic inference at line speed. However, data planes pose significant constraints due to the limited memory and minimal support for mathematical operations and data types. As a result, the only unsupervised ML models implemented in data planes to date are Isolation Forests (iForests). However, conventional iForest models yield suboptimal malicious traffic detection performance in various traffic use cases. To address this limitation, this paper proposes iGuard, the first iForest implementation that can accurately detect malicious traffic by incorporating the "knowledge" of more powerful autoencoders. We deploy iGuard in the form of a small set of whitelist rules that could be easily installed in the switch data planes. We implement iGuard using the P4 language, and assess its performance in an experimental platform based on Intel Tofino switches. Upon evaluating iGuard on various attack traffic use cases, our model can improve accuracy up to 48.3% while maintaining a similar or lower switch memory footprint over previous approaches to implement iForest models in real-world equipment.

CCS Concepts

- Security and privacy → Intrusion/anomaly detection and malware mitigation; Network security;
- Computing methodologies → Machine learning.

Keywords

Network security, intrusion/anomaly detection, programmable switches, software defined networking, machine learning.

*This work was done while the author was at IIT Hyderabad.

†Both authors contributed equally to this research.



This work is licensed under a Creative Commons Attribution International 4.0 License.

CoNEXT '24, December 9–12, 2024, Los Angeles, CA, USA

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1108-4/24/12

<https://doi.org/10.1145/3680121.3697807>

ACM Reference Format:

Sankalp Mittal, Harikrishnan V., Patel Heetkumar, and Praveen Tammana. 2024. iGuard: Efficient Isolation Forest Design for Malicious Traffic Detection in Programmable Switches . In *Proceedings of the 20th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '24), December 9–12, 2024, Los Angeles, CA, USA*. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3680121.3697807>

1 Introduction

The emergence of programmable switches (e.g., P4 switches [1, 9]) has introduced a new research direction aimed at low-latency and high-throughput (line-speed) traffic classification using machine learning (ML) models. Programmable switch data planes can achieve significantly higher throughput, faster packet forwarding rates, and lower latency than control planes while maintaining similar infrastructure and maintenance costs [15, 27]. However, they pose several constraints [9] due to limited switch memory [12, 25] and per-packet operations [29, 31, 32]. As a result, complex ML models (e.g., neural networks, autoencoders [20]) cannot be implemented, and only decision-tree-based models have been successfully deployed in these data planes.

Many works [2, 3, 10, 13, 17, 18, 21, 34–36, 39–44] have utilized decision tree-based supervised ML models in programmable switch data planes to detect malicious traffic (network attacks). However, supervised methods assume the presence of labeled datasets, which are costly and impractical for real-world deployment [6]. Additionally, these methods struggle to detect unseen attacks [15, 26], suffer from concept drift of anomalous samples [22], and require large-scale anomaly datasets, which are difficult to obtain [5–7]. Unsupervised ML methods, which only require normal datasets, offer a potential solution as they are more readily available in real-world scenarios [7].

To the best of our knowledge, the only work that has implemented unsupervised iForest models in programmable switch data planes is [15]. However, iForest is insufficient for high-performance malicious traffic detection across various attack scenarios (§3.1). This is because benign and malicious samples often end up in the same or nearby leaf nodes, making it challenging for iForest to distinguish between them using expected path lengths across different iTrees (§3.1).

To address the above limitation, we propose iGuard, an iForest model-based design that accurately detects malicious traffic at

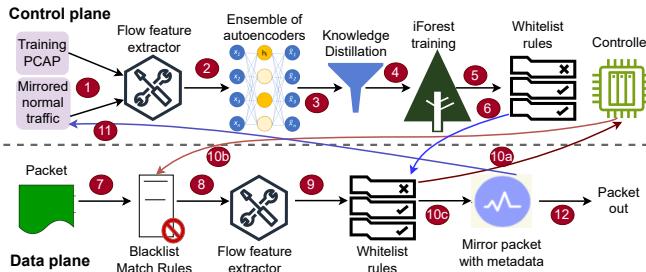


Figure 1: Overview of iGuard

line speed across various attack traffic use cases. Our approach incorporates knowledge of a trained ensemble of autoencoders¹ to train an iForest such that the autoencoders learn to distinguish malicious samples from benign samples generated at leaf nodes (autoencoder-guided iForest training, §3.2.1). Next, we propose a knowledge distillation (§3.2.2) scheme to transfer knowledge of autoencoders into the trained iForest. This involves embedding reconstruction errors from autoencoders into iForest to label each leaf node, and the final prediction for a test sample is determined by a majority vote across all iTrees. Finally, we convert our designed iForest model into a set of whitelist rules (generally a small set) that can be installed on the target switch (§3.2.3).

In short, the main contributions of our paper are as follows,

- We present a novel iForest model (iGuard) that accurately detects malicious traffic in the data plane at line speed. This model design is achieved through autoencoder-guided training (§3.2.1), knowledge distillation from autoencoders to trained iForest (§3.2.2), and conversion of iForest to a small set of whitelist rules that can be installed on target switch (§3.2.3).
- We clarify the challenges of implementing iGuard on Intel’s Tofino switch and developing a working prototype² (§3.3).
- We extensively evaluate iGuard in a real-world testbed, revealing iGuard’s increasing accuracy gains over previous implementations of iForest models [15] on various attack datasets [8, 14, 15, 23, 26] and normal datasets [15, 30] while maintaining similar or lower switch memory footprint (§4).

2 Overview of iGuard

iGuard is a novel design of switch-tailored iForest deployed entirely in the data plane (i.e., programmable switches). Further, iGuard is an efficient design of iForest to *accurately* detect malicious traffic at line speed. We present the overview of iGuard in Fig. 1.

Control plane. Using PCAP traces in the training dataset of benign traffic, we extract flow-level (FL) features³ (1). Using the extracted FL features, we train an ensemble of autoencoders (e.g., [15, 26]) as shown in (2) of Fig. 1. We then use these trained autoencoders to guide the training of the iForest model (3) and perform knowledge distillation from an ensemble of autoencoders into the trained iForest (4). Finally, as shown in (5), the trained and knowledge-distilled

¹We can show that autoencoders are best candidates for guiding iForest models to achieve higher detection accuracy (§4)

²<https://github.com/networked-systems-iith/iGuard>

³FL feature extraction mechanism in control plane and data plane is already covered in prior arts [3, 15, 37, 44]. We follow in these footsteps.

iForest is converted into a small set of whitelist rules⁴ and installed on the target switch data plane (6). For details, see §3.2.

Data plane. This module first matches the incoming packet’s flow_ID (5-tuple) in the blacklist match rules table. If matched, it can drop the malicious packets or forward them to the desired port (7). Then the module extracts FL features from the incoming packets using the flow-level feature extractor [2, 15, 44] (8). The extracted FL features are matched with whitelist rules at a timeout or flow’s per-packet count threshold (9). Regardless of the match, a digest is sent to the controller in the control plane (10a), and the controller can then install a blacklist rule (10b) for a malicious flow_ID. If there is a match, then FL features are added to the incoming packet’s metadata (10c), and the packet’s payload is truncated and mirrored to the control plane (11). FL features from benign traffic may be used to update the whitelist rules table. Finally, the packet is sent to the egress port as shown in (12).

