# HorusEye: A Realtime IoT Malicious Traffic Detection Framework using Programmable Switches

Yutao Dong[1,2], Qing Li [2*], Kaidong Wu[1,2], Ruoyu Li[1,2], Dan Zhao[2], Gareth Tyson[3], Junkun Peng[1,2], Yong Jiang[1,2], Shutao Xia[1,2] and Mingwei Xu[4]

[1] *Tsinghua Shenzhen International Graduate School, Shenzhen, China*
[2] *Peng Cheng Laboratory, Shenzhen, China*
[3] *Hong Kong University of Science and Technology (GZ), Guangzhou, China*
[4] *Tsinghua University, Beijing, China*

## Abstract

The ever-growing volume of IoT traffic brings challenges to IoT anomaly detection systems. Existing anomaly detection systems perform all traffic detection on the control plane, which struggles to scale to the growing rates of traffic. In this paper, we propose HorusEye, a high throughput and accurate two-stage anomaly detection framework. In the first stage, preliminary burst-level anomaly detection is implemented on the data plane to exploit its high-throughput capability (e.g., 100Gbps). We design an algorithm that converts a trained iForest model into white list matching rules, and implement the first unsupervised model that can detect unseen attacks on the data plane. The suspicious traffic is then reported to the control plane for further investigation. To reduce the false-positive rate, the control plane carries out the second stage, where more thorough anomaly detection is performed over the reported suspicious traffic using flow-level features and a deep detection model. We implement a prototype of HorusEye and evaluate its performance through a comprehensive set of experiments. The experimental results illustrate that the data plane can detect 99% of the anomalies and offload 76% of the traffic from the control plane. Compared with the state-of-the-art schemes, our framework has superior throughput and detection performance.

## 1 Introduction

The number of Internet of Things (IoT) connections is increasing dramatically and is expected to reach 24.6 billion by 2025 (more than doubled the number in 2019 [16]). However, many IoT devices are unable to deploy complex security mechanisms due to their limited hardware. Thus, they have become a major target for attackers [1]. For example, most IoT devices (e.g., cameras, lights sensor) are exposed outdoors, making it easy for hackers to attack through physical connections. Due to a large number of deployed devices (e.g. cameras, sensors), achieving low-cost and real-time anomaly detection on such a massive scale is challenging.

Traditionally, rule-based systems are utilized for anomaly detection with high throughput requirements [12]. Since flow-level rules are high cost to be deployed, network operators commonly use packet-level rule-based detection systems (e.g., a port-based and signature-based firewall [28, 30]), supplemented by offline sampling analysis [33]. However, such detection systems fail to detect unseen attacks and can be easily bypassed [12, 26].

One promising avenue of investigation is the use of unsupervised learning. A number of works perform anomaly detection using this, such as [26, 37], which explicitly claim the ability of discerning unseen attacks. For instance, in [26], the authors deploy an ensemble of autoencoders to distinguish between normal and abnormal traffic patterns. However, these methods are designed to deploy detection on the control plane, which cannot operate at a sufficient throughput to reflect practical IoT scenarios [29]. Moreover, abnormal traffic usually constitutes a very small portion of the entire traffic. Uploading all traffic to the control plane for detection causes excessive communication overhead and is highly inefficient.

The emergence of programmable switches brings about a new perspective on anomaly detection. Compared with the control plane, programmable switches (e.g., P4 switches [7]) can achieve higher throughput (20x), lower latency (20x), and faster packet forwarding rate (75x) at the same cost [29]. Thus, various tasks have been offloaded to the switch data plane for line-speed processing, e.g., in-network intelligence [20, 22, 39, 42, 43, 45, 47]. The key limitation is that programmable switches only allow simple instructions (e.g., integer additions and bit shifts) to guarantee high-speed processing. Hence, existing works focus on decision tree (DT) [42, 43] or threshold-based methods [22, 45], which can easily transform into switch rules. However, DT is a supervised method that requires a large-scale anomaly dataset. Unfortunately, high-quality network intrusion anomaly datasets are difficult to obtain [3], not to mention the need to deal with the drift of anomalous samples. In contrast, unsupervised meth-

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

ods only need to maintain normal datasets, which are easy to obtain in real-world scenarios.

In this paper, we propose a two-stage IoT anomaly detection framework, named HorusEye,[1] with high-throughput processing and powerful detection capabilities. In the first stage, we design an unsupervised model on the data plane, called Gulliver Tunnel,[2] which filters out a small amount of suspicious traffic at line speed. In the second stage, we propose a novel unsupervised deep learning model, named Magnifier, which is deployed on the control plane. This further investigates the traffic flagged as suspicious (in the first ) and produces more accurate detection results. To design HorusEye, we must overcome three key challenges: ***(i)*** It is difficult to deploy an unsupervised model with both high anomaly recall and offloading capabilities on a programmable switch that only supports simple instructions and has limited resources; ***(ii)*** it is challenging to extract and maintain the required flow features on the limited switch memory (e.g., 120 Mb SRAM); ***(iii)*** it is challenging to achieve a low false-positive rate using a high throughput deep model, since the control plane is a major throughput bottleneck.

To solve these challenges, first, we propose a novel algorithm to convert an isolation forest (iForest) [21] with many isolation trees (iTrees) into a small set of white list rules, which can be easily deployed on the programmable switches. Notably, this is the first solution to deploy a powerful yet lightweight unsupervised model on a switch. Then, we design an IoT flow feature extraction scheme that can be deployed on programmable switches. The bi-hash and double hash table methods are proposed to match bidirectional flow with only $O(1)$ computational complexity, based on which burst-based features can be obtained to distinguish abnormal behavior. Finally, we design an asymmetric lightweight autoencoder and implement a model quantization on the control plane, thereby achieving a throughput of millions of packets per second without significantly degrading detection performance.

We implement the prototype of HorusEye.[3] and conduct comprehensive experiments on a real IoT testbed. The results show that HorusEye can achieve single-port 100Gbps detection on the switch. It also exhibits excellent anomaly detection accuracy, achieving a recall rate as high as 99%. Moreover, HorusEye can offload 76% of the normal traffic away from the control plane. Compared with the state-of-the-art schemes, Kitsune [26] and Mousika [42], HorusEye has superior throughput and detection performance.

## 2 Threat Model

This paper mainly concerns the cyber threats initiated by compromised IoT devices. Due to the prevalent vulnerability of IoT systems, attackers can get unauthorized access and transform the devices into part of a botnet. A compromised device can be controlled by the botmaster to launch a DDoS attack, leak sensitive data, scan and infect more devices and conduct other malicious activities. Additionally, certain assumptions are necessarily made. (*i*) A newly produced IoT device is initially benign and trusty, and no malware or backdoor is pre-installed. (*ii*) All active attacks from IoT bots leave traces in and above the IP layer traffic. Accordingly, attacks like eavesdropping attacks [27] or MAC spoofing are not considered.

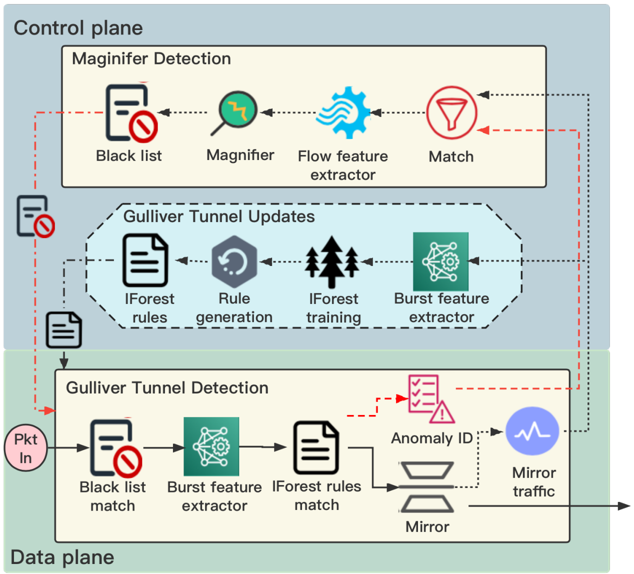


Figure 1: Overview of HorusEye.

## 3 Overview of HorusEye

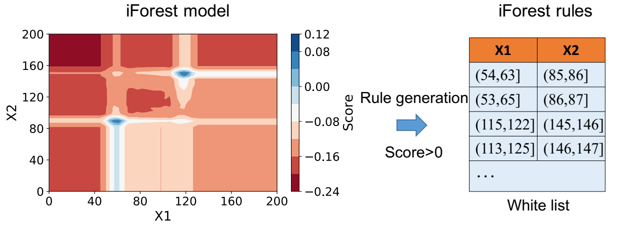
HorusEye is a novel two-stage IoT anomaly detection framework consisting of Gulliver Tunnel and Magnifier, deployed on the data plane (i.e., the programmable switches) and the control plane (e.g., x86 server), respectively, as shown in Figure 1. By taking advantage of the high throughput of the data plane, Gulliver Tunnel conducts preliminary anomaly detection over the high-speed incoming traffic and reports any suspicious traffic to the control plane. Gulliver Tunnel mainly aims at filtering as much normal traffic as possible to alleviate the burden of the control plane. The control plane then further investigates the traffic reported using a more powerful anomaly detection model, i.e., Magnifier.

**Gulliver Tunnel detection.** Gulliver Tunnel first extracts burst-level features from the incoming traffic using the burst feature extractor (§ 4.2). It detects anomalies based on the iForest model. However, since the iForest consists of a large number of iTrees, it is impractical to deploy the trained iFor-

[1] The Eye of Horus is believed to have a protective magical power that can turn away harm or evil influences.

[2] Gulliver Tunnel is believed to transform objects from large to small.

[3] https://github.com/vicTorKd/HorusEye.



| X1 | X2 |
|---|---|
| (54,63] | (85,86] |
| (53,65] | (86,87] |
| (115,122] | (145,146] |
| (113,125] | (146,147] |
| . . . | |

Figure 2: Transformation of a trained iForest with two features into white list rules on the programmable switch.

est directly on the programmable switch due to the limited resources. Therefore, Gulliver Tunnel transforms the trained iForest model into white list rules using a novel rule generation algorithm (§ 4.1). The generated rules on the switch produce almost the same detection results as the original model, while requiring only integer match operations, as supported by the data plane. After the data plane anomaly detection, Gulliver Tunnel sends the suspicious flow ID to the control plane for a more precise investigation.

**Magnifier detection.** Upon receiving the suspicious flow ID from the data plane, Magnifier (on the control plane) first matches the flow ID with those in its suspicious flow table. If the flow's suspicious anomaly frequency exceeds a threshold, Magnifier further analyzes the flow pattern from the historical mirrored traffic using a lightweight deep-learning model. Finally, Magnifier adds the confirmed malicious flow ID to the blacklist on the data plane to block or limit the speed of this flow in the future. In addition, we update the unsupervised deep model and the iForest model on the control plane when new flow information is obtained.