3 iGuard System Design

3.1 Motivation

We motivate iGuard by demonstrating the inefficiency of conventional iForest models when detecting malicious traffic.

iForest obtains a label (malicious or benign) using the anomaly score of test sample x as $\text{label} = \mathbb{1}\{\text{score}(x) < \tau\}$ (1 for anomaly), $\tau > 0$, and $\text{score}(x)$ is the anomaly score⁵ of x obtained using expected path length [24]. *We show that iForest cannot distinguish malicious samples from benign ones using expected path lengths.*

We consider 15 different attacks from attack datasets [8, 14, 15, 23, 26]. FL features considered are the same as that of Magnifier [15]. We divide benign traffic dataset [15] into training and test sets (as shown in [15]). The training set is then divided into training and validation (4:1) sets, and we further add 20% of attack traffic to the validation set and test set (one attack added at a time, meaning we validate and test per each attack). We select optimal hyperparameters⁶ (t , Ψ , contamination) from a search space using *grid search* to obtain maximal macro F1 score on validation set. We plot the distribution of test set samples (benign and malicious) as a function of expected path lengths in Fig. 2 for 5 attacks. We clearly see a significant overlap across benign and malicious samples for various attacks. Therefore, we can conclude that expected path length is not an adequate metric to differentiate between attack and normal samples (conventional iForest models are inaccurate). We defer the supplementary results for 10 other attacks to Fig. 7 in Appendix.

Key takeaway. Conventional iForest models lead to inaccurate malicious traffic detection across various attacks.

3.2 iForest Design

We saw in §3.1 that conventional iForest models are inadequate for malicious traffic detection tasks. This motivates us to propose a novel design for iForest.

⁴Since the majority of traffic is benign traffic, whitelist rules help separate anomalies (malicious traffic) from the regular traffic.

⁵The anomaly score of a sample x in iForest is given by $\text{score}(x) = 2^{-\frac{\mathbb{E}(h(x))}{c(n)}}$, where $\mathbb{E}(h(x))$ is the expected path length traversed by x over all iTrees and $c(n)$ is the normalization factor based on #samples in dataset, i.e. n .

⁶ t denotes the number of iTrees, Ψ is sub-sample size, and contamination stands for contamination ratio, meaning estimated fraction of malicious samples in validation dataset (it also controls threshold τ).

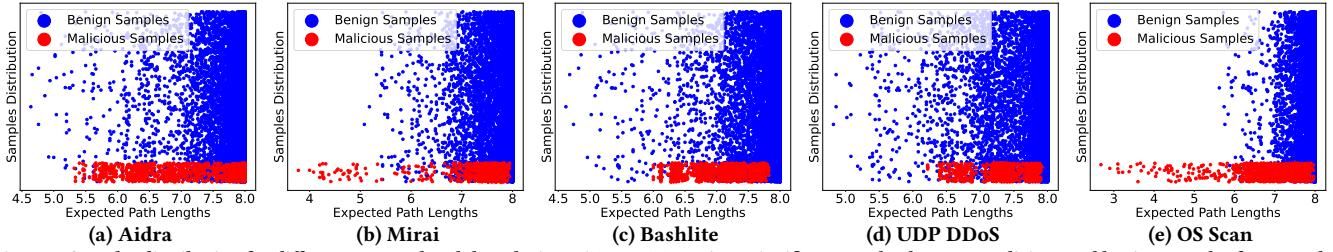


Figure 2: Samples distribution for different expected path lengths in an iForest. We notice a significant overlap between malicious and benign samples for 5 attacks. Other 10 attacks follow the same trend. Thus, iForest struggles to distinguish between malicious and benign samples.

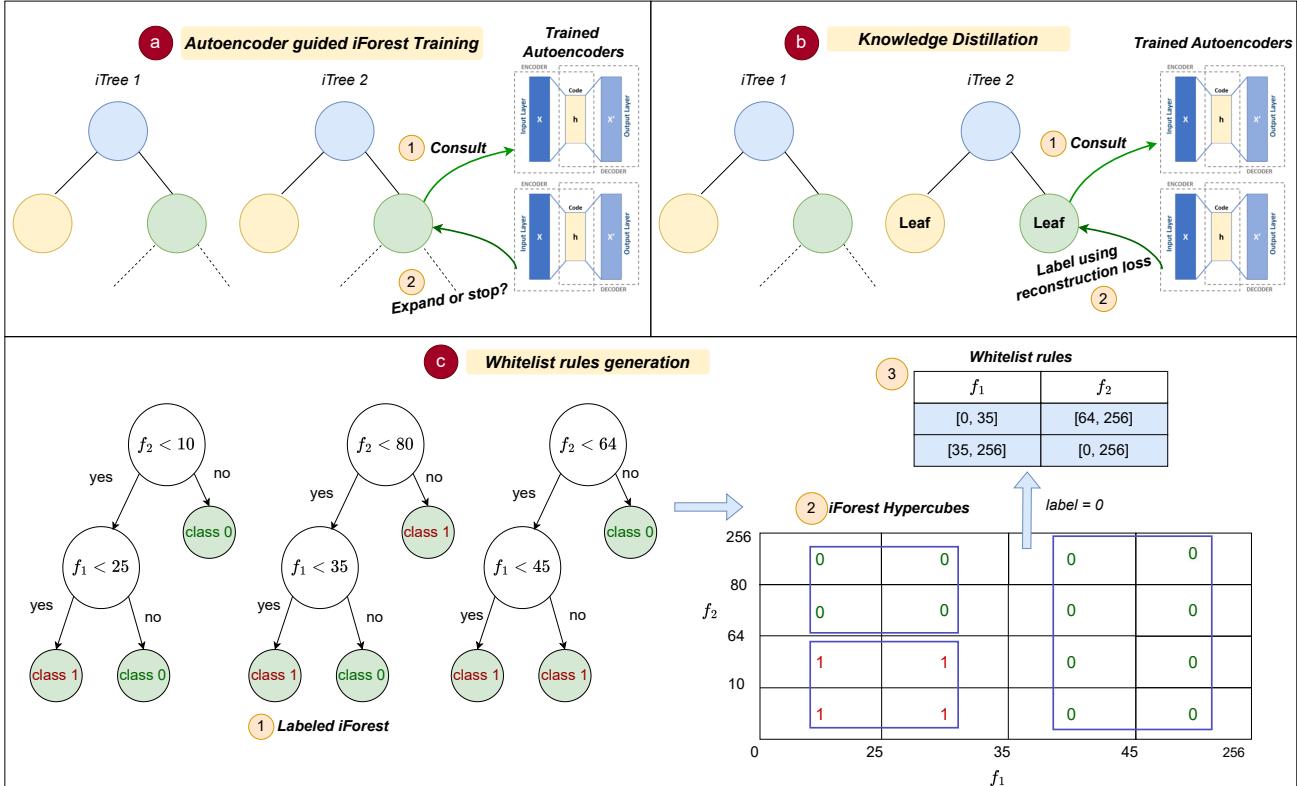


Figure 3: iForest design : (a) Autoencoder-guided iForest training (b) Knowledge distillation (c) Whitelist rules generation.