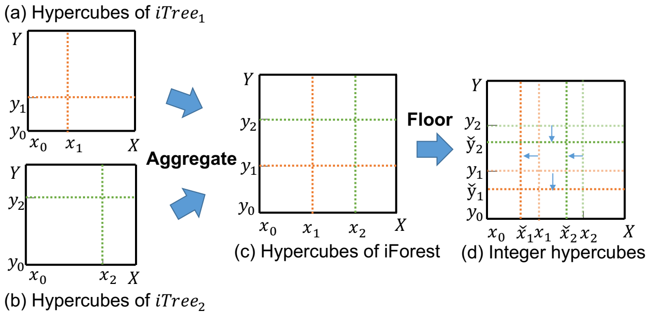
In summary, HorusEye integrates the respective advantages of the two modules to achieve high throughput and precision anomaly detection. We next describe Gulliver Tunnel (§4) and Magnifier (§5) modules in detail.

## 4 Gulliver Tunnel

In Gulliver Tunnel, we propose an efficient algorithm to generate a series of rules from the iForest model. We design a feature extraction algorithm considering the limitations of switch resources. We also analyze the complexity of the whole module and implement the prototype in P4.

### 4.1 IForest Rule Generation

We propose a novel rule generation algorithm, which converts a trained iForest with any number of iTrees into white list rules, as illustrated in Figure 2. These can then be installed on the data plane. Such rules are interpretable and can be combined with expert knowledge.



(a) Hypercubes of $iTree_1$

(b) Hypercubes of $iTree_2$

(c) Hypercubes of iForest

(d) Integer hypercubes

Figure 3: An example of the rule generation algorithm (with 2 features). Each iTree divides the hyper-dimension feature space into hypercubes. The algorithm aggregates all iTrees' divisions to form the iForest's hypercubes.

An iForest is formed by ensembling $t$ iTrees, as shown in Algorithm 2 (in Appendix), where each iTree is generated by Algorithm 3 (in Appendix) using $\psi$ sub-samples randomly selected from the samples. The iTree randomly selects a feature to branch the tree using a random threshold within the valid range of this feature. This step generates a hyperplane that divides the current feature space into two sub-spaces. Then, the same branching process is repeated recursively until a child has a single sample, or reaches the maximum depth. To test a data point, its anomaly score is calculated based on its average path length when traversing all iTrees. Intuitively, anomalies usually stop at sub-spaces reached within fewer branching steps (i.e., having smaller anomaly scores). Compared with the supervised method Random Forest, iForest is more dependent on the path of the branch. That is why the existing forest rule generation method [4, 11, 25, 40] using the public shortest path is not suitable for iForest.

We observe that each iTree essentially conducts a round of hyper-dimension feature space dividing. That is, each iTree divides the hyper-dimension feature space into hypercubes (called iTree hypercubes) through binary tree branching, as illustrated in Figures 3(a) and 3(b). To deploy the iForest rules on the data plane, we aggregate the space divisions, i.e., branches, of all iTrees in the iForest and obtain finer hypercubes in the hyper-dimension feature space (called iForest hypercubes), as illustrated in Figure 3(c). Specifically, the following label consistency property of each hypercube after aggregation is observed.

**Property.** *(**Label Consistency**) All samples inside an iForest hypercube have the same anomaly score.*

*Proof.* For each iTree, the labels in an iTree hypercube are consistent. Following the aggregated division, all points inside an iForest hypercube must belong to the same iTree hypercube obtained by the original dividing of each iTree, i.e., they have the same path length. As such, their average path lengths, scores and classification labels are the same as well. □

The branching thresholds in the iForest are real values. As such, the hypercubes are characterized by non-integer boundaries. However, the data plane does not support floating-point arithmetic. Fortunately, most packet-related features are inherently integers (e.g., header features and burst size). Thus, for these integer features, we can further "shift" these hypercubes slightly for integer boundaries by rounding down the branches of each feature to their nearest integers, as illustrated by Figure 3(d), as follows:

$$\breve{b}_i \leftarrow \lfloor b_i \rfloor, b_i \in \mathbf{b}, \tag{1}$$

where $\mathbf{b}$ is the set of branching thresholds of the iForest.

**Theorem 1.** *The shifting in Equation (1) does not change which hypercube an integer data point falls into, i.e., for any integer value* $\alpha$*,* $\alpha \in (b_{i-1}, b_i] \Rightarrow \alpha \in (\breve{b}_{i-1}, \breve{b}_i]$.

*Proof.* If $b_i$ is an integer, $(b_{i-1}, b_i] \subseteq (\breve{b}_{i-1}, \breve{b}_i]$. Therefore, in this case, $\alpha \in (b_{i-1}, b_i] \Rightarrow \alpha \in (\breve{b}_{i-1}, \breve{b}_i]$.

If $b_i$ is not an integer, suppose $\alpha$ does not satisfy the inference, i.e., $\alpha \in (b_{i-1}, b_i]$ but $\alpha \notin (\breve{b}_{i-1}, \breve{b}_i]$. In this case, $\alpha \in (\breve{b}_i, b_i]$. However, since $\breve{b}_i$ is the nearest integer that is smaller than $b_i$, there is no integer in the interval of $(\breve{b}_i, b_i]$. This contradicts the existence of such $\alpha$. Therefore, $\alpha \in (b_{i-1}, b_i] \Rightarrow \alpha \in (\breve{b}_{i-1}, \breve{b}_i]$.

□

After obtaining the hypercubes with integer boundaries, we need to decide the label of samples in each hypercube. According to the label consistency property, a hypercube's label can be obtained using any data point in it. Without loss of generality, for each hypercube, we choose the vertex that has the largest value, i.e., the right bound, in each feature dimension. We then use the vertex's label predicted by the iForest model as the label of the hypercube. Finally, the hypercubes marked with "normal" labels will be turned into white list rules on the data plane (Figure 1).

**Algorithm 1** *ruleGenereation*(*iTrees*)

```
Require: a set of $t$ $iTrees$
Ensure: $rules$
 1: initial $feature\_branch = \{ 'feature\_name' : [branch\_set] \}$
 2: $feature\_branch \leftarrow getFeature\_Branch(iTrees)$
 3: for $f$ in $feature\_branch.keys()$ do
 4:     $branch\_set \leftarrow Set(floor(feature\_branch[f]))$
 5:     $vertex \leftarrow vertex \otimes branch\_set$
 6: end for
 7: $sort(vertex)$
 8: $hypercubes.label \leftarrow iTrees.predict(vertex)$
 9: #Merge the spatial domains of consecutive and identical labels.
10: $rules \leftarrow range\_generation(hypercubes)$
11: return rules
```

The above process of generating data plane rules from iForest is summarized in Algorithm 1. First, we record all branch values of iForest (lines 1-2), which are the boundaries of the hypercubes. Then, we shift them down to obtain the integer boundaries according to Equation (1), use the set function to avoid duplicate boundaries (line 4), and generate each vertex by the Cartesian product (line 5). Furthermore, we employ a greedy algorithm to generate regular intervals by sorting the vertexes from small to large, and consider the domain between two adjacent vertexes as a hypercube (line 7). Then, we predict the labels of these hypercubes using the trained iForest (line 8). After that, the hypercube of the same label is merged. Finally, the range rules are generated (line 10). Notably, we add the maximum value INF as the boundary branch for each feature to avoid the problem that multi-dimensional feature sorting makes the later branch's value smaller than the previous value.

Compared with the enumeration methods of [43], we propose executable-level enumerations by taking advantage of the consistent labels within the hypercubes in the iForest.

## 4.2 Burst Feature Extractor

To apply the rules, it is first necessary to extract bursts of packets from the traffic. A burst is defined as a long sequence of continuously sent packets in a flow, where the inter-arrival time does not exceed a certain threshold $\tau$. Although [2, 23, 34, 46] use bursts to infer the event activity categories of IoT devices, such solutions require excessive resources to maintain the long sequence of packets, e.g., long HTTP flows will cause persistent storage in memory. This is not practical for Tbps switches. To address this, we propose burst segmentation to reduce long-term resource occupation by keepalive traffic, bi-hash to achieve bi-directional flow matching, and double hash table to solve the hash conflict problem. It will be described in the following subsections.

### 4.2.1 Burst segmentation

We set a packet number segmentation threshold to achieve low resource usage and high real-time anomaly detection. Meanwhile, we use the packet number and sizes of segmented bidirectional bursts as features.

### 4.2.2 Bi-hash.

To get bidirectional burst features, it is necessary to first match the bidirectional flows. Conventionally, the matching of bidirectional flows is to calculate the ID through a five-tuple hash function *hash(dstIP, srcIP, dstPort, srcPort, protocol)*, and then use a dictionary to maintain the ID of the reverse flow, which is highly complex. To reduce the computation burden on the switches, we design a linear complexity bi-hash algorithm. Specifically, we divide the original five-tuple hash function into an XOR of bi-hash: *hash(dstIP, dstPort, protocol)* $\oplus$ *hash(srcIP, srcPort, protocol)*. This is through the commutative law of XOR to achieve bidirectional flow mapping to the same address. Then, we do bidirectional flow matching on the data plane by deploying the two hashes and one binary XOR operation. In Appendix B, we discuss the operator selection of bi-hash and find that choosing XOR for bi-hash has a lower collision rate than the addition operator, and bi-hash is close to the original five-tuple's hash collision rate.

### 4.2.3 Double hash table

To mitigate hash conflicts, we implement the double hash table algorithm. We divide one hash table into two hash tables. If the value conflicts in the first hash table, the algorithm executes the hash function on the first hash value and allocates it to the second hash table. We investigate the impact of different resource allocations on the conflict rate in Appendix B. Surprisingly, the equally divided double hash table is approximately optimal. Moreover, it only requires segmenting the length of the hash values but not extra operations (e.g., mod). Compared with using one hash table of the same bit width, the double hash table reduces hash collision by ten times.

To implement a double hash table, the switch needs to judge whether the first hash table conflicts. We calculate an additional bi-hash value named hash-check through different hash algorithms, and store it in the mapping address corresponding to bi-hash. When the original bi-hash hits, we determine whether the flow belongs to a collision flow or a hit flow by checking the hash-check value, so as to determine whether to allocate the second hash bucket. We find that when the number of streams is 32,000 and the total bit width is 18, the collision rate is as low as 0.6%, which saves resources more than using stored five-tuple for judgment.