Our idea. Our key idea is to use a superior ensemble of trained (on the same features as iForest) autoencoders to label each leaf node of iForest by using *expected reconstruction error* of the samples generated from that leaf node. This will help distinguish malicious samples from benign ones that were initially not possible using expected path lengths (the autoencoders can better identify subtle differences in the feature values).

Challenge. When the vast majority of samples contained in a leaf node are of the same class, the label from autoencoders can rectify the inference result of iForest. However, when a leaf node itself contains a similar number of benign and malicious samples, using an ensemble of autoencoders to label that leaf node based on expected reconstruction errors might not be adequate for accurate malicious traffic detection.

Our approach. Our approach to tackle the above challenge is to use trained *autoencoders* to guide iForest training such that the samples generated at leaf nodes are *heavily skewed towards samples*

being either benign or malicious. Once iForest is trained this way, we can then embed the expected reconstruction errors of the samples generated at each leaf node to label that leaf node as 0 (benign) or 1 (malicious). We demonstrate our approach in detail in subsequent subsections.

3.2.1 Autoencoder guided iForest training (Fig. 3a) We first train an ensemble of r autoencoders (independently) on a benign training set consisting of m features. Then for u^{th} autoencoder AE_u , the reconstruction error $RE_u(x)$ for test sample x is given by $RE_u(x) = \sqrt{\frac{1}{m} \sum_{i=1}^m (AE_u(x)_i - x_i)^2}$.

The label assigned to sample x by autoencoder AE_u is given by $label_u(x) = \mathbb{1}\{RE_u(x) > T_u\}$, where T_u is RMSE threshold of u^{th} autoencoder AE_u . For an ensemble of r autoencoders, $Autoencoders.predict(x) = \mathbb{1}\{\sum_{u=1}^r w_u \times label_u(x) > 0.5\}$, where $w_u \in [0, 1]$ is weight given to AE_u and $\sum_{u=1}^r w_u = 1$.

iForest training. A conventional iForest is trained by ensembling t iTrees, where each iTree is formed from Ψ randomly selected sub-samples from the training set. At every node of iTree, a feature

$q \in Q$ is randomly selected from features set Q , and a split point p is randomly selected lying between the minimum and maximum value of feature q . The left and right subtrees are recursively created using $q < p$ and $q \geq p$ until number of samples in a node $|X_{node}| \leq 1$ or height of the tree $h \geq \lceil \log_2(\Psi) \rceil$. We make two key changes to this algorithm: node expansion i.e. how to expand an iTree recursively, and when this algorithm needs to be stopped.

Node expansion criterion. At each node of an iTree, we have associated feature ranges or feature boundaries. Also, we have X_{node} samples associated with that node and $|X_{node}| \leq \Psi$. We additionally perform data augmentation to generate k additional points from that node (sampled from a distribution⁷ from features range) as $X_{aug} \sim features_range$, $|X_{aug}| = k$. Expanding an iTree from a node into left and right subtrees depends on samples $X_{decision} = X_{node} \cup X_{aug}$. Next, we use autoencoders to separate out malicious samples from benign samples in $X_{decision}$ as follows,

$$X_{malicious} = \{x \in X_{decision} \mid Autoencoders.predict(x) = 1\} \quad (1)$$

and $X_{benign} = X_{decision} \setminus X_{malicious}$. Let $pr = \frac{|X_{malicious}|}{|X_{decision}|}$. Then entropy at the node is given by,

$$H(node) = -pr \log_2(pr) - (1 - pr) \log_2(1 - pr) \quad (2)$$

We then explore all possibilities of search space of (q, p) , where $q \in Q$ is a feature in features set Q , and p is a possible value for feature q where iTree is split into left and right subtrees. If we split an iTree at (q, p) , then $X_{left} = X_{decision}[q < p]$ and $X_{right} = X_{decision}[q \geq p]$. Let $w_{left} = \frac{|X_{left}|}{|X_{decision}|}$. Then,

$$H(node.children) = w_{left} \cdot H(node.left) + (1 - w_{left}) \cdot H(node.right) \quad (3)$$

where $H(node.left)$ and $H(node.right)$ are calculated using Eq(1) and Eq(2), except in place of $X_{decision}$, X_{left} and X_{right} are used. Finally, we split on feature q^* at value p^* based on,

$$(q^*, p^*) = \underset{(q,p)}{\operatorname{argmax}} H(node) - H(node.children) \quad (4)$$

The Eq(4) is in line with the logic of splitting an iTree into left and right subtrees recursively to obtain a maximal split between malicious and benign samples at the leaf node. We recursively repeat the same node expansion procedure on the left and right child of the node by giving them $X_{node.left} = X_{node}[q^* < p^*]$ and $X_{node.right} = X_{node}[q^* \geq p^*]$ samples respectively. The node expansion continues until any one of the stopping criteria at a node is met (as discussed below).

Stopping criterion. The criteria to stop an iTree's node expansion is when either $|X_{node}| \leq 1$, current height of iTree at that node $h \geq \lceil \log_2(\Psi) \rceil$, or $\min(|X_{malicious}|, |X_{benign}|) < \tau_{split}$ ⁸. The last criterion denotes that the vast majority of samples at a node are either malicious or benign, and τ_{split} is the sample split threshold. Any node satisfying any one of the 3 stopping criteria becomes a leaf node.

Key takeaway. An iTree is expanded recursively based on maximum information gain (entropy loss) until a maximum height is

⁷We found that sampling from a normal distribution with mean as average of feature boundaries and standard deviation as quartile range of feature boundaries worked.

⁸We found that $\tau_{split} = 10^{-2}$ worked well for our evaluation.

reached or a vast majority of samples at a node are skewed towards being malicious or benign (as per the prediction from a trained ensemble of autoencoders).

3.2.2 Knowledge distillation (Fig. 3b) Once our iForest is trained, we need to transfer knowledge of autoencoders into the leaves of iForest. This is done as follows. We traverse a sample $x \in X_{train}$ on each of the t iTrees and reach t respective leaf nodes. This way, we map x to a leaf in every iTree. We repeat for all $x \in X_{train}$. Let the samples mapped to the leaf node be $X_{leaf} \subseteq X_{train}$. For every leaf, we also add k additional data points sampled from an arbitrary distribution $X_{aug} \sim features_range(leaf)$ and update for each leaf $X_{leaf} \leftarrow X_{leaf} \cup X_{aug}$. Now, we embed expected reconstruction errors into each leaf and transform them into a label. The expected reconstruction error for each leaf node is given by,

$$RE_{leaf_u} = \frac{1}{|X_{leaf}|} \sum_{x \in X_{leaf}} RE_u(x) \quad (5)$$

The expected reconstruction error is then transformed into a label for every leaf node as follows,

$$label_{leaf} = \mathbb{1}\left\{\sum_{u=1}^r w_u \times \mathbb{1}\{RE_{leaf_u} > T_u\} > 0.5\right\} \quad (6)$$

iForest inference. Given a test sample x_{test} , we traverse each of the t iTrees and end up at t leaves (one in each iTree). We then retrieve labels from each leaf node and take the *majority vote* over all the t iTrees. That is, $label(x_{test}) = majority_vote(label_{leaf} \text{ from } t \text{ leaves})$.

Time complexity overhead. In our design, the primary time complexity overhead stems from the autoencoder-guided iForest training step. This step contributes to the worst-case time complexity for training iGuard's model, which is more aligned with a random forest approach rather than a standard iForest. The added complexity arises from maximizing information gain by identifying the optimal feature split.

3.2.3 Whitelist rules generation We show the whitelist rules generation from the trained iForest design of iGuard in Fig. 3c. As shown, iForest hypercubes are formed from labeled iForest by considering all combinations of feature ranges (obtained from all root-to-leaf paths of iTrees), i.e., a cartesian product of all feature boundaries at leaf nodes. Next, we randomly select a sample inside an iForest hypercube and infer its class using our labeled iForest. We repeat the process for all hypercubes. Further, we can merge adjacent hypercubes sharing the same label (as shown by purple boxes in Fig. 3c). Finally, hypercubes with $label = 0$ are transformed into whitelist rules which can be easily installed on the target switch data plane. It is worth noting that *each sample lying inside an iForest hypercube will share the same label*.

Results. To check the fidelity of the whitelist rules generated, we use consistency C given by, $C = \frac{1}{N} \sum_{i=1}^N \mathbb{1}\{iForest_{distilled}(x_i) = R(x_i)\}$, where R is the set of whitelist rules. We obtain the consistency on distilled iForest design based on optimal hyperparameter configuration (see §3.1) and average across all 15 attacks. We get $C = 0.992$ to 0.996 , which demonstrates that our whitelist rules generation retains the original performance of distilled iForest.

3.3 Switch-tailored Implementation

To implement iGuard on a programmable switch, we must tailor our model as per the switch requirements and constraints.

3.3.1 Switch-tailored model design We encounter the following key challenges while deploying our iForest model on programmable switch data planes.

Limited stateful memory. Due to limited stateful memory and high traffic volume, it is not feasible to maintain stateful flow-level (FL) features persistently as they can be untimely overwritten due to storage hash collisions [2, 44] (it is also not practical for Tbps switches). Therefore, to reduce long-term resource consumption of switches by keep-alive traffic and handle hash collisions, we set per-flow packet count threshold n (FL features are maintained until $PktCount \geq n$) based on pdf of per-flow packet counts [2, 3, 15]. We also set a timeout, meaning a flow's stateful storage should be released when the flow is idle for a duration more than δ [3, 15, 44]. To further mitigate hash collisions and enable bi-directional flow indexing, we use the bi-hash algorithm and double hash tables [15].

In summary, we tailor our iForest model to be trained on FL features truncated at packet count threshold n^9 and timeout δ .

Early packets are ignored. Extracting FL features in the switch can cause delay, and therefore *early malicious packets (that remain ignored) of a flow may flood into the network and harm it* [2]. To avoid this, we train a conventional iForest only on the packet-level (PL) features of the early packets of flows, generate whitelist rules, and merge these rules with the whitelist rules of our labeled iForest. This way, the final set of whitelist rules may be able to distinguish early packets as malicious/benign before the packet count or timeout threshold of a flow is reached.

3.3.2 Data plane implementation We show the data plane implementation of iGuard in Fig. 4 through 6 possible packet execution paths (represented in different colors).

Red path. If the incoming packet's 5-tuple matches with the blacklist rules table (initially empty and updated by the control plane), then we can block the deemed malicious packet early.

Brown path. If the incoming packet does not match the blacklist rules table, is not under collision, and is 1 to $n - 1^{th}$ packet of a flow and there is no timeout, then we update the stateful storage but match only the packet's PL features with whitelist rules (FL features not yet reliable). Based on the outcome, we can decide whether to forward or drop it.

Blue path. If the incoming packet does not match the blacklist rules table, is not under collision, and the packet is either n^{th} packet of a flow or there is a timeout, then we update the stateful storage and match PL+FL features with the whitelist rules. A digest is sent to the controller to install the blacklist rule (if no match), clear the storage, and the packet is mirrored to the loopback port to update the class (0 or 1). Moreover, in egress, the packet is also mirrored

⁹Some malicious flow samples may manifest after the packet count threshold n , potentially causing misclassification. Attackers could also exploit this delayed manifestation. While n is a tunable hyperparameter optimized via grid search for maximum F1 score, one solution could be using 2-3 threshold points instead of a single value. We would prefer to block the flow as malicious if it is judged malicious on at least any one of the points. This analysis could be explored as a part of future work.

	TCAM	SRAM	sALUs	VLIWs	Stages
iForest [15]	16.47%	11.55%	19.59%	7.75%	12
iGuard	13.34%	11.51%	19.62%	7.79%	12

Table 1: Average resource consumptions in a switch across all 15 attacks. The comparison of iGuard is made to previous iForest implementations [15].

to the CPU to update whitelist rules using FL features of benign traffic.

Orange path. If the packet's 5-tuple does not match with blacklist rules and there is a collision, flow label storage is checked. If it is either 0 or 1 (residing flow is classified), then we clear and initialize the stateful storage with the incoming packet's header values, match PL features with whitelist rules, and mirror the packet to the loopback port (to initialize flow ID). Otherwise, if flow label storage is -1 (residing flow not yet classified), then we match the packet's PL features to whitelist rules and take an appropriate decision.

Purple path. If the packet's 5-tuple does not match with blacklist rules, there is no collision, and flow label storage is either 0 or 1, then we can take the decision early, whether to forward this packet or drop it.

Green path. If the packet is a mirrored packet on the loopback port, then we take the decision based on its metadata value. If there was a timeout, then we update the flow label storage and match PL features with whitelist rules (because that latest packet was unaccounted for). If it was n^{th} packet, we simply modify the flow label storage. If there was a collision, we modify the flow ID storage.

Controller. Once the controller receives the digest (when the flow class is determined in the data plane), it clears the stateful storage based on the flow indexed by the 5-tuple it received. Moreover, if the flow classified was malicious, the controller installs an appropriate rule in the blacklist rules table. The controller can also delete old rules from the blacklist table based on FIFO or LRU [44].

4 Evaluation

We use attack datasets [8, 14, 15, 23, 26] and normal datasets [15, 30]. The division of normal/benign datasets into training and test datasets is entailed in HorusEye [15]. The normal training set is further divided into training and validation in a 4 : 1 ratio. In both validation and test set, 20% attack traffic is added (one attack at a time). Best configuration is obtained using validation set while final results (§4.1, §4.2) are obtained using the test set.

To select the best candidate for guiding iForest and knowledge distillation, see Appendix §A.