## 4.3 Model Computational Complexity

The complexity of each phase of Gulliver Tunnel is as follows.
**Detection phase.** In this phase, the iForest model has been converted into rules and deployed on the switch. As the programmable switch hardware implements the match action function of the pipeline, model detection can achieve linear processing speed without adding extra overhead.
**Burst-level feature extraction phase.** In this phase, only the bi-hash and XOR operations are used, so the complexity of flow feature extraction in the switch is $O(1)$.
**Training phase.** In the training phase, it has been proved in [21] that the computational complexity of an iForest is $O(t\psi log\psi)$, where $t$ is the number of iTree, and $\psi$ is the size of sub-samples.
**Rule generation.** During the conversion from an iForest model to rules, the computational complexity is related to the number of branches of each feature. In the worst case, the maximum number of branches of a single tree is $2^{l+1}-2$ when the tree is a binary complete tree. Meanwhile, the height limit of a single tree is $l=\lceil log_2\psi\rceil$ (line 2 of algorithm 2). That is, the upper bound of the total number of branches in an iForest is $2t*(\psi-1)$.

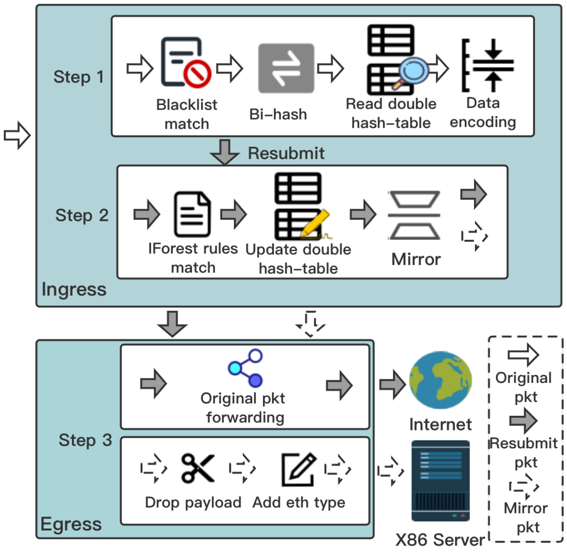


Figure 4: The implementation on the data plane.

Since we use the Cartesian product to record hypercubes, the complexity is the product of different features' branch numbers. We only use packet numbers and flow sizes of segmented bursts as burst-level features. With a segmentation threshold of 15, the packet number feature has only 15 branches at most. Thus, the worst complexity of the rule generation process in our scheme is $30t*(\psi-1)$, which is $O(t\psi)$.

To summarize, in our scheme the complexity of the model generation process does not exceed $O(t\psi log\psi)$, indicating the model can be placed on less powerful devices.

## 4.4 Hardware Implementation

We implement Gulliver Tunnel on programmable switches using the P4 programming language. As shown in Figure 4, the switch consists of the ingress phase and the egress phase. In the ingress phase, we implement flow identification, burst feature extraction, storage, as well as anomaly detection using the iForest white list rules. Due to the limitations of the Tofino chip, a register in the switch pipeline can be either read or written. Hence, the read and update operation of the double hash table is realized by resubmitting. In the egress phase, packet mirroring and forwarding are performed. The implementation details are as follows.
**Step 1: Parsing.** When a packet passes through the pipeline for the first time, the switch conducts flow identification. The switch first uses the bi-hash algorithm to calculate the index of the five-tuple and calculates the location of the data storage according to the double hash table algorithm. Then, the switch reads the burst features according to the double bucket

position. According to the read feature, the switch decides whether to perform anomaly detection or update burst feature information. At the end of this pipeline, the switch adds the decision into the additional metadata in the form of flags and resubmits the packet to the pipeline, instead of sending all the burst features (e.g., timestamp).

**Step 2: Detection and update.** The second time the packet passes through the pipeline, the switch performs anomaly detection or feature update. According to the flags in the metadata, if the switch notices the burst interval times out, or the number of packets exceeds the segmentation threshold, it will perform abnormal flow identification using the iForest white list rules. We compress flag judgment and anomaly detection into the same rule matching table for parallel processing. After the anomaly detection is performed, the contents of the corresponding registers are released. Otherwise, the burst feature content in the double hash table is updated. Next, the switch mirrors the packet to achieve asynchronous processing.

**Step 3: Post-processing.** Last, the egress handles the mirrored packet while forwarding the original packet as usual. For the mirrored packet, if the flow is normal, we drop the payload and upload the header to the server for storage. Otherwise, we encapsulate the last mirrored packet header, add the abnormal burst information (burst size, number of packets) to the new packet header, and send the packet to the control plane. Then, the control plane extracts any stored historical packets using the five-tuple of the abnormal mirrored packet (reported by the switch), before performing further detection.

## 5 Magnifier

To further reduce false positives, we propose a lightweight autoencoder, called Magnifier (Figure 5), which inspects suspicious traffic reported by Gulliver Tunnel. This operates in the centralized control plane. We design a series of lightweight components to reduce the false-positive rate, while achieving high throughput. First, we design an asymmetric autoencoder to reduce the number of parameters while ensuring the performance of the model. Second, we use separable convolution [15] rather than a basic convolutional layer, which further reduces the computational complexity. Third, we use dilation convolution [44] to replace the original convolution operation to achieve a larger receptive field without increasing layers and parameters. Finally, we compress the model through model quantization to improve the model throughput.

### 5.1 Asymmetric Autoencoder

We design an asymmetric autoencoder (AAE) inspired by [14]. The encoder requires deeper layers for better representation extraction, while the decoder can reconstruct features without overly complex operations. Compared with the symmetric autoencoders used by many existing methods [26], this asymmetric structure significantly (*i*) reduces the number of parameters; (*ii*) increases the training speed and throughput; and (*iii*) maintains similar performance.

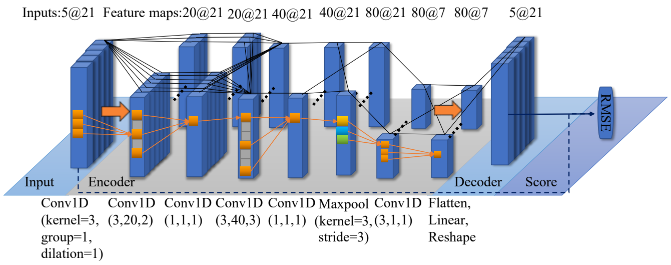


Figure 5: Magnifier model, an asymmetric lightweight autoencoder with separable and dilated convolutions.

The AAE contains two parts: the encoder and the decoder. Given the input features $X = \{x_i, \dots, x_m\}$, the encoder extracts $X$ through a multi-layer 1-dimensional convolution layer to generate a hidden representation $H$. Then $H$ is reconstructed back to $X' = \{x'_i, \dots, x'_m\}$ by a linear layer decoder. Using benign traffic as training data, the AAE is trained to minimize the reconstruction errors, i.e., the Root Mean Squared Error Loss, $RMSE = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(x_i - x'_i)^2}$, between the reconstructed features $X'$ and the input features $X$.

Through this, the AAE learns normal traffic patterns and is able to reconstruct normal traffic features without a large deviation from the original features. When conducting anomaly detection on traffic, the model identifies malicious traffic if the reconstruction error exceeds a threshold. This is because the autoencoder does not learn the pattern of malicious traffic, and is unable to reconstruct the features of malicious traffic, resulting in an excessively large RMSE.

To further reduce the number of parameters in the encoder layer and effectively utilize the information processing capabilities of the convolutional layer for different channels, we reshape the features obtained from the Kitsune [26] feature extraction into different channels according to different time windows. The features contain five time windows (100 ms, 500 ms, 1.5 sec, 10 sec, and 1 min), and each time window contains 20 features of the same type, such as the standard deviation of the packet size and packet jitter. In addition, we add the normalized source and destination port features to the first two channels, pad 0 for the remaining three channels, and finally obtain the input matrix of 5@21.

### 5.2 Separable Convolution

Inspired by MobileNets [15], a two-dimensional (2D) convolutional layer can be replaced by two 2D convolutional layers, i.e., a separable convolution, which consists of a depthwise convolutional layer and a pointwise convolutional layer. We use the separable convolution under 1D convolution that replacing one 1D convolutional layer with two 1D convolutional layers can also reduce computational complexity without sacrificing accuracy.

## 5.3 Dilated Convolutions

We enlarge the receptive field of Magnifier by utilizing dilated convolutions [6, 44], which skip some convolutional operations of the input at a certain dilation rate. Intuitively, instead of taking a fine-grain view of a small input region, dilated convolutions allow the network to take a coarse but wider view of the input. This allows our model to achieve a receptive field as large as a deep model with fewer layers, and obtain more information from features. Since the filter size used in dilated convolutions is the same as in regular convolutions, the number of parameters and training costs do not increase.

## 5.4 Model quantization

We optimize Magnifier with model quantization. We convert the model parameters from floating-point representation to lower-precision representation, e.g., 8-bit integers, to facilitate higher throughput, lower latency and less memory consumption. Magnifier adopts a symmetric variant of scale quantization [41] to transform the model parameters. To yields a $b$-bit integer value with a chosen representable range of $[-\alpha, \alpha]$, the scale factor is given as $s = \frac{2^{b-1}-1}{\alpha}$. The scale quantization of a real value $\chi$ is obtained by:

$$\chi_q = \begin{cases} -2^{b-1}+1 & \chi_s < -2^{b-1}+1, \\ \chi_s & -2^{b-1}+1 \le \chi_s \le 2^{b-1}-1 \ , \\ 2^{b-1}-1 & \chi_s > 2^{b-1}-1, \end{cases} \quad (2)$$

where $\chi_s = round(s * \chi)$.

To minimize precision loss, an appropriate $\alpha$ for the model parameters should be selected to achieve model calibration. For weights, Magnifier chooses max calibration [35], which uses the maximum absolute value seen during calibration as $\alpha$. It maintains precision because the distribution of weights is relatively concentrated. For activation, the use of max calibration could significantly reduce the precision due to the presence of extreme outliers caused by model inputs. Therefore, Magnifier chooses entropy calibration, which utilizes relative entropy as a criterion to select the $\alpha$ that minimizes the information loss from quantization.

# 6 Experimental Evaluation

## 6.1 Implementation

We prototype HorusEye using P4 (version $P_{16}$) and Python (version 3.8.0) with more than 2000 lines of code (LOC).

**Gulliver Tunnel.** We use Python to implement iForest training and rule generation. We use P4 programming language to deploy Gulliver Tunnel on a H3C S9830-32H-H data center switch with an Intel Tofino switch ASIC.

**Magnifier.** We use PyTorch to implement Magnifier and use TensorRT to implement quantization operations. We deploy Magnifier on a GeForce RTX 2080 SUPER.

**Hyperparameter settings.** We set the burst interval threshold $\tau$ to 1 second [46] and the segmentation length threshold to 15 based on the probability density function of the burst sequence length (In Appendix C). The dilation rates in our convolutional layers are $\{1,2,1,3,1,1\}$, as recommended by [38], which avoids the grid effect problem. The settings of other parameters (e.g., $t$, $\psi$) are discussed later in §6.3.