4.1 CPU Experiments

We implement iGuard and conventional iForest models in python3 on 40-core, 2 x Intel(R) Xeon(R) Silver 4316 CPU @ 2.30GHz, and 256GB DDR4 memory. We use state-of-the-art autoencoder Magnifier [15] to train and perform knowledge distillation in iGuard.

We compare macro F1 score, PR_{AUC} and ROC_{AUC} (area under PR and ROC curve) of iGuard with iForest on 5 different attacks as shown in Fig. 5. FL features used in both are similar to those of the autoencoder Magnifier. Both iGuard and iForest's best versions¹⁰ are implemented. iGuard yields higher macro F1 score, PR_{AUC} and

¹⁰For iGuard the best version is selected based on grid search on (t, Ψ, k, T) for maximum value of average of macro F1 score, PR_{AUC} and ROC_{AUC}. k is the number of data points augmented from the nodes during training and knowledge distillation,

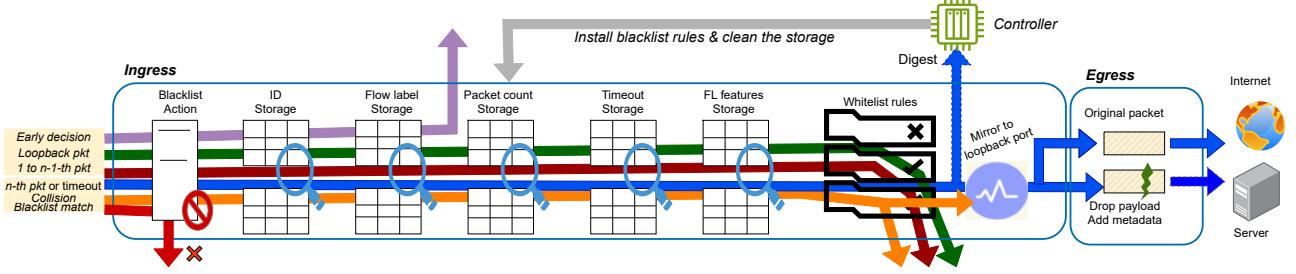


Figure 4: iGuard’s packet-processing in the data plane

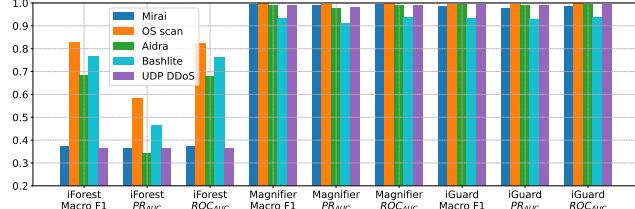


Figure 5: Detection performance comparison of iGuard with iForest and Magnifier [15] on CPU.

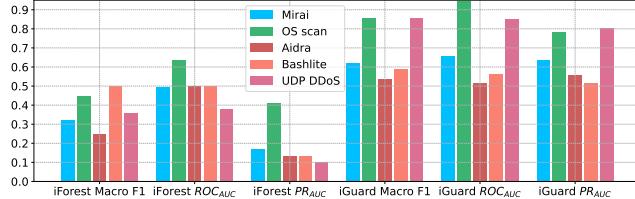


Figure 6: Detection performance comparison of iGuard with iForest on Tofino switch tested.

ROC_{AUC} (but similar to Magnifier) compared to iForest by 1.8–62.9%, 5.7–72.2% and 1.8–62.8% respectively. This is because as shown in §3.1, iForest struggles to distinguish malicious samples from benign based on expected path lengths. This is rectified by Magnifier which is used to train iGuard. We defer the results for 10 other attacks to Fig. 8 in the Appendix.

4.2 Testbed Experiments

We implement iGuard and iForest in P4₁₆ language and deploy them on Edgecore 32X Tofino 1 switch with a forwarding rate of 6.4 Tbps. We use tcpreplay to generate traffic from PCAP traces at 40 Gbps on a 40 Gbps link. Since all the features used by Magnifier cannot be extracted in the data planes, we only consider 13 FL features such as per-flow packet count, total/average/standard deviation/variance/min/max of packet size, average/minimum/variance/standard deviation/max of inter-packet delay, and flow duration (as in [44]). For PL features (to detect early malicious packets), we consider destination port, protocol, packet’s length and TTL. We use our custom (asymmetric) autoencoder¹¹ for training and knowledge distillation. Additional experiments in Appendix §B.

4.2.1 Detection performance. We use per-packet metrics [2] to compare best versions¹² of iGuard and iForest under the given switch memory budget. In other words, best version is based on maximizing reward given by $\frac{\alpha}{3}(F1 + PR_{\text{AUC}} + ROC_{\text{AUC}}) + (1 - \alpha)(1 - \rho)$

and T is RMSE threshold of autoencoders. For iForest, grid search is performed on $(t, \Psi, \text{contamination})$.

¹¹Data planes are unable to extract 2D statistics used by Magnifier.

¹²By performing grid search on packet count and timeout thresholds (n, δ) besides hyperparameters already mentioned in §4.1.

where ρ is a measure of memory footprint of the system, expressed as a fraction of the total available resources in the target switch. We put $\alpha = 0.5$ to balance out the two factors in our experiments.

The comparison trend (shown in Fig. 6) shows that iGuard on Tofino testbed improves macro F1 score by 5–48%, ROC_{AUC} by 2–55.7% and PR_{AUC} by 26–70% compared to previous hardware implementations of iForest [15]. The reasoning is the same as given in §4.1. Performance of iGuard on the testbed is generally lower than on the CPU because only a few FL features can be extracted on the switch data plane due to limited memory and per-packet operations. Supplementary experiments deferred to Fig. 9 (more attacks), and Table 2 and Table 3 (adversarial attacks) in Appendix.

4.2.2 Switch memory overheads. Resource consumption on the switch for iGuard is similar or lower (lower TCAM) compared to previous iForest implementations (Table 1) because of an additional stopping criterion (which is to stop the iTree growth if samples at that node are skewed towards malicious or benign). This further restricts the number of whitelist rules and reduces TCAM consumption in particular.

5 Related Work

We divide the works that perform malicious traffic detection using unsupervised ML methods into two categories.

Control plane based. Works such as [11, 19, 26, 33, 38] cannot scale to multi-Tbps because they perform attack detection (using autoencoders) in the control plane. In contrast, iGuard is implemented in data planes and thus can scale to Tbps. Although [16] uses enhanced iForest by combining X-means algorithm for anomaly detection, it does not leverage data planes.

Programmable switch based. Only works that leverage programmable switches for anomaly detection using unsupervised ML models are [4, 15]. Peregrine [4] extracts FL features in the data plane but uses KitNet [26] in the control plane for anomaly detection. On the contrary, HorusEye [15] deploys an unsupervised iForest model in the switch data plane but it takes the support of the autoencoder in the control plane for more accurate anomaly detection. iGuard follows a different approach of offloading the entire detection in data planes by coming up with a novel and efficient iForest design.

6 Conclusion and Future work

We proposed iGuard, an efficient iForest design to detect malicious traffic at line speed in data planes. iGuard leverages powerful autoencoders to rectify the attack detection of conventional iForests. We saw that iGuard is implemented in data planes as whitelist rules and can improve the accuracy of previous switch-based iForest implementations up to 48.3%. For future work, see App. §C.