**Protocol-specific training.** Since different protocols are orthogonal when matching, we deploy rule sets for different protocols (e.g., TCP, UDP) within one table. We train an iForest model for each protocol flow and convert it to rules.

**Auxiliary rules for Gulliver Tunnel.** To further improve the detection capability, in addition to burst-level features, we add the IoT destination port feature $P$ as an auxiliary feature. Since $P$ is a non-distance vector feature, we train a new iForest, namely port-level iForest, with $P$ as an independent feature, and convert it to rules as a supplementary whitelist. We integrate auxiliary rules with burst-level rules to detect suspicious traffic (w/ $P$). As will be shown in 6.3.1, the auxiliary port rules can help with the detection, though the auxiliary port rules alone can be easily bypassed by attacks. The main detection capability is owed to Gulliver Tunnel with burst-level features.

## 6.2 Dataset and Feature Processing

### 6.2.1 Dataset

In the experiments, we use data generated in our real IoT testbed, as well as public datasets, with a total of 26 types of IoT devices and 10 types of attacks. Details of datasets can be found in Table 9 in Appendix D.

**Normal dataset - Ours.** To demonstrate a realistic functional IoT network, we set up a real-world testbed with 8 types of popular cameras, 7 types of sensors and one smart gateway, as IP cameras and sensors are among the IoT devices producing dominant traffic in IoT private networks. We generate device traffic by two means: (*i*) placing devices in a laboratory to detect the movements of people or objects; and (*ii*) configuring a laptop with Android Debug Bridge (ADB) and an emulator installed with control apps and running a Python script to request devices for streaming. This testbed runs for a month, generating 13.9 GB of PCAP data. To protect pedestrian privacy, we anonymize the data and discard the payload. We use the first 4 days of IoT normal traffic as the training set and the following 2 days as the test set.

**Normal dataset - [32].** We also use five types of cameras and five types of sensors in a public dataset [32] to demonstrate the generality of our scheme. We randomly select five days of data as the training set and the day after each of them as the test set. The training and test sets do not overlap.

**Attack dataset.** Since our scheme is unsupervised, we mix the attack datasets with the normal test set for testing. We use public attack datasets [5, 10, 18, 26]. In addition, to prevent data artifacts that may be caused by network mixing, we infect one of the IoT devices, and reproduce Mirai, OS scan, service scan, TCP DDoS, UDP DDoS attacks for testing. The total attack datasets mainly include four abnormal categories: (1) botnet infection, including Aidra, Bashlite [5] and Mirai; (2) data exfiltration, including keylogging and data theft [18]; (3) scanning attacks, including service scan and OS scan; (4) distributed denial of service attack (DDoS), including HTTP DDoS, TCP DDoS, and UDP DDoS.

### 6.2.2 Dataset processing

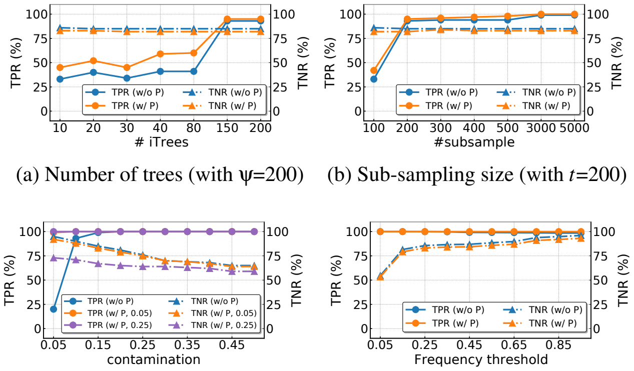
We divide the normal training set into validation set and training set with a ratio of 2:8 and only use one attack dataset (HTTP DDoS) as the validation set for hyperparameter selection. As suggested by [3], the frequency of Mirai attacks in the Kitsune dataset is too high to reflect real botnet infection. Therefore, we randomly down-sampled Kitsune's Mirai to a size similar to our Mirai dataset, i.e., 40,000 packets.

## 6.3 Gulliver Tunnel Hyperparameter Analysis

Gulliver Tunnel has four main hyperparameters: the number of trees ($t$), the sub-sampling size ($\psi$), the contamination, and the abnormal frequency. Under the validation set, we explore the effects of them on model performance (Figure 6). Here we only consider true-negative rate (TNR) and true-positive rate (TPR), because filtering out normal traffic and detecting anomalies are the most important targets for Gulliver Tunnel. We also discuss the impact on TPR and TNR of Gulliver Tunnel with or without destination port ($P$) rules. Among them, the basic model only uses burst-level iForest (w/o $P$), which is used to describe the proportion of attackers who bypass port defenses and are still detected by Gulliver Tunnel.

### 6.3.1 Number of trees

In Figure 6(a), we illustrate that the basic model (w/o $P$) has poor anomaly recall when the sub-sampling size is 200 and the number of trees is less than 80. This shows that it is easier for attacks to bypass iForest when there are fewer trees. The reason is that a larger forest reduces the uncertainty introduced by iTree random generation, and learns more about the behavioral patterns of normal data. This result demonstrates that iForest does not work well when the number of trees is small. It also proves that the previous method [47] of deploying iForest on the switch by direct coding cannot achieve good performance (because the number of trees cannot exceed 12).



(a) Number of trees (with $\psi$=200)

(b) Sub-sampling size (with $t$=200)

(c) Contamination setting (with $t$=200, $\psi$ =5000)

(d) Abnormal frequency setting (with $t$=200, $\psi$=5000, contamination-P=0.05, contamination-Burst=0.15).

Figure 6: Hyperparameter analysis for Gulliver Tunnel

### 6.3.2 Sub-sampling size

In Figure 6(b), the performance of both models increases as the sub-sampling size increases until the sub-sampling size reaches 3000, after which the performance gain is minimal. Therefore, we recommend not using a high sub-sampling size (>3000) when the server has insufficient computing power. Since our server has enough computation, we choose the sub-sampling size of 5000 to reach better performance.

### 6.3.3 Contamination

The contamination parameter controls the percentage of points in the dataset to be anomalous and affects the final label. As shown in Figure 6(c), we can indirectly limit the rule bounds and control the traffic offload by setting the contamination value. There is a trade-off between anomaly recalls and flow offloading rate by controlling contamination. As the contamination increases, more anomalies are recalled, but there is less normal traffic offloading.

To explore the effect of contamination on Gulliver tunnel performance, we set up three models, namely the basic model (w/o $P$) and the basic model supplemented with the port-based iForest model (w/ $P$), where the port-based iForest uses the contamination value as 0.05 or 0.25. Experimental results show that the basic model (w/o $P$) with 0.15 contamination has both large anomaly recall and normal recall. Yet the performance is poor at 0.05. This is because the range of rules generated by iForest is more extensive than expected when the training set is a clean normal dataset with multiple behavioral events. Thus, it is necessary to set the higher contamination value (e.g., contamination=0.15) for burst-level features, and we get tighter rule bounds.

Comparing the port ensemble models with contamination values of 0.05 and 0.25, we find that the larger contamination value does not increase the anomaly recall, but greatly reduces the normal recall. This is because the port-level rules are relatively weak, and many anomalies use the same ports as normal traffic, so generating tighter whitelist rules does not increase the anomaly recall, but only filters out normal traffic. Therefore, we choose a port contamination value of 0.05 to ban redundant ports that are barely used. The network administrator can modify the port whitelist according to the actual situation, instead of automatically generating port rules. Nonetheless, all the rules in this paper are automatically generated to avoid human interference.

### 6.3.4 Abnormal frequency threshold

To further reduce the burden on the control plane, we calculate the flow's abnormal frequency and set a threshold to further decide whether a flow is abnormal. The abnormal frequency $F(ID = i)$ of flow $i$ is calculated as

$$F(ID = i) = \frac{\sum_{n=1}^{N_i} \mathrm{I}(y_n = anomaly)}{N_i}, \tag{3}$$

where $N_i$ represents the burst number of flow $i$ and I is the indicator function.

In Figure 6(d), we vary the abnormal frequency threshold to study the impact on recall. No matter how large the threshold is set, the recall of anomalies is very high, which indicates that most of the anomalies have an extremely large warning frequency. When the frequency threshold is set to 0.95, Gulliver Tunnel without $P$ can detect 98.6% of malicious traffic and offload 96.0% of normal traffic. Accordingly, we can set a higher abnormal frequency threshold to improve the ability of Gulliver Tunnel to offload normal traffic.

## 6.4 Performance of Rule Generation

Converting iForest models to white list rules is the main challenge of HorusEye. In order to facilitate the calculation of the label consistency between the original iForest and the newly generated rules, we do not set the abnormal frequency in this experiment. We show the fidelity of our rule generation algorithm by examining the label consistency under various parameter combinations. Furthermore, we demonstrate the low resource occupancy and low computational complexity of our algorithm by recording the number of generated rules (#$R$) and the number of enumerations (#$Enum$). In addition, we show the number of burst and port rules for UDP and TCP, respectively, where burst rules, range-type rules and port rules are exact-type rules. In particular, the label consistency rate (C) is calculated as

$$Consistency(C) = \frac{\sum_{i=1}^{N} \mathrm{I}(\mathrm{iForest}(x_i) = \mathrm{R}(x_i))}{N}, \tag{4}$$

where $\mathrm{iForest}(x)$ means the prediction of iForest and $\mathrm{R}(x)$ means the prediction of rules.

Table 1: Performance of rule generation algorithm

| $\psi$ | $t$ | $\#R^{TCP}_{burst}$ | $\#R^{UDP}_{burst}$ | $\#R^{TCP}_{port}$ | $\#R^{UDP}_{port}$ | $C(\%)$ | #Enum |
|---|---|---|---|---|---|---|---|
| 400 | 10 | 11 | 21 | 46803 | 39249 | 99.46 | 144947 |
| | 50 | 7 | 28 | 26202 | 34958 | 99.58 | 191687 |
| | 100 | 15 | 17 | 24866 | 36264 | 99.63 | 237137 |
| | 200 | 19 | 18 | 29063 | 31440 | 99.62 | 309197 |
| 1000 | 200 | 13 | 37 | 15549 | 9132 | 99.68 | 402977 |
| 2000 | | 10 | 47 | 9120 | 1253 | 99.63 | 489392 |
| 5000 | | 29 | 48 | 3460 | 125 | 99.66 | 612932 |

Table 2: Interpretability analysis of TCP rules

| # Packets | Burst size | Expert's Understanding |
|---|---|---|
| $(1,2]$ | $(154,239]$ | Keepalive |
| $(2,3]$ | $(577,630]$ | State exchange |
| $(4,5]$ | $(432,460]$ | Command issue |
| $(0,15]$ | $(4533,4541]$ | Data stream |

Table 1 summarizes the results for the above metrics. Regardless of the increase of iForest tree number, our rule generation converts the iForest model to rules with nearly 100% label consistency. Furthermore, as the tree and subsampling size increase, the number of burst-type whitelist rules does not change much. This is because the burst-level rule is a range rule and will automatically reduce the number through rule merging. In addition, we find that by increasing the tree and subsampling size, we can significantly reduce the number of port-level rules, which can obtain more exact whitelist rules.