Acknowledgements

We thank our shepherd and the reviewers for their insightful comments and thorough feedback. This work is supported by National Security Council Secretariat (NSCS), India.

References

- [1] 2021. Barefoot Networks, Tofino Switch. (2021).
- [2] Aristide Tanyi-Jong Akem, Beyza Bütin, Michele Guicciardo, Marco Fiore, et al. 2024. Jewel: Resource-Efficient Joint Packet and Flow Level Inference in Programmable Switches. In *IEEE Conference on Computer Communications*.
- [3] Aristide T.-J. Akem, Michele Guicciardo, and Marco Fiore. 2023. Flowrest: Practical Flow-Level Inference in Programmable Switches with Random Forests. *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications* (2023).
- [4] João Romeiras Amado, Francisco Pereira, David Pissarra, Salvatore Signorello, Miguel Correia, and Fernando Ramos. 2024. Peregrine: ML-based Malicious Traffic Detection for Terabit Networks. *arXiv preprint arXiv:2403.18788* (2024).
- [5] Giuseppina Andresini, Feargus Pendlebury, Fabio Pierazzi, Corrado Loglisci, Annalisa Appice, and Lorenzo Cavallaro. 2021. Insomnia: Towards concept-drift robustness in network intrusion detection. In *Proceedings of the 14th ACM workshop on artificial intelligence and security*.
- [6] Giovanni Apruzzese, Pavel Laskov, and Aliya Tastemirova. 2022. SoK: The impact of unlabelled data in cyberthreat detection. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*.
- [7] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2022. Dos and don'ts of machine learning in computer security. In *31st USENIX Security Symposium (USENIX Security 22)*.
- [8] Vitor Hugo Bezerra, Victor G Turrisi da Costa, Ricardo Augusto Martins, Sylvio Barbon Junior, Rodrigo Sanches Miani, and Bruno Bogaz Zarpelao. 2018. Providing IoT host-based datasets for intrusion detection research. In *Anais do XVIII Simpósio Brasileiro de Segurança da Informação Sistemas Computacionais*.
- [9] Pat Bosschart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* (2014).
- [10] Coralie Busse-Growitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. 2019. pforest: In-network inference with random forests. *arXiv preprint arXiv:1909.05680* (2019).
- [11] Z Berkay Celik, Gang Tan, and Patrick D McDaniel. 2019. Iotguard: Dynamic enforcement of security and safety policy in commodity IoT. In *NDSS*.
- [12] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, et al. 2017. drmt: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*.
- [13] Bruno Coelho and Alberto Schaeffer-Filho. 2022. BACKORDERS: using random forests to detect DDoS attacks in programmable data planes. In *Proceedings of the 5th International Workshop on P4 in Europe (EuroP4 '22)*.
- [14] F. Ding. 2017. IoT Malware. (2017). <https://github.com/fidning/iot-malware>
- [15] Yutao Dong, Qing Li, Kaidong Wu, Ruoyu Li, Dan Zhao, Gareth Tyson, Junkun Peng, Yong Jiang, Shutao Xia, and Mingwei Xu. 2023. HorusEye: A Realtime IoT Malicious Traffic Detection Framework using Programmable Switches. In *32nd USENIX Security Symposium (USENIX Security 23)*.
- [16] Yifan Feng, Weihong Cai, Haoyu Yue, Jianlong Xu, Yan Lin, Jiaxin Chen, and Zijun Hu. 2022. An improved X-means and isolation forest based methodology for network traffic anomaly detection. *Plos one* (2022).
- [17] Kurt Friday, Elias Bou-Harb, and Jorge Crichtigno. 2022. A Learning Methodology for Line-Rate Ransomware Mitigation with P4 Switches. In *International Conference on Network and System Security*.
- [18] Kurt Friday, Elie Kfoury, Elias Bou-Harb, and Jorge Crichtigno. 2022. INC: In-Network Classification of Botnet Propagation at Line Rate. In *European Symposium on Research in Computer Security*.
- [19] Chuantu Fu, Qi Li, Meng Shen, and Ke Xu. 2021. Realtime robust malicious traffic detection via frequency domain analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 3431–3446.
- [20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.
- [21] Syed Usman Jafri, Sanjay Rao, Vishal Shrivastav, and Mohit Tawarmalani. 2024. Leo: Online ML-based Traffic Classification at Multi-Terabit Line Rate. *NSDI*.
- [22] Roberto Jordaney, Kumar Sharad, Santanu K Dash, Zhi Wang, Davide Papini, Ilya Nouretdinov, and Lorenzo Cavallaro. 2017. Transcend: Detecting concept drift in malware classification models. In *26th USENIX security symposium*.
- [23] Nickolaos Koroniots, Nour Moustafa, Elena Sitnikova, and Benjamin Turnbull. 2019. Towards the development of realistic botnet dataset in the internet of things for network forensic analytics: Bot-iot dataset. *Future Generation Computer Systems* (2019).
- [24] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. 2008. Isolation forest. In *2008 eighth ieee international conference on data mining*. IEEE, 413–422.
- [25] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*.
- [26] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. 2018. Kitsune: an ensemble of autoencoders for online network intrusion detection. *Network and Distributed System Symposium (NDSS '18)* (2018).
- [27] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, et al. 2021. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*.
- [28] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* (2011).
- [29] Naveen Kr Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. 2017. Evaluating the power of flexible packet processing for network resource allocation. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*.
- [30] Arunan Sivanathan, Hassan Habibi Gharakheili, Franco Loi, Adam Radford, Chamith Wijenayake, Arun Vishwanath, and Vijay Sivaraman. 2018. Classifying IoT devices in smart environments using network traffic characteristics. *IEEE Transactions on Mobile Computing* (2018).
- [31] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*.
- [32] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. 2017. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*. 164–176.
- [33] Runming Tang, Zheng Yang, Zeyan Li, Weibin Meng, Haixin Wang, Qi Li, Yongqian Sun, Dan Pei, Tao Wei, Yanfei Xu, et al. 2020. ZeroWall: Detecting zero-day web attacks through encoder-decoder recurrent neural networks. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*.
- [34] Bruno Missi Xavier, Rafael Silva Guimaraes, Giovanni Comarella, and Magnos Martinello. 2021. Programmable Switches for in-Networking Classification. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*.
- [35] Guorui Xie, Qing Li, Yutao Dong, Guanglin Duan, Yong Jiang, and Jingpu Duan. 2022. Mousika: Enable general in-network intelligence in programmable switches by knowledge distillation. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*.
- [36] Guorui Xie, Qing Li, Guanglin Duan, Jiaye Lin, Yutao Dong, Yong Jiang, Dan Zhao, and Yuan Yang. 2024. Empowering In-Network Classification in Programmable Switches by Binary Decision Tree and Knowledge Distillation. *IEEE/ACM Transactions on Networking* (2024).
- [37] Yifan Yuan, Omar Alama, Jiawei Fei, Jacob Nelson, Dan R. K. Ports, Amedeo Sapio, Marco Canini, and Nam Sung Kim. 2022. Unlocking the Power of Inline Floating-Point Operations on Programmable Switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*.
- [38] Wei Zhang, Yan Meng, Yugeng Liu, Xiaokuan Zhang, Yinqian Zhang, and Haojin Zhu. 2018. Homonit: Monitoring smart home apps from encrypted traffic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*.
- [39] Xiaoquan Zhang, Lin Cui, Fung Po Tso, and Weijia Jia. 2021. pHHeavy: Predicting heavy flows in the programmable data plane. *IEEE Transactions on Network and Service Management* (2021).
- [40] Changgang Zheng, Zhaoqi Xiong, Thanh T Bui, Siim Kaupmees, Riyad Bensoussane, Antoine Bernabeu, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. 2022. Illy: Practical in-network classification. *arXiv preprint arXiv:2205.08243* (2022).
- [41] Changgang Zheng, Zhaoqi Xiong, Thanh T. Bui, Siim Kaupmees, Riyad Bensoussane, Antoine Bernabeu, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. 2024. Illy: Hybrid In-Network Classification Using Programmable Switches. *IEEE/ACM Transactions on Networking* (2024).
- [42] Changgang Zheng, Mingyuan Zang, Xinpeng Hong, Riyad Bensoussane, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. 2022. Automating in-network machine learning. *arXiv preprint arXiv:2205.08824* (2022).
- [43] Changgang Zheng and Noa Zilberman. 2021. Planter: seeding trees within switches. In *Proceedings of the SIGCOMM'21 Poster and Demo Sessions*.
- [44] Guangmeng Zhou, Zhoutao Liu, Chuantu Fu, Qi Li, and Ke Xu. 2023. An Efficient Design of Intelligent Network Data Plane. In *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association.

APPENDIX

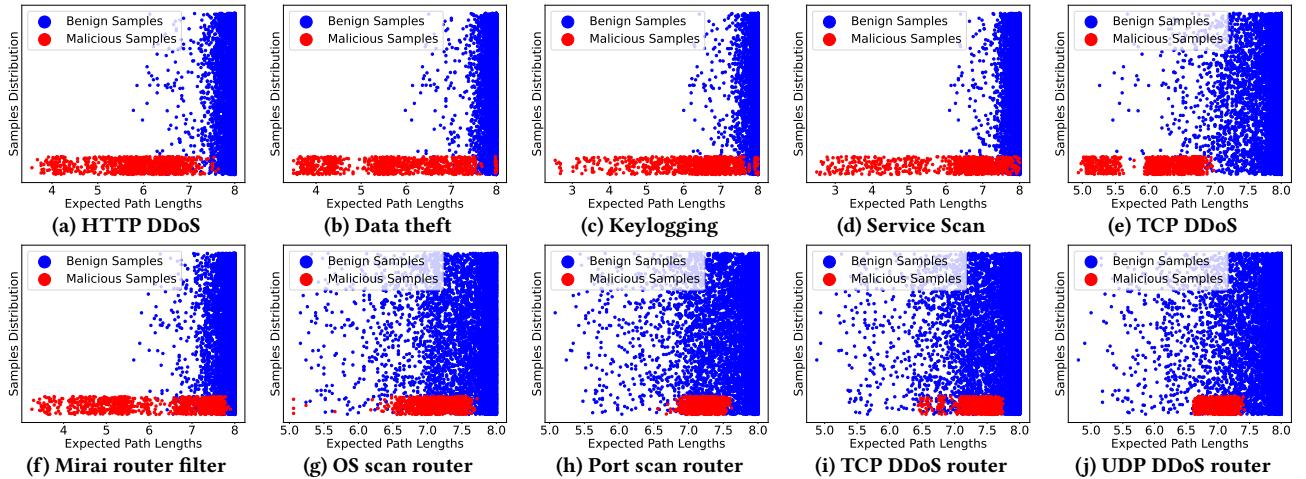


Figure 7: Samples distribution for different expected path lengths in an iForest. We notice a significant overlap between malicious and benign samples for 10 attacks. Thus, iForest struggles to distinguish between malicious and benign samples.

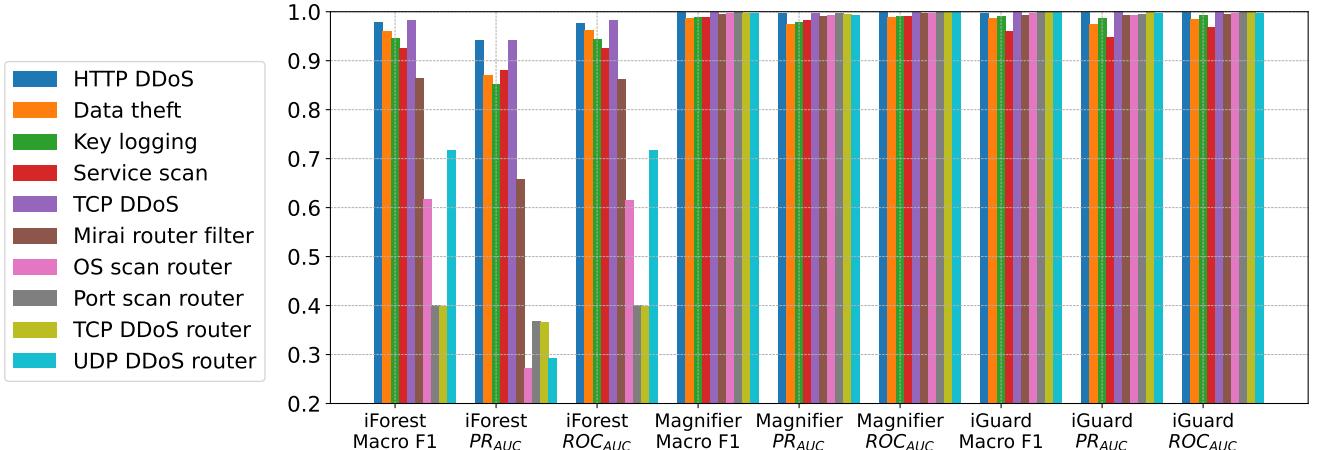


Figure 8: Detection performance of iGuard compared to iForest and Magnifier for 10 attacks on CPU. iGuard yields similar metrics compared to Magnifier but compared to iForest it improves macro F1 score by 1.8-62.9%, PR_{AUC} by 5.7-72.2% and ROC_{AUC} by 1.8-62.8% respectively