We also find an interesting phenomenon. Compared with UDP, TCP needs to use more destination ports, but its burst-type behavior is less. This shows that when IoT devices use TCP connections, the traffic features are relatively stable, and most IoT-based network services are relatively fixed. There are many destination port rules, indicating that IoT will often use dynamic ports. In our experiments, there are only 3585 port exact rules and 77 Burst range rules in the whitelist. We put the exact filtering of port rules in the blacklist in Figure 4, without adding additional tables. For burst-level range features, we compress the matching of multi-dimensional burst features into one rule by encoding and concatenating. Compared with previous work [47], which encodes the iTrees on the data plane by encoding each path of iTrees, our algorithm can be deployed on the data plane with low resource consumption (i.e., adding one table). Furthermore, the trend in the number of enumerations corroborates that the computational complexity of our rule generation algorithm is $O(tlog\psi)$.

## 6.5 Interpretability Analysis

A benefit of converting the iForest model into rules is that it is easier to interpret and debug. To demonstrate the interpretability of the generated rules, we take the burst-level rule for TCP traffic as an example (containing 29 rules). Through analysis, an expert could easily divide the traffic into four categories: (*i*) keepalive, which is sent periodically by a camera to maintain the connection with the server; (*ii*) state exchange, which is sent periodically by an IoT device to inform the server whether its status is active or idle; (*iii*) command issue, such as switching on and off, which is sent by the server to the camera; and (*iv*) data stream, i.e., the surveillance video sent by the camera to the server with adaptive bit rate. Table 2 shows a simple example to highlight that our scheme can mine the traffic behaviors, and assist experts in further debugging.

## 6.6 Hardware Performance

We install Gulliver Tunnel on the hardware switch and evaluate its efficiency using three metrics: (*i*) the resource occupancy, i.e., the occupancy rate of SRAM and TCAM on the switch; (*ii*) throughput, i.e, the receive packet throughput and transmit packet throughput of the switch after loading the program; (*iii*) latency, i.e., the time taken for the switch to process each packet.

The resource usage on the switch is shown in Table 3. When a packet first passes through the pipeline, stage 0 is used to deploy the blacklist, where port and protocol are filtered at the same time; stages 1 and 2 calculate the hash key value of the flow, and two hash functions are required to calculate bi-hash and hash-check respectively; stage 3 and 4 obtain the feature values in the first and second hash tables; stages 5-8 are used to judge whether the burst stream is over, each occupying a table. Meanwhile, two tables in stage 6 are used to set up the resubmit operation. When a packet passes through the pipeline for the second time, stage 3-5 is used for feature update operations; stage 6 matches the encoding results of burst-level features using the whitelist rules, which are stored in one table. It is worth noting that, the whitelist can be easily extended by adding new whitelist items, which are stored in the TCAM, without interrupting the functioning of the switch.

Thanks to the burst feature extraction scheme (burst segmentation, bi-hash and double hash table), even though storing and maintaining burst-level features (in stages 3-5) consumes the most table and register resources in the pipeline, the whole Gulliver Tunnel deployment still only requires very little resources, i.e., 2.78% of TCAM and 9.90% of SRAM.

We use the SPIRENT N11U traffic generator for high-speed traffic simulation. Since the data packet size is not fixed in real network environments, we test various combinations of different packet sizes, called Internet Mix (IMIX), to evaluate the forwarding capability of the switch. We use the pre-configured IMIX parameters of the traffic generator manufacturer for the real environment simulation.

Table 4 shows the switch packet throughput under 100Gbps. After loading Gulliver Tunnel P4 program, we send a large number of packets for testing. We find that there is no packet loss caused by the overhead of our framework. Moreover, the average processing delay per packet is as small as 1065.6 (ns).

Table 3: Resource occupancy

| Stage | Table | SRAM(kbit) | TCAM(kbit) |
|---|---|---|---|
| 0 | 8 | 512 | 0 |
| 1 | 4 | 0 | 0 |
| 2 | 4 | 128 | 11 |
| 3 | 15 | 3456 | 0 |
| 4 | 18 | 4480 | 0 |
| 5 | 11 | 2944 | 0 |
| 6 | 4 | 384 | 44 |
| 7 | 1 | 128 | 22 |
| 8 | 1 | 128 | 11 |
| 9 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 |
| 11 | 0 | 0 | 0 |
| Total | 66 | 12160 (9.90%) | 88 (2.78%) |

Table 4: Single port forwarding performance at 100 Gbps

| Packet size | Latency | #Transmit | #Loss |
|---|---|---|---|
| 256 | 1027 (ns) | 40760869565 | 0 |
| 512 | 1033 (ns) | 21146616541 | 0 |
| 1024 | 1052 (ns) | 10775862068 | 0 |
| 1500 | 1074 (ns) | 7401315789 | 0 |
| IMIX | 1142 (ns) | 24832244393 | 0 |

## 6.7 Detection Performance

We evaluate the true-positive rates (TPRs) of different schemes under low FPR$\leq$5e-5 and FPR$\leq$5e-4 by setting RMSE abnormality thresholds. We measure the detection ability of different schemes by $\text{PR}_{\text{AUC}}$, which reflects the performance of the model on imbalanced data sets. Besides, we randomly down-sample the normal data set in the test set with a ratio of 10% to alleviate the data imbalance problem.

To reflect HorusEye's detection performance, we compare it with the state-of-the-art (SOTA) unsupervised learning anomaly detection model Kitsune [26]. Table 5 summarizes the detection performance trained on our real-world testbed IoT dataset, which contains multi-behavior interactions. Due to limited space, the detection performance trained on the public device set is presented in Table 10 in Appendix E. Both tables can draw the same conclusions as follow.

First, HorusEye notably outperforms the Kitsune model in most attacks, especially in low false positive scenarios. Under the FPR of 5e-5, the TPRs of HorusEye are 37.6% and 20.9% higher than those of Kitsune on the public attack dataset and our attack dataset, respectively. Meanwhile, with reduced frequency Mirai attacks, Kitsune could only achieve

Table 5: Detection performance of models trained on our dataset. The best performance is highlighted in bold. The percentages next to the colored arrows are the relative increments/decrements of HorueEye's TPRs compared with Kitsune. ≤5e-5 indicates FPR≤5e-5 and ≤ 5e-4 indicates FPR≤5e-4.

| Dataset | Attack | Kitsune | | | Magnifier | | | HorusEye | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | TPR | | $PR_{AUC}$ | TPR | | $PR_{AUC}$ | TPR | | $PR_{AUC}$ |
| | | ≤5e-5 | ≤5e-4 | | ≤5e-5 | ≤5e-4 | | ≤5e-5 | ≤5e-4 | |
| [5]<br>[18]<br>[26] | Aidra | 0.228 | 0.406 | **0.716** | 0.370 | 0.451 | 0.631 | **0.383** ↑ 68.1% | **0.469** ↑ 15.3% | 0.657 ↓ 8.33% |
| | Bashlite | 0.605 | 0.677 | **0.818** | 0.698 | 0.730 | 0.806 | **0.713** ↑ 17.7% | **0.735** ↑ 8.58% | 0.817 ↓ 0.09% |
| | Mirai | 0.105 | 0.183 | 0.949 | 0.962 | **0.966** | 0.976 | **0.964** ↑ 815% | **0.966** ↑ 428% | **0.980** ↑ 3.23% |
| | Keylogging | **0.527** | 0.527 | 0.602 | **0.527** | **0.528** | 0.779 | **0.527 -** | **0.528** ↑ 0.03% | **0.806** ↑ 33.9% |
| | Data theft | **0.508** | 0.508 | 0.587 | **0.508** | 0.508 | 0.785 | **0.508 -** | **0.510** ↑ 0.04% | **0.810** ↑ 38.1% |
| | Service scan | 0.217 | 0.274 | 0.833 | 0.318 | 0.358 | 0.915 | **0.334** ↑ 53.6% | **0.363** ↑ 32.5% | **0.934** ↑ 12.1% |
| | OS scan | 0.367 | 0.504 | 0.939 | 0.461 | 0.561 | 0.933 | **0.498** ↑ 35.9% | **0.577** ↑ 14.5% | **0.946** ↑ 0.77% |
| | HTTP DDoS | 0.055 | 0.211 | 0.779 | 0.235 | 0.382 | 0.927 | **0.285**↑ 421% | **0.408** ↑ 93.8% | **0.942** ↑ 21.0% |
| | TCP DDoS | 0.903 | 0.936 | 0.969 | **0.959** | **0.971** | **0.989** | 0.903 - | 0.912 ↓ 2.65% | 0.929 ↓ 4.13% |
| | UDP DDoS | 0.904 | 0.936 | 0.968 | 0.959 | 0.972 | 0.989 | **0.965** ↑ 6.70% | **0.973** ↑ 4.02% | **0.990** ↑ 2.31% |
| | macro | 0.442 | 0.516 | 0.816 | 0.600 | 0.643 | 0.873 | **0.608** ↑ 37.6% | **0.644** ↑ 24.8% | **0.881** ↑ 7.97% |
| Ours | Mirai | 0.000 | 0.012 | 0.636 | 0.196 | 0.412 | 0.842 | **0.303** ↑ ∞ | **0.424** ↑ 340% | **0.868** ↑ 36.4% |
| | Service scan | 0.918 | 0.956 | 0.998 | 0.989 | 0.995 | 0.999 | **0.991** ↑ 8.06% | **0.996** ↑ 4.05% | **1.000** ↑ 0.14% |
| | OS scan | 0.617 | 0.810 | 0.994 | 0.943 | 0.983 | **0.999** | **0.968** ↑ 56.9% | **0.985** ↑ 21.7% | **0.999** ↑ 0.54% |
| | TCP DDoS | 0.994 | 0.996 | **1.000** | **0.997** | **0.998** | **1.000** | **0.997** ↑ 0.39% | **0.998** ↑ 0.25% | **1.000 -** |
| | UDP DDoS | 0.995 | 0.997 | **1.000** | 0.997 | **0.998** | **1.000** | **0.998** ↑ 0.27% | **0.998** ↑ 0.15% | **1.000 -** |
| | macro | 0.705 | 0.754 | 0.925 | 0.825 | 0.877 | 0.968 | **0.852** ↑ 20.9% | **0.880** ↑ 16.7% | **0.973** ↑ 5.19% |

TPR of 0.105 and 0 on the two datasets under the condition of FPR of 5e-5, respectively. This illustrates that it is difficult to detect low-frequency Mirai attacks under a low false-positive rate. Nevertheless, HorusEye can still reach TPRs of 0.964 and 0.303 on the two datasets, respectively. This is owing to the better representation extraction of HorusEye's deeper encoder layers, and the large receptive field brought by the dilated convolution in Magnifier.

Second, HorusEye and Magnifier achieve comparable detection performance, while HorusEye is even slightly better at detecting most anomalies than Magnifier. This indicates that the preliminary screening conducted by Gulliver Tunnel on the data plane does not hurt the performance of Magnifier in most cases, since Gulliver Tunnel has a high TPR. This also shows that the low false-positive capability mainly comes from Magnifier, while the burst features exploited by Gulliver Tunnel also help the detection of attacks.

Last, we find that all schemes have lower $PR_{AUC}$ against botnet infection and data exfiltration than the other two types of attacks. This is because botnet infection and data exfiltration use SSH transmissions, which resemble patterns of normal traffic, whereas frequency attacks such as DDoS or scan differ from normal traffic more obviously. Fortunately, HorusEye can still recall more than 50% of these attacks even at FPR≤5e-5, thereby effectively blocking attacks by restricting hackers from reaching a complete attack chain. Furthermore, we find that all schemes can recall more than 90% of TCP DDoS and UDP DDoS attacks under low false-positive conditions. This is because these two kinds of anomalies are attacked through high frequency, which can be easily detected. However, all schemes struggle to recall HTTP DDoS attacks at FPR≤5e-5. This is because HTTP DDoS constructs real HTTP requests, which appear just like the normal HTTP traffic from some of the cameras. HTTP DDoS detection requires the server side to make more detailed judgments based on the request content and frequency.

Table 6: Comparison of $PR_{AUC}$ on unknown bot attacks between supervised and unsupervised schemes

| Proposal | Label | Data plane | Flow-level | $PR_{AUC}$ |
|---|---|---|---|---|
| Mousika | Yes | Yes | No | 0.646 |
| Kitsune | No | No | Yes | 0.723 |
| HorusEye | No | Yes | Yes | 0.781 |

To highlight the advantage of unsupervised methods, we further compare HorusEye with the supervised method Mousika [42], a SOTA solution for deploying decision tree on the data plane, which uses packet header features. We train and test this model using the same normal data and three categories of botnet infection (i.e., ours Mirai, Aidra and Bashlite). For supervised learning, we use two of the attacks for training and the other one as the unknown attack for testing. For unsupervised learning, these attacks are only used for testing,

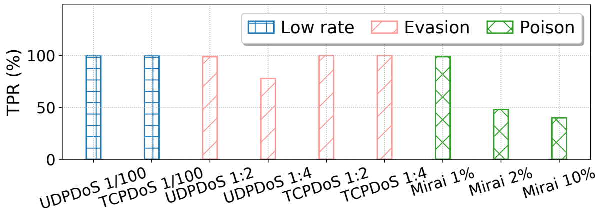


Figure 7: Robustness of Gulliver Tunnel against different adversarial attacks.

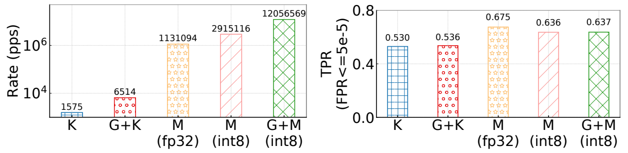
i.e., they are not in the training set and validation set. The macro $PR_{AUC}$ results are summarized in Table 6. We find that Mousika is significantly weaker than the unsupervised schemes Kitsune and HorusEye in detecting new attacks. It also shows that the existing scheme of deploying a decision tree on the data plane is not suitable for IoT intrusion detection tasks, where new abnormalities frequently appear.

## 6.8 Robustness of Detection

Since the data plane module is a weak part of the two-stage detection, we examine the robustness of Gulliver Tunnel. We assume that attackers know the existence of malicious traffic detection. Attackers can construct adversarial attacks: (*i*) low rate DDoS attacks, we respectively reduce the rates of TCP DDoS and UDP DDoS by 100 times (900 packets/second) as low rate DDoS attacks; (*ii*) evasion attacks, we inject benign TLS traffic and QUIC traffic, which are obtained from the camera training set, into the TCP DDoS and UDP DDoS and disguise them as benign traffic for evasion. TLS and QUIC are chosen since they are the two main protocols commonly used by cameras. We consider two different ratios, 1:2 and 1:4, of malicious traffic to the benign traffic; (*iii*) poison attacks, we mix our Mirai attack dataset into the training set by up-sampling to achieve 1%, 2% and 10% pollution ratios.

The experimental results are summarized in Figure 7. We first observe that Gulliver Tunnel is resilient to low-rate attack and still retains a TPR of 100%. This is because 900 (packets/second) and 90000 (packets/second) in DDoS make no difference in segmented burst features on the data plane. In terms of evasion attacks, Gulliver Tunnel can resist most of the one-way flow evasion attacks because it adopts the features of the bidirectional flow.

Regarding poison attacks, without changing the hyperparameters, Gulliver Tunnel can retain a TPR of 99% when the pollution ratio is 1%. This is because iForest adopts the subsampling and contamination hyperparameters to let the model learn the main patterns of normal traffic, which provides some natural resilience against mild pollution. As such, Gulliver Tunnel is resistant to chronic data poisoning, where the proportion of pollution caused by chronic data poisoning is often very low since the normal data is constantly updated and accumulated. However, more than 2% data pollution causes a drop in Gulliver Tunnel's TPR. Nevertheless, security engineers can detect such pollution attacks using old models, since over 2% pollution ratio of normal traffic will lead to significant changes in data distribution.



(a) Thoughput of models

(b) Macro TPR at FPR≤5e-5

Figure 8: Throughput and detection capability. K and M represent Kitsune and Magnifier, respectively. G indicates Gulliver Tunnel.

Overall, Gulliver Tunnel is fairly robust to three common black-box attacks. However, it may not defend against white-box attacks from sophisticated hackers, because it deploys explainable whitelist rules. In future work, we will improve the robustness against white-box attacks.

## 6.9 Throughput and Detection Capability

As shown in Table 4, Gulliver Tunnel can nearly reach line speed, i.e., 100Gbps on a single port. The throughput bottleneck is the model processing capability on the control plane. Luckily, Gulliver Tunnel is able to confine the processing of most normal traffic to the data plane. On the test set of our dataset, we find that Gulliver Tunnel offloads 76% of normal traffic, i.e., the throughput gains are 4.13x. We present the throughput and detection capability of HorusEye on our dataset in Figure 8.

Note that Gulliver Tunnel also works with other control plane solutions, e.g., Kitsune. Thus, Figure 8(a) compares the packet throughput of Kitsune and Magnifier (with/without Gulliver Tunnel). Kitsune only provides a Python version of the implementation, and its feature map module cannot be quantified. Therefore, we reproduce the Python version of Kitsune and compare the throughput with our model under the same precision fp32. Although Kitsune claims that its C++ version has 100 times the throughput of the Python version (1575*100 packets/sec), Magnifier (fp32) still has significantly better packet throughput (1131094 packets/sec). The outstanding throughput of Magnifier comes from its lightweight designs (e.g., AAE, separate convolution, and dilation convolution). Furthermore, HorusEye, i.e., Magnifier (int8) + Gulliver Tunnel, achieves a throughput of 12 million packets/sec, which is a remarkable 7654x higher than Kitsune.

In Figure 8(b), we present the performance of HorusEye in terms of macro TPR at FPR≤5e-5. We find that quantizing Magnifier from 32-bit float (fp32) to 8-bit int (int8) brings about a 2.5x throughput gain while the TPR only slightly drops from 0.675 to 0.636. This is because quantization only affects the abnormal judgment near the decision boundary, while most abnormal samples with low FPR are far away from the decision boundary. As such, the degree of influence is relatively small. Moreover, we can see that the adoption of Gulliver Tunnel does not affect the Magnifier and Kitsune's anomaly detection.

In conclusion, HorusEye has excellent throughput and anomaly detection capabilities.

# 7 RELATED WORK

## 7.1 IoT Anomaly Detection

In an IoT private network (e.g, smart city), the devices (e.g., cameras) are numerous, with limited computational resources and protection. They are often deployed over several years and perform frequent traffic exchanges (via public IP addresses). As such, they are particularly vulnerable to attacks, e.g. Mirai [1, 17]. Worse, these attacks can also be disguised under TLS encryption, which renders traditional deep packet inspection no longer applicable [1, 12] in the network.

Some works distinguish normal and abnormal traffic using traffic patterns [8, 12, 24, 26, 33, 36, 46]. However, [24, 26] use complex deep models, which cannot achieve real-time high-throughput detection. [8,36,46] are deployed under home gateways to learn behavioral interactions between a fixed number and type of devices. However, the computational complexity grows exponentially with the number of devices, due to the increased interactions between devices. Therefore, they cannot be deployed in smart cities. In [12], an anomaly detection algorithm is implemented by Fourier transform and clustering, and deployed on the control plane of the CPU architecture, whose control plane cannot keep up with the growth of the traffic in the network [29]. Overall, these anomaly detection solutions cannot scale to multi-Tbps because they perform detection in the control plane. Moreover, [12, 26] are orthogonal to our approach and can benefit from Gulliver Tunnel by pre-filtering suspicious traffic in the data plane.

## 7.2 Programmable Switches

In the network, most data processing is placed in the control plane. This limits the throughput of the traffic. The advent of programmable switches has made it possible to process traffic in the data plane. Hence, various tasks have been implemented in the data plane, such as DDoS detection [22, 45], flow size prediction [39, 42], and others [19, 20, 29]. Among them, [45] is the first to detect DDoS attacks in the data plane. Furthermore, [22] uses the sketch method to further measure the occurrence of DDoS attacks on the data surface. Nevertheless, they measure whether the amount of traffic sent by the source IP address exceeds a certain threshold. Thus, it is not applicable to the detection of attacks that have a large number and a wide dispersion of IoT devices. Therefore, it is necessary to implement complex rule matching on the switch.

A key contribution of our work is mapping complex iForest models into rules. [39, 42] both propose methods to convert a decision tree into rules and deploy them to P4. This method uses machine learning to mine more complex rules from data. Unfortunately, the decision tree belongs to supervised learning. It is difficult to maintain a high-quality intrusion dataset and detect unknown attacks. [39] proposes an exhaustive approach to transform SVM [9] or K-means [13] models into rules and deploy them in programmable switches. Nevertheless, SVM and K-means require brute force to traverse all feature domains and incur an impractical overhead. Finally, [47] stores M features by allocating them into M stages, and uses N trees as N encoding stages, thus occupying M+N stages in total. However, the number of stages of a programmable switch is generally around 12, with some of them reserved for basic forwarding functions. Therefore, only a very limited number of ensemble model trees can be deployed.