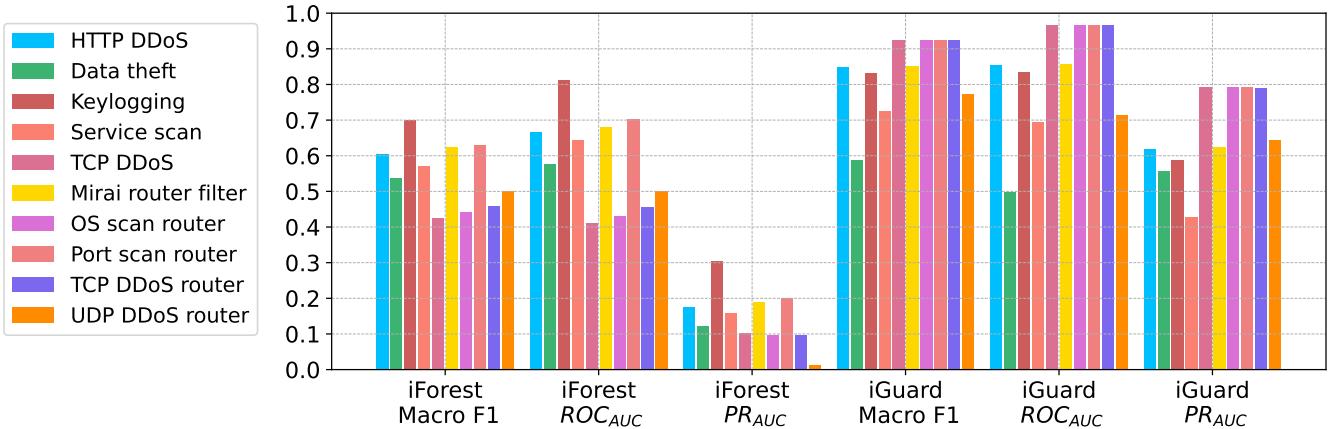


Figure 9: Detection performance of iGuard compared to iForest for 10 attacks on Tofino switch testbed. Improvement of macro F1 score by 5-48.3%, PR_{AUC} by 26-70% and ROC_{AUC} by 2-55.7% respectively.

	Low rate (UDPDDoS 1/100)	Low rate (TCPDDoS 1/100)	Poison (Mirai 2%)	Poison (Mirai 10%)
iForest [15]	43.43%/44.42%/14.92%	57.43%/57.5%/23.8%	28.52%/29.56%/14.78%	15.55%/18.56%/6.24%
iGuard	65.92%/66.67%/59.01%	88.84%/89.12%/70.93%	65.75%/61.56%/30.54%	65.21%/61.5%/30.06%

Table 2: Detection performance of iGuard (in terms of macro F1/ROC_{AUC}/PR_{AUC}) compared to iForest for black-box low rate and poison adversarial attacks [15]. There is an improvement up to 22% - 57%.

	Evasion (UDPDDoS 1:2)	Evasion (TCPDDoS 1:2)	Evasion (UDPDDoS 1:4)	Evasion (TCPDDoS 1:4)
iForest [15]	33.33%/34.45%/20.51%	38.83%/39.68%/20%	40.52%/41.11%/28.87%	42.26%/42.62%/19.2%
iGuard	72.23%/78.85%/70.51%	100%/100%/100%	72.12%/77.55%/68.82%	87.23%/81.43%/68.39%

Table 3: Detection performance of iGuard (in terms of macro F1/ROC_{AUC}/PR_{AUC}) compared to iForest for black-box adversarial evasion attacks [15]. There is an improvement up to 30% - 80%.

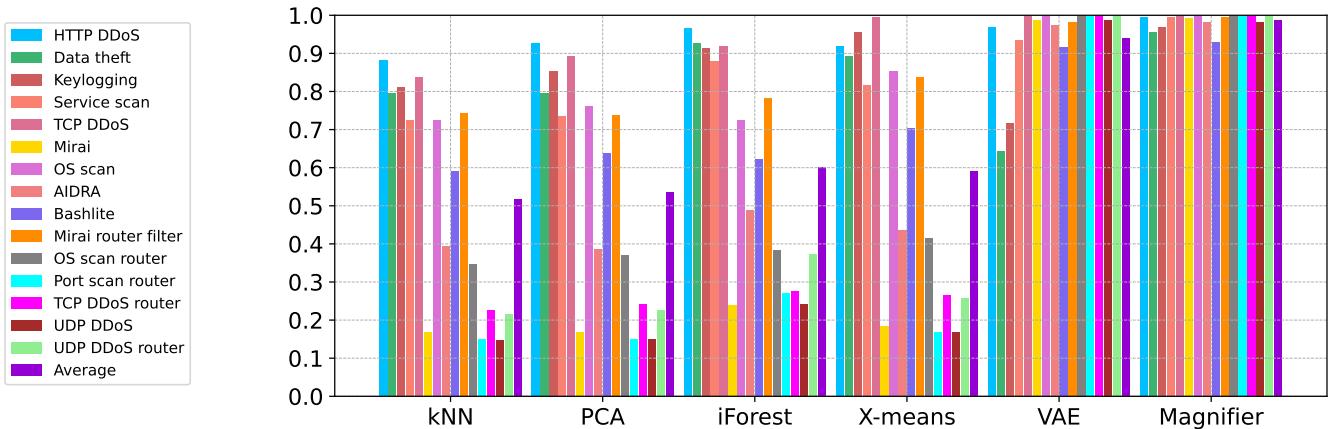


Figure 10: We compare macro F1 score on the test set among best versions (fine-tuned on the validation set) of k-NN, PCA, iForest, X-means, Variational Autoencoder, and Magnifier for 15 different attacks. We used the features similar to that of Magnifier. The architecture of VAE was similar to Magnifier, except for the use of asymmetry and dilated convolutions [15]. We notice that Magnifier outperforms all other models and therefore is chosen as a candidate to guide iForest to be deployed on switch data planes.

A Candidates for iGuard

Candidates for guiding iForest and knowledge distillation. We compare the macro F1 score among the following unsupervised ML models: iForest, PCA, k-Nearst neighbors (kNN), variational autoencoders (VAEs) and Magnifier (asymmetric autoencoders). We find out the best performers to be VAEs and Magnifier, out of which Magnifier surpassed VAE in all but one of 15 attacks. Therefore, we choose Magnifier as the final candidate for performing experiments using iGuard. See Fig. 10.

B Supplementary Experiments

B.1 Throughput and latency

On a 40 Gbps link, iGuard yields a packet processing throughput (averaged across all 15 attacks) of 39.6 Gbps. This is an improvement of 66.47% over HorusEye [15] as HorusEye makes use of a control plane for anomaly detection. We observe that the average per-packet latency of iGuard across 15 attacks is 532.8ns.

B.2 Control plane overhead

Whenever a flow class (benign/malicious) is determined by iGuard in the data plane, a digest is sent to the control plane carrying flow ID in the form of 5-tuple (13B) and a flow label (1-bit). Assuming the existence of 50k digests during a window of 30 seconds, iGuard rate of control plane overhead is 21 KBps. In contrast, recent works [4, 15] that use programmable switch-based unsupervised ML methods need an extra ~ 52 B per digest as FL features to perform detection in the control plane, an overhead of 110 KBps, which is 5.2x more than iGuard. Therefore, the control plane interactions are managed efficiently.

C Future Work

As a part of future work, we will add our new iForest design as a module in scikit-learn [28]. We will also try to improve the detection performance of iGuard in data planes. Lastly, we will try to reduce the time complexity overhead to train iGuard.