## 7.3 IoT Behavior Analysis

Since most modern traffic is encrypted, it is impossible to detect the anomalies through the payloads. Kitsune [26] use statistical features to measure the behavior of the network, such as the mean or variance of the packet size, or the correlation coefficient. However, these statistical features cannot be tracked in the programmable switches since floating-point arithmetic is not supported [31]. [2, 23, 34, 46] find that even with encryption, the behavior of IoT devices can be identified using the sequence of packet sizes or relatively fixed behavior patterns. Our feature extraction scheme is the first that considers computational constraints and resource consumption, where we propose burst segmentation, bi-hash, and double hash tables.

# 8 Conclusion

In this paper, we propose HorusEye, an unsupervised Internet of Things (IoT) anomaly detection framework. It offloads abnormal traffic detection into the data plane, thereby freeing resources in the control plane to recheck results for higher accuracy. In the data plane, we propose a rule generation algorithm for iForest and a new flow feature extraction scheme, which implement the first unsupervised model that can reflect the limited resources of switches. We prove that the computational complexity of HorusEye in the data plane satisfies real-time performance constraints in all phases, from training to execution. On the control plane, we adopt a lightweight unsupervised model and a high-speed inference scheme. Extensive experiments show that HorusEye can offload a high proportion of traffic from the control plane, and detect diverse attacks even in very high throughput networks.

## Acknowledgments

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 Shenzhen Key Lab of Software Defined Networking under grant No.ZDSYS20140509172959989, and Research Center for Computer Network (Shenzhen) Ministry of Education.

## References

[1] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. Sok: Security evaluation of home-based iot deployments. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP)*, pages 1362–1380. IEEE, 2019.

[2] Noah Apthorpe, Dillon Reisman, and Nick Feamster. A smart home is no castle: Privacy vulnerabilities of encrypted iot traffic. *arXiv preprint arXiv:1705.06805*, 2017.

[3] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. Dos and don'ts of machine learning in computer security. In *Proceedings of the 2022 USENIX Security Symposium*, pages 3971–3988. USENIX Association, 2022.

[4] Clément Bénard, Gérard Biau, Sébastien Da Veiga, and Erwan Scornet. Interpretable random forests via rule extraction. In *International Conference on Artificial Intelligence and Statistics*, pages 937–945. PMLR, 2021.

[5] Vitor Hugo Bezerra, Victor G Turrisi da Costa, Ricardo Augusto Martins, Sylvio Barbon Junior, Rodrigo Sanches Miani, and Bruno Bogaz Zarpelao. Providing iot host-based datasets for intrusion detection research. In *Anais do XVIII Simpósio Brasileiro de Segurança da Informaçao e de Sistemas Computacionais*, pages 15–28. SBC, 2018.

[6] Sanjit Bhat, David Lu, Albert Hyukjae Kwon, and Srinivas Devadas. Var-cnn: A data-efficient website fingerprinting attack based on deep learning. *Proc. Priv. Enhancing Technol.*, 2019(4):292–310, 2019.

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[8] Z Berkay Celik, Gang Tan, and Patrick D McDaniel. Iotguard: Dynamic enforcement of security and safety policy in commodity iot. In *Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2019.

[9] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[10] F Ding. Iot malware. https://github.com/ifding/iot-malware, 2017.

[11] Menachem Domb, Elisheva Bonchek-Dokow, and Guy Leshem. Lightweight adaptive random-forest for iot rule generation and execution. *Journal of Information Security and Applications*, 34:218–224, 2017.

[12] Chuanpu Fu, Qi Li, Meng Shen, and Ke Xu. Realtime robust malicious traffic detection via frequency domain analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2021.

[13] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)*, 28(1):100–108, 1979.

[14] Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked autoencoders are scalable vision learners. *arXiv preprint arXiv:2111.06377*, 2021.

[15] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[16] GSMA Intelligence. The mobile economy 2020. https://www.gsmaintelligence.com, 2020.

[17] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. Ddos in the iot: Mirai and other botnets. *Computer*, 50(7):80–84, 2017.

[18] Nickolaos Koroniotis, Nour Moustafa, Elena Sitnikova, and Benjamin Turnbull. Towards the development of realistic botnet dataset in the internet of things for network forensic analytics: Bot-iot dataset. *Future Generation Computer Systems*, 100:779–796, 2019.

[19] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael M Swift. Atp: In-network aggregation for multi-tenant learning. In *Proceedings of the 2021 USENIX Symposium on Network System Design and Implementation (NDSI)*, pages 741–761. USENIX Association, 2021.

[20] Yiran Lei, Yu Zhou, Yunsenxiao Lin, Mingwei Xu, and Yangyang Wang. Dove: Diagnosis-driven slo violation detection. In *Proceedings of the 2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2021.

[21] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *Proceedings of the 2008 IEEE International Conference on Data Mining*, pages 413–422. IEEE, 2008.

[22] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric ddos attacks with programmable switches. In *Proceedings of the 30th USENIX Security Symposium*. USENIX Association, 2021.

[23] Xiaobo Ma, Jian Qu, Jianfeng Li, John CS Lui, Zhenhua Li, and Xiaohong Guan. Pinpointing hidden iot devices via spatial-temporal traffic fingerprinting. In *Proceedings of the 2020 IEEE INFOCOM Conference on Computer Communications*, pages 894–903. IEEE, 2020.

[24] Gonzalo Marín, Pedro Casas, and Germán Capdehourat. Deep in the dark-deep learning-based malware traffic detection without expert knowledge. In *Proceedings of the 2019 IEEE Security and Privacy Workshops (SPW)*, pages 36–42. IEEE, 2019.

[25] Morteza Mashayekhi and Robin Gras. Rule extraction from random forest: the rf+ hc methods. In *Canadian Conference on Artificial Intelligence*, pages 223–237. Springer, 2015.

[26] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2018.

[27] Gabi Nakibly, Alex Kirshon, Dima Gonikman, and Dan Boneh. Persistent ospf attacks. In *Proceedings of the 2012 Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2012.

[28] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 226–241. IEEE, 2005.

[29] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, et al. Sailfish: accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 194–206, 2021.

[30] M Zubair Rafique and Juan Caballero. Firma: Malware clustering and network signature generation with mixed network behaviors. In *International Workshop on Recent Advances in Intrusion Detection*, pages 144–163. Springer, 2013.

[31] Dominik Scholz, Henning Stubbe, Sebastian Gallenmüller, and Georg Carle. Key properties of programmable data plane targets. In *Proceedings of the 2020 32nd International Teletraffic Congress (ITC 32)*, pages 114–122. IEEE, 2020.

[32] Arunan Sivanathan, Hassan Habibi Gharakheili, Franco Loi, Adam Radford, Chamith Wijenayake, Arun Vishwanath, and Vijay Sivaraman. Classifying iot devices in smart environments using network traffic characteristics. *IEEE Transactions on Mobile Computing*, 18(8):1745–1759, 2018.

[33] Ruming Tang, Zheng Yang, Zeyan Li, Weibin Meng, Haixin Wang, Qi Li, Yongqian Sun, Dan Pei, Tao Wei, Yanfei Xu, et al. Zerowall: Detecting zero-day web attacks through encoder-decoder recurrent neural networks. In *Proceedings of the 2020 IEEE INFOCOM Conference on Computer Communications*, pages 2479–2488. IEEE, 2020.

[34] Rahmadi Trimananda, Janus Varmarken, Athina Markopoulou, and Brian Demsky. Packet-level signatures for smart home devices. In *Proceedings of the 2020 Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2020.

[35] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS*, 2011.

[36] Juan Wang, Shirong Hao, Ru Wen, Boxian Zhang, Liqiang Zhang, Hongxin Hu, and Rongxing Lu. Iotpraetor: Undesired behaviors detection for iot devices. *IEEE Internet of Things Journal*, 8(2):927–940, 2020.

[37] Mowei Wang, Yong Cui, Xin Wang, Shihan Xiao, and Junchen Jiang. Machine learning for networking: Workflow, advances and opportunities. *IEEE Network*, 32(2):92–99, 2017.

[38] Panqu Wang, Pengfei Chen, Ye Yuan, Ding Liu, Zehua Huang, Xiaodi Hou, and Garrison Cottrell. Understanding convolution for semantic segmentation. In *2018 IEEE winter conference on applications of computer vision (WACV)*, pages 1451–1460, Lake Tahoe, NV, USA, 2018. IEEE Computer Society.

[39] Shuhe Wang, Chen Sun, Zili Meng, Minhu Wang, Jiamin Cao, Mingwei Xu, Jun Bi, Qun Huang, Masoud Moshref, Tong Yang, et al. Martini: bridging the gap between network measurement and control using switching asics. In *Proceedings of the 2020 IEEE 28th International Conference on Network Protocols (ICNP)*, pages 1–12. IEEE, 2020.

[40] Sutong Wang, Yuyan Wang, Dujuan Wang, Yunqiang Yin, Yanzhang Wang, and Yaochu Jin. An improved random forest-based rule extraction method for breast cancer diagnosis. *Applied Soft Computing*, 86:105941, 2020.

[41] Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. Integer quantization for deep learning inference: Principles and empirical evaluation. *arXiv preprint arXiv:2004.09602*, 2020.

[42] Guorui Xie, Qing Li, Yutao Dong, Guanglin Duan, Yong Jiang, and Jingpu Duan. Mousika: Enable general in-network intelligence in programmable switches by knowledge distillation. In *Proceedings of the 2022 IEEE INFOCOM Conference on Computer Communications*, 2022.

[43] Zhaoqi Xiong and Noa Zilberman. Do switches dream of machine learning? toward in-network classification. In *Proceedings of the 18th ACM workshop on hot topics in networks (HotNets)*, pages 25–33. ACM, 2019.

[44] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. In *Proceedings of the 2016 International Conference on Learning Representations (ICLR)*, 2016.

[45] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qianqian Li, Mingwei Xu, and Jianping Wu. Poseidon: Mitigating volumetric ddos attacks with programmable switches. In *Proceedings of the 2020 Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2020.

[46] Wei Zhang, Yan Meng, Yugeng Liu, Xiaokuan Zhang, Yinqian Zhang, and Haojin Zhu. Homonit: Monitoring smart home apps from encrypted traffic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1074–1088. ACM, 2018.

[47] Changgang Zheng and Noa Zilberman. Planter: seeding trees within switches. In *Proceedings of the SIGCOMM'21 Poster and Demo Sessions*, pages 12–14. 2021.

## A Background of IForest

**Algorithm 2** iForest$(X,t,\psi)$

```
Require: $X-$ input data, $t-$number of trees, $\psi-$ sub-sampling size
Ensure: a set of $t$ iTrees
initialize Forest
set height limit $l = ceiling(log_2\psi)$
for $i = 1$ to $t$ do
    $X' \leftarrow sample(X,\psi)$
    $Forest \leftarrow Forest \cup iTree(X',0,l)$
end for
return Forest
```

**Algorithm 3** iTree$(X,e,l)$

```
Require: $X-$ input data, $e-$ current tree height, $l-$ height limit
Ensure: an iTree
if $e \geq l$ or $|X| \leq 1$ then
    return exNodeSize $\leftarrow |X|$
else
    let $Q$ be a list of attributes in $X$
    randomly select an attribute $q \in Q$
    randomly select a split point $p$ from max and min values of
    attribute $q$ in $X$
    $X_l \leftarrow filter(X, q \leq p)$
    $X_r \leftarrow filter(X, q > p)$
    return inNode { $Left \leftarrow iTree(X_l, e+1, l)$
                    $Right \leftarrow iTree(X_r, e+1, l)$,
                    $SplitAtt \leftarrow q$,
                    $SplitValue \leftarrow p$ }
end if
```

## B Hash Collision

First, we explore the choice of operators for bi-hash operations. We set the experiment as the number of streams is $2^{16}$, and the total register size is $2^{17}$. We find that bi-hash has a hash collision rate of 54.8% using the addition operation. The hash collision rate of bi-hash using XOR operation is 31.8%. The XOR operation is significantly better than the addition operation. This is because binary addition operation will cause a large amount of overflow in limited resources, resulting in the loss of information. Further, we measure that the collision rate under the traditional five-tuple hash algorithm is 29.0%. Compared with it, the bi-hash algorithm only increases the hash collision rate lightly, but realizes the bidirectional flow matching with lower resources.

Second, we investigate the impact of different resource allocations on the conflict rate through a probabilistic model. Considering $K$ flows, the numbers of conflicts in the first and the second buckets, denoted as $C_1$ and $C_2$, can be obtained as:

$$C_1 = K - M_1 * (1 - e^{-\frac{K}{M_1}}),\\ C_2 = C_1 - (M - M_1) * (1 - e^{-\frac{C1}{M - M_1}}), \tag{5}$$

where $M$ is the total hash table size, and $M_1$ and $M - M_1$ are the sizes of the first and second hash table, respectively. The collision rate of the double hash table is $P(C_2) = C_2/K$.

Here we show some simulation double-bucket hash collision experiments (Table 7), when the total bucket size is 131072 and the size of the flow is 32000, and the first hash table size M1 starts from 32768 and increases with 16384 steps. We find that when M1 is 65546, i.e., the double hash table resources are equally divided, the hash collision 1.03% is not much different from the lowest 0.93% in the simulation, which is approximately optimal. Hence, we use the equally divided double hash table in our experiments to exploit its advantage of avoiding further mod operations.

Table 7: Bucket resource allocation v.s. collision rate

| M | #Flow | M1 | Collision rate |
|---|---|---|---|
| | | 32768 | 2.05 % |
| | | 49152 | 1.33 % |
| | | 65536 | 1.03 % |
| 131072 | 32000 | 81920 | 0.93 % |
| | | 98304 | 1.00 % |
| | | 114688 | 1.46 % |
| | | 131072 | 11.27 % |

Then, we measure the average collision rate of single hash table and double hash table under the same bit resource. Specifically, we illustrate hash collisions based on the CRC32 hash algorithm, as shown in Table 8. The results show that under the same resources, the collision rate of the double hash table is nearly ten times lower than that of the single hash table. Theoretically, using the double hash table, only 15-bit hash index width is required to support 8000 flows with an acceptable collision rate of 1.03%. As the width increases, the number of flows that can be maintained grows exponentially, while 18-bit width can support 64000 flows simultaneously.

Table 8: Hash collision

| Bit width | #Flow | Collision rate | |
|---|---|---|---|
| | | Single hash table | Double hash table |
| 15 | 8000 | 11.38% | 1.03% |
| 16 | 16000 | 11.28% | 1.02% |
| 17 | 32000 | 11.25% | 1.05% |
| 18 | 64000 | 11.31% | 1.02% |

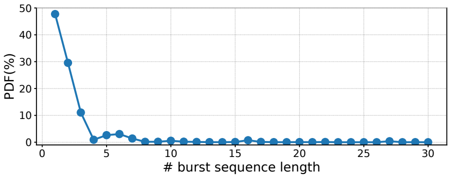


Figure 9: PDF of burst sequence length.

## C Data Analysis

We analyze the burst length distribution of the training data. As shown in Figure 9, due to the long tail distribution, we only show the probability density function (PDF) of the burst sequence length less than 30. We find that for long video streams, the burst is often particularly long, and it is difficult to summarize the fixed rules of its length. However, for state transitions or heartbeat packets of sensors and smart cameras, it is a sequence of short bursts with a fixed pattern. Therefore, we select 15 as the segmentation according to the PDF of the burst sequence length and divide a long burst sequence video stream into multiple segmentation bursts. In addition, each camera has several fixed transmission bit rates, so the sizes and packets number of segmentation bursts have a fixed pattern.

## D Dataset

Table 9: Details of datasets

| Type | Dataset | Name | Name |
|---|---|---|---|
| Normal | Ours | 360 camera | Ezviz camera |
| | | Philips camera | Skyworth camera |
| | | Tplink camera | Mercury wirecamera |
| | | Xiaomi camera | Hichip battery camera |
| | | iHorn-temperature | iHorn-door sensor |
| | | iHorn-body sensor | Xiaomi-light sensor |
| | | Xiaodu-doorbell | Aqara-water sensor |
| | | TCL-body sensor | Linksys WRT32X |
| | [32] | Dropcam camera | Samsung camera |
| | | Insteon camera | Nest camera |
| | | Tplink camera | smart sleep sensor |
| | | Netatmo Welcome | NEST smoke alarm |
| | | Smart Baby Monitor | motion sensor |
| Attacks | Ours | Mirai | Service scan |
| | | OS scan | TCP DDoS |
| | | UDP DDoS | |
| | [18, 26] | Aidra | Bashlite |
| | | Mirai | Keylogging |
| | | Data theft | Service scan |
| | | OS scan | HTTP DDoS |
| | | TCP DDoS | UDP DDoS |

Table 10: Detection performance of models trained on the public device set [32]. ≤5e-5 indicates FPR≤5e-5 and ≤ 5e-4 indicates FPR≤5e-4.

| Dataset | Attack | Kitsune |  |  | Magnifier |  |  | HorusEye |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
|  |  | TPR |  | $\text{PR}_{\text{AUC}}$ | TPR |  | $\text{PR}_{\text{AUC}}$ | TPR |  | $\text{PR}_{\text{AUC}}$ |
|  |  | ≤5e-5 | ≤5e-4 |  | ≤5e-5 | ≤5e-4 |  | ≤5e-5 | ≤5e-4 |  |
| [5]<br>[18]<br>[26] | Aidra | 0.595 | 0.611 | 0.854 | 0.620 | 0.657 | 0.823 | **0.623** ↑ 4.7% | **0.662** ↑ 8.3% | **0.880** ↑ 3.0% |
|  | Bashlite | 0.784 | 0.795 | 0.890 | 0.809 | 0.843 | 0.909 | **0.814** ↑ 3.9% | **0.845** ↑ 6.3% | **0.935** ↑ 5.0% |
|  | Mirai | 0.965 | 0.966 | 0.993 | **0.967** | **0.967** | 0.993 | **0.967** ↑ 0.1% | **0.967** ↑ 0.1% | **0.996** ↑ 0.2% |
|  | Keylogging | 0.580 | 0.607 | 0.869 | 0.592 | 0.660 | 0.923 | **0.600** ↑ 3.6% | **0.677** ↑ 11.5% | **0.941** ↑ 8.4% |
|  | Data theft | 0.591 | 0.617 | 0.868 | 0.598 | 0.667 | 0.919 | **0.604** ↑ 2.3% | **0.682** ↑ 10.6% | **0.938** ↑ 8.0% |
|  | Service scan | **0.891** | 0.896 | 0.982 | 0.889 | **0.905** | 0.984 | 0.887 ↓ 0.4% | 0.902 ↑ 0.7% | **0.990** ↑ 0.9% |
|  | OS scan | 0.602 | 0.667 | 0.989 | 0.630 | 0.873 | **0.995** | **0.659** ↑ 9.4% | **0.880** ↑ 14.5% | 0.987 ↓ 0.2% |
|  | HTTP DDoS | 0.674 | 0.744 | 0.984 | 0.728 | 0.851 | **0.994** | **0.731**↑ 8.5% | **0.853** ↑ 14.7% | 0.984 - |
|  | TCP DDoS | 0.961 | 0.965 | 0.989 | 0.969 | 0.980 | 0.993 | **0.970** ↑ 1.0% | **0.981** ↑ 1.7% | **0.994** ↑ 0.5% |
|  | UDP DDoS | 0.961 | 0.966 | 0.989 | 0.970 | 0.980 | 0.993 | **0.969** ↑ 0.8% | **0.980** ↑ 1.5% | **0.993** ↑ 0.5% |
|  | macro | 0.760 | 0.783 | 0.941 | 0.777 | 0.838 | 0.952 | **0.782** ↑ 2.9% | **0.843** ↑ 7.7% | **0.964** ↑ 2.4% |
| Ours | Mirai | 0.661 | 0.761 | 0.986 | 0.812 | 0.871 | 0.994 | **0.817** ↑ 23.5% | **0.888** ↑ 16.7% | **0.995** ↑ 1.0% |
|  | Service scan | **0.993** | 0.994 | **1.000** | 0.991 | **0.997** | **1.000** | 0.991 ↓ 0.2% | **0.997** ↑ 0.3% | **1.000 -** |
|  | OS scan | **0.987** | 0.990 | **1.000** | 0.986 | 0.994 | **1.000** | **0.987 -** | **0.994** ↑ 0.4% | **1.000 -** |
|  | TCP DDoS | 0.997 | 0.998 | **1.000** | **0.998** | 0.998 | **1.000** | **0.998** ↑ 0.1% | **0.999** ↑ 0.1% | **1.000 -** |
|  | UDP DDoS | 0.998 | **0.999** | **1.000** | 0.999 | **0.999** | **1.000** | **0.999** ↑ 0.1% | **0.999 -** | **1.000 -** |
|  | macro | 0.927 | 0.948 | 0.997 | 0.957 | 0.972 | **0.999** | **0.958** ↑ 3.3% | **0.975** ↑ 2.9% | **0.999** ↑ 0.1% |

## E Detection Performance on Public Device Set

Table 10 summarizes the detection performance of models trained on the public device sets [32]. All schemes achieve good results on the dataset [32]. This is because the dataset has fewer devices and does not trigger multiple behavioral interactions like our testbed. Therefore, its traffic can be easily distinguished. Through the experiments on this public dataset, we demonstrate our schemes have excellent generality